Ben-Gurion University of the Negev
Faculty of Natural Science
Department of Computer Science

Principles of Programming Languages

Mira Balaban

Lecture Notes

May 6, 2017

Many thanks to Tamar Pinhas, Ami Hauptman, Eran Tomer, Barak Bar-Orion, Azzam Maraee, Yaron Gonen, Ehud Barnea, Rotem Mairon, Igal Khitron, Rani Etinger, Ran Anner, Tal Achimeir, Michael Elhadad, Michael Frank for their great help in preparing these notes and the associated code.

Contents

Introduction

This course is about building ***computational processes***. We need computational processes for computing functions, and for performing computational tasks. The means for performing computational processes are ***programs***. The ***power*** and ***weakness*** of a computational process, realized within a program depends on:

1. ***modeling***:
   - How good is the description/understanding of the computational process;
   - How it is split and combined from simpler processes;
   - How clear are the structures used;
   - How natural is the organization of the process;
   - and more.
2. ***language***: How powerful is the language used to write the program:
   - Does it support the needed structures;
   - Does it enable high level thinking, i.e., ***abstraction***, ignoring irrelevant details;
   - Does it enable modular construction;
   - and more.

This course deals with both aspects, with a greater emphasis on programming languages and their properties. The course emphasizes the value of ***modularity*** and ***abstraction*** in modeling, and insists on writing contracts for programs.

What is this course about?
It is about the building, implementation and usage of programming languages in a way that enables construction of manageable, useful and reusable products. The secret word for all this is ***ABSTRACTION!***

The course introduces the following ***subjects***, in various contexts:

1. **Elements of programming languages:**

   (a) **Language building blocks:** Atomic elements; composition means; abstraction means.
   (b) **Language syntax:** Concrete syntax, abstract syntax.

2. **Meaning of a programming language:**
   (a) **Operational semantics:** Applicative (eager), normal (lazy); lexical scoping, dynamic scoping.
   (b) **Types:** Type inference, type checking; dynamic typing, static typing techniques; polymorphic types; type specification language.

3. **Using programming languages for problem solving:**
   (a) **Procedure abstraction:** As a parameter/argument, returned value, data structure component.
   (b) **Abstract data types (ADT):** Abstraction – separation between application to implementation; invariants, operation contracts, ADT embedding, Design by Contract; hierarchical data types; lazy lists.

4. **Meta-programming tools** Interpreter, compiler; static (compile) time evaluation, run-time evaluation.

5. ***Programming styles***:
   (a) Iteration vs. recursion.
   (b) Continuation Passing Style.
   (c) Lazy lists.

6. **Software considerations and conventions:** Contracts, tests, asymptotic complexity, compile time, runtime.

7. **Imperative programming:** State, mutation and change in programming.

The following programming languages ***techniques*** are used:

1. **Substitution operation:** Used in the operational semantics algorithms and in type inference algorithms.

2. **Renaming operation:** Used in the operational semantics algorithms and in type inference algorithms.

3. **Pattern matching, unification operations:** Used in the operational semantics algorithms and in type inference algorithms.

4. **Lazy/eager approaches:** Used in various contexts – semantics, ADT implementation, partial evaluation.

5. **Delayed evaluation**.

6. **Partial evaluation:** Currying.

7. Lexical scoping.

Programming paradigms:

1. ***Functional programming*** (***Scheme, Lisp, ML***): Its origins are in the ***lambda calculus***.

2. ***Logic programming*** (***Prolog):*** Its origins are in mathematical logic.

3. ***Imperative programming*** (***ALGOL-60, Pascal, C):*** Its origins are in the Von-Neumann computer architecture.

For each computational paradigm we define its syntax and operational semantics and implement a meta tool for its evaluation. We use it to solve typical problems,and study essential properties.

Most subjects and techniques are taught using the ***scheme*** language: A small and powerful language, designed for educational purposes.

– ***small*** – It has a very simple syntax, with few details. Can be taught in half an hour.

– ***powerful*** – It combines, in an elegant way, the ideas of functional and imperative programming. It can be easily used to demonstrate **all** programming approaches.

Chapter 1

The Elements of Programming

Functional Programming is a paradigm of programming that is most similar to evaluation of expressions in mathematics. In functional programming a program is viewed as an expression, which is evaluated by successive applications of functions to their arguments, and substitution of the result for the functional expression. Its origin is in the *lambda calculus* of Church.

The most characteristic feature of functional programming is the lack of ***state*** during a computation. That is, a computation is not a sequence of states, created by triggers that modify the states. Rather, a computation is a sequence of expressions, that result from the successive evaluation of sub-expressions. Computation in functional programming has no side-effects, because there are no variables to be assigned. That is, the only result of a functional computation is the computed value, and there are no additional changes that can take place during computation. Variables in functional programming denote values, rather than locations, as in other kinds of programming (***imperative programming***). Functional programming does not include a notion of ***variable assignment***.

We concentrate on functional programming because it is a small and powerful paradigm for introducing fundamental subjects and techniques in teaching programming languages.

Partial sources: SICP [1] 1.1, 1.2, 1.3, 2.2.2 (online version `http://deptinfo.unice.fr/~roy/sicp.pdf`);
HTDP [2] 2.5. (online version `http://htdp.org/2003-09-26/Book/`)

1.1 The Elements of Programming

A language for expressing computational processes needs:

1. ***Primitive expressions***: Expressions whose ***evaluation process*** and hence their ***values*** are built into the language tools.

2. ***Combination means***: Create compound expressions from simpler ones.
3. ***Abstraction***: Manipulate compound objects as stand alone units.
4. ***Reference***: Reference abstracted units by their names.

Scheme possesses these four features in an extremely clean and elegant way:

– ***Common syntax for all composite objects***: Procedure definitions, procedure applications, collection objects: lists. There is a SINGLE simple syntax for all objects that can be ***combined, abstracted, named***, and ***referenced***.

– ***Common semantic status*** for all composite objects, including procedures.

This elegance is based on a uniform approach to ***data*** and ***procedures***: All composite expressions can be viewed both as ***data*** and as ***procedures***. It all depends on how objects are used. Procedures can be data to be combined, abstracted into other procedures, named, and ***applied***.

1.1.1 Expressions (SICP 1.1.1)

Scheme programs are ***expressions***. There are ***atomic*** expressions and ***composite*** expressions. Expressions are ***evaluated*** by the Scheme interpreter. The evaluation process returns a Scheme ***value***, which is an element in a Scheme ***type***. The Scheme interpreter operates in a ***read-eval-print*** loop: It reads an expression, evaluates it, and prints the resulting value.

Atomic expressions:

Some atomic expressions are ***primitive***: Their evaluation process and the returned values are already built into Scheme semantics, and all Scheme tools can evaluate them.

Numbers are primitive atomic expressions in Scheme.

```
> 467
467
;;; This is a primitive expression.
```

Booleans are primitive atomic expressions in Scheme.

```
> #t
#t
```

Primitive procedures are primitive atomic expressions in Scheme.

```
> +
#<procedure:+>
```

Composite expressions:

```
> (+ 45 78)
123
> (- 56 9)
47
> (* 6 50)
300
> (/ 25 5)
5
```

Sole punctuation marks: "(", ")", " ".

Composite expressions are called ***forms*** or ***combinations***. When evaluated, the left-most expression is taken as the ***operator***, the rest are the ***operands***. The ***value*** of the combination is obtained by applying the procedure specified by the operator to the ***arguments***, which are the ***values*** of the operands. Scheme composite expressions are written in Prefix notation.
More examples of combinations:

```
(5 9), (*5 9), (5 * 9), (* (5 9)).
```

Which forms can be evaluated?

Nesting forms:

```
> (/ 25. 6)
4.16666666666667
> (+ 1.5 3)
4.5
> (+ 2 10 75.7)
87.7
> (+ (* 3 15) (- 9 2))
52
> (+ (* 3 (+ (* 2 4) (+ 13 5))) (+ (- 10 5) 3))
86
```

Pretty printing:

```
> (+ (* 3
        (+ (* 2 4)
           (+ 13 5)))
      (+ (- 10 5)
         3))
 86
```

1.1.2 Abstraction and Reference: Variables and Values (SICP 1.1.2)

Naming computational objects is an essential feature of any programming language. Whenever we use naming we have a ***Variable*** that identifies a ***value***. Values turn into ***named objects***. `define` is the Scheme's ***special operator*** for naming. It declares a variable, binds it to a value, and adds the ***binding*** to the ***global environment***.

```
 > (define size 6)
 >
```

The variable `size` denotes the value 6.

> The ***value*** of a `define` combination is `void` – the single value in type `Void`.
> A form with a special operator is called a ***special form***.

```
> size
6
> (* 2 size)
12
> (define a 3)
> (+ a (* size 1.5))
12
> (define result (+ a (* size 1.5)) )
> result
12
```

Note: `size` is an ***atomic expression*** but is ***not a primitive***. `define` provides the simplest means of ***abstraction***: It allows for using names for the results of complex operations.

The global environment: The global environment is a ***function*** from a finite set of variables to values, that keeps track of the name-value bindings. The bindings defined by `define` are added to the `global environment` function. A variable-value pair is called a ***binding***. The global environment mapping can be viewed as (and implemented by) a data structure that ***stores*** bindings.
Characters in variable names: Any character, except space, parentheses, “,”, “ ‘ “, and no “ ’ ” in the beginning. Numbers cannot function as variables.

Note: Most Scheme applications allow redefinition of primitives. But then we get “disastrous” results:

```
> (define + (* 2 3))
> +
6
> (+ 2 3)
```

```
ERROR: Wrong type to apply:  6
; in expression: (... + 2 3)
; in top level environment.
> (* 2 +)
12
```

and even:

```
> (define define 5)
> define
5
> (+ 1 define)
6
> (define a 3)
. . reference to undefined identifier: a
>
```

Note: Redefinition of primitives is a bad practice, and is forbidden by most language applications. ***Evaluation rule for*** `define` ***special forms*** `(define ⟨variable⟩ ⟨expression⟩)`:

1. Evaluate the 2nd operand, yielding the value ***value***.
2. Add the binding: ⟨`⟨variable⟩` ***value***⟩ to the global environment mapping.

1.1.3 Evaluation of Scheme Forms (SICP 1.1.3)

Evaluation of atomic expressions:

1. Special operator symbols are not evaluated.
2. ***Variables*** evaluate the values they are mapped to by the ***global environment*** mapping (via `define` forms).
3. ***Primitive expressions*** evaluate to their denoted values:
   - ***Numbers*** evaluate to their number values;
   - The ***Booleans*** atomic expressions `#t, #f` evaluate to the boolean values ***#t, #f***, respectively;
   - ***Primitive procedures*** evaluate to the machine instruction sequences that perform the denoted operation. We say that "primitives evaluate to themselves".

Note the status of the ***global environment*** mapping in determining the meaning of atomic symbols. The global environment is consulted first. Apart from primitive and special symbols, all evaluations are global environment dependent.

Evaluation of special forms – forms whose first expression is a special operator: Special forms are evaluated by the special evaluation rules of their special operators. For example: in `define` forms the second expression is **not** evaluated.

Evaluation of non-special forms: `(`$expr_0 \ldots expr_n$`)`:

1. ***Evaluate*** all subexpressions $expr_i, i \geq 0$ in the form. The value of $expr_0$ must be of type Procedure, otherwise the evaluation ***fail***s (run-time error).

2. ***Apply*** the procedure which is the ***value*** of $expr_0$, to the ***values*** of the other subexpressions.

The evaluation rule is ***recursive***:

```
> (* (+ 4 (+ 5 7))
     (* 6 1 9))
864
```

Note that `(5* 9)`, `(5 * 9)` and `(not a b)` are syntactically correct Scheme forms, but their evaluation fails.

We can visualize the evaluation process by drawing an ***evaluation tree***, in which each form is assigned an internal node, whose direct descendents are the operator and operands of the form. The evaluation process is carried out from the leaves to the root, where every form node is replaced by the value of applying its operator child to its operands children:

```
------------------864-------------
|                                 |
*   ------16----              ----54-------
    |   |       |             |    |   |   |
    +   4   ---12----         *    6   1   9
            |   |   |
            +   5   7
```

1.1.4 User Defined Procedures (compound procedures)

Procedure construction is an abstraction mechanism that turns a compound operation into a single unit.

Procedure definition: A ***compound (user defined) procedure*** is a value, constructed by the ***special operator "lambda"***. For example, a procedure with a ***parameter*** `x` and ***body*** `(* x x)` is created by evaluation of the following ***lambda form***:

```
> (lambda (x) (* x x))
#<procedure>
```

The procedure created by the evaluation of this `lambda` form is called ***closure***, and denoted ⟨`Closure (x) (* x x)`⟩. Its ***parameters*** is `(x)` and its ***body*** is `(* x x)`. The body is not evaluated when the closure is created. "`lambda`" is called the ***value constructor*** of the Procedure type. The evaluation of the syntactic form `(lambda (x) (* x x))` creates the semantic value ⟨`Closure (x) (* x x)`⟩, i.e., the closure. A procedure is a ***value*** like any other value. The origin of the name ***lambda*** is in the ***lambda calculus***.

Procedure application: Applications of the above procedure:

```
> ( (lambda (x) (* x x))
    5 )
25
> ( (lambda (x) (* x x))
    (+ 2 5) )
49
```

In an application of a compound procedure, occurrences of the parameters in the body are replaced by the arguments of the application. The body expressions are evaluated in sequence. The value of the procedure application is the value of the last expression in the body.
Nested applications:

```
 > ( + ( (lambda (x) (* x x))
         3)
       ( (lambda (x) (* x x))
         4) )
 25
 >
```

Multiple expression body and side-effects: The ***body*** of a ***lambda*** expression can include several Scheme expressions:

```
> ( (lambda (x) (+ x 5) 15 (* x x))
    3)
9
```

But – no point in including several expressions in the body, since the value of the procedure application is that of the last one. Having several expressions in a body is useful only when their evaluations have ***side effects***, i.e., affect some external state of the computation environment. For example, a sequence of scheme expressions in a procedure body can be used for debugging purposes:

```
> ( (lambda (x) (display x)
               (* x x))
    3)
39
> ( (lambda (x) (display x)
                (newline)
                (* x x))
    3)
3
9
>
```

`display` is a primitive procedure of Scheme. It evaluates its argument and displays it:

```
> display
#<procedure:display>
> newline
#<procedure:newline>
>
```

`display` is a ***side effect*** primitive procedure! It displays the value of its argument, but its returned value (like the special operator `define`), is `void`, the single value in `Void`, on which no procedure is defined.
Style Rule: `display` and `newline` forms should be used only as internal forms in a procedure body. A deviation from this style rule is considered an ***bad style***. The procedure below is a ***bad style*** procedure – why? What is the type of the returned value?

```
(lambda (x) (* x x)
             (display x)
             (newline))
```

Demonstrate the difference from:

```
(lambda (x) (display x)
            (newline)
            (* x x))

> ( (lambda (x) (* x x)
            (display x)
            (newline))
   3)
3
> (+ ( (lambda (x) (* x x)
             (display x)
             (newline))
       3)
     4)
3
. . +: expects type <number> as 1st argument, given: #<void>; other
 arguments were: 4
 >
```

Summary:

1. The Procedure type consists of ***closures***: user defined procedures.
2. A closure is created by application of `lambda` – the ***value constructor*** of the Procedure type: It constructs values of the Procedure type.
   The syntax of a lambda form: (`lambda` ⟨parameters⟩ ⟨body⟩).
   ⟨parameters⟩ syntax is: (⟨`variable`⟩ ... ⟨`variable`⟩), 0 or more.
   ⟨body⟩ syntax is ⟨Scheme-expression⟩ ... ⟨Scheme-expression⟩, 1 or more.
3. A Procedure value is ***composite***: It has 2 parts: ***parameters*** and ***body***. It is symbolically denoted: ⟨Closure ⟨parameters⟩ ⟨body⟩⟩.
4. When a procedure is created it is not necessarily applied! Its body is **not** evaluated.

Naming User Procedures (compound procedures): A definition of a compound procedure associates a name with a procedure value. Naming a compound procedure is an abstraction means that allows for multiple applications of a procedure, defined only once. This is a major abstraction means: The procedure name stands for the procedure operation. An explicit application using ***anonymous*** procedures:

```
> ((lambda (x) (* x x)) 3)
9
> ( + ( (lambda (x) (* x x))
        3)
      ( (lambda (x) (* x x))
        4) )
25
```

Can be replaced by:

```
> (define square (lambda (x) (* x x)))
> (square 3)
9
```

```
> square
#<procedure:square>

> (define sum-of-squares
     (lambda (x y)
        (+ (square x) (square y ))))
> (sum-of-squares 3 4)
25
```

Recall that the evaluation of the `define` form is dictated by the `define` evaluation rule:

1. Evaluate the 2nd parameter: The `lambda` form – returns a procedure value: ⟨Closure (x) (* x x)⟩
2. Add the following ***binding*** to the global environment mapping:
   `square -- #<procedure>`

Note that a procedure is treated as just **any** value, that can be given a name!! Also, distinguish between ***procedure definition*** to ***procedure call/application***.
Syntactic sugar for named procedure-definition: A special, more convenient, syntax for procedure definition:

```
 > (define (square x) (* x x))
```

This is just a ***syntactic sugar***: A special syntax, introduced for the sake of convenience. It is replaced by the ***real syntax*** during pre-processing. It is not evaluated by the interpreter.
The syntax of the syntactic sugar syntax of named procedure-definition:

```
  (define (<name> <parameters>) <body>)
```

It is transformed into:

```
  (define <name> (lambda (<parameters>) <body>))

> (square 4)
16
> (square (+ 2 5))
49
> (square (square 4))
256
> (define (sum-of-squares x y)
    (+ (square x) (square y )))

> (sum-of-squares 3 4)
25
```

```
> (define (f a)
    (sum-of-squares (+ a 1) (* a 2)) )

> (f 3)
52
```

Intuitively explain these evaluations! Note: We did not provide, yet, a formal semantics!

Summary:

1. In a `define` special form for procedure definition: First: the 2nd argument is evaluated (following the `define` evaluation rule), yielding a new procedure.
   Second: The binding ⟨`variable - `⟨`Closure <parameters> <body>`⟩⟩ is added to the global environment.
2. The `define` special operator has a syntactic sugar that hides the call to `lambda`.
3. Procedures that are not named are called ***anonymous***.

1.1.5 Conditional Expressions

So far, the evaluation of a procedure application yields the same computation path for all inputs. Addition of computation branching is enabled by evaluating ***condition expressions***. They are formed by two special operators: `cond` and `if`.

```
(define abs
  (lambda (x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          (else (- x)))
  ))
> (abs -6)
6
> (abs (* -1 3))
3
```

The syntax of a `cond` form:

```
 (cond (<p_1> <e_11> ... <e_1k_1>)
       (<p_2> <e_21> ... <e_2k_2>)
       ...
       (else <e_n1> ... <e_nk_n>) )
```

The arguments of `cond` are ***clauses*** `(<p> <e1> ...<en>)`, where `<p>` is a ***predication*** (boolean valued expression) and the `<ei>`s are any Scheme expressions. A predication is an expression whose value is false or true. The operator of a predication is called a ***predicate***. The ***false*** value is represented by the symbol #f, and the ***true*** value is represented by the symbol #t.

```
> #f
#f
> #t
#t
> (> 3 0)
#t
> (< 3 0)
#f
>
```

Evaluation of a conditional expressions:

$$
\begin{array}{l}
\texttt{(cond (<}p_1\texttt{> <}e_{11}\texttt{> ... <}e_{1k_1}\texttt{>)} \\
\qquad\quad \texttt{(<}p_2\texttt{> <}e_{21}\texttt{> ... <}e_{2k_2}\texttt{>)} \\
\qquad\quad \texttt{...} \\
\qquad\quad \texttt{(else <}e_{n1}\texttt{> ... <}e_{nk_n}\texttt{>))}
\end{array}
$$

$<p_1>$ is evaluated first. If the value is false then $<p_2>$ is evaluated. If its value is false then $<p_3>$ is evaluated, and so on until a predication $<p_i>$ with a non-false value is reached. In that case, the $<e_i>$ elements of that ***clause*** are evaluated, and the value of the last element $<e_{ik_i}>$ is the value of the `cond` form. The last `else` clause is an ***escape*** clause: If no predication evaluates to true (anything not false), the expressions in the `else` clause are evaluated, and the value of the `cond` form is the value of the last element in the `else` clause.
Definition: A ***predicate*** is a procedure (primitive or not), that returns the values true or false.

```
>(cond (#f #t)
      ( else #f) )
#f
```

Note: Scheme considers every value different from `#f` as true.

Another syntax for conditionals: `if` forms

```
(define abs
  (lambda (x)
    (if (< x 0)
        (- x)
        x)))
```

`if` is a restricted form of `cond`. The syntax of `if` expressions: `(if <predication> <consequent> <alternative>)`.
The `if` evaluation rule: If `<predication>` is true the `<consequent>` is evaluated and returned as the value of the `if` special form. Otherwise, `<alternative>` is evaluated and its value returned as the value of the `if` form. Conditional expressions can be used like any other expression:

```
> (define a 3)
> (define b 4)
> (+ 2 (if (> a b) a b) )
6
> (* (cond ( (> a b) a)
           ( (< a b) b)
           (else -1   ) )
     (+ a b) )
28
> ( (if (> a b) + -) a b)
-1
```

Example 1.1 (Newton's method for computing square roots)**.** *(SICP 1.1.7)*

In order to compute `square-root` we need a method for computing the function `square-root`. One such method is Newton's method. The method consists of successive application of steps, each of which improves a former approximation of the square root. The improvement is based on the ***proved property*** that if `y` is a non-zero guess for a square root of `x`, then `(y + (x/y)) / 2` is a better approximation. The computation starts with an arbitrary non-zero guess (like 1). For example:

```
x=2, initial-guess=1
1st step:  guess=1   improved-guess= (1+ (2/1))/2 = 1.5
2nd step:  guess=1.5   improved-guess= (1.5 + (2/1.5) )/2 = 1.4167
```

A close look into Newton's method shows that it consists of a ***repetitive*** step as follows:

1. Is the current guess close enough to the `square-root`? (`good-enough?`)
2. If not – compute a new guess. (`improve`).

Call the step `sqrt-iter`. Then the method consists of repeated applications of `sqrt-iter`.
The following procedures implement Newton's method:

```
(define sqrt
   (lambda (x) (sqrt-iter 1 x)))

(define sqrt-iter
  (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x)
                   x))))

(define improve
  (lambda (guess x)
     (average guess (/ x guess))))

(define average
  (lambda (x y)
     (/ (+ x y) 2)))

(define good-enough?
  (lambda (guess x)
     (< (abs (- (square guess) x)) .001)))

 > (sqrt 6.)
 ERROR: unbound variable:  square
 ; in expression: (... square guess)
 ; in scope:
 ;   (guess x)

> (define square (lambda (x) (* x x)))

> square
  #<procedure:square>
> (sqrt 6.)
2.44949437160697
> (sqrt (+ 100 44.))
12.0000000124087
> (sqrt (+ (sqrt 2) (sqrt 9.)))
2.10102555114187
> (square (sqrt 4.))
4.00000037168919
```

Summary: Informal Syntax and Evaluation Semantics:

Syntax: There are 2 kinds of Scheme language expressions:

1. **Atomic:**
   (a) Primitives:
      – Number symbols.
      – Boolean symbols: `#t, #f`.
      – Primitive procedure symbols.

   (b) Non-primitives:
      – Variable symbols.
      – Special operator symbols.

2. **Composite:**
   – Special forms: `( <special operator> <exp> ... <exp> )`.
   – Forms: `( <exp> ... <exp> )`.

Evaluation semantics:

1. Atomic expressions: Special operators are not evaluated; Variables evaluate to their associated values given by the global environment mapping; primitives evaluate to their defined values.

2. Special forms: Special evaluation rules, depending on the special operator.

3. Non-special forms:
   (a) Evaluate all subexpressions.
   (b) Apply the procedure which is the value of the first subexpression to the values of the other subexpressions. For user defined procedures, the application involves ***substitution*** of the argument values for the procedure parameters, in the procedure body expressions. The returned value is the value of the last expression in the body.

1.2 Procedure Design Using Contracts

Following Felleison, Findler, Flatt and Krishnamurthi: How to Design Programs
`http://www.htdp.org/2003-09-26/Book/`

Assume that we need to compute the area of a ring, given its outer and inner radius. We start with a ***contract***, which provides information about the desired procedure:

```
Signature: area-of-ring(outer,inner)
Purpose: To compute the area of a ring whose radius is
            'outer' and whose hole has a radius of 'inner'
Type: [Number*Number -> Number]
Example: (area-of-ring 5 3) should produce 50.24
Pre-conditions: outer >= 0, inner >= 0, outer >= inner
Post-condition: result = PI * outer^2 - PI * inner^2
Tests:  (area-of-ring 5 3) ==> 50.24
```

Signature specifies the name of the procedure, and its parameters.
Purpose is a short textual description.
Type specifies the types of the input parameters and of the returned value. The types of number-valued and boolean-valued expressions are Number and Boolean, respectively. The type of a procedure is denoted: `[<type of arg1>*...*<type of argn> -> <return type>]`.
Example gives examples of procedure applications.
Pre-conditions specify conditions that the input parameters are obliged to satisfy.
Post-conditions specify conditions that the returned value is responsible to satisfy.
Invariants describe inter-relationships that must hold between the input parameters.
Tests provide test cases.

Once the contract is written and validated, it is time to provide an implementation that *satisfies* the contract.

```
Signature: area-of-ring(outer,inner)
Purpose: To compute the area of a ring whose radius is
            'outer' and whose hole has a radius of 'inner'
Type: [Number*Number -> Number]
Example: (area-of-ring 5 3) should produce 50.24
Pre-conditions: outer >= 0, inner >= 0, outer >= inner
Post-condition: result = PI * outer^2 - PI * inner^2
Tests:  (area-of-ring 5 3) ==> 50.24
Definition: [refines the contract]
 (define area-of-ring
    (lambda (outer inner)
       (- (area-of-disk outer)
          (area-of-disk inner))))
```

In this implementation `area-of-ring` is a ***caller*** of `area-of-disk`.

A procedure ***contract*** that includes:

1. Signature
2. Purpose

3. Type
4. Example
5. Pre-conditions
6. Post-conditions
7. Invariantss
8. Tests

Signature, purpose and type are **mandatory** to all procedures. Examples are **desirable**. Tests are **mandatory** for complex procedures. Pre/post-conditions and invariants are relevant only if non-trivial (not vacuous).

The specification of ***Types, Pre-conditions*** and ***Post-conditions*** requires special ***specification languages***. The keyword `result` belongs to the specification language for post-conditions.

1.2.1 The Design by Contract (DbC) approach:

DbC is an approach for designing computer software. It prescribes that software designers should define precise verifiable interface specifications for software components based upon the theory of abstract data types and the conceptual metaphor of business contracts. The approach was introduced by Bertrand Meyer in connection with his design of the Eiffel object oriented programming language and is described in his book "Object-Oriented Software Construction" (1988, 1997).

The central idea of DbC is a metaphor on how elements of a software system collaborate with each other, on the basis of mutual ***obligations*** and ***benefits***. The metaphor comes from business life, where a ***client*** and a ***supplier*** agree on a ***contract***. The contract defines ***obligations*** and ***benefits***. If a routine provides a certain functionality, it may:

– ***Impose*** a certain obligation to be guaranteed on entry by any client module that calls it: The routine's ***precondition*** – an ***obligation*** for the client, and a ***benefit*** for the supplier.

– ***Guarantee*** a certain property on exit: The routine's postcondition is an ***obligation*** for the supplier, and a ***benefit*** for the client.

– ***Maintain*** a certain property, assumed on entry and guaranteed on exit: An ***invariant***.

The contract is the formalization of these obligations and benefits.

```
---------------------------------------------------------------------
            |        Client                   Supplier
------------|--------------------------------------------------------
Obligation: | Guarantee precondition       Guarantee postcondition
Benefit:    | Guaranteed postcondition     Guaranteed precondition
---------------------------------------------------------------------
```

DbC is an approach that emphasizes the value of developing program specification together with programming activity. The result is more reliable, testable, documented software.

DbC is crucial for software correctness.

Many languages have now tools for writing and enforcing contracts: Java, C#, C++, C, Python, Lisp, Scheme:
`http://www.ccs.neu.edu/scheme/pubs/tr01-372-contract-challenge.pdf`
`http://www.ccs.neu.edu/scheme/pubs/tr00-366.pdf`
The contract language is a language for specifying constraints. Usually, it is based in Logic. There is no standard, overall accepted contract language: Different languages have different contract languages. In Eiffel, the contracts are an integral part of the language. In most other languages, contract are run by additional tools.

Policy of the PPL course:

1. All assignment papers must be submitted with contracts for procedures.
2. Contract ***mandatory*** parts: The ***Signature, Purpose, Type, Tests*** are mandatory for every procedure!
3. The ***Examples*** part is always recommended as a good documentation.
4. ***Pre-conditions*** should be written when the type does not prevent input for which the procedure does not satisfy its contract. The pre-condition can be written in English. When a pre-condition exists it is recommended to provide a ***precondition-test*** procedure that checks the pre-condition. This procedure **is not** part of the supplier procedure (e.g., not part of area-of-ring) (why?), but should be called by a client procedure, prior to calling the supplier procedure.
5. ***Post-conditions*** are recommended whenever possible. They clarify what the procedure guarantee to supply. Post-conditions provide the basis for tests.

Continue the `area-of-ring` example: The area-of-ring is a ***client*** (a ***caller***) of the `area-of-disk` procedure. Therefore, it must consider its contract, to verify that it fulfills the necessary pre-condition. Here is a contract for the `area-of-disk` procedure:

```
Signature: area-of-disk(radius)
Purpose: To compute the area of a disk whose radius is the
            'radius' parameter.
Type: [Number -> Number]
Example: (area-of-disk 2) should produce 12.56
Pre-conditions: radius >= 0
Post-condition: result = PI * radius^2
Tests:  (area-of-disk 2) ==> 12.56
Definition: [refines the contract]
  (define area-of-disk
    (lambda (radius)
      (* 3.14 (* radius radius))))
```

`Area-of-ring` must fulfill `area-of-disk` precondition when calling it. Indeed, this can be proved as correct, since both parameters of `area-of-disk` are not negative. The post condition of `area-of-ring` is correct because the post-condition of `area-of-disk` guarantees that the results of the 2 calls are indeed, $PI * outer^2$ and $PI * inner^2$, and the definition of `area-of-ring` subtracts the results of these calls.

> We expect that whenever a client routine calls a supplier routine the client routine will either explicitly call a pre-condition test procedure, or provide an argument for the correctness of the call!

Defensive programming is a style of programming, where each procedure first tests its pre-condition. This is **not recommended**, as it might interfere with the clarity of the intended procedure code. Guaranteeing correctness of the pre-conditions is the responsibility of the clients.

Determining the type of conditionals: The type of a conditional expression depends on the type of the values of the clauses of the conditional. But what if different conditionals evaluate to values that belong to different types. For example, the value of `(if x 3 #f)` depends on the value of `x`: might be `3` and might be `#f`. So, what is the type of such conditional expressions? `Number`? `Boolean`?

Example 1.2 (Bounded computation of Newton's method for computing square roots).

Example 1.1 presents an implementation for Newton's approximation method for computing square roots. The contract and implementation are as follows:

```
Signature: sqrt(x)
Purpose: To compute the square root of x, using Newton's approximations method
Type: [Number -> Number]
Example: (sqrt 16.) should produce 4.000000636692939
```

```
Pre-conditions: x >= 0
(define sqrt
   (lambda (x) (sqrt-iter 1 x)))
Signature: sqrt-iter(guess,x)
Purpose: To compute the square root of x, starting with 'guess' as initial guess
Type: [Number*Number -> Number]
Example: (sqrt 1 16.) should produce 4.000000636692939
Pre-conditions: x >= 0, guess != 0
(define sqrt-iter
  (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x)
                   x))))
```

Assume that we have a request to compute the square root only if the computation does not exceed a given number of iterations. Otherwise, the necessary action varies by the caller request. Therefore, if the number of approximations exceeds the bound, we cannot just return an error. One possibility is to return a value that marks this case, for example `#f`. We need to precede each iteration by testing against the bound:

```
Signature: bounded-sqrt(x,bound)
Purpose: To compute the square root of x, using Newton's approximations
method, if number of iterations does not exceed 'bound'
Type: [Number*Number -> ?]
Example: (sqrt 16. 7) should produce 4.000000636692939
         (sqrt 16. 4) should produce #f
Pre-conditions: x >= 0, bound >= 0
(define bounded-sqrt
   (lambda (x bound)
     (bounded-sqrt-iter 1 x bound)))

Signature: bounded-sqrt-iter(guess,x,bound)
Purpose: To compute the square root of x, starting with 'guess' as
initial guess, if number of iterations does not exceed 'bound'
Type: [Number*Number*Number -> ?]
Example: (sqrt 1 16. 7) should produce 4.000000636692939
         (sqrt 1 16. 4) should produce #f
Pre-conditions: x >= 0, bound >= 0, guess != 0
(define bounded-sqrt-iter
  (lambda (guess x bound)
    (if (zero? bound)
        #f
        (if (good-enough? guess x)
            guess
            (bounded-sqrt-iter (improve guess x) x (sub1 bound))))
  ))

> (bounded-sqrt 16. 7)
4.000000636692939
> (bounded-sqrt 16. 6)
4.000000636692939
>  (bounded-sqrt 16. 5)
#f
```

What is the return type of `bounded-sqrt,bounded-sqrt-iter`?
The problem is in the conditional that has clauses that return values from different types: `Number` and `Boolean`. In order to accommodate such conditionals we allow `union` types in contract specifications. The resulting contracts:

```
Signature: bounded-sqrt(x,bound)
Purpose: To compute the square root of x, using Newton's approximations
method, if number of iterations does not exceed 'bound'
Type: [Number*Number -> Number union Boolean]
Example: (sqrt 16. 7) should produce 4.000000636692939
         (sqrt 16. 4) should produce #f
Pre-conditions: x >= 0, bound >= 0

Signature: bounded-sqrt-iter(guess,x,bound)
Purpose: To compute the square root of x, starting with 'guess' as
initial guess, if number of iterations does not exceed 'bound'
Type: [Number*Number*Number -> Number union Boolean]
Example: (sqrt 1 16. 7) should produce 4.000000636692939
         (sqrt 1 16. 4) should produce #f
Pre-conditions: x >= 0, bound >= 0, guess != 0
```

Union typed conditionals (and procedures) is a bad programming style: Union types are problematic for type checking, let alone, for type inference. Therefore, they are a major cause for runtime errors. We should strive to avoid using such conditionals.

Side comment on artificial "flag" values: The returned value `#f` in the last example is a flag value, that signs a failure to compute the square root under the given bound. The only role of this value is to signal this case. Using flags is not a good style. There are various programming approaches that help in avoiding them. For example, an approach that abstracts the results of the clauses of a conditional as procedure parameter can help here. Such an approach, named *Continuation Passing Style* (*CPS*) is taught in Chapter 3 (an example appears also at the end of Section 1.5.5 in this chapter). The idea is to replace flags in "conditional-legs" by ***continuation procedures***, that are given as additional parameters:

```
(define f
  (lambda (x)
    (if (<predicate> x)
        #f
        (<some-computation> x)) ))
==>
(define f
  (lambda (x consequent-procedure alternative-procedure)
    (if (<predicate> x)
        (consequent-procedure x)
        (alternative-procedure x)) ))
```

This way, a caller of `f` instead of receiving the flag value `#f` and determining the desired action based on it, directly pass the consequent action and the alternative actions as arguments, when calling `f`.

1.3 Hierarchical Data types: Pairs and Lists

We already saw that abstractions can be formed by ***compound procedures*** that model processes and general methods of computation. This chapter introduces abstractions formed by ***compound data***. The ability to combine data entities into compound entities, that can be further combined adds additional power of abstraction: The entities that participate in the modeled process are no longer just atomic, unrelated entities, but can be organized into some relevant structures. The modeled world is not just an unordered collection of elements, but has an internal structure.

Hierarchical data types also complement the fourth feature of procedures as first class citizens in functional languages: They can be included in data structures.

1.3.1 The Pair Type

Pairs are a basic compound data object in data modeling. A pair combines 2 data entities into a single unit, that can be further manipulated by higher level conceptual procedures.

In Scheme, Pair is the basic type, for which the language provides primitives. The ***value constructor*** for pairs in Scheme is the primitive procedure `cons`, and the primitive procedures for ***selecting*** the first and second elements of a pair are `car` and `cdr`, respectively. Its identifying predicate is `pair?`, and the equality predicate is `equal?`.

```
> (define x (cons 1 2))
> (car x)
1
> (cdr x)
2
> x
(1 . 2)
```

This is an example of the ***dotted notation*** for pairs: This is the way Scheme prints out values of the Pair type.
Syntax of pair creation: `(cons <exp> <exp>)`
Value (semantics) of pair creation: A dotted pair

```
> (define y (cons x 5))
> (car y)
(1 . 2)
> (cdr y)
5
> y
((1 . 2) . 5)
> (define z (cons x y))
> (car z)
(1 . 2)
> (cdr z)
((1 . 2) . 5)
> (car (car z))
1
> (car (cdr z))
(1 . 2)
> z
((1 . 2) (1 . 2) . 5)
```

This is the way Scheme denotes (prints out) Pair values whose `cdr` is a pair. It results from the way Scheme denotes (prints out) ***List values*** – data objects that will be discussed later.

```
> (cdr (cdr z))
5
> (car (car y))
1
> (car (cdr y))
ERROR: car: Wrong type in arg1 2
```

```
Mixed-type pairs:
> (define pair (cons 1 #t))
> (pair? pair)
#t
> (pair? (car pair))
#f
> (car (car pair))
car: expects argument of type <pair>; given 1
> (procedure? pair?)
#t

Pairs whose components are procedures:
> (define proc-pair (cons + -))
> proc-pair
'(#<procedure:+> . #<procedure:->)
> (car proc-pair)
#<procedure:+>
> ((cdr proc-pair) 3)
-3
```

Denotation of Pair values: The distinction between Pair typed **Scheme expressions** (syntax), to their Pair **values** (semantics) must be made clear. `(cons 1 2)` is a Scheme expression that evaluates into the Pair value `(1 . 2)`. The Pair value cannot be evaluated by the interpreter. Evaluation of `(1 . 2)` causes an error.

Example 1.3 (A procedure for constructing pairs of successive numbers)**.**

```
Signature: successive-pair(n)
Purpose: Construct a pair of successive numbers: (n . n+1).
Type: [Number -> Pair(Number,Number)]
(define successive-pairself
  (lambda(n) (cons n (+ n 1))))
> (successive-pair 3)
(3 . 4)
> (pairself (+ 3 4))
(7 . 8)
```

The type of pair-valued expressions is denoted `Pair(<type of 1st>,<type of 2nd>)`.

1.3.2 Symbol values

Symbols, which are variable names, can be used as variable values. They form the `Symbol` type. This is Scheme's approach for supporting symbolic information. The values of the

Symbol type are **atomic names**, i.e., (unbreakable) sequences of keyboard characters (to distinguish from strings). Values of the Symbol type are introduced via the special operator `quote`, that can be abbreviated by the ***macro character*** '. `quote` is the value constructor of the Symbol type; its parameter is any sequence of keyboard characters. The Symbol type identifying predicate is `symbol?` and its equality predicate is `eq?`. It has no operations (a degenerate type).

```
> (quote a)
'a
;the " ' " in the returned value marks that this is a returned semantic
;value of the Symbol type, and not a syntactic variable name
> 'a
'a
> (define a 'a)
> a
'a
> (define b 'a)
> b
'a
> (eq? a b)
#t
> (symbol? a)
#t
> (define c 1)
> (symbol? c)
#f
> (number? c)
#t
> (= a b)
. =: expects type <number> as 2nd argument, given: a; other arguments
  were: a
>
```

Notes:

1. The Symbol type differs from the String type: Symbol values are unbreakable names. String values are breakable sequences of characters.

2. The preceding " ' " letter is a ***syntactic sugar*** for the `quote` special operator. That is, every `'a` is immediately pre-processed into `(quote a)`. Therefore, **the expression `'a` is not atomic!** The following example is contributed by Mayer Goldberg:

   ```
   > (define 'a 5)
   ```

```
> 'b
. . reference to undefined identifier: b
> quote
#<procedure:quote>
> '0
5
> '5
5
```

Explain!

3. `quote` is a special operator. Its parameter is any sequence of characters (apart of few punctuation symbols). It has a special evaluation rule: It returns its argument as is – no evaluation. This differs from the evaluation rule for primitive procedures (like `symbol?` and `eq?`) and the evaluation rule for compound procedures, which first evaluate their operands, and then apply.
   Question: What would have been the result if the operand of `quote` was evaluated as for primitive procedures?

Pair components can be used for tagging information:

```
A tagging procedure:
Signature: tag(tg,n)
Purpose: Construct a pair of tg and n.
Type: [Symbol*(Number union Symbol) -> Pair(Symbol,(Number union Symbol))]
(define tag
  (lambda (tg n)
    (cons 'tg n)))

(define get-tag
  (lambda (tagged-pair)
    (car tagged-pair)))

(define get-content
  (lambda (tagged-pair)
    (cdr tagged-pair)))

> (define tagged-name (cons 'roman-name 'Augustus))
> (get-tag tagged-name)
'roman-name
> (get-content tagged-name)
'Augustus
```

1.3.3 The List Type

Lists represent finite ***sequences***, i.e., ordered collections of data elements (compound or not). The sequence $\langle v_1, v_2, \ldots, v_n\rangle$ is represented by the list $(v_1 v_2 \ldots v_n)$. The empty sequence is represented by the ***empty list***. `List` is recursive type (inductively defined):

1. The ***empty list*** `()`.
2. For a list $(v_1 v_2 \ldots v_n)$, and a value v_0, $(v_0 v_1 v_2 \ldots v_n)$ is a list, whose ***head*** is v_0, and ***tail*** is $(v_1 v_2 \ldots v_n)$.

The `List` type has two ***value constructors***: `cons` and `list`.

1. The empty list is constructed by the value constructor `list`: The value of the expression `(list)` is `()`. The empty list `()` is also the value of two built in variables: `null` and `empty`.
2. A non-empty list can be constructed either by `cons` or by `list`.
   (a) `(cons head tail)`: `cons` takes two parameters. `head` is an expression denoting some value v_0, and `tail` is an expression whose value is a list $(v_1 v_2 \ldots v_n)$. The value of `(cons head tail)` is the list $(v_0 v_1 v_2 \ldots v_n)$.
   (b) `(list e1 e2 ... en)`: `list` takes indefinite number of parameters. If `e1, e2..., en` ($n \geq 0$) are expressions whose values are $(v_1, v_2, \ldots v_n)$, respectively, then the value of `(list e1 e2 ... en)` is the list $(v_1 v_2 \ldots v_n)$. For $n = 0$ `(list)` evaluates to the empty list.

List implementation: Scheme lists are implemented as nested pairs, starting from the empty list value. The list created by $(conse1(conse2(cons...(consen(list))...)))$ is implemented as a nested pair (it's the same value constructor, anyway), and the list created by `(list e1 e2 ... en)` is translated into the former nested `cons` application.
Printing form: The ***printing form*** of lists is: `(<a1> <a2> ... <an>)`. The printing form describes the value (semantics) of List-typed expressions (syntax).

The selectors of the List type are `car` – for the 1st element, and `cdr` – for the tail of the given list (which is a list). The predicates are `list?` for identifying List values, and `null?` for distinguishing the empty list from all other lists. The equality predicate is `equal?`.

```
> (define one-through-four (cons 1 (cons 2 (cons 3 (cons 4 (list) )))))
> one-through-four
(1 2 3 4)
> (car one-through-four)
1
> (cdr one-through-four)
(2 3 4)
```

```
> (car (cdr one-through-four))
2
> (cons 10 one-through-four)
(10 1 2 3 4)

> (list 3 4 5 7 8)
(3 4 5 7 8)
> (define x (list 5 6 8 2))
> x
(5 6 8 2)

> (define one-through-four (list 1 2 3 4))
>one-through-four
(1 2 3 4)
>(car one-through-four)
1
>(cdr one-through-four)
(2 3 4)
>(car (cdr one-through-four))
2
>(cons 10 one-through-four)
(10 1 2 3 4)
```

Note: The value constructor `cons` is considered more basic for lists. `(list <a1> <a2> ... <an>)` can be (and is usually) implemented as `(cons <a1> (cons <a2> (cons...(cons <an> (list))...)))`.
Note on the Pair and List value constructors and selectors: The Pair and the List types have the same value constructor `cons`, and the same selectors `car, cdr`. This is unfortunate, but is actually the Scheme choice. Scheme can live with this confusion since it is not statically typed (reminder: a language is statically typed if the type of its expressions is determined at compile time.) A value constructed by `cons` can be a Pair value, and also a List value – in case that its 2nd element (its `cdr`) is a List value. At run time, the selectors `car` and `cdr` can be applied to every value constructed by `cons`, either a list value or not (it is always a Pair value).
Note: Recall that some Pair values are not printed "properly" using the printed form of Scheme for pairs. For example, we had:

```
 > (define x (cons 1 2))
 > (define y (cons x (quote a)))
 > (define z (cons x y))
 > z
((1 . 2) (1 . 2) . a)
```

while the printed form of `z` should have been: `((1 . 2) . ( (1 . 2) . a))`. The reason is that the principal type of Scheme is List. Therefore, the Scheme interpreter tries to interpret every `cons` value as a list, and only if the scanned value encountered at the list end appears to be different than `(list)`, the printed form for pairs is restored. Indeed, in the last case above, `z = (cons (cons 1 2) (cons (cons 1 2) 'a))` is not a list.

Visual representation of lists – Box-Pointer diagrams: Box-Pointer diagrams are a helpful visual mode for clarifying the structure of hierarchical lists and complex Pair values.

– The empty list `( )` is visualized as a box:

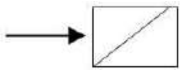


Figure 1.1: Visual representation for the empty list

– A non empty list is visualized as a sequence of ***2-cell boxes***. Each box has a pointer to its content and to the next box in the list visualization. The list `((1 2) ((3)) 4)` is visualized by:

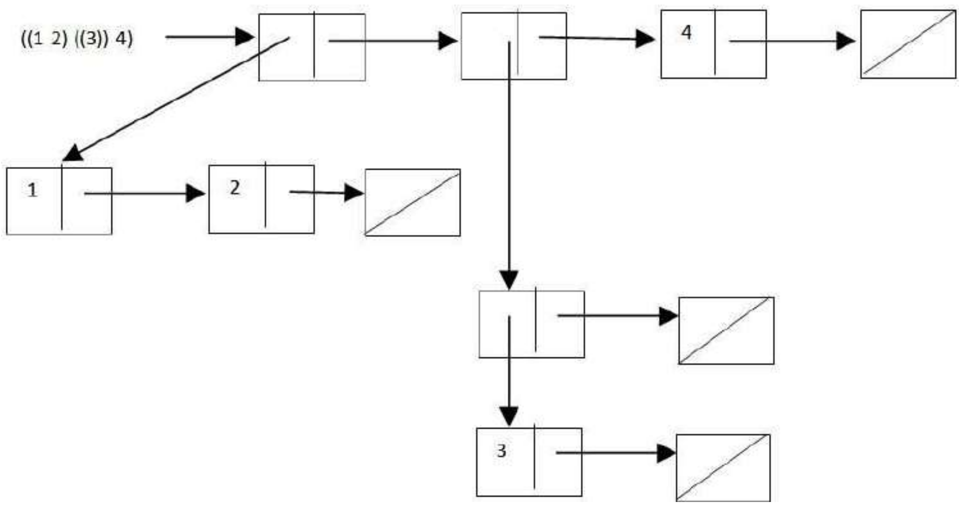


Figure 1.2: Box-pointer Visual representation of lists

Complex list values that are formed by nested applications of the list value constructor, are represented by a ***list skeleton*** of box-and-pointers, and the nested elements form the box contents.

Note: The layout of the arrows in the box-and-pointer diagrams is irrelevant. The arrow pointing to the overall diagram is essential – it stands for the hierarchical data object as a whole.
Predicate `null?`: Tests if the list is empty or not.

```
> null?
#<primitive:null?>
> (null? (list))
#t
> null
'()
> (equal? '( ) null)
#t
> (null? null)
#t
> (define one-through-four (cons 1 (cons 2 (cons 3 (cons 4 (list) )))))
> (null? one-through-four)
#f
```

Identification predicate `list?`: Tests whether its argument is a list.

```
> list?
#<procedure:list?>
> (list? (list))
#t
> (list? one-through-four)
#t
> (list? 1)
#f
> (list? (cons 1 2))
#f
> (list? (cons 1 (list)))
#t
```

Homogeneous and Heterogeneous lists Lists of elements with a common type are called ***homogeneous lists***, while lists whose elements have no common type are termed ***heterogeneous lists***. For example, `(1 2 3), ((1 2) (3 4 5))` are homogeneous lists, while `((1 2) 3 ((4 5)))` is a heterogeneous list. This distinction divides the List type into two types: Homogeneous-List and Heterogeneous-List. The empty list belongs both to the Homogeneous-List and the Heterogeneous-List types.

The Homogeneous-List type is a polymorphic type, with a type constructor `List` that takes a single parameter. For example:

`List(Number)` is the type of Number lists;
`List(Pair(Number, Number))` is the type of number pair lists;
`List(List(Symbol))` is the type of symbol list lists.
The heterogeneous-List type includes all heterogeneous lists. It is not polymorphic. Its type constructor is `List` with no parameters.

All statically typed languages, e.g., JAVA and ML, support only homogeneous list values. Heterogeneous list values like the above are defined as values of some hierarchical type like n-TREE.

1.3.3.1 List operations and procedures:

1. **Composition of `cons`, `car`, `cdr`:**

```
> (define x (list 5 6 8 2))
> x
(5 6 8 2)
> (car x)
5
> (cdr x)
(6 8 2)
> (cadr x)
6
> (cddr x)
(8 2)
> (caddr x)
8
> (cdddr x)
(2)
> (cadddr x)
2
> (cddddr x)
()
> (cons 2 x)
(2 5 6 8 2)
> (cons x x)
((5 6 8 2) 5 6 8 2)
>
```

Question: Consider the expressions `(cons 1 2)` and `(list 1 2)`. Do they have the same value?

2. **Selector `list-ref`:** Selects the nth element of a list. Its type is `[List*Number -> T]` or `[List(T)*Number -> T]`.

   ```
   > (define list-ref
       (lambda (items n)
          (if (= n 0)
              (car items)
              (list-ref (cdr items) (- n 1)))))

   > (define squares (list 1 4 9 16 25 36))

   > (list-ref 4 squares)
   25
   ```

3. **Operator `length`:**
   Reductional (recursive, inductive) definition:

   – The length of a list is the length of its tail (`cdr`) + 1.
   – The length of the empty list is 0.

   ```
   > (define length
       (lambda (items)
         (if (null? items)
             0
             (+ 1 (length (cdr items))))))
   WARNING: redefining built-in length

   > (length squares)
   6
   ```

4. **Operator `append`:** Computes list concatenation.

   ```
   Type: [List * List -> List]
   > (define append
       (lambda (list1 list2)
         (if (null? list1)
             list2
             (cons (car list1)
                   (append (cdr list1) list2)))))
   ```

```
WARNING: redefining built-in append

> (append squares (list squares squares))
(1 4 9 16 25 36 (1 4 9 16 25 36) (1 4 9 16 25 36))
> (append (list squares squares) squares)
((1 4 9 16 25 36) (1 4 9 16 25 36) 1 4 9 16 25 36)
```

5. **Constructor make-list:** Computes a list of a given length with a given value:

```
Type: [Number * T -> List(T)]
> (make-list 7 'foo)
'(foo foo foo foo foo foo foo)
> (make-list 5 1)
'(1 1 1 1 1)
```

Example 1.4 (Newton's squarer root with stored approximations).

Consider Example 1.1 for implementing Newton's method for computing square roots. The implementation in that example computes a sequence of approximations, which is lost, once a result is returned. Assume that we are interested in obtaining also the sequence of approximations, for example, in order to observe it, measure its length, etc. The implementation below returns the approximation sequence as a list.

```
Type: [Number  -> Number]
(define sqrt
   (lambda (x) (car (storage-sqrt-iter (list 1) x))))

Type: [List(Number) * Number -> List(Number)]
(define storage-sqrt-iter
  (lambda (guess-lst x)
    (if (good-enough? (car guess-lst) x)
        guess-lst
        (storage-sqrt-iter (cons (improve (car guess-lst) x) guess-lst)
                   x))))

Type: [Number  -> Number]
(define length-sqrt
  (lambda (x)
    (length (storage-sqrt-iter (list 1) x))))
```

1.3.3.2 Using heterogeneous lists for representing hierarchical structures (SICP 2.2.2)

Scheme does not enable user defined types, and does not offer built-in types for hierarchical data like trees. Therefore, lists are used for representing hierarchical data. This is possible because Scheme supports heterogeneous lists.

Unlabeled trees, i.e., trees with unlabeled internal nodes and leaves labeled with atomic values, can be represented by lists of lists. A nested list represents a branch, and a nested atom represents a leaf. For example, the unlabeled tree in Figure 1.3 can be represented by the list `(1 (2 3))`. This representation has the drawback that a Tree is represented either by a List value or by a value of the labels type:

1. The empty tree is represented by the empty list `( )`.
2. A leaf tree is represented by some leaf value.
3. A non-empty branching tree is represented by a non-empty list.

We'll experience the drawbacks of this representation when client procedures of trees will have many end cases.

A better representation of unlabeled trees uses lists for representing all trees: A leaf tree `l` is represented by the list `(l)`, and a branching tree as in Figure 1.3 is represented by `((1) ((2) (3)))`

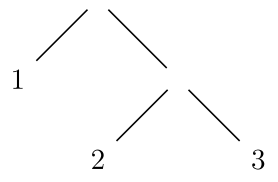


Figure 1.3: An unlabeled tree

In order to represent a labeled tree, the first element in every nesting level can represent the root of the sub-tree. A leaf tree `l` is represented by the singleton list `(l)`. A non-leaf tree, as for example, the sorted number labeled tree in Figure 1.4 is represented by the list `(1 (0) (3 (2) (4)))`.

Example 1.5. *An unlabeled tree operation – using the first representation, where a leaf tree is not a list:*

```
Signature: count-leaves(x)
Purpose: Count the number of leaves in an unlabeled tree (a selector):
   ** The count-leaves of an empty tree is 0.
```

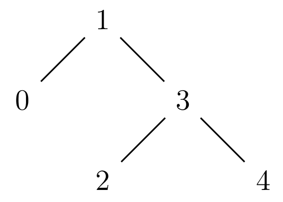


Figure 1.4: A labeled tree

```
   ** The count-leaves of a leaf is 1.
      A leaf is not represented by a list.
   ** The count-leaves of a non-empty and not leaf tree T is the count-leaves of
      the "first" branch (car) + the count-leaves of all other branches (cdr).
Type:  [List union Number union Symbol union Boolean -> Number]
>(define (count-leaves x)
    (cond ((null? x) 0)
          ((not (list? x)) 1)
          (else (+ (count-leaves (car x))
                   (count-leaves (cdr x))))))

 > (define x (cons (list 1 2) (list 3 4)))
 > x
 ((1 2) 3 4)
 > (length x)
 3
 > (count-leaves x)
 4
 > (list x x)
 (((1 2) 3 4) ((1 2) 3 4))
 > (length (list x x))
 2
 > (count-leaves (list x x))
 8
```

1.4 Procedures and the Processes they Generate (SICP 1.2)

Iteration in computing refers to a process of repetitive computations, following a single pattern. In ***imperative programming languages*** (e.g., Java, C++, C) iteration is specified by ***loop*** constructs like `while, for, begin-until`. Iterative computations (loops) are managed by ***loop variables*** whose changing values determine loop exit. Loop constructs provide abstraction of the looping computation pattern. Iteration is a central computing feature.

Functional languages like the Scheme part introduced in this chapter do not posses looping constructs like `while`. The only provision for computation repetition is repeated function application. The question asked in this section is whether iteration by function call obtains the advantages of iteration using loop constructs, as in other languages. We show that recursive function call mechanism can simulate iteration. Moreover, the conditions under which function call simulates iteration can be syntactically identified: A computing agent (interpreter, compiler) can determine, based on syntax analysis of a procedure body, whether its application can simulate iteration.

For that purpose, we discuss the ***computational processes*** generated by procedures. We distinguish between ***procedure expression*** – a syntactical notion, to ***process*** – a semantical notion. Recursive procedure expressions can create iterative processes. Such procedures are called ***tail recursive***.

1.4.1 Linear Recursion and Iteration (SICP 1.2.1)

Consider the computation of the `factorial` function. In an imperative language, it is natural to use a looping construct like `while`, that increments a factorial computation until the requested number is reached. In Scheme, factorial can be computed by the following two procedure definitions:

Recursive factorial:

```
Signature: factorial(n)
Purpose: to compute the factorial of a number 'n'.
    This procedure follows the rule: 1! = 1, n! = n * (n-1)!
Type: [Number -> Number]
Pre-conditions: n > 0, an integer
Post-condition: result = n!
Example: (factorial 4) should produce 24
Tests:  (factorial 1) ==> 1
        (factorial 4) ==> 24

(define factorial
   (lambda (n)
      (if (= n 1)
          1
          (* n (factorial (- n 1))))
    ))
```

Alternative: Iterative factorial

```
(define factorial
  (lambda (n)
    (fact-iter 1 1 n) ))

fact-iter:
Signature: fact-iter(product,counter,max-count)
Purpose: to compute the factorial of a number 'max-count'.
 This procedure follows the rule:
       counter = 1;  product = 1;
       repeat the simultaneous transformations:
       product <-- counter * product,    counter <-- counter + 1.
       stop when counter > n.
Type: [Number*Number*Number -> Number]
Pre-conditions:
        product, counter, max-count > 0
        product * counter * (counter + 1) * ... * max-count = max-count!
Post-conditions: result = max-count!
Example: (fact-iter 2 3 4) should produce 24
Tests:  (fact-iter 1 1 1)  ==> 1
        (fact-iter 1 1 4) ==> 24

(define fact-iter
   (lambda (product counter max-count)
      (if (> counter max-count)
          product
          (fact-iter (* counter product)
                     (+ counter 1)
                     max-count))))
```

Recursion vs. iteration:
Recursive factorial: The evaluation of the form `(factorial 6)` yields the following sequence of evaluations:

```
(factorial 6)
(* 6 (factorial 5))
...
(* 6 (* 5 (...(* 2 factorial 1 )...)
(* 6 (* 5 (...(* 2 1)...)
...
(* 6 120)
720
```

We can see it in the trace information provided when running the procedure:

```
> (require (lib "trace.ss"))
> (trace factorial)
> (trace *)
> (factorial 5)
"CALLED" factorial 5
"CALLED" factorial 4
  "CALLED" factorial 3
   "CALLED" factorial 2
    "CALLED" factorial 1
    "RETURNED" factorial 1
    "CALLED" * 2 1
    "RETURNED" * 2
   "RETURNED" factorial 2
   "CALLED" * 3 2
   "RETURNED" * 6
  "RETURNED" factorial 6
  "CALLED" * 4 6
  "RETURNED" * 24
 "RETURNED" factorial 24
 "CALLED" * 5 24
 "RETURNED" * 120
"RETURNED" factorial 120
120
 >
```

Every recursive call has its own information to keep and manage – input and procedure-code evaluation, but also a return information to the calling procedure, so that the calling procedure can continue its own computation. The space needed for a procedure call evaluation is called ***frame***. Therefore, the implementation of such a sequence of recursive calls requires keeping the frames for all calling procedure applications, which **depends on the value of the input**. The computation of (factorial 6) requires keeping 6 frames simultaneously open, since every calling frame is waiting for its called frame to finish its computation and provide its result.

Iterative factorial: The procedure admits the following pseudo-code:

```
define fact-iter
   function (product,counter,max-count)
      {while (counter <= max-count)
            { product := counter * product;
            counter := counter + 1;}
        return product;}
```

That is, the iterative factorial computes its result using a ***looping construct***: Repetition of a fixed process, where repetitions (called ***iterations***) vary by **changing** the values of variables. Usually there is also a variable that functions as the **loop variable**. In contrast to the evaluation process of the recursive factorial, the evaluation of a loop iteration does not depend on its next loop iteration: Every loop iteration hands-in the new variable values to the loop manager (the `while` construct), and the last loop iteration provides the returned result. Therefore, **all loop iterations can be computed using a fixed space, needed for a single iteration**. That is, the procedure can be computed using a fixed space, which **does not depend on the value of the input**. This is a great advantage of looping constructs. Their great disadvantage, though, is the reliance on variable value change, i.e., assignment.

In functional languages there are no looping constructs, since variable values cannot be changed – **No assignment in functional languages**. Process repetition is obtained by procedure (function) calls. In order to achieve the great space advantage of iterative looping constructs, procedure calls are postponed to be **the last evaluation action**, which means that once a procedure-call frame calls for a new frame, the calling frame is done, no further actions are needed, and it can be abandoned. Therefore, as in the looping construct case, every frame hands-in the new variable values to the next opened frame, and the last frame provides the returned result. Therefore, **all frames can be computed using a fixed space, needed for a single frame**.

A procedure whose body code includes a procedure call only as a last evaluation step[1], is called ***iterative***. If the evaluation application is "smart enough" to notice that a procedure is iterative, it can use a fixed space for procedure-call evaluations, and enjoy the advantages of iterative loop structures, without using variable assignment. Such evaluators are called ***tail recursive***.

Indeed, there is a single procedure call in the body of the iterative factorial, and it occurs last in the evaluation actions, implying that it is an iterative procedures. Since Scheme applications are all tail recursive, the evaluation of `(factorial 6)` using the iterative version, yields the following evaluation sequence:

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
...
(fact-iter 720 7 6)
720
```

[1]There can be several embedded procedure calls, each occurs last on a different branching computation path.

The trace information, after tracing all procedures::

```
 > (factorial 3)
"CALLED" factorial 3
 "CALLED" fact-iter 1 1 3
  "CALLED" * 1 1
  "RETURNED" * 1
  "CALLED" fact-iter 1 2 3
   "CALLED" * 2 1
   "RETURNED" * 2
   "CALLED" fact-iter 2 3 3
    "CALLED" * 3 2
    "RETURNED" * 6
    "CALLED" fact-iter 6 4 3
    "RETURNED" fact-iter 6
   "RETURNED" fact-iter 6
  "RETURNED" fact-iter 6
 "RETURNED" fact-iter 6
"RETURNED" factorial 6
6
```

In the first case – the number of deferred computations grows linearly with n. In the second case – there are no deferred computations. A computation process of the first kind is called ***linear recursive***. A computation process of the second kind is called ***iterative***. In a ***linear recursive process***, the ***time and space*** needed to perform the process, are proportional to the input size. In an ***iterative process***, the ***space is constant*** – it is the space needed for performing a single iteration round. These considerations refer to the space needed for procedure-call frames (the space needed for possibly unbounded data structures is not considered here). In an ***iterative process***, the status of the evaluation process is completely determined by the variables of the procedure (parameters and local variables). In a ***linear recursive process***, procedure call frames need to store the status of the deferred computations.

Note the distinction between the three notions:

– ***recursive procedure*** - a syntactic notion;

– ***linear recursive process, iterative process*** - semantic notions.

`fact-iter` is a recursive procedure that generates an iterative process.

Recursive processes are, usually, clearer to understand, while iterative ones can save space. The method of ***tail-recursion***, used by compilers and interpreters, executes iterative processes in constant space, even if they are described by recursive procedures. A recursive

procedure whose application does not create deferred computations can be performed as an iterative process.
Typical form of iterative processes: Additional parameters for a ***counter*** and an ***accumulator***, where the partial result is ***stored***. When the counter reaches some ***bound***, the accumulator gives the result.

Example 1.6. *(no contract):*

```
> (define count1
      (lambda (x)
          (cond ((= x 0) (display x))
                (else (display x)
                      (count1 (- x 1))))))
> (define count2
      (lambda (x)
         (cond ((= x 0) (display x))
               (else (count2 (- x 1))
                      (display x)))))
> (trace count1)
> (trace count2)
> (count1 4)
|(count1 4)
4|(count1 3)
3|(count1 2)
2|(count1 1)
1|(count1 0)
0|#<void>

> (count2 4)
|(count2 4)
| (count2 3)
| |(count2 2)
| | (count2 1)
| | |(count2 0)
0| | |#<void>
1| | #<void>
2| |#<void>
3| #<void>
4|#<void>
```

`count1` generates an ***iterative process***; `count2` generates a ***linear-recursive process***.

1.4.2 Tree Recursion (SICP 1.2.2)

Consider the following procedure definition for computing the n-th element in the sequence of Fibonacci numbers:

Recursive FIB

```
Signature: (fib n)
Purpose: to compute the nth Fibonacci number.
 This procedure follows the rule:
    fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2).
Type: [Number -> Number]
Example: (fib 5) should produce 5
Pre-conditions: n >= 0
Post-conditions: result = nth Fibonacci number.
Tests: (fib 3) ==> 2
       (fib 1) ==> 1

  (define fib
    (lambda (n)
      (cond ((= n 0) 0)
            ((= n 1) 1)
            (else (+ (fib (- n 1))
                     (fib (- n 2)))))
     ))
```

The evaluation process generated by this procedure has a ***tree structure***, where nested forms lie on the same branch:

```
                 +-----------------(fib 5)-----------------+
                 |                                         |
         +-----(fib 4)---------+                 +-----(fib 3)---------+
         |                     |                 |                     |
     +--(fib 3)--+       +--(fib 2)-+       +-(fib 2)-+            (fib 1)
     |           |       |          |       |         |                |
  +-(fib 2)-+   (fib 1) (fib 1)   (fib 0)  (fib 1)   (fib 0)           1
  |         |    |       |          |       |         |
 (fib 1) (fib 0) 1       1          0       1         0
  1         0
```

The ***time*** required is proportional to the ***size*** of the tree, since the evaluation of `(fib 5)` requires the evaluation of all `fib` forms. Hence, the time required is ***exponential*** in the input of `fib`. The ***space*** required is proportional to the ***depth*** of the tree, i.e., ***linear*** in

the input.
Note: The exponential growth order applies to balanced (or almost balanced) trees. Highly pruned computation trees can yield a smaller growth order.

Iterative FIB

```
(define fib
   (lambda (n) (fib-iter 0 1 n)))

fib-iter:
Signature: fib-iter(current,next,count)
Purpose: to compute the nth Fibonacci number.
 We start with current = 0, next = 1, and count as the Fibonacci goal,
 and repeat the simultaneous transformation 'count' times:
 next <-- next + current,  current <-- next,
 in order  to compute  fib(count).
Type: [Number*Number*Number -> Number]
Example: (fib-iter 0 1 5) should produce 5
Pre-conditions: next = (n+1)th Fibonacci number, for some n >= 0;
                    current = nth Fibonacci number;
Post-conditions: result = (n+count)th Fibonacci number.
Tests:  (fib-iter 1 2 3) ==> 5
        (fib-iter 0 1 1) ==> 1

  (define fib-iter
    (lambda (current next count)
       (if (= count 0)
           current
           (fib-iter  next (+ current next) (- count 1)))
     ))
```

Example 1.7. – *Counting Change (without contract)*

Given an amount `A` of money, and types of coins (5 agorot, 10 agorot, etc), ordered in some fixed way. Compute the number of ways to change the amount `A`. Here is a rule:

> The number of ways to change `A` using `n` kinds of coins (ordered) =
> number of ways to change `A` using the last `n-1` coin kinds +
> number of ways to change `A - D` using all `n` coin kinds, where `D` is the denomination of the first kind.

Try it!

```
(define count-change
  (lambda (amount)
     (cc amount 5)))

(define cc
  (lambda (amount kinds-of-coins)
     (cond ((= amount 0) 1)
           ((or (< amount 0) (= kinds-of-coins 0)) 0)
           (else (+ (cc (- amount
                           (first-denomination kinds-of-coins))
                        kinds-of-coins)
                    (cc amount
                        (- kinds-of-coins 1)))))))

(define first-denomination
  (lambda (kinds-of-coins)
     (cond ((= kinds-of-coins 1) 1)
           ((= kinds-of-coins 2) 5)
           ((= kinds-of-coins 3) 10)
           ((= kinds-of-coins 4) 25)
           ((= kinds-of-coins 5) 50))))
```

What kind of process is generated by count-change? Try to design a procedure that generates an iterative process for the task of counting change. What are the difficulties?

1.4.3 Orders of Growth (SICP 1.2.3)

Processes are evaluated by the amount of ***resources of time and space*** they require. Usually, the amount of resources that a process requires is measured by some agreed ***unit***. For ***time***, this might be number of machine operations, or number of rounds within some loop. For ***space***, it might be number of registers, or number of cells in a Turing machine performing the process.

The ***resources*** are measured in terms of the ***problem size***, which is some attribute of the input that we agree to take as most characteristic. The resources are represented as functions $Time(n)$ and $Space(n)$, where n is the problem size.

$Time(n)$ and $Space(n)$ have ***order of growth*** of $O(f(n))$ if for some constant C: $Time(n) <= C * f(n), Space(n) <= C * f(n)$, for any sufficiently large n.

– For the linear recursive factorial process: $Time(n) = Space(n) = O(n)$.

– For the iterative factorial and Fibonacci processes: $Time(n) = O(n)$, but $Space(n) = O(1)$.

- For the tree recursive Fibonacci process: $Time(n) = O(C^n)$, and $Space(n) = O(n)$.

Order of growth is an indication of the ***change*** in resources implied by changes in the ***problem size***.

- $O(1)$ – ***Constant growth***: Resource requirements do not change with the size of the problem. For all iterative processes, the space required is constant, i.e., $Space(n) = O(1)$.
- $O(n)$ – ***Linear growth***: Multiplying the problem size ***multiplies*** the resources by the same factor.
  For example: if $Time(n) = Cn$ then
  $Time(2n) = 2Cn = 2Time(n)$, and
  $Time(4n) = 4Cn = 2Time(2n)$, etc.
  Hence, the resource requirements grow ***linearly*** with the problem size.
  A ***linear iterative*** process is an iterative process that uses linear time ($Time(n) = O(n)$), like the iterative versions of `factorial` and of `fib`.
  A ***linear recursive*** process is a recursive process that uses linear time and space ($Time(n) = Space(n) = O(n)$), like the recursive version of `factorial`.
- $O(C^n)$ – ***Exponential growth***: Any increment in the problem size, ***multiplies*** the resources by a constant number.
  For example: if $Time(n) = C^n$, then
  $Time(n+1) = C^{n+1} = Time(n) * C$, and
  $Time(n+2) = C^{n+2} = Time(n+1) * C$, etc.
  $Time(2n) = C^{2n} = (Time(n))^2$.
  Hence, the resource requirements grow ***exponentially*** with the problem size. The tree-recursive Fibonacci process uses exponential time.
- $O(\log n)$ – ***Logarithmic growth***: ***Multiplying*** the problem size implies a ***constant increase*** in the resources.
  For example: if $Time(n) = \log(n)$, then
  $Time(2n) = \log(2n) = Time(n) + \log(2)$, and
  $Time(6n) = \log(6n) = Time(2n) + \log(3)$, etc.
  We say that the resource requirements grow ***logarithmically*** with the problem size.
- $O(n^a)$ – ***Power growth***: Multiplying the problem size ***multiplies*** the resources by a power of that factor.
  For example: if $Time(n) = n^a$, then
  $Time(2n) = (2n)^a = Time(n) * (2^a)$, and
  $Time(4n) = (4n)^a = Time(2n) * (2^a)$, etc.
  Hence, the resource requirements grow as a ***power*** of the problem size. Linear grows is a special case of power grows ($a = 1$). Quadratic grows is another special common case ($a = 2$, i.e., $O(n^2)$).

Example 1.8. – *Exponentiation (SICP 1.2.4)*

This example presents procedures that generate several processes for computing exponentiation, that require different resources, and have different orders of growth in time and space.
Linear recursive version (no contracts): Based on the recursive definition: $b^0 = 1, b^n = b * b^{n-1}$,

```
(define expt
  (lambda (b n)
     (if (= n 0)
         1
         (* b (expt b (- n 1))))))
```

$Time(n) = Space(n) = O(n)$.
Linear iterative version: Based on using `product` and `counter`, with initialization: `counter = n, product = 1`, and repeating the simultaneous transformations: `counter <- counter - 1, product <- product * b` until `counter` becomes zero.

```
(define expt
  (lambda (b n)
    (exp-iter b n 1)))

(define exp-iter
  (lambda (b counter product)
    (if (= counter 0)
        product
        (exp-iter b
                  (- counter 1)
                  (* b product)))))
```

$Time(n) = O(n), \quad Space(n) = O(1)$.
Logarithmic recursive version: Based on the idea of successive squaring, instead of successive multiplications:
For even n: $a^n = (a^{n/2})^2$.
For odd n: $a^n = a * (a^{n-1})$.

```
(define fast-exp
  (lambda (b n)
     (cond ((= n 0) 1)
           ((even? n) (square (fast-exp b (/ n 2))))
           (else (* b (fast-exp b (- n 1)))))))
```

Note: `even?` and `odd?` are primitive procedures in Scheme:

```
> even?
#<primitive:even?>
```

They can be defined via another primitive procedure, `remainder` (or `modulo`), as follows:

```
(define even?
  (lambda (n)
     (= (remainder n 2) 0)))
```

The complementary procedure to `remainder` is `quotient`, which returns the integer value of the division: `(quotient n1 n2) ==> n1/n2`.
$Time(n) = Space(n) = O(\log n)$, since `fast-exp(b, 2n)` adds a single additional multiplication to `fast-expr(b, n)` (in the even case). In this approximate complexity analysis, the application of primitive procedures is assumed to take constant time.

Example 1.9. – *Greatest Common Divisors (GCD) (no contracts) (SICP 1.2.5)*

The GCD of 2 integers is the greatest integer that divides both. The ***Euclid's algorithm*** is based on the observation:

Lemma 1.4.1. If r is the remainder of a/b, then: $GCD(a,b) = GCD(b,r)$. Successive applications of this observation yield a pair with 0 as the second number. Then, the first number is the answer.

Proof. Assume $a > b$. Then $a = qb + r$ where q is the quotient. Then $r = a - qb$. Any common divisor of a and b is also a divisor of r, because if d is a common divisor of a and b, then $a = sd$ and $b = td$, implying $r = (s - qt)d$. Since all numbers are integers, r is divisible by d. Therefore, d is also a common divisor of b and r. Since d is an arbitrary common divisor of a and b, this conclusion holds for the greatest common divisor of a and b. □

```
(define gcd
  (lambda (a b)
     (if (= b 0)
         a
         (gcd b (remainder a b)))))
```

Iterative process:
$Space(a,b) = O(1)$.
$Time(a,b) = O(\log(n))$, where $n = \min(a,b)$.
The Time order of growth results from the theorem: $n \geq Fib(Time(a,b)) = (\Theta(C^{Time(a,b)})/\sqrt{5})$ which implies: $Time(a,b) \leq \log(n * \sqrt{5}) = \log(n) + \log(\sqrt{5})$ Hence: $Time(a,b) = O(\log(n))$, where $n = \min(a,b)$.

Example 1.10. – *Primality Test (no contracts) (SICP 1.2.6)*

Straightforward search:

```
(define smallest-divisor
  (lambda (n)
    (find-divisor n 2)))

(define find-divisor
  (lambda (n test-divisor)
    (cond ((> (square test-divisor) n) n)
          ((divides? test-divisor n) test-divisor)
          (else (find-divisor n (+ test-divisor 1))))))

(define divides?
  (lambda (a b)
    (= (remainder b a) 0)))

(define prime?
  (lambda (n)
    (= n (smallest-divisor n))))
```

Based on the observation that if n is not a prime, then it must have a divisor less than or equal than its square root.
Iterative process. $Time(n) = O(\sqrt{n}), Space(n) = O(1)$.
Another algorithm, based on Fermat's Little Theorem:

Theorem 1.4.2. If n is a prime, then for every positive integer a, $(a^n)\ mod\ n = a\ mod\ n$.

The following algorithm picks randomly positive integers, less than n, and applies Fermat's test for a given number of times: `expmod` computes $b^e\ mod\ n$. It is based on the observation:
$(x * y)\ mod\ n = ((x\ mod\ n) * (y\ mod\ n))\ mod\ n$, a useful technique, as the numbers involved stay small.

```
(define expmod
  (lambda (b e n)
     (cond ((= e 0) 1)
           ((even? e)
            (remainder (square (expmod b (/ e 2) n))
                       n))
           (else
            (remainder (* b (expmod b (- e 1) n))
                       n)))))
```

The created process is recursive. The rate of time grows for expmod is $Time(e) = O(\log(e))$, i.e., logarithmic grow in the size of the exponent, since: $Time(2 * e) = Time(e) + 2$.

```
(define fermat-test
  (lambda (n a)
      (= (expmod a n n) a))))

(define fast-prime?
  (lambda (n times)
     (cond ((= times 0) t)
           ((fermat-test n (+ 2 (random (- n 2))))
            (fast-prime? n (- times 1)))
           (else #f))))
```

`random` is a Scheme primitive procedure. `(random n)` returns an integer between 0 to `n-1`.

1.5 Higher-Order Procedures

Partial source: (SICP 1.3.1, 1.3.2, 1.3.3, 1.3.4)

Variables provide abstraction of values. ***Procedures*** provide abstraction of compound operations on values. In this section we introduce:

Higher-order procedures: Procedures that manipulate procedures.

This is a common notion in mathematics, where we discuss notions like $\Sigma f(x)$, without specifying the exact function f. Σ is an example of a higher order procedure. It introduces the concept of ***summation***, independently of the particular function whose values are being summed. It allows for discussion of general properties of sums.

In ***functional programming*** (hence, in Scheme) procedures have a ***first class*** status:

1. Can be named by variables.
2. Can be passed as arguments to procedures.
3. Can be returned as procedure values.
4. Can be included in data structures.

In this section, we introduce higher-order procedures that have the first three features listed above. In the next section (Section 1.3), we also introduce data structures that can include procedures as their components.

1.5.1 Procedures as Parameters (SICP 1.3.1)

Summation Consider the following 3 procedures:

1. `sum-integers`:

```
Signature: sum-integers(a,b)
Purpose: to compute the sum of integers in the interval [a,b].
Type: [Number*Number ->  Number]
Post-conditions: result = a + (a+1) + ... + b.
Example: (sum-integers 1 5) should produce 15
Tests:  (sum-integers 2 2) ==> 2
        (sum-integers 3 1) ==> 0

(define sum-integers
  (lambda (a b)
     (if (> a b)
       0
       (+ a (sum-integers (+ a 1) b)))))
```

2. `sum-cubes`:

```
Signature: sum-cubes(a,b)
Purpose: to compute the sum of cubic powers of
             integers in the interval [a,b].
Type: [Number*Number -> Number]
Post-conditions: result = a^3 + (a+1)^3 + ... + b^3.
Example: (sum-cubes 1 3) should produce 36
Tests: (sum-cubes 2 2) ==> 8
       (sum-cubes 3 1) ==> 0

(define sum-cubes
  (lambda (a b)
     (if (> a b)
       0
       (+ (cube a) (sum-cubes (+ a 1) b)))))
```

where `cube` is defined by: `(define cube (lambda (x) (* x x x)))`.

3. `pi-sum`:

```
Signature: pi-sum(a,b)
Purpose: to compute the sum
       1/(a*(a+2)) + 1/((a+4)*(a+6)) + 1/((a+8)*(a+10)) + ...
       (which converges to PI/8, when started from a=1).
Type: [Number*Number -> Number]
Pre-conditions: if a < b, a != 0.
Post-conditions:
     result = 1/a*(a+2) + 1/(a+4)*(a+6) + ... + 1/(a+4n)*(a+4n+2),
              a+4n =< b, a+4(n+1) > b
Example: (pi-sum 1 3) should produce 1/3.
Tests: (pi-sum 2 2) ==> 1/8
       (pi-sum 3 1) ==> 0

(define pi-sum
  (lambda (a b)
     (if (> a b)
       0
       (+ (/ 1 (* a (+ a 2))) (pi-sum (+ a 4) b)))))
```

The procedures have the same pattern:

```
(define <name>
  (lambda (a b)
     (if (> a b)
       0
       (+ (<term> a)
          (<name> (<next> a) b)))))
```

The 3 procedures can be abstracted by a single procedure, where the empty slots `<term>` and `<next>` are captured by formal parameters that specify the `<term>` and the `<next>` functions, and <name> is taken as the defined function `sum`:
`sum`:

```
Signature: sum(term,a,next,b)
Purpose: to compute the sum of terms, defined by <term>
    in predefined gaps, defined by <next>, in the interval [a,b].
Type: [[Number -> Number]*Number*[Number -> Number]*Number -> Number]
Post-conditions: result = (term a) + (term (next a)) + ... (term n),
                     where n = (next (next ...(next a))) =< b,
                                                    (next n) > b.
Example: (sum identity 1 add1 3) should produce 6,
             where 'identity' is (lambda (x) x)
Tests: (sum square 2 add1 2) ==> 4
       (sum square 3 add1 1) ==> 0
```

```
(define sum
   (lambda (term a next b)
     (if (> a b)
       0
       (+ (term a)
          (sum term (next a) next b)))))
```

Using the `sum` procedure, the 3 procedures above have different implementations (same contracts):

```
 (define sum-integers
  (lambda (a b)
     (sum identity a add1 b)))

(define sum-cubes
  (lambda (a b)
     (sum cube a add1 b)))

(define pi-sum
(lambda (a b)
     (sum pi-term a pi-next b)))

(define pi-term
  (lambda (x)
       (/ 1 (* x (+ x 2)))))

(define pi-next
  (lambda (x)
       (+ x 4)))
```

Discussion: What is the advantage of defining the `sum` procedure, and defining the three procedures as concrete applications of `sum`?

1. First, the `sum` procedure prevents ***duplications*** of the computation pattern of summing a sequence elements between given boundaries. Duplication in software is bad for many reasons, that can be summarized by management difficulties, and lack of abstraction – which leads to the second point.

2. Second, and more important, the `sum` procedure expresses the mathematical notion of sequence summation. Having this notion, further abstractions can be formulated, on top of it. This is similar to the role of ***interface*** in object-oriented languages.

Definite integral – Definition based on `sum`: Integral of f from a to b is approximated by: $[f(a+dx/2)+f(a+dx+dx/2)+f(a+2dx+dx/2)+...]\times dx$ for small values of dx. The definite integral can be computed (approximated) by the procedure:

```
(define dx 0.005)
(define integral
  (lambda (f a b)
     (* (sum f (+ a (/ dx 2)) add-dx b)
        dx)))

(define add-dx
  (lambda (x) (+ x dx)))
```

For example:

```
> (integral cube 0 1 0.01)
 0.2499875
> (integral cube 0 1 0.001)
 0.249999875
```

True value: 1/4.

Sequence-operation – Definition based on `sum`:

```
Signature: sequence-operation(operation,start,a,b)
Purpose: to compute the repeated application of an operation on
             all integers in the interval [a,b], where <start> is
             the neutral element of the operation.
Type: [[Number*Number -> Number]*Number*Number*Number -> Number]
Pre-conditions: start is a neutral element of operation:
                    (operation x start) = x
Post-conditions:
     result = if a =< b: a operation (a+1) operation ... b.
              if a > b: start
Example: (sequence-operation * 1 3 5) is 60
Tests: (sequence-operation + 0 2 2) ==> 2
       (sequence-operation * 1 3 1) ==> 1

(define sequence-operation
  (lambda (operation start a b)
     (if (> a b)
       start
       (operation a (sequence-operation operation start (+ a 1) b)))))
```

where `operation` stands for any binary procedure, such as +, *, -, and `start` stands for the neutral (unit) element of `operation`, i.e., 0 for +, and 1 for *. For example:

```
> (sequence-operation * 1 3 5)
60
> (sequence-operation + 0 2 7)
27
> (sequence-operation - 0 3 5)
4
> (sequence-operation expt 1 2 4)
2417851639229258349412352
> (expt 2 (expt 3 4))
2417851639229258349412352
```

1.5.2 Constructing procedure arguments at run-time

Procedures are constructed by the value constructor of the Procedure type: `lambda`. The evaluation of a `lambda` form creates a `closure`. For example, the evaluation of

```
(lambda (x) (+ x 4))
```

creates the closure:

```
<Closure (x)(+ x 4)>
```

`lambda` forms can be evaluated during computation. Such closures are termed ***anonymous procedures***, since they are not named. Anonymous procedures are useful whenever a procedural abstraction does not justify being named and added to the global environment mapping.

For example, in defining the `pi-sum` procedure, the procedure `(lambda (x) (/ 1 (* x (+ x 2))))` that defines the general form of a term in the sequence is useful only for this computation. It can be passed directly to the `pi-sum` procedure:

```
(define pi-sum
   (lambda (a b)
     (sum (lambda (x) (/ 1 (* x (+ x 2))))
          a
          (lambda (x) (+ x 4))
          b)))
```

The body of the `pi-sum` procedure includes two anonymous procedures that are created at runtime, and passed as arguments to the `sum` procedure. The ***price*** of this elegance is that the anonymous procedures are redefined in every application of `pi-sum`.

The `integral` procedure using an anonymous procedure:

```
(define integral
   (lambda (f a b dx)
     (* (sum f
             (+ a (/ dx 2.0))
             (lambda (x) (+ x dx))
             b)
        dx)))
```

Note that once the `next` procedure is created anonymously within the integral procedure, the `dx` variable can become a parameter rather than a globally defined variable.

1.5.3 Defining Local Variables – Using the `let` Abbreviation

Local variables are an essential programming technique, that enables the declaration of variables with a restricted scope. Such variables are used for saving repeated computations. In a restricted scope, a local variable can be initialized with the value of some computation, and then substituted where ever this computation is needed. In imperative programming local variables are also used for storing changing values, like values needed for loop management. Local variables are characterized by:

1. ***Declaration*** combined with a ***restricted scope***, where the variable is recognized – a restricted program region, where occurrences of the variable are ***bound*** to the variable declaration.
2. A one-time initialization: A local variable is initialized by a value that is computed only once.
3. Substitution of the initialization value for variable occurrences in the scope.

In Scheme, variables are declared only in `lambda` forms and in `define` forms. Therefore, the core language presented so far has no provision for local variable. Below, we show how local variable behavior can be obtained by plain generation and immediate application of a run time generated closure to initialization values. We then present the Scheme syntactic sugar for local values – the `let` special form. This special operator does not need a special evaluation rule (unlike `define, lambda, if, cond, quote`) since it is just a syntactic sugar for a plain Scheme form.

Parameters, scope, bound and free variable occurrences: A `lambda` form includes ***parameters*** and ***body***. The parameters act as ***variable declarations*** in most programming languages. They ***bind*** their ***occurrences*** in the body of the form, unless, there is a nested `lambda` form with the same parameters. The body of the `lambda` form is the ***scope*** of the parameters of the `lambda` form. The occurrences that are bound by the parameters are ***bound occurrences***, and the ones that are not bound are ***free***. For example, in the `integral` definition above, in the `lambda form`:

```
(lambda (f a b dx)
     (* (sum f
             (+ a (/ dx 2.0))
             (lambda (x) (+ x dx))
             b)
        dx))
```

The ***parameters***, or declared variables, are `f, a, b, dx`. Their ***scope*** is the entire `lambda` form. Within the body of the `lambda` form, they bind all of their occurrences. But the occurrences of `+, *, sum are` ***`free`*** `.` Within the inner `lambda` form `(lambda (x) (+ x dx))`, the occurrence of `x` is bound, while the occurrence of `dx` is free.
A `define` form also acts as a variable declaration. The defined variable binds all of its free occurrences in the rest of the code. We say that a defined variable has a ***universal scope***.

Example 1.11. *Local variables in computing the value of a polynomial function:*

Consider the following function:

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

In order to compute the value of the function for given arguments, it is useful to define two local variables:

```
a = 1+xy
b = 1-y
```

then:

$$f(x,y) = xa^2 + yb + ab$$

The local variables save repeated computations of the $1+xy$ and $1-y$ expressions. The body of $f(x,y)$ can be viewed as a function in a and b (since within the scope of f, the occurrences of x and y are already bound). That is, the body of $f(x,y)$ can be viewed as the function application

$$f(x,y) = f\_helper(1+xy, 1-y)$$

where for given x, y:

$$f\_helper(a,b) = xa^2 + yb + ab$$

The Scheme implementation for the f_helper function:

```
(lambda (a b)
      (+ (* x (square a))
         (* y b)
         (* a b)))
```

$f(x, y)$ can be implemented by applying the helper function to the values of a and b, i.e., $1 + xy$ and $1 - y$:

```
(define f
   (lambda (x y)

     ((lambda (a b)
        (+ (* x (square a))
           (* y b)
           (* a b)))

      (+ 1 (* x y))
      (- 1 y))
   ))
```

The important point is that this definition of the polynomial function `f` provides the behavior of local variables: The initialization values of the parameters `a, b` are computed only once, and substituted in multiple places in the body of the `f_helper` procedure.

Note: The helper function cannot be defined in the global environment, since it has x and y as free variables, and during the evaluation process, while the occurrences of `a` and `b` are replaced by the argument values, `x` and `y` stay unbound:

```
> (define f_helper
    (lambda (a b)
        (+ (* x (square a))
           (* y b)
           (* a b))))

>  (f_helper (+ 1 (* x y)) (- 1 y))
reference to undefined identifier: x
```

The `let` abbreviation: A conventional abbreviation for this construct, which internally turns into a nested `lambda` form application, is provided by the `let` special operator. That is, a `let` form is just a ***syntactic sugar*** for application of a lambda form. Such abbreviations are termed ***derived*** expressions. The evaluation of a `let` form creates an anonymous closure and applies it.

```
(define f
   (lambda ( x y)
     (let ((a (+ 1 (* x y)))
           (b (- 1 y)))
       (+ (* x (square a))
          (* y b)
          (* a b)))))
```

The general syntax of a `let` form is:

```
(let ( (<var1> <exp1>)
          (<var2> <exp2>)
          ...
          (<varn> <expn>) )
        <body> )
```

The evaluation of a `let` form has the following steps:

1. Each `<expi>` is evaluated (simultaneous binding).
2. The values of the `<expi>`s are replaced for all free occurrences of their corresponding `<vari>`s, in the `let` body.
3. The `<body>` is evaluated.

These rules result from the internal translation to the lambda form application:

```
( (lambda ( <var1> ... <varn> )
            <body> )
   <exp1>
   ...
   <expn> )
```

Therefore, the evaluation of a `let` form does not have any special evaluation rule (unlike the evaluation of `define` and `lambda` forms, which are ***true*** special operators).
Notes about `let` evaluation:

1. `let` provides variable declaration and an embedded scope.
2. Each `<vari>` is associated (***bound***) to the ***value*** of `<expi>` (simultaneous binding). Evaluation is done only once.
3. The `<expi>`s reside in the outer scope, where the `let` resides. Therefore, variable occurrences in the `<expi>`s are ***not bound*** by the `let` variables, but by binding occurrences in the outer scope.
4. The `<body>` is the `let` scope. All variable occurrences in it are bound by the `let` variables (substituted by their values).
5. The evaluation of a `let` form consists of ***creation of an anonymous procedure***, and its immediate application to the initialization values of the local variables.

```
> (define x 5)
> (+ (let ( (x 3) )
          (+ x (* x 10)))
      x)
==>
> (+ ( (lambda (x) (+ x (* x 10)))
        3)
      x)
38
```

Question: How many times the `let` construct is computed in:

```
> (define x 5)
> (define y (+ (let ( (x 3) )
                  (+ x (* x 10)))
               x))
> y
38
> (+ y y)
76
```

In evaluating a `let` form, variables are bound simultaneously. The initial values are evaluated ***before*** all `let` local variables are substituted.

```
> (define x 5)
> (let ( (x 3) (y (+ x 2)))
        (* x y))
21
```

1.5.4 Procedures as Returned Values (SICP 1.3.4)

The ability to have procedures that create procedures provides a great expressiveness. Many applications provide skeletal procedures that employ changing functions (algorithms). For example, Excel enables ***filter*** creation with a changing filtering function. General procedure handling methods can be implemented using ***procedures that create procedures***.

> A function definition whose body evaluates to the value of a lambda form is a high order procedure that returns a procedure as its value.

First, let us see how repeated lambda abstractions create procedures (closures) that create procedures.

– A form: `(+ x y y)`, evaluates to a number.

- A `lambda` abstraction: `(lambda (x) (+ x y y))`, evaluates to a procedure definition.
- A further `lambda` abstraction of the `lambda` form: `(lambda (y) (lambda (x) (+ x y y)))`, evaluates to a procedure with formal parameter `y`, whose application (e.g., `((lambda (y) (lambda (x) (+ x y y))) 3)` evaluates to a procedure in which `y` is already substituted, e.g., `<Closure (x) (+ x 3 3)>`.

```
> (define y 0)
> (define x 3)
> (+ x y y)
3
> (lambda (x) (+ x y y))
#<procedure>
> ((lambda (x) (+ x y y)) 5)
5
> (lambda (y) (lambda (x) (+ x y y)))
#<procedure>
> ((lambda (y) (lambda (x) (+ x y y))) 2)
#<procedure>
> (((lambda (y) (lambda (x) (+ x y y))) 2) 5)
9
> (define f (lambda (y) (lambda (x) (+ x y y)) )  )
> ((f 2) 5)
9
> (define f (lambda (y) (lambda (x) (+ x y y))))
> ((f 2) 5)
9
> ((lambda (y) ((lambda (x) (+ x y y)) 5)) 2)
9
> ((lambda (x) (+ x y y)) 5)
5
>
```

The following examples demonstrate high order procedures that return procedures (closures) as their returned value.

Example 1.12. *Average damp:*

Average damping is average taken between a value `val` and the value of a given function `f` on `val`. Therefore, every function defines a different average. This function-specific average can be created by a procedure generator procedure:

```
average-damp:
Signature: average-damp(f)
Purpose: to construct a procedure that computes the average damp
    of a function average-damp(f)(x) = (f(x) + x )/ 2
Type: [[Number -> Number] -> [Number -> Number]]
Post-condition: result = closure r,
                             such that (r x) = (average (f x) x)
Tests: ((average-damp square) 10) ==> 55
       ((average-damp cube) 6) ==> 111

(define average-damp
   (lambda (f)
      (lambda (x) (average x (f x)))))
```

For example:

```
> ((average-damp (lambda (x) (* x x))) 10)
55
> (average 10 ((lambda (x) (* x x)) 10))
55

((average-damp cube) 6)
111
> (average 6 (cube 6))
111
> (define av-damped-cube (average-damp cube))
> (av-damped-cube 6)
111
```

Example 1.13. *The derivative function:*

For every number function, its derivative is also a function. The derivative of a function can be created by a procedure generating procedure:

```
derive:
Signature: derive(f dx)
Purpose: to construct a procedure that computes the derivative
    dx approximation of a function:
     derive(f dx)(x) = (f(x+dx) - f(x) )/ dx
Type: [[Number -> Number]*Number -> [Number -> Number]]
Pre-conditions: 0 < dx < 1
Post-condition: result = closure r, such that
    (r y) =   (/ (- (f (+ x dx)) (f x))
                        dx)
Example: for f(x)=x^3, the derivative is the function 3x^2,
    whose value at x=5 is 75.
Tests: ((derive cube 0.001) 5) ==> ~75

  (define derive
     (lambda (f dx)
       (lambda (x)
         (/ (- (f (+ x dx)) (f x))
            dx))))
```

The value of `(derive f dx)` is a procedure!

```
> (define cube (lambda (x) (* x x x)))

> ((derive cube .001) 5)
75.0150010000254
> ((derive cube  .0001) 5)
75.0015000099324
>
```

Closure creation time – Compile time vs runtime Creation of a closure takes time and space. In general, it is preferred that expensive operations are done before a program is run, i.e., in ***static compile time***. When a closure (i.e., a procedure created at runetime) depends on other closures, it is preferred that these auxiliary closures are created at compile time. The following example, presents four procedures that differ in the timing of the creation of the auxiliary procedures.

Example 1.14 (Compile time vs runtime creation of auxiliary closures)**.**

Given a definition of a procedure that approximates the derivative of a function of one argument, `f`: `x`:

```
(define dx 0.00001)

Signature: derive(f)
Type: [[Number -> Number] -> [Number -> Number]]
(define derive
  (lambda (f)
    (lambda(x)
      (/ (- (f (+ x dx))
            (f x))
         dx))))
```

The following four procedures approximate the nth derivative of a given function of one argument, `f`:

```
Signature: nth-deriv(f,n)
Type: [[Number -> Number]*Number -> [Number -> Number]]
(define nth-deriv
  (lambda (f n)
          (lambda (x)
              (if (= n 0)
                  (f x)
                  ( (nth-deriv (derive f) (- n 1)) x)))
  ))

(define nth-deriv
  (lambda (f n)
          (if (= n 0)
              f
              (lambda (x)
                  ( (nth-deriv (derive f) (- n 1)) x)))
  ))

(define nth-deriv
  (lambda (f n)
          (if (= n 0)
              f
              (nth-deriv (derive f) (- n 1)) )
  ))

(define nth-deriv
  (lambda (f n)
          (if (= n 0)
              f
              (derive  (nth-deriv f (- n 1)) ))
  ))
```

The four procedures are equivalent in the sense that they evaluate to the same function. However, there is a crucial difference in terms of the ***creation time*** of the auxiliary closures, i.e., the closures used by the final `n`-th derivative procedure. Consider for example:

```
(define five-exp (lambda (x) (* x x x x x)))
(define fourth-deriv-of-five-exp (nth-deriv five-exp 4))
```

1. The first and second versions create **no closure** at compile time. Every application of `fourth-deriv-of-five-exp` to an argument number, repeatedly computes all the lower derivative closures prior to the application. In the first version a closure is created even for the 0-derivative, and a test for whether the derivative rank is zero is performed as part of the application to an argument number, while in the second version this test is performed prior to the closure creation, and no extra closure is created for the 0 case.

2. The third and forth versions create **four closures** – the first, second, third and fourth derivatives of `five-exp` at compile time. No closure is created when `fourth-deriv-of-five-exp` is applied to an argument number. They differ in terms of the process recursive-iterative properties: The third version is iterative, while the fourth is recursive.

As a conclusion we see that closures can be compared on the iterative-recursive dimension and on the compile-run time dimension. Clearly, an iterative version that maximizes the compile time creation is the preferred one, i.e., the third version, in the above example.

1.5.5 Procedures that Store Delayed (Future) Computations

An important usage of lambda abstraction is ***delaying*** a computation, since when a procedure returns a closure, the computation embedded in a returned closure is not applied. We say that the computation is ***delayed***. We present two cases where delayed computation using lambda abstraction solves a hard problem in an elegant (simple) way.

Delayed computation for distinguishing ambiguous values:

Example 1.15.

The task in this example is to write Scheme implication (`if` forms) in terms of Scheme `or` and `and`. Recall that Scheme `or` and `and` are optimized to save redundant evaluation (they are special operators, not primitive procedures).
First try:

```
(if condition consequence alternative) ==>
(or (and condition consequence) alternative)
```

For example,

```
> (define x 0)
> (define y 6)
> (if (zero? x) 100000 (/ y x))
100000
> (or (and (zero? x) 100000) (/ y x))
100000
```

But what about

```
> (if (zero? x) #f #t)
#f
> (or (and (zero? x) #f) #t)
#t
```

The problem is, of course, that once the condition does not evaluate to `#f`, the value of the overall expression is the value of the consequence, even if the consequence value is `#f`. But in this case the `or` operator evaluates the alternative. The `or` form can be corrected by delaying the computation of the consequence and the alternative, using lambda abstraction:

```
(if condition consequence alternative) ==>
((or (and condition (lambda () consequence)) (lambda () alternative)))

> (if (zero? x) #f #t)
#f
> ((or (and (zero? x) (lambda () #f)) (lambda () #t)))
#f
```

Delayed computation for obtaining iterative processes Recall the distinction between recursive to iterative processes. A procedure whose computations are recursive is characterized by a storage (frames in the Procedure-call stack) that stores the future computations, once the recursion basis is computed. In Section 1.4 we provided iterative versions for some recursive procedures. But they were based on new algorithms that compute the same functions. In many cases this is not possible.

A recursive procedure can be turned iterative by using high order procedures that store the delayed computations. These delayed computation procedures are created at run-time and passed as arguments. Each delayed computation procedure corresponds to a frame in the Procedure-call stack. The resulting iterative procedure performs the recursion base, and calls itself with an additional delayed computation argument procedure.

Example 1.16. *Turning the factorial recursive procedure iterative by using procedures that store the delayed computations:*

Consider the recursive factorial procedure:

```
Signature: factorial(n)
Purpose: to compute the factorial of a number 'n'.
    This procedure follows the rule: 1! = 1, n! = n * (n-1)!
Type: [Number -> Number]
Pre-conditions: n > 0, an integer
Post-condition: result = n!
```

```
Example: (factorial 4) should produce 24
Tests:  (factorial 1) ==> 1
        (factorial 4) ==> 24

(define factorial
   (lambda (n)
      (if (= n 1)
          1
          (* n (factorial (- n 1))))
    ))
```

We turn it into a procedure `fact$` that accepts an additional parameter for the delayed (future) computation. This procedure will be applied once the `factorial` procedure apply, i.e., in the base case. Otherwise, it will be kept in a run-time created closure that waits for the result of `factorial` on $n - 1$:

```
(define fact$
    (lambda (n cont)
       (if (= n 0)
           (cont 1)
           (fact$ (- n 1) (lambda (res) (cont (* n res)))))
    ))
```

The `cont` parameter stands for the delayed computation (`cont` stands for `continuation`). If the given argument is the base case, the continuation is immediately applied. Overawes, a closure is created that “stores” the `cont` procedure and waits until the result on $n - 1$ is obtained, and then applies. The created continuations are always procedures that just apply the delayed computation and a later continuation to the result of a recursive call:

```
(fact$ 3 (lambda (x) x))
==>
(fact$ 2 (lambda (res)
            ( (lambda (x) x)
              (* 3 res))))
==>
(fact$ 1 (lambda (res)
            ( (lambda (res)
                 ( (lambda (x) x) (* 3 res)))
              (* 2 res))))
==>
(fact$ 0 (lambda (res)
            ( (lambda (res)
                 ( (lambda (res)
                      ( (lambda (x) x) (* 3 res)))
                    (* 2 res)))
              (* 1 res))))
==>
( (lambda (res)
     ( (lambda (res)
          ( (lambda (res)
               ( (lambda (x) x) (* 3 res)))
            (* 2 res)))
       (* 1 res)))
  1)
==>
( (lambda (res)
     ( (lambda (res)
          ( (lambda (x) x) (* 3 res)))
       (* 2 res)))
  1)
==>
( (lambda (res)
     ( (lambda (x) x) (* 3 res)))
  2)
==>
( (lambda (x) x) 6)
==>
6
```

The continuations are built one on top of the other, in analogy to the frames in the Procedure-call stack. In Section 4.2 we present this idea as the *Continuation Passing Style* (*CPS*) approach.

1.6 Polymorphic Procedures

So far we have created and manipulated powerful procedures that demonstrate complex programming features. The type of a procedure was intuitively inferred from its code, based on the primitive elements in it. For example, the type of the `average` procedure below is determined by the known type of the primitive procedures `+` and `/`. The type of `average-damp` is derived from the type of `average`.

```
average-damp:
Signature: average-damp(f)
```

```
Type: [[Number -> Number] -> [Number -> Number]]
(define average-damp
   (lambda (f)
      (lambda (x) (average x (f x)))))

Type: Number*Number -> Number]
(define average
  (lambda (x y)
     (/ (+ x y) 2)))
```

But, consider the following abstraction on `average-damp`, where the application (call) to `average-damp` is abstracted as a call to a parameter procedure `op`:

```
(define op-damp
   (lambda (f op)
      (lambda (x) (op x (f x)))))
```

What is the type of `op-damp`?

```
(define op-damp
   (lambda (f op)
      (lambda (x) (op x (f x)))))
```

Consider its applications:

```
> (op-damp - +)
#<procedure>
> ((op-damp - +) 3)
0
> (op-damp not (lambda (x y) (and x y)))
#<procedure>
> ((op-damp not (lambda (x y) (and x y))) #t)
#f
```

We see that `op-damp` returns a closure, but of which type? What is the type of its parameters? Based on the code we can determine that both parameters are one-parameter procedures (primitives or closures), but we cannot determine their types. Moreover, these closures have varying types: The type of `(op-damp - +)` is `number -> number` while the type of `(op-damp not (lambda (x y) (and x y)))` is `Boolean -> Boolean`.

We say that procedures that can accept arguments and return values of varying types are ***polymorphic***, i.e., have multiple types and the expressions that create these procedures are ***polymorphic expressions***. Such procedures belong to multiple Procedure types, based on their argument types.

Here are further examples of polymorphic procedures:

```
> ((lambda (x) x) 3)
3
> ((lambda (x) x) #t)
#t
> ((lambda (x) x) (lambda (x) (- x 1)))
#<procedure:x>
> ((lambda (x) x) (lambda (x) x))
#<procedure:x>

> ( (lambda (f x) (f (f x)))
    abs -3)
3
> ( (lambda (f x) (f (f x)))
    (lambda (x) (* x x))
    3)
81
```

We see that the closure ⟨`closure (x) x`⟩ can belong to the types
`[Number -> Number]`,
`[Boolean -> Boolean]`,
`[[Number -> Number] -> [Number -> Number]]`.

In order to assign a single type expression to polymorphic language expressions (and procedures) we introduce ***type variables***, denoted `T1, T2, ...`. Type variables in type expressions range over all types. Type expressions that include type variables are called ***polymorphic type expressions*** since they denote multiple types.

- The type of the identity `lambda` expression and procedure is `[T -> T]`.
- The type of the expression `(lambda (f x) (f x))` is `[[T1 -> T2]*T1 -> T2]`.
- The type of the expression `(lambda (f x) (f (f x)))` is `[[T -> T]*T -> T]`.
- The type of `op-damp` is `[[T1 -> T2]*[T1*T2 -> T3] -> [T1 -> T3]]`.

Polymorphic pair/list procedures: `cons, car, cdr` are ***polymorphic procedures***: They apply to arguments, and return results of multiple types.

Example 1.17 (A polymorphic pair-valued procedure).

```
Signature: pairself(x)
Purpose: Construct a pair of a common component.
type: [T -> Pair(T,T)]
```

```
(define pairself (lambda(x)
                   (cons x x)))
> (pairself 3)
(3 . 3)
> (pairself (+ 3 4))
(7 . 7)
> (pairself (lambda(x) x))
(#<procedure> . #<procedure>)
```

Example 1.18.

```
Signature: firstFirst(pair)
Purpose: Retrieve the first element of the first element of a pair.
Type: Pair(Pair(T1,T2),T3) -> T1
(define firstFirst (lambda(pair)
                   (car (car pair))))

Type: [T -> Boolean]
(define firstFirst-argument-type-test
     (lambda (pair) (and (pair? pair) (pair? (car pair)))))
```

Example 1.19.

```
Signature: member(el, pair)
Purpose: Find whether the symbol el occurs in pair.
Type: [Symbol*Pair(T1,T2) -> Boolean]
(define member (lambda (el pair)
                (cond ((and (pair? (car pair))
                            (member el (car pair))) #t)
                      ((eq? el (car pair)) #t)
                      ((and (pair? (cdr pair))
                            (member el (cdr pair))) #t)
                      ((eq? el (cdr pair)) #t)
                      (else #f))))
```

Note that before recursively applying `member`, we test the calling arguments for their type. Indeed, according to the Design by Contract policy, it is the responsibility of clients to call procedures with appropriate arguments, i.e., arguments that satisfy the procedure's precondition and type. Procedures that take over the responsibility for testing their arguments demonstrate ***defensive programming***.

Below is a better version, that uses a type-test procedure. The type of the arguments to `member` is not checked in the procedure.

```
(define member (lambda (el pair)
         (cond ((and (member-type-test el (car pair))
                     (member el (car pair))) #t)
               ((eq? el (car pair)) #t)
               ((and (member-type-test el (cdr pair))
                     (member el (cdr pair))) #t)
               ((eq? el (cdr pair)) #t)
               (else #f))))

Type: [T1*T2 -> Boolean]
(define member-type-test
  (lambda (el pair-candidate)
    (and (symbol? el)
         (pair? pair-candidate))))
```

Here is a defensive programming version:

```
(define member (lambda (el pair)
         (cond ((not (member-type-test el pair))
                (error 'member "wrong type arguments: el = ~s pair = ~s" el pair))
               ((and (member-type-test el (car pair))
                     (member el (car pair))) #t)
               ((eq? el (car pair)) #t)
               ((and (member-type-test el (cdr pair))
                     (member el (cdr pair))) #t)
               ((eq? el (cdr pair)) #t)
               (else #f))))
```

Summary of the demonstrated computing features:

In this chapter we have shown how Scheme handles several essential computing features:

1. Performing iteration computation, without losing the advantages of iteration.
2. Procedures as arguments to procedures. This feature enables the formulation of essential abstract notions, like sequence summation, that can further support more abstract notions.
3. Run time created procedures (anonymous procedures) that save the population of a name space with one-time needed procedures.
4. Local variables.
5. Procedures that create procedures as their returned values.

6. Polymorphic procedures.
7. Compile (design) time vs. run time computation.
8. Delayed computation.
9. Hierarchical data types.
10. Procedure contracts.

Glossary: Language expressions – atomic, composite; Primitive elements; Semantics; Types – atomic, composite; Type specification language; Value constructors; Type constructors; Design by contract; Preconditions; Postconditions; Procedure signature; High order procedures; Process, Linear recursive process, Iterative process, space/time resources, recursive procedure, tail recursion, iterative procedures; Side-effect, syntactic sugar, derived expressions.

Chapter 2

Theoretical Introduction of Programming Languages: Syntax, Semantics, Types

Partial sources:
SICP [1] 1.1.5, 1.1.7, 1.3;
Krishnamurthi [8] 3, 24-26 (online version `http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-04-26.pdf`);
Friedman and Wand [3] 7 (online version: `http://dl.taq.ir/computer/essentials_of_programming_languages_friedman.pdf`).

The theoretical specification of a programming language has two "legs": ***Syntax*** and ***semantics***. The syntax of a programming language consists of ***atomic*** elements, some of which are ***primitives***, and from combination and abstraction means. The specification of syntax is done using grammars. The semantics of a programming language has two aspects: The ***operational semantics*** aspect, which deals with algorithms for evaluation of language expressions, and the theory of ***types*** which deals with the specification and management of the ***values*** that are computed by language expressions.

2.1 Syntax: Concrete and Abstract

Syntax of languages can be specified by a ***concrete syntax*** or by an ***abstract syntax***. The concrete syntax includes all syntactic information needed for parsing a language element (program), e.g., punctuation marks. The abstract syntax include only the essential information needed for language processing, e.g., for executing a program. The abstract syntax is an abstraction of the concrete syntax: There can be many forms of concrete syntax for a single abstract syntax. The abstract syntax provides a layer of ***abstraction*** that protects against modifications of the concrete syntax.

2.1.1 Concrete Syntax:

The concrete syntax of a language defines the actual language. The concrete syntax of Scheme is a small and simple context free grammar (Scheme is a context free language, unlike most programming languages).

We use the BNF notation for specifying the syntax of Scheme. Quote from Wikipedia:

> In computer science, ***Backus-Naur Form*** (***BNF***) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages. John Backus and Peter Naur developed a context free grammar to define the syntax of a programming language by using two sets of rules: i.e., lexical rules and syntactic rules.

BNF is widely used as a notation for the grammars of computer programming languages, instruction sets and communication protocols, as well as a notation for representing parts of natural language grammars.

1. Syntactic categories (non-terminals) are denoted as `<category>`.
2. Terminal symbols (tokens) are surrounded with `'`.
3. Optional items are enclosed in square brackets, e.g. `[<item-x>]`.
4. Items repeating 0 or more times are enclosed in curly brackets or suffixed with an asterisk, e.g. `<word> -> <letter> <letter>*`.
5. Items repeating 1 or more times are followed by a `+`.
6. Alternative choices in a production are separated by the `|` symbol, e.g., `<alternative-A>` | `<alternative-B>`.
7. Grouped items are enclosed in simple parentheses.

Concrete syntax of the subset of Scheme introduced so far:

```
<scheme-exp>       -> <exp> | '(' <define> ')'
<exp>              -> <atomic> | '(' <composite> ')'
<atomic>           -> <number> | <boolean> | <variable> | <quoted-variable>
<composite>        -> <special> | <macro> | <form>
<number>           -> Numbers
<boolean>          -> '#t' | '#f'
<variable>         -> Restricted sequences of letters, digits,punctuation marks
<quoted-variable>  -> A variable starting with '
```

```
<special>          -> <lambda> | <quote> | <cond> | <if>
<macro>            -> <let>
<form>             ->  <exp>+
<define>           ->  'define' <variable> <exp>
<lambda>           ->  'lambda' '(' <variable>* ')' <exp>+
<quote>            ->  'quote' <variable>
<cond>             ->  'cond' <condition-clause>* <else-clause>
<condition-clause> -> '(' <exp> <exp>+ ')'
<else-clause>      -> '(' 'else' <exp>+ ')'
<if>               ->  'if' <exp> <exp> <exp>
<let>              ->  'let' '(' <var-definition>* ')' <exp>+
<var-definition>   ->  '(' <variable> <exp> ')'
```

Note that the <define> expression cannot be nested in other combined expressions. Therefore, a <define> expression can appear only at the top level of Scheme expressions.

The expressions of the Scheme language are obtained from the concrete syntax by terminal derivations from the start symbol <Scheme-EXP>. For example, the expression `(if #f (/ 1 0) 2)` is syntactically correct in Scheme because it can be derived in the above syntax. Here is a derivation which produces it:

```
<scheme-exp> ->
<exp> ->
( <composite> ) ->
( <special> ) ->
( <if> ) ->
( if <exp> <exp> <exp> )->
( if <atomic> <exp> <exp> )->
( if <boolean> <exp> <exp> ) ->
( if #f <exp> <exp> ) ->
( if #f <exp> <atomic> ) ->
( if #f <exp> number) ->
( if #f <exp> 2) ->
( if #f ( <composite> ) 2) ->
( if #f ( <form> ) 2) ->
( if #f ( <exp> <exp> <exp> ) 2) ->
( if #f ( <atomic> <exp> <exp> ) 2) ->
( if #f ( <variable> <exp> <exp> ) 2) ->
( if #f ( / <exp> <exp> ) 2) ->
( if #f ( / <atomic> <exp> ) 2) ->
( if #f ( / <number> <exp> ) 2) ->
( if #f ( / 1 <exp> ) 2) ->
( if #f ( / 1 <atomic> ) 2) ->
( if #f ( / 1 <number> ) 2) ->
( if #f ( / 1 0 ) 2).
```

Write a derivation tree for (if #f (/ 1 0) 2).

2.1.2 Abstract Syntax

The abstract syntax of a language emphasizes the ***content parts*** of the language, and ignores syntactical parts that are irrelevant for the semantics. For example, the exact ordering of the arguments in a `define` special form, or the exact parentheses or phrasing symbols, are irrelevant. In Scheme, we could have replaced the "(" and ")" by "<" and ">", respectively; or replace the white space by commas, without changing the denotation of the expressions. A single abstract syntax can be an abstraction of multiple concrete syntax grammars.

Abstract syntax singles out alternative ***kinds*** of a category, and the ***components*** of a composite element. For each component, the abstract syntax emphasizes its ***role*** in the composite sentence, its ***category***, its ***amount*** in the composite sentence, and whether its instances are ordered.

Abstract syntax choices and structure can be represented by an ***and/or tree*** whose nodes are labeled by grammatical categories. Alternative kinds of a category are represented by an ***or-branch***, and the components of a composite grammatical category are represented by an ***and-branch***, whose child nodes are labeled by the component categories. Component branches are labeled by their roles, and the child nodes are also annotated by the amount using the `+, *` symbols, and ordering.

The abstract syntax of a concrete syntax can be constructed by following the rules of the concrete syntax, and creating an or-branch for a rule choice and an and-branch for a composite rule. For example, the concrete syntax rule for the `scheme-exp`

```
<scheme-exp>        -> <exp> | '(' <define> ')'
```

is represented by the and/or tree in Figure 2.1.
The concrete syntax rule for a `lambda` form:

```
<lambda>  -> '(' 'lambda' '(' <variable>* ')' <exp>+ ')'
```

is represented by the and/or tree in Figure 2.2.
The full abstract syntax of a language is obtained by combining the and/or trees of all language categories. For example, the abstract syntax of the `<scheme-exp>, <exp>` and `<define>` categories of Scheme are represented by the and/or tree in Figure 2.3.

Compilers and interpreters use an ***Abstract Syntax Parser*** (***ASP***) which defines an **interface to the syntax**, to be used by all clients of the parser. The separation of abstract syntax from the concrete syntax provides an additional degree of freedom to compilers and interpreters.

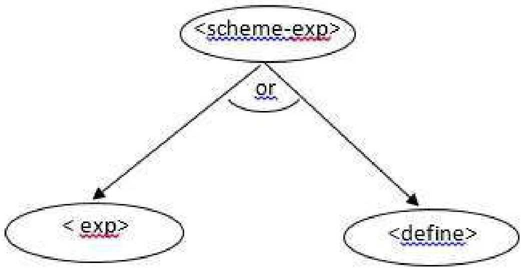


Figure 2.1: And/or tree for `<scheme-exp>`

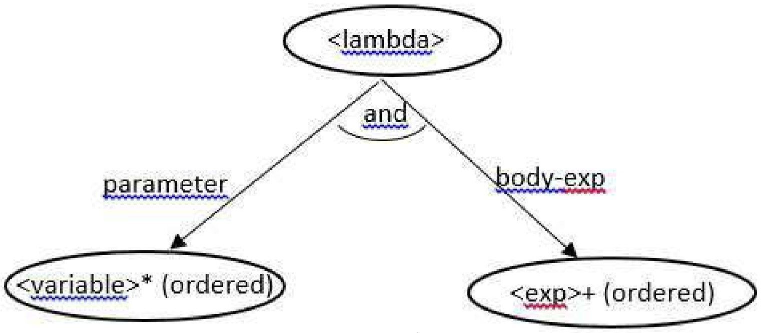


Figure 2.2: And/or tree for `<lambda>`

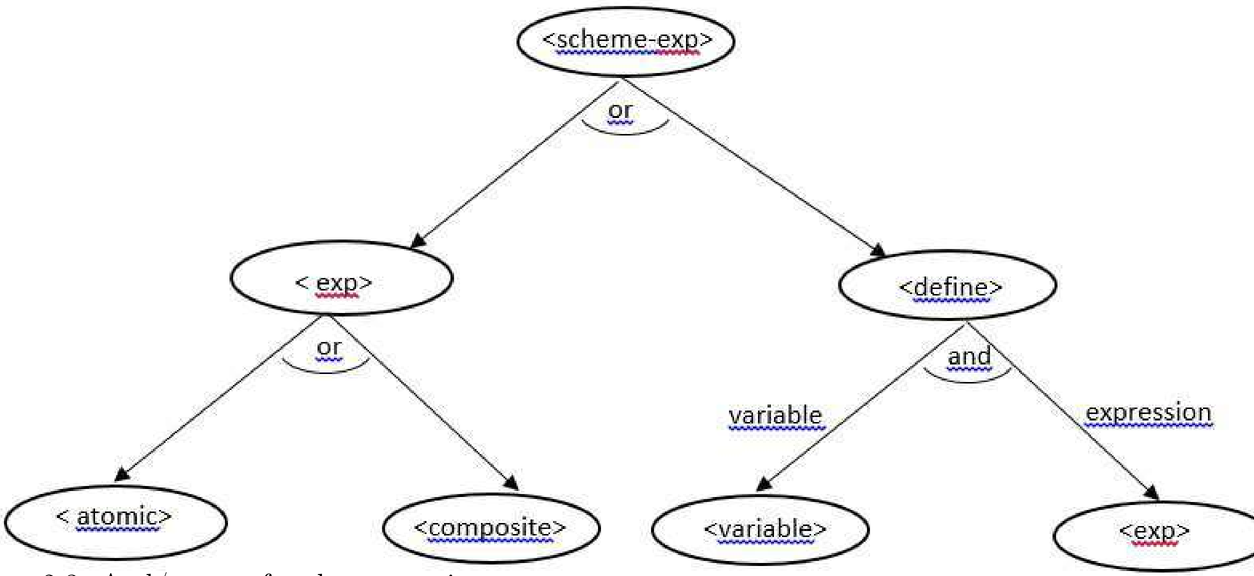


Figure 2.3: And/or tree for the categories `<scheme-exp>, <exp>, <define>`

The and/or tree of the abstract syntax of a language determines the ***Abstract Syntax Tree*** (***AST***) representations of language expressions. For a language expressions `e`, AST(`e`) is a tree whose internal nodes are labeled by language categories, roles or operators, and its leaves are labeled by atomic elements of the expression (language terminal symbols) that are not punctuation marks. For example, an AST of the expression `(lambda (x y) (+ x y))` might be the tree in Figure 2.4. It can be created by following the `and/or-branches` of the appropriate ASD. The UML Class Diagram is a good language for specifying abstract syntax: It can describe the grammar categories and their inter-relationships, while ignoring the concrete syntax details.

Symbolic description of the scheme abstract syntax: We specify, for each category, its ***kinds*** or its ***components*** (parts).

```
<scheme-exp>:
    Kinds: <exp>, <define>
<exp>:
    Kinds: <atomic>, <composite>
<atomic>:
    Kinds: <number>, <boolean>, <variable>
<composite>:
    Kinds: <special>, <form>
<number>:
    Kinds: numbers.
<boolean>:
    Kinds: #t, #f
```

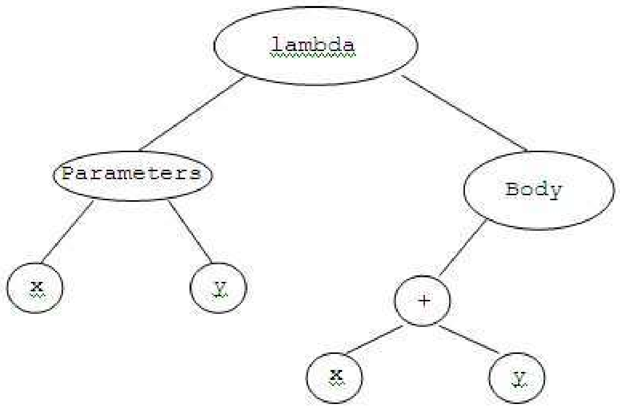


Figure 2.4: A possible AST for `(lambda (x y) (+ x y))`

```
<variable>:
    Kinds: Restricted sequences of letters, digits, punctuation marks
<special>:
    Kinds: <lambda>, <quote>, <cond>, <if>, <let>
<form>:
    Components: Expression: <exp>. Amount: >= 1. Ordered.
<define>:
    Components: Variable: <variable>
                Expression: <exp>
<lambda>:
     Components: Parameter: <variable>. Amount: >= 0. Ordered.
                 Body: <exp>. Amount: >= 1 . Ordered.
<quote>:
     Components: Quoted-name: <variable>
<cond>:
     Components: Clause: <condition-clause>. Amount >= 0. Ordered.
                 Else-clause: <else-clause>
<condition-clause>:
     Components: Predicate: <exp>
                 Action: <exp>. Amount: >= 1. Ordered.
<else-clause>:
     Components: Action: <exp>. Amount: >= 1. Ordered.
<if>:
     Components: Predicate: <exp>
                 Consequence: <exp>
                 Alternative: <exp>
```

```
<let>:
     Components: Var: <variable-initialization>. Amount: >= 0.
                 Body: <exp>. Amount: >= 1 . Ordered.
<variable-initialization>:
     Components: Variable: <variable>
                 Expression: <exp>
```

The Scheme interpreters and compilers in chapter 6 use an abstract syntax based parser for analyzing Scheme expressions. Figure 2.5 presents a ***UML*** class diagram [12, 9] for the

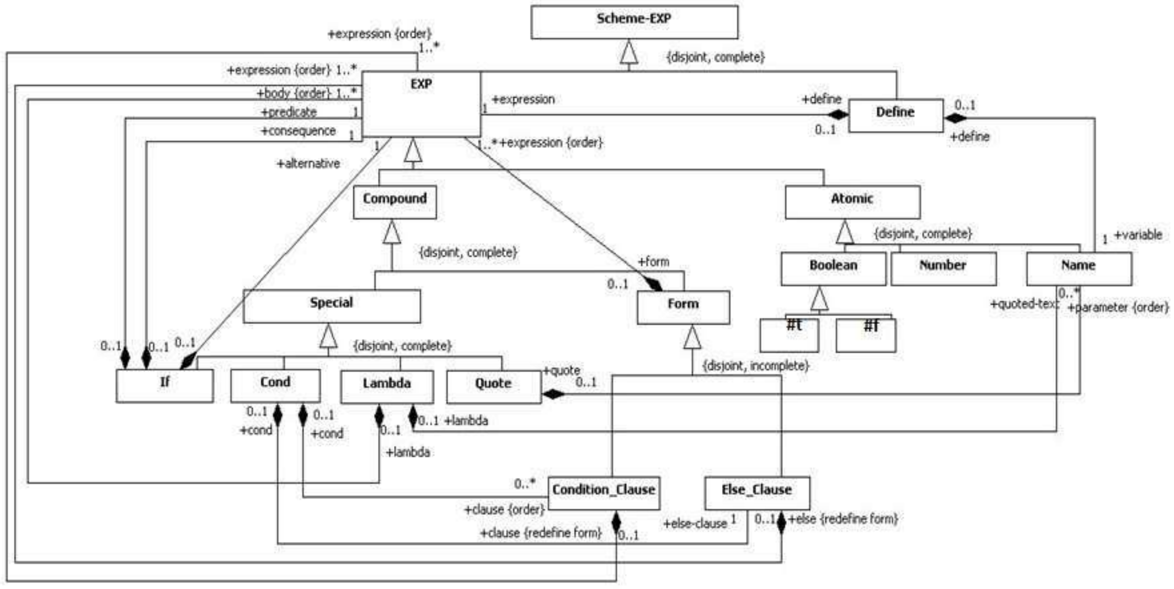


Figure 2.5: Scheme Abstract Syntax formulated in UML class diagram

1. ***Categories*** are represented as classes.
2. ***Kinds*** are represented by class hierarchy relationships.
3. ***Components*** are represented as composition relationships between the classes.

2.2 Operational Semantics: The Substitution Model

The operational semantics is specified by a set of formal ***evaluation rules*** that can be summarized as an ***algorithm*** `eval(exp)` for evaluation of Scheme expressions.

In order to formally define `eval(exp)` we first introduce several concepts:

1. ***Binding instance*** or ***declaration***: A ***binding instance*** (***declaration***) of a variable is a variable occurrence to which other occurrences refer.
   In Scheme:
   (a) Variables occurring as `lambda` parameters are ***binding instances*** (***declarations***).
   (b) Top level (non-nested) defined variables (in a `define` form) are ***binding instances*** (***declarations***).
   (c) Variables occurring as `let` local variables are ***binding instances*** (***declarations***).
2. ***Scope***: The scope of a binding instance x is the region of the program text in which occurrences of x refer to (are ***bound*** by) the declaration of x (where the variable declaration is ***recognized***).
   In Scheme:
   (a) The scope of `lambda` parameters is the entire `lambda` expression.
   (b) The scope of `define` variables is the entire program, from the `define` expression and on: ***Universal scope***.
   (c) The scope of `let` local variables is the entire `let` body (not including the initialization expressions).
3. ***Bound occurrence in an expression***: Occurrences of a variable `x` (not as binding instances (declarations)) within the scope of a binding instance `x` are called ***bound***. The ***binding instance*** (***declaration***) that ***binds*** an occurrence of `x` is the most nested declaration of `x` that includes the `x` occurrence in its scope.
4. ***Free occurrence in an expression***: Occurrences of a variable `x` (not as binding instances) that are not bound are ***free***.

```
(lambda (x) (+ x 5))
  ;x is bound by its declaration in the parameter list.

(define y 3)   ;A binding y instance.

(+ x y)   ;x, + are free, y is bound (considering the above evaluations).

(+ x ( (lambda (x) (+ x 3)) 4))
            ;the 1st x occurrence is free,
            ;the 2nd is a binding instance,
            ;the 3rd is bound by the declaration in the lambda parameters.

(lambda (y)     ;a binding instance
  (+ x y
      ;the x occurrence is free
      ;the y occurrence is bound by the outer declaration
     ((lambda (x y) (+ x y))
      ;the x and y occurrences are bound by the internal declarations.
       3 4)
))

An equivalent form:
(lambda (y)
  (+ x y
    (let ((x 3)
          (y 4))   ;declarations of x and y
       (+ x y))
          ;the x and y occurrences are bound by the let declarations.
))
```

Note: A variable is bound or free with respect to an expression in which it occurs. A variable can be free in one expression but bound in an enclosing expression.

5. ***Renaming***: Bound variables in expressions can be consistently renamed by ***new*** variables (not occurring in the expression) without changing the intended meaning of the expression. That is, expressions that differ only by consistent renaming of bound variables are considered ***equivalent***. For example, the following are equivalent pairs:

```
(lambda (x) x)                     ==>      (lambda (x1) x1)

(+ x ( (lambda (x) (+ x y)) 4))    ==>   (+ x ( (lambda (x1) (+ x1 y)) 4))

Incorrect renaming:
(+ x ( (lambda (y) (+ y y)) 4))    ==>   (+ x1 ( (lambda (x1) (+ x1 y)) 4)))
```

6. ***The substitute operation***: An operation of ***replacement*** of ***free occurrences*** of variables in an expression by given ***expressions***.

Definition: A ***substitution*** s is a mapping from a finite set of variables to a finite set of syntactic expressions. A ***binding*** is a pair $\langle x, s(x)\rangle$ such that $x = s(x)$.

Substitutions are written using set notation. For example:

– `{x = 3, y = a, z = #t, w = (lambda(x) (+ x 1))}` is a substitution, while

– `{x = 3, x = a, z = #t, w = (lambda(x) (+ x 1))}` is not a substitution.

Definition: ***Composition (combination) of substitutions*** The composition of substitutions s and s', denoted $s \circ s'$, is a substitution s'' that extends s with a binding $\langle x, s'(x)\rangle$ for every variable x for which $s(x)$ is not defined.

For example,
`{x = 3, y = a}`∘`{z = #t, w = (lambda(x)(+ x 1))}` =
`{x = 3, y = a, z = #t, w = (lambda(x)(+ x 1))}`
The empty substitution { } is the ***neutral*** (unit) element of the substitution-composition operation: For every substitution s, $\{\ \} \circ s = s \circ \{\ \} = s$.

Definition: ***The substitute operation*** is an application of a substitution s to an expression E. It is denoted $E \circ s$ (or just Es if no confusion arises), and involves replacement of free variable occurrences by values:

(a) Consistent ***renaming*** of the expression E and the values in s.

(b) Simultaneous ***replacement*** of all ***free*** occurrences of the variables of `s` in (the renamed) `E` by (the renamed) corresponding values.

Example 2.1 (Substitution applications $E \circ s$).

(a) `10`∘`{x = 5} = 10`: No renaming; no replacement.

(b) `(+ x y)`∘`{x = 5} = (+ 5 y)`: No renaming; just replacement.

(c) `(+ x y)`∘`{x = 5, y = 'x} = (+ 5 'x)`: No renaming; just replacement.
Note the difference between the variable `x` to the Symbol value `'x`.

(d) `((+ x ((lambda (x) (+ x 3)) 4)))`∘`{x = 5} =`

  i. Renaming: E turns into `((+ x ((lambda (x1) (+ x1 3)) 4)))`

  ii. Substitute: E turns into `((+ 5 ((lambda (x1) (+ x1 3)) 4)))`

(e) `(lambda (y)(((lambda (x) x) y) x))`∘`{x = (lambda(x)(y x))}`:
Variable `y` in the substitution is free. It should stay free after the substitution application.

  i. Renaming:
  The substitution turns into `(lambda (x1) (y x1))`;
  E turns into `(lambda (y2) (((lambda (x3) x3) y2) x))`;

  ii. Substitute: E turns into

```
(lambda (y2) ( ((lambda (x3) x3) y2)
               (lambda (x1) (y x1)))
)
```

What would be the result without renaming? Note the difference in the binding status of the variable `y`.

(f) `(lambda (f) (g (f 3)))∘{g = (lambda (x) (f x))}:`
Variable `f` in the substitution is free. It should stay free after the substitution is applied.

i. Renaming:
The substitution turns into `(lambda (x1) (f x1))`;
E turns into `(lambda (f2) (g (f2 3)))`;

ii. Substitute:
E turns into `(lambda (f2) ((lambda (x1) (f x1)) (f2 3)))`

Applying a substitution to multiple expressions: The notion is extended into $E_1, E_2, \ldots, E_n \circ s$. It will be useful for lambda expressions and closures with multiple expressions in their bodies.
Writing agreement: In manual descriptions of substitutions, if there is no variable overlapping between the substitution values and the substituted expression, we save the explicit writing of the renaming step.

2.2.1 The Substitution Model – Applicative Order Evaluation:

The substitution model uses ***applicative order evaluation***, which is an ***eager approach*** for evaluation. The rules formalize the informally stated rules in Chapter 1:

1. ***Eval***: Evaluate the elements of the compound expression;

2. ***Apply***: Apply the procedure which is the value of the operator of the expression, to the ***arguments***, which are the values of the operands of the expression. For user defined procedures (closures) this step is broken into 2 steps: ***Substitute*** and ***reduce***.

   (a) **Substitute:** Replaces the evaluated arguments for free occurrences of the parameters in the procedure body, yielding the so called *reduced body*.

   (b) **Reduce:** Recursive application of the evaluation algorithm on the reduced body.

Therefore, the model is also called ***eval-substitute-reduce***. The algorithm that defines the operational semantics is called `applicative-eval`. The input to the algorithm can be a plain scheme expression, or the result of the application of a substitution to a scheme expression. Therefore, the algorithm is a function:
`applicative-eval: <scheme-exp>`$\rightarrow$ *Scheme_type*
Scheme_type = $Number \cup Boolean \cup Symbol \cup Procedure \cup Pair \cup List \cup Void \cup Empty$.

We use the identifying predicates `atom?, composite? number?, boolean?`, and `variable?`, for identifying atomic, composite, number, boolean, and variable expressions, respectively.

The predicates `primitive-procedure?`, and `procedure?` are used for identifying primitive procedures and user defined procedures, respectively.
How to avoid mixing syntactic expressions with values: The rules for application of a user procedure create a confusion between syntactic expressions and computed values. The *eval* step creates values, which should be substituted for parameter occurrences in the procedure body. But this might create an undesirable mixture of values within a syntactic expression. In order to avoid that, the algorithm uses the following assumption:

> Every type in *Scheme_type* has *value->syntax* function that maps values to a Scheme expression that evaluates to the given value:
> **Numbers, booleans:** *value->syntax*(v) = `v`
> **Symbols:** *value->syntax*('v) = `(quote v)`
> **Pairs:** for a pair *p* whose literal expression is `(a1 . a2)`, *value->syntax*(*p*) = `(cons a1 a2)`
> **Lists:** For a list *lst* whose literal expression is `(a1 a2 ... an)`, *value->syntax*(*lst*) = `(cons a1 (const a2 ... (cons an ()) ... ))`
> **Closures:** For a closure *cl* whose literal expression is `<closure(x1 ... xn) e1 e2 ... em>`, *value->syntax*(*cl*) = `(lambda (x1 ... xn) e1 e2 ... em)`

The `applicative-eval` algorithm:

```
Signature: applicative-eval(e)
Purpose: Evaluate a Scheme expression
Type: [<scheme-exp> → Scheme-type]

Definition:
applicative-eval[e] =
  I. atom?(e):
     1. number?(e) or boolean?( e) or empty-list?(e):
        applicative-eval[e] = make-number(e) or make-boolean(e) or make-empty(e)
        ;a number or a boolean or an empty-list atom evaluates into a number or
         a boolean  or the empty list value, respectively
     2. variable?(e):
        a. If GE(e) is defined:
           applicative-eval[e] = GE(e)
        b. Otherwise: e must be a variable denoting a Primitive procedure:
           applicative-eval[e] = primitive-value(e).

  II. composite?(e): e = (e0 e1 ... en)(n >= 0):
     1. e0 is a Special Operator:
        applicative-eval[e] is defined by the special evaluation rules
```

```
      of e0 (see below).
   2. a. Evaluate: compute applicative-eval[ei] = vi’ for i=0..n.
      b. primitive-procedure?(v0’):
         applicative-eval[e] = system application v0’(v1’, ..., vn’).
      c. procedure?(v0’):
         v0’ is a closure: <Closure (x1 ... xn) b1 ... bm>
         i.  value->syntax: let ei’ = value->syntax(vi’), for i=1..n

         ii.  Substitute (preceded by renaming):
              b1, ..., bm ∘{x1 = e1’, ..., xn = en’} = b1’, ..., bm’

         iii. Reduce: applicative-eval[b1’], ..., applicative-eval[b(m-1)’].
              applicative-eval[e] = applicative-eval[bm’].
```

Special operators evaluation rules:

```
1.  e = (define x e1):
    GE = GE∘{x=applicative-eval[e1]}
    applicative-eval[e]=void

2.  e = (lambda (x1 x2 ... xn) b1 ... bm), m>=1:
    applicative-eval[e] = make-closure((x1 ...xn),(b1 ... bm)) =
                          <Closure (x1 ...xn) b1 ... bm>
3.  e = (quote a):    ;a must be a symbol
    applicative-eval[e] = make-symbol(a)
4.  e = (cond (p1 e11 ...) ... (else en1 ...)):
    If not(false?(applicative-eval[p1])):
       applicative-eval[e11], applicative-eval[e12], ...
       applicative-eval[e] = applicative-eval[last e1i]
    Otherwise, continue with p2 in the same way.
    If for all pi-s applicative-eval[pi] = #f:
       applicative-eval[en1], applicative-eval[en2], ...
       applicative-eval[e] = applicative-eval[last eni]
5.  e = (if p con alt):
    If not(false?(applicative-eval[p])):
       then applicative-eval[e] = applicative-eval[con]
       else applicative-eval[e] = applicative-eval[alt]
```

Note: `value?(e)` holds for all values computed by `applicative-eval`, i.e., for numbers, boolean values (`#t, #f`), symbols (computed by the `Symbol` value constructor `quote`), closures (computed by the `Procedure` value constructor `lambda`), and pairs and lists.

Example 2.2. *Assume that the following definitions are already evaluated:*

```
(define square (lambda (x) (* x x)))
(define sum-of-squares (lambda (x y)
                          (+ (square x) (square y))
                      ))
(define f (lambda (a)
              (sum-of-squares (+ a 1) (* a 2) )
         ))
(define sum
  (lambda (a term b next)
    (if (> a b)
        0
        (+ (term a) (sum (next a) term b next)))))
```

Apply `applicative-eval`:

```
> (f 5)
136
> (sum 3 (lambda(x)(* x x)) 5 (lambda(x)(+ x 1)))
50

applicative-eval[ (f 5) ] ==>
        applicative-eval[ f ] ==>
                       <Closure (a) (sum-of-squares (+ a 1) (* a 2) )>
        applicative-eval[ 5 ] ==> 5
 ==>
 applicative-eval[ (sum-of-squares (+ 5 1) (* 5 2)) ] ==>
        applicative-eval[sum-of-squares] ==>
                        <Closure (x y) (+ (square x) (square y))>
        applicative-eval[ (+ 5 1) ] ==>
               applicative-eval[ + ] ==> <primitive-procedure +>
               applicative-eval[ 5 ] ==> 5
               applicative-eval[ 1 ] ==> 1
        ==> 6
        applicative-eval[ (* 5 2) ] ==>
               applicative-eval[ * ] ==> <primitive-procedure *>
               applicative-eval[ 5 ] ==> 5
               applicative-eval[ 2 ] ==> 2
        ==> 10
 ==>
 applicative-eval[ (+ (square 6) (square 10)) ] ==>
        applicative-eval[ + ] ==> <primitive-procedure +>
        applicative-eval[ (square 6) ] ==>
```

```
               applicative-eval[ square ] ==> <Closure (x) (* x x)>
               applicative-eval[ 6 ] ==> 6
        ==>
        applicative-eval[ (* 6 6) ] ==>
              applicative-eval[ * ] ==> <primitive-procedure *>
              applicative-eval[ 6 ] ==> 6
              applicative-eval[ 6 ] ==> 6
        ==> 36
        applicative-eval[ (square 10) ]
               applicative-eval[ square ] ==> <Closure (x) (* x x))>
               applicative-eval[ 10 ] ==> 10
        ==>
        applicative-eval[ (* 10 10) ] ==>
              applicative-eval[ * ] ==> <primitive-procedure *>
              applicative-eval[ 10 ] ==> 10
              applicative-eval[ 10 ] ==> 10
        ==> 100
 ==> 136
```

Example 2.3. *A procedure with no formal parameters, and with a primitive expression as its body:*

```
> (define five (lambda () 5))
> five
<Closure () 5>
> (five)
5

applicative-eval[ (five) ] ==>
        applicative-eval[ five ] ==> <Closure () 5)>
==>
applicative-eval[ 5 ] ==>
5
```

Example 2.4.

```
> (define four 4)
> four
4
> (four)
```

```
ERROR: Wrong type to apply:  4
; in expression: (... four)
; in top level environment.

applicative-eval[ (four) ] ==>
       applicative-eval[ four ] ==> 4
the Evaluate  step yields a wrong type.
```

Example 2.5.

```
> (define y 4)
> (define f (lambda (g)
                (lambda (y) (+ y (g y)))))
> (define h (lambda (x) (+ x y)))

> (f h)
<Closure (y1) (+ y1 ((lambda (x) (+ x y))
                       y1))>
> ( (f h) 3)
10

applicative-eval[ (f h) ] ==>
        applicative-eval[ f ] ==>
                       <Closure (g) (lambda (y) (+ y (g y)))>
        applicative-eval[ h ] ==>
                       <Closure (x) (+ x y)>
 ==>
```

Apply `value->syntax(<Closure (x) (+ x y)>)`, followed by substitute – rename both expressions and replace:
`(lambda (y2) (+ y2 (g y2)))∘{g = (lambda (x1) (+ x1 y))}`

```
 ==>
  applicative-eval[ (lambda (y2) (+ y2
                                  ((lambda (x1) (+ x1 y)) y2 ) )) ]
 ==>
  <Closure (y2) (+ y2  ((lambda (x1) (+ x1 y)) y2) ) >
```

Note the essential role of renaming here. Without it, the application `((f h) 3)` would replace all free occurrences of `y` by 3, yielding 9 as the result.

Example 2.6 (The essential role of the `value->syntax` translation:).

I. Repeated evaluation of a symbol value:

```
applicative-eval[((lambda (x)(display x) x) (quote a))] ==>
Eval:
  applicative-eval[(lambda (x)(display x) x)] ==> <Closure (x)(display x) x>
  applicative-eval[(quote a)] ==> the symbol 'a
Value->syntax and substitute:
```

(display x),x ∘ {x = (quote a)} = (display (quote a)), (quote a)

```
Reduce:
applicative-eval[ (display (quote a)),(quote a) ] ==>
Eval:
applicative-eval[ (display (quote a)) ] ==>
  applicative-eval[display] ==> Code of display.
  applicative-eval[(quote a)] ==> 'a ,                            (*)
==> void
Eval:
applicative-eval[(quote a)] ==> 'a
==> 'a
```

II. Repeated evaluation of a Procedure value:

```
> ((lambda (f x)(f x)) (lambda (x) x) 3)
3

applicative-eval[ ((lambda (f x)(f x)) (lambda (x) x) 3) ] ==>
Eval:
  applicative-eval[(lambda (f x)(f x))] ==> <Closure (f x)(f x)>
  applicative-eval[(lambda (x) x) ] ==> <Closure (x) x)>
  applicative-eval[3] ==> 3
Value->syntax, followed by renaming and substitution:

(f1 x1)∘{f1 = (lambda (x2) x2), x1 = 3} = ((lambda (x2) x2) 3)

Reduce:
applicative-eval[((lambda (x2) x2) 3)] ==>
Eval:
  applicative-eval[(lambda (x2) x2)] ==> <Closure (x2) x2>          (*)
  applicative-eval[3] ==> 3
Substitute x2 by 3 in x2 ==> 3
Reduce:
applicative-eval[3]= 3
```

In both evaluations, canonical syntactic expressions are created by `value->syntax` (the symbol `a` and the closure `<Closure (x2) x2>`). The evaluation completes correctly because at the point of repeated evaluation of `applicative-eval` (the lines marked by `(*)`, `applicative-eval` is given the canonical syntactic expressions, rather the already computed values. Otherwise, the first evaluation would have failed with an "unbound variable" error, and the second with "unknown type of argument". Try it!

> The ***substitution model – applicative order*** uses the ***call-by-value*** method for parameter passing.
> ***Parameter passing method***: In procedure application, the ***values*** of the actual arguments are substituted for the formal parameters. This is the ***standard*** evaluation model in Scheme (LISP), and the most frequent method in other languages (Pascal, C, C++, Java).

2.2.2 The Substitution Model – Normal Order Evaluation:

`applicative-eval` implements the ***eager approach in evaluation***. The eagerness is expressed by immediate evaluation of arguments, prior to closure application. An alternative algorithm, that implements the ***lazy approach in evaluation*** avoids argument evaluation until essential:

1. Needed for deciding a computation branch.
2. Needed for application of a primitive procedure.

The `normal-eval` algorithm is similar to `applicative-eval`. The only difference, which realizes the lazy approach, is moving the argument evaluation from step II.2.a. into step II.2.b., just before a primitive procedure is applied. Otherwise, the algorithm is unchanged, and the computation rules for the special operators are the same:

```
Signature: normal-eval(e)
Purpose: Evaluate a Scheme expression
Type: [<scheme-exp> -> Scheme-type]
Definition:
normal-eval[e] =
  I. atom?(e):
     1. number?(e) or boolean?( e) or empty-list?(e):
        normal-eval[e] = make-number(e) or make-boolean(e), or make-empty(e)
        ;a number or a boolean or an empty-list atom evaluates into a number or
         a boolean  or the empty list value, respectively
     2. variable?(e):
        a. If GE(e) is defined:
           normal-eval[e] = GE(e)
        b. Otherwise: e must be a variable denoting a Primitive procedure:
           normal-eval[e] = Scheme-value of e.
II. (composite? e): e = (e0 e1 ... en)(n >= 0):
    1. e0 is a Special Operator:
       normal-eval[e] is defined by the special evaluation rules of e0.
    2. a. Evaluate: compute normal-eval[e0] = e0'.
       b. (primitive-procedure? e0'):
          Evaluate: compute normal-eval[ei] = ei' for all ei.
          normal-eval[e] = system application e0'(e1', ..., en').
       c. (procedure? e0'):
          e0' is a closure: <Closure (x1 ... xn) b1 ... bm >:

           i.  Substitute (preceded by renaming):
               b1, ..., bm ∘{x1 = e1, ..., xn = en} = b1', ..., bm'

          ii. Reduce: normal-eval[b1'], ..., normal-eval[bm']
          iii. Return: normal-eval[e] = normal-eval[bm']
```

Note that `normal-eval` is defined only on Scheme expressions (as expected from an interpreter). Why?

Example 2.7. *The **Expansions** step for Example 2.2– In this step there are no evaluations, just replacements.*

```
normal-eval[ (f 5) ] ==>
      normal-eval[f] ==> <Closure (a) (sum-of-squares (+ a 1) (* a 2))>
==>
 normal-eval[ (sum-of-squares (+ 5 1) (* 5 2)) ] ==>
       normal-eval[ sum-of-squares ] ==>
                       <Closure (x y) (+ (square x) (square y))>
==>
 normal-eval[ (+  (square (+ 5 1))  (square (* 5 2))) ] ==>
        normal-eval[ + ] ==> <primitive-procedure +>
        normal-eval[ (square (+ 5 1)) ] ==>
               normal-eval[ square ] ==> <Closure (x) (* x x)>
               ==>
               normal-eval[ (* (+ 5 1) (+ 5 1)) ] ==>
                     normal-eval[ * ] ==> <primitiv-procedure *>
                     normal-eval[ (+ 5 1) ] ==>
                         normal-eval[ + ] ==> <primitive-procedure +>
                         normal-eval[ 5 ] ==> 5
                         normal-eval[ 1 ] ==> 1
                     ==> 6
```

```
                    normal-eval[ (+ 5 1) ] ==>
                        normal-eval[ + ] ==> <primitive-procedure +>
                        normal-eval[ 5 ] ==> 5
                        normal-eval[ 1 ] ==> 1
                    ==> 6
              ==> 36
        normal-eval[ (square (* 5 2)) ] ==>
              normal-eval[ square ] ==> <Closure (x) (* x x)>
              ==>
              normal-eval[ (* (* 5 2) (* 5 2)) ] ==>
                    normal-eval[ * ] ==> <primitiv-procedure *>
                    normal-eval[ (* 5 2) ] ==>
                        normal-eval[ * ] ==> <primitive-procedure *>
                        normal-eval[ 5 ] ==> 5
                        normal-eval[ 2 ] ==> 2
                    ==> 10
                    normal-eval[ (* 5 2) ] ==>
                        normal-eval[ * ] ==> <primitive-procedure *>
                        normal-eval[ 5 ] ==> 5
                        normal-eval[ 2 ] ==> 2
                    ==> 10
              ==> 100
==> 136
```

2.2.3 Comparison: The applicative order and the normal order of evaluations:

1. If both orders terminate (no infinite loop): They compute the same value.
2. Normal order evaluation repeats many computations.
3. Whenever applicative order evaluation terminates, normal order terminates as well.
4. There are expressions where normal order evaluation terminates, while applicative order does not:

   ```
   (define loop (lambda (x) (loop x)))
   (define g (lambda (x) 5))
   (g (loop 0))
   ```

   In normal order, the application `(loop 0)` is not evaluated. In applicative order: Better, do not try! Most interpreters use applicative-order evaluation.

5. ***Side effects*** (like printing – the `display` primitive in Scheme) can be used to identify the evaluation order. Consider, for example,

   ```
   > (define f (lambda (x) (display x) (newline) (+ x 1)))
   > (define g (lambda (x) 5))
   > (g (f 0))
   0
   5
   ```

   – What evaluation order was used?

   – What are the ***side effect*** and the ***result*** in the other evaluation order?

   Explain the results by applying both evaluation orders of the `eval` algorithm.

   The normal-order evaluation model uses the ***call-by-name*** method for ***parameter passing***: In procedure application, the actual arguments themselves are substituted for the formal parameters. This evaluation model is used in Scheme (LISP) for ***special forms***. The call by name method was first introduced in Algol-60.

2.2.4 Local Recursive Procedures

Recall the special operator `let`, which is a syntactic sugar for application of an anonymous `lambda`, i.e., runtime creation of a closure and its immediate application. Example 1.11 in Chapter 1, introduces a procedure for solving a polynomial function, using component substitution:

```
(define f
   (lambda ( x y)
     (let ((a (+ 1 (* x y)))
           (b (- 1 y)))
       (+ (* x (square a))
          (* y b)
          (* a b)))))
```

is actually the procedure:

```
(define f
   (lambda (x y)

     ((lambda (a b)
```

```
        (+ (* x (square a))
           (* y b)
           (* a b)))

      (+ 1 (* x y))
      (- 1 y))
   ))
```

Therefore:

```
applicative-eval[ (f 3 1) ] ==>*
```

Substitute: `<body of f>`∘`{x = 3, y = 1} ==>`

```
applicative-eval[ ((lambda (a b) (+ (* 3 (square a))
                                    (* 1 b)
                                    (* a b)))
                     (+ 1 (* 3 1))
                     (- 1 1)) ] ==>*
 applicative-eval[ (+ (* 3 16) (* 1 0) (* 4 0)) ] ==>
 48
```

The symbol `==>*` is used to denote application of several `applicative-eval` steps. Alternatively, we can define the internal procedure as a local procedure `f-xy`, and apply it to the components `(+ 1 (* x y))` and `(- 1 y)`:

```
(define f
  (lambda (x y)
     (let ( (f-xy (lambda (a b)
                        (+ (* x (square a))
                           (* y b)
                           (* a b)))
             ) )
        (f-xy (+ 1 (* x y)) (- 1 y)))
  ))

applicative-eval[ (f 3 1) ] ==>
      applicative-eval[ f ] ==> <Closure (x y) (let ...)>
      applicative-eval[ 3 ] ==> 3
      applicative-eval[ 1 ] ==> 1
```

Substitute: `<body of f>`∘`{x = 3, y = 1} ==>`

```
applicative-eval[ ( (lambda (f-xy) (f-xy (+ 1 (* 3 1)) (- 1 1)))
                    (lambda (a b)(+ (* 3 (square a))
                                    (* 1 b)
                                    (* a b))) ) ] ==>*
applicative-eval[ ( <Closure (a b)(+ (* 3 (square a))
                                    (* 1 b)
                                    (* a b)) >
                    (+ 1 (* 3 1))
                    (- 1 1)) ] ==>*
 applicative-eval[ (+ (* 3 16) (* 1 0) (* 4 0)) ] ==>*
 48
```

Indeed, since procedures are first class citizens in functional programming, we expect that local variables can have any Scheme value, be it a number value or a closure.

But, what happens if the closure computes a recursive function? Consider:

```
(define factorial
   (lambda (n)
     (let ( (iter (lambda (product counter)
                     (if (> counter n)
                         product
                         (iter (* counter product)
                               (+ counter 1))))
             ) )
        (iter 1 1))))
```

In order to clarify the binding relationships between declarations and variable occurrences we add numbering, that unifies a declaration with its bound variable occurrences:

```
(define factorial
   (lambda (n1)
     (let ( (iter2 (lambda (product3 counter3)
                     (if (> counter3 n1)
                       product3
                       (iter4 (* counter3 product3)
                              (+ counter3 1))))
             ) )
        (iter2 1 1))))
```

This analysis of declaration scopes and variable occurrences in these scopes clarifies the problem:

– The binding instance `n1` has the whole `lambda` body as its scope.

- The binding instance `iter2` has the `let` body as its scope.
- The binding instances `product3` and `counter3` have the body of the `iter` lambda as their scope.
- In the body of the `iter2 lambda` form:
  - The variable occurrences `n1, counter3, product3` are bound by the corresponding binding instances.
  - The variable occurrences `>, *, +` denote primitive procedures.
  - 1 is a number.
  - `if` denotes a special operator.
  - `iter4` is a ***free variable***!!!!! Causes a runtime error in evaluation:

```
applicative-eval[ (factorial 3) ] ==>*
```

Substitute: `<body of factorial>o{n = 3} ==>`

```
applicative-eval[ ( (lambda (iter) (iter 1 1))
                     (lambda (product counter)
                         (if (> counter 3)
                             product
                             (iter (* counter product)
                                   (+ counter 1))))
                    ) ] ==>*
applicative-eval[ ( <Closure (product counter)
                         (if (> counter 3)
                             product
                             (iter (* counter product)
                                   (+ counter 1))) >
                      1 1) ] ==>*
applicative-eval[ (if (> 1 3) 1 (iter (* 1 1) (+ 1 1))) ] ==>*
applicative-eval[ (iter (* 1 1) (+ 1 1)) ] ==>
 *** RUN-TIME-ERROR: variable iter undefined ***
```

The problem is that `iter` is a ***recursive procedure***: It applies the procedure `iter` which is not globally defined. `iter` is just a parameter that was substituted by another procedure. Once `iter` is substituted, its occurrence turns into a ***free variable***, that must be already bound when it is evaluated. But, unfortunately, it is not! This can be seen clearly, if we replace the `let` abbreviation by its meaning expression:

```
1. (define factorial
2.   (lambda (n)
3.     ( (lambda (iter) (iter 1 1))
4.       (lambda (product counter)
5.           (if (> counter n)
6.               product
7.               (iter (* counter product)
8.                     (+ counter 1))))
9.     )))

> (factorial 3)
reference to undefined identifier: iter
```

We see that the occurrence of `iter` in line 3 is indeed bound by the `lambda` parameter, which is a declaration. Therefore, this occurrence of `iter` is replaced at the ***substitute*** step in the evaluation. But the occurrence of `iter` on line 7 is not within the scope of any `iter` declaration, and therefore is ***free, and is not replaced***!

Indeed, the substitution model cannot simply compute recursive functions. For globally defined procedures like `factorial` it works well because we have strengthened the lambda calculus substitution model with the ***global environment*** mapping. But, that does not exist for local procedures.
Therefore:

> For local recursive functions there is a special operator `letrec`, similar to `let`, and used only for local procedure (function) definitions. It's syntax is the same as that of `let`. But, the scope of the local procedure declarations **includes all of the initialization procedures!**

```
(define factorial
  (lambda (n)
     (letrec ( (iter (lambda (product counter)
                        (if (> counter n)
                            product
                            (iter (* counter product)
                                  (+ counter 1))))
                ) )
        (iter 1 1))
  ))
```

Usage agreement: The special operator `let` is used for introducing ***non Procedure*** local variables, while the special operator `letrec` is used for introducing ***Procedure*** local variables. Note that the substitution model presented above ***does not*** account for local recursive procedures.

In lambda calculus, which is the basis for functional programming, recursive functions are computed with the help of ***fixed point operators***. A recursive procedure is ***rewritten***, using a fixed point operator, in a way that defines it as a recursive procedures.

2.2.4.1 Side comment: Fixed point operators

Based on excerpt from Wikipedia:
A ***fixed point combinator*** (or fixed-point operator) is a higher-order function which computes fixed points of functions. This operation is relevant in programming language theory because it allows the implementation of recursion in the form of a rewrite rule, without explicit support from the language's runtime engine. A fixed point of a function f is a value x such that $f(x) = x$. For example, 0 and 1 are fixed points of the function $f(x) = x^2$, because $0^2 = 0$ and $1^2 = 1$.

Recursive functions can be viewed as high order functions, that take a function parameter as well as other arguments. Viewed this way, the intended semantics is to find a function argument, which is equal to the original definition, so that recursive calls use the same function. This is achieved by replacing the definition of f (its `lambda` expression) by an application of a ***fixed point operator*** to the `lambda` expression that defines f.
What is a fixed-point of a high order function?
Whereas a fixed-point of a first-order function (a function on "simple" values such as integers) is a first-order value, a fixed point of a higher-order function F is another function `f-fix` such that

```
F(F-fix) = F-fix.
```

A fixed point operator is a function `FIX` which produces such a fixed point `f-fix` for any function `F`:

```
FIX(F) = F-fix.
```

Therefore: `F( FIX(F) ) = FIX(F).`

Fixed point combinators allow the definition of anonymous recursive functions (see the example below). Somewhat surprisingly, they can be defined with non-recursive lambda abstractions.

Example 2.8. *Consider the factorial function:*

```
factorial(n) = (lambda (n)(if (= n 0) 1 (* n (factorial (- n 1)))))
```

Function abstraction:

```
F = (lambda (f)
        (lambda (n)(if (= n 0) 1 (* n (f (- n 1))))))
```

Note that `F` is a high-order procedure, which accepts an integer procedure argument, and returns an integer procedure. That is, `F` creates integer procedures, based on its integer procedure argument. For example, `F( +1 ), F( square ), F( cube )` are three procedures, created by `F`.

The question is: Which argument procedure to `F` obtains the intended meaning of recursion? We show that `FIX(F)` has the property that it is equal to the body of `F`, with recursive calls to `FIX(F)`. That is:
`FIX(F) = (lambda (n)(if (= n 0) 1 (* n (FIX(F) (- n 1)))))`
which is the intended meaning of recursion.
We know that: `FIX(F) = F(FIX(F))`. Therefore:

```
FIX(F)      ==>
F( FIX(F) ) ==>
( (lambda (f)
        (lambda (n)(if (= n 0) 1 (* n (f (- n 1))))))
   FIX(F) )  ==>
(lambda (n)(if (= n 0) 1 (* n (FIX(F) (- n 1)))))
```

That is, `FIX(F)` performs the recursive step. Hence, in order to obtain the recursive factorial procedure we can replace all occurrences of factorial by `FIX(F)`, where `FIX` is a fixed point operator, and `F` is the above expression.
An example of an application:

```
FIX(F) (1)       ==>
F( FIX(F) ) (1)  ==>
( ( (lambda (f)
        (lambda (n)(if (= n 0) 1 (* n (f (- n 1))))))
     FIX(F) )
   1)             ==>
( (lambda (n)(if (= n 0) 1 (* n (FIX(F) (- n 1)))))
  1)             ==>
(if (= 1 0) 1 (* 1 (FIX(F) (- 1 1))))  ==>
(* 1 (FIX(F) (- 1 1)))                 ==>
(* 1 (FIX(F) 0))                       ==>* (skipping steps)
(* 1 (if (= 0 0) 1 (* 0 (FIX(F) (- 0 1)))))  ==>
(* 1 1)                                ==> 1
```

We see that `FIX(F)` functions exactly like the intended factorial function. Therefore, factorial can be viewed as an abbreviation for `FIX(F)`:

```
(define factorial (FIX F))
```

where `F` is the high-order procedure above.

Scheme applications saves us the need to explicitly use fixed point operators in order to define local recursive procedures. The `letrec` special operator is used instead.

Summary:

- Local procedures are defined with `letrec`.
- Local non-procedure variables are defined with `let`.
- The substitution model (without using fixed-point operators) cannot compute local (internal) recursive procedures. The meaning of the `letrec` special operator can be defined within an imperative operational semantics (accounts for changing variable ***state***s). Within the substitution model, we have only an intuitive understanding of the semantics of local recursive procedures (`letrec` works by "magic").

2.2.5 Partial evaluation using Function Currying

Currying is a technique for turning a function (procedure) of n parameters into a chain of n single parameter procedures. It is called after Haskell Curry. It is used when parameters are not all available, and fixing part of them enables ***partial evaluation*** of a procedure code. Currying turns a procedure `(lambda (x1 x2 ... xn) body)` into a chain of high order procedures `(lambda (x1) (lambda (x2) ... (lambda (xn) body)...))`. The Currying technique is used in every language that supports closures, i.e., run-time created procedures: All functional languages, Javascript, C#.

Example 2.9.

```
Signature: add(x,y)
Type: [Number*Number -> Number]
(define add
  (lambda (x y) (+ x y)))

Can be Curried into:
Signature: c-add(x)
Type: [Number -> [Number -> Number]]
(define c-add
  (lambda (x)
    (lambda (y) (add x y))))

(define add3 (c-add 3))
> (add3 4)
7
```

Example 2.10. *Recall the `sum` procedure for summing finite sequences:*

```
Signature: sum(term a next b)
Type: [[Number -> Number]*Number*[Number -> number]*Number ->Number]
```

```
(define sum
   (lambda (term a next b)
     (if (> a b)
       0
       (+ (term a)
          (sum term (next a) next b)))))
```

For a given mathematical sequence, the procedures `term, next` are fixed and maybe frequently used for varying intervals. In that case, `sum` can be Curried into `c-sum` and reused for multiple interval boundaries:

```
Signature: c-sum(term)
Type: [[Number -> Number] -> [[Number -> number] -> [Number*Number ->Number]]]
(define c-sum
  (lambda (term)
    (lambda (next)
      (lambda (a b)
        (sum term a next b)))))
For the sequence of squares with skips of 2:
```

$\langle a^2, (a+2)^2, \ldots \rangle$:

```
(define sum-sqrts-skip2 ((c-sum (lambda (x)(* x x)))
                         (lambda (x) (+ x 2)) ))
> (sum-sqrts-skip2 2 6)
56
```

Further generalization can involve general Currying procedures:

```
Signature: curry2(f)
Type: [[T1*T2 -> T3] -> [T1 -> [T2  -> T3]]]
(define curry2
  (lambda (f)
    (lambda (x)
      (lambda (y) (f x y)))))

> (((curry2 add) 3) 4)
7
(define add3 ((curry2 add) 3))
> (add3 4)
7
> (add3 7)
10
```

Example 2.11 (Naive Currying).

Consider a more complex version of the `add` procedure, where a complex computation, say the recursive `fib` procedure, is applied to the available (non-delayed) parameter:

```
Signature: add-fib(x,y)
Type: [Number*Number -> Number]
(define add-fib
  (lambda (x y)
    (+ (fib x) y)))
> (add-fib 4 5)
10

Naive Currying:
Signature: c-add-fib(x)
Type: [Number -> [Number -> Number]]
(define c-add-fib
  (lambda (x)
     (lambda (y)
       (+ (fib x) y))))

(define add-fib4 (c-add-fib 4))
> (add-fib4 5)
10
> (add-fib4 3)
8
> (add-fib4 5)
10
```

This Currying is naive because it does not meet the expectations of partial evaluation: In every application of `add-fib4`, `(fib 4)` is re-computed, although we could have computed it once at the time of Currying, i.e., before the definition of `add-fib4`.

The following version demonstrates the full idea of Currying: ***Delaying*** computations in unknown variables, while ***applying*** computations in known variables.

```
(define c-add-fib
  (lambda (x)
    (let ((fib-x  (fib x)))   ;Applying known computation
      (lambda (y)             ;delaying unknown computations
        (+ fib-x y)))
    ))
```

Using the new version, `(fib 4)` is computed once, while `add-fib-4` is defined.

Example 2.12. *Consider the recursive procedure* ***expt****:*

```
Signature: expt(x,y)
Type: [Number*Number -> Number]
(define expt
  (lambda (b n)
     (if (= n 0)
         1
         (* b (expt b (- n 1))))))

Naive Currying:
Signature: c-expt(x,y)
Type: [Number -> [Number -> Number]]
(define c-expt
  (lambda (b)
    (lambda (n)
      (expt b n))))

> ((c-expt 3) 4)
81

Another naive Currying:
(define c-expt
  (lambda (b)
    (lambda (n)
      (if (= n 0)
          1
          (* b ((c-expt b) (- n 1)) )))
      ))
```

Both Curried versions are naive since the Curried procedures still need to apply computations or to handle substitutions of the known variable `b`. In order to cope with the recursion, we define a local recursive procedure that resides in the scope of the known variable and therefore does not need it as a parameter:

```
(define c-expt
  (lambda (b)
    (letrec ((helper (lambda (n)
                       (if (= n 0)
                           1
                           (* b (helper (- n 1)))))
                     ))
```

```
      helper
      )))

(define expt3 (c_expt 3) )
> (expt3 4)
81
```

`expt3` is a procedure in the single delayed variable `n`, in which all occurrences of the known variable `b` are already substituted.

Example 2.13.

If `expt` would have included some complex computation in the known variable `b`, it would have been computed prior to the definition of the internal recursive procedure `helper`:

```
> (define expt'
      (lambda (b n)
        (if (even? (fib b))
            (if (= n 0)
                1
                (* b (expt b (- n 1))))
            0)
        ))

Curring:
> (define c-expt'
      (lambda (b)
        (let ((fib-b (fib b)))
          (letrec ((helper (lambda (n)
                             (if (even? fib-b)
                                 (if (= n 0)
                                     1
                                     (* b (helper (- n 1))))
                                 0)
                             )))
            helper
          ))
        ))
```

2.2.6 Revisiting: Primitive procedures, User-defined procedures, Derived expressions (Syntactic sugar), Special operators

With the operational semantics formally defined, and the experience gained so far, it is time to revisit the four different ways to implement functions using procedures. We demonstrate it via an example, and emphasize the requirements and advantages/disadvantages of each.

The example involves the `or` logic operator which is implemented in most programming languages (including Scheme). The syntax of `or` expressions in Scheme is:

```
(or <expression>*)  zero or more expressions.
> (or)
#f
> (or (= 1 2) #f #t (= y 4))
#t
```

In order to optimize the performance, most implementations use an optimized operational semantics that evaluates just as much as needed for determining the resulting value.
The Scheme optimized evaluation rule for `or`:

1. Evaluate the arguments from left to right.
2. If an argument evaluates to a non `#f` value, stop. The value of this argument is the value of the `or` expression.
3. Otherwise, the value of the `or` expression is `#f`.

The four alternatives for implementing `or`:

1. **Primitive procedure:** But following the substitution model (applicative or normal), all sub-expressions are evaluated before the the whole `or` expression is evaluated (as in `+, map` expressions). Therefore, this option is not appropriate for implementing the `or` operator using the optimized evaluation rule.
2. **User-defined procedure:** For a binary `or`, the implementation can be:

```
> (define user-or
    (lambda (first second)
        (if (eq? first #f)
            second
            first)))

> user-or
#<procedure: user-or>
> (user-or #f (> 2 3)
#f
> (user-or 1 2)
1
> (user-or #t (fib 250))
#t
```

Does this user-defined procedure provides the desired operational semantics?
The answer depends on the version of the Substitution model we use:

(a) **Applicative order:** The optimized evaluation rule is not implemented, since all arguments are evaluated before the whole `or` expression is evaluated.

(b) **Normal order:** The optimized evaluation rule is implemented, since the `if` operator does not evaluate the alternative, if the consequence is not false.

3. **Derived expression (syntactic sugar):** The `or` operator is translated in pre-processing time, before evaluation starts (like the macro character) into an alternative Scheme code. We consider three options for a binary `or`, showing how name clash can be prevented by creating separate lexical scopes (using lambda abstraction).

   (a) **Try 1:** Translate an `or` expression into a conditional expression:

   ```
   (or exp1 exp2) ==>
   (if exp1 exp1 exp2)
   ```

   For example,

   ```
   (or (> (fib 250) 300) (> y 1)) ==>
   (if (> (fib 250) 300)
       (> (fib 250) 300)
       (> y 1))
   ```

   Problem: Repeated evaluation of the first expression. In addition to inefficiency, if there are side effects (e.g., printouts), or in the context of imperative programming, repetition might change the results and effects of evaluation.

   (b) **Try 2:** Evaluate the first expression just once, and use a local variable to refer to the returned value:

   ```
   (or exp1 exp2) ==>
   (let ((test1 exp1))
      (if test11 test11 exp2))
   ```

   For example,

   ```
   (or (> (fib 250) 300) (> y 1)) ==>
   (let ((test1 (> (fib 250) 300)))
      (if test1
          test1
          (> y 1)))
   ```

Problem: The `let` operator defines a lexical scope for the `body`. In this scope, all references to the newly defined local variable `test1` refer to the `let` variable. But what if the arguments of the `or` expression also include the variable `test1` as a ***free variable***? Then their evaluation is modified to refer to the local variable instead. For example:

```
> (define test1 #t)
> (or (= 1 2) test1)
#t
while
> (let ((test1 (= 1 2)))
     (if test1
         test1
         test1))
#f
```

The local `test1` variable overrides the declaration in the surrounding scope.

(c) **Try 3:** Insert a separate lexical scope to prevent name clashes. In this case, the local variable might have a clash with free variables in the second argument. Therefore, a new lexical scope is inserted for the second expression – using lambda abstraction:

```
(or exp1 exp2) ==>
(let ((test1 exp1)
      (thunk (lambda () exp2)))
   (if test1 test11 (thunk)))
```

For the last example,

```
> (define test1 #t)
> (or (= 1 2) test1)
#t
and indeed
> (let ((test1 (= 1 2))
   (thunk (lambda () test1)))
     (if test1
         test1
         (thunk)))
#f
```

and another example,

```
> (define test1 #t)
> (define thunk (lambda () #t))
> (or (= 1 2) (thunk))
#t
and indeed
> (let ((test1 (= 1 2))
        (thunk (lambda() (thunk))))
     (if test1
         test1
         (thunk)))
#t
```

There is no clash between the local `thunk` variable to the free occurrence of this variable in the second argument, which refers to the global declaration. The clash is prevented due to the scope separation.
Recall that we have already used lambda abstraction for ***delaying*** computation. Here we see another use of lambda abstraction: for separating ***lexical scopes***.

4. **Special operator:** The `or` operator is inserted to the language as a special operator. That is, the concrete and abstract syntax is extended to include `or` expressions, and the operational semantics is extended with the special evaluation rule. This option is inferior to the derived expression one.

2.2.7 Formalizing tail recursion – analysis of expressions that create iterative processes

Based on the operational semantics, language expressions can be analyzed so to identify whether they create recursive or iterative processes. This analysis can be formalized (and automated), so that an expression can be ***proved*** to create iterative processes. Having an iterative process automated identifier, a compiler (interpreter) can be ***tail recursive***, i.e., can identify iterative expressions and evaluate them using bounded space.

Head and tail positions: The tail recursion analysis starts with the concepts of ***head*** and ***tail positions***, which characterize positions in expressions where user procedures can be called without affecting the iterative nature of the processes that the expression creates. Tail positions are positions whose evaluations is the last to occur. Head positions are all other positions. Head positions are marked `H` and tail positions are marked `T`:

```
1. (<PRIMITIVE> H ... H)
2. (define var H)
3. (if H T T)
```

```
4. (lambda (var1 ... varn) H ... T)
5. (let ( (var1 H) ...) H ...T)
6. Application: (H ... H)
```

Example 2.14. *(define x (let ((a (+ 2 3)) (b 5)) (f a b)))*

- The `let` sub-expression is in head position.
- The `(+ 2 3)` and 5 sub-expressions of the `let` expressions are in head positions.
- The `(f a b)` sub-expression of the `let` expression is in tail position.
- The 2 and 3 sub-expressions of `(+ 2 3)` are in head positions.
- The `f, a, b` sub-expressions of `(f a b)` are in head positions.

An expression is in ***tail form*** if its head positions do not include calls to user procedures, and its sub-expressions are in tail form. By default, atomic expressions are in tail form.

Example 2.15.

- `(+ 1 x)` is in tail form.
- `(if p x (+ 1 (+ 1 x)))` is in tail form.
- `(f (+ x y))` is in tail form.
- `(+ 1 (f x))` is not in tail form (but `(f x)` is in tail form).
- `(if p x (f (- x 1)))` is in tail form.
- `(if (f x) x (f (- x 1)))` is not in tail form.
- `(lambda (x) (f x))` is in tail form.
- `(lambda (x) (+ 1 (f x)))` is not in tail form.
- `(lambda (x) (g (f 5)))` is not in tail form.

Proposition 2.2.1. Expressions in tail form create iterative processes.

2.3 Types

The evaluation of a language expression results a ***value***. The set of all returned values is the set of ***computed values*** of a language. This set is usually divided into subsets, called ***types***, on which common operations are defined. That is, a ***type*** is a distinguished ***set of values*** on which common operations are defined. The theory of types of a programming language associates a type $\mathcal{T}$ with the set of language expressions whose evaluations yield values in $\mathcal{T}$. That is:

```
Language expressions  ---- Evaluation algorithm ---->  Types (sets of values)
     (syntax)                                                (semantics)
```

The theory provides methods and techniques for identifying, for every language expression, what is the type (or types) of the values that result from its evaluation(s).
Syntax notions: *expressions, variables, symbols, forms*
Semantic notions: *types, values*

The purpose of type management in programming languages is to prevent unfeasible computations, i.e., computations that cannot be completed due to improper applications of procedures to values. For example, prevent:

```
> (+ ((lambda (x) x) (lambda (x) x)) 4)
+: expects type <number> as 1st argument, given: #<procedure:x>; other
arguments were: 4
```

Language expressions (syntax) are assigned a type (semantics), so that well typing rules can be checked. For example, the above expression is not well typed, since that type of the `+` primitive procedure is `[number*Number -> Number]`, while the types of the given arguments were `[T -> T]` and `Number`.

But, can we guarantee that expressions are well typed? Consider:

```
> (define x 4)
> (+ 3
     (if (> x 0)
         (+ x 1)
         "non-positive value"))
5
```

and

```
> (define x 0)
> (+ 3
     (if (> x 0)
         (+ x 1)
         "non-positive value"))
```

```
+: expects type <number> as 2nd argument, given: "non-positive value";
other arguments were: 3
```

What happened? The expression `(if (> x 0) (+ x 1) ‘‘non-positive value’’)` is not well typed. Depending on the runtime value of the variable `x`, it evaluates either to a number or to a string. Such expressions might cause runetime errors when combined with other operations.

Programming languages that enforce type checking at ***static time*** (before runtime), usually prohibit conditional expressions that might evaluate to different types. In Scheme (Dr. Racket), since types are checked at runetime, such conditionals are allowed, but should be handled with much care.

Typing information is not part of Scheme syntax. However, all primitive procedures have predefined types for their arguments and result. The Scheme interpreter checks type correctness at run-time: ***dynamic typing***.

Type classification: Primitive, atomic, composite types:
Primitive types are types already built in the language. That is, the language can compute their values and provides operations for operating on them. All types in the Scheme subset that we study are primitive, i.e., the user cannot add new types (unlike in ML or Java).
Atomic and Composite types: Atomic types are sets of atomic values, i.e., values that are not decomposable into values of other types. Composite types are sets of values that are constructed from values of other types.

Standard specification of a type: The standard specification includes information about the values in the type, language expressions that return these values, ***value constructors***, operations for identification (***identification predicate***), comparison (***equality predicate***), and management:

1. **Values:** The set of values of the type
2. **Value constructors:** Operations that construct values in the type. For atomic types, every value is a constructor. For composite types, the value constructors are functions from the component types to the specified type.
3. **Identifying predicates:** Identify the values of the type
4. **Equality predicates:** Compare values of the type
5. **Operations:** Operate on values of the type

2.3.1 Atomic Types

Scheme supports several primitive ***atomic types***, of which we discuss only ***Number***, ***Boolean*** and ***Symbol***.

The Number Type: Scheme allows number type overloading: integer and real expressions can be combined freely in Scheme expressions. We consider a single Number type. Values of the Number type are denoted by number symbols:

> Scheme does not distinguish between the ***syntactic number symbol*** to its ***semantic number value***: The number symbol stands both for the syntactic sign and its number value.

The identifying predicate of the Number type is `number?`, the equality predicate is the primitive procedure `=`, and the operations are the regular arithmetic primitive operations and relations. Each number symbol is evaluated to the number it stands for, i,e., each number symbol is a value constructor for a single Number value (there is infinity of value constructors).

The Boolean Type: The Boolean type is a 2 valued set {#t, #f}, with the identifying predicate `boolean?`, the equality predicate `eq?`, and the regular boolean connectives (`and, or, not`, and more) as operations (note that `and, or` are special operators since they optimize their argument evaluation). The values of the Boolean type (semantics) are syntactically denoted by 2 Scheme symbols: `#t, #f`. `#t` denotes the value #t, and `#f` denotes the value #f. Therefore, there are two value constructors: `#t, #f`.

> Scheme does not distinguish between the ***syntactic boolean symbol*** to its ***semantic boolean value***.

The Symbol Type: The values of the Symbol type are **atomic names**, i.e., (unbreakable) sequences of keyboard characters (to distinguish from strings). `quote` is the value constructor of the Symbol type (with `'` as a macro character). Its parameter is any sequence of keyboard characters. The Symbol type identifying predicate is `symbol?` and its equality predicate is `eq?`. The Symbol type is a degenerate type, i.e., there are no operations defined on Symbol types.

2.3.2 Composite Types

A composite type is a type whose values are composed from values of other types (including self values, in the case of a recursive type). The values of a composite type can be decomposed into values of other (or self) types. For example, the rational numbers can be viewed as the composite ***Rational-number*** type, whose values are constructed, each, from two integer values (and therefore, can be decomposed into their nominator and denominator integer components).

In the Scheme subset used in this course, the composite types are ***Procedure, Pair*** and ***List***. As already said above, there are no user defined types. For example, we cannot define a new Rational-number type, but rather, decide how to represent rational numbers using values of the primitive types, e.g., as pairs of integers.

Type and value constructors: In order to describe a composite type we need to

1. specify the **types** from which its values are constructed;
2. describe how **values** of the component types are composed to form a new value of the defined type.

The first need is provided by a ***type constructor***. The second need is provided by ***value constructors***. That is, a type constructor is a function that takes type arguments (sets) and returns a new set – the defined composite type. A value constructor is a function that takes value arguments and returns a value in the type of the value constructor.

1. **Procedure types:** The type constructor for procedure types is `->`. It is used for all procedure arities. $[type_1 * \ldots * type_n \to type]$ $(n \geq 1)$ denotes the type (set) of all n-ary procedures, that take n arguments from types $type_1, \ldots, type_n$, and return a value in type $type$. $[Empty \to type]$ is the type of all parameter-less procedures that return a value in type $type$. That is, `->` is a $(n+1)$-ary $n \geq 0$ function:

   $$\texttt{->}\text{: } \times_0^{n+1} TYPES \to Procedure$$
   $$\texttt{->}(type_1, \ldots, type_n, type) = [type_1 * \ldots * type_n \to type] \quad n \geq 1$$
   $$\texttt{->}(type) = [Empty \to type]$$

   where $TYPES$ is the set of all types (including procedure types), and $Procedure$ is the set of all procedure types.

   The value constructor for all procedure types is `lambda`. It takes as arguments a list of n variables $(n \geq 0)$, and a body that consists of m expressions $(m \geq 1)$. It constructs a procedure value in a procedure type $[type_1 * \ldots * type_n - > type]$ $(n \geq 1)$, or in $[Empty - > type]$ $(n = 0)$. That is, `lambda` is the function:

`lambda`:	n-variables-list × m-expression-list → $[type_1 * \ldots * type_n \to type]$	$n \geq 1$
`lambda`:	empty-variable-list × m-expression-list → $[Empty \to type]$	$n = 0$

   The procedure value (closure) that is constructed by application of the value constructor `lambda` is determined by the operational-semantics algorithm. The challenge of type inference systems is to identify the type of this procedure value (closure) based on:

   (a) the length of the variable list
   (b) the syntax of the expressions in the procedure body
   (c) the rules of the evaluation algorithm

2. **Pair types:** The type constructor for pair types is `Pair`. It takes two type arguments $type_1, type_2$ and returns a pair type $Pair(type_1, type_2)$:

   $$Pair\colon\ TYPES \times TYPES \rightarrow Pair$$

   where $TYPES$ is the set of all types, and $Pair$ is the set of all pair types. The pair type returned by application of $Pair$ to types $type_1, type_2$ is denoted $Pair(type_1, type_2)$. `Pair(Number,Number)` denotes the type (set) of all number pairs, `Pair(Number,[T -> Boolean])` denotes the type of all pairs of a number and a boolean 1-ary procedure, and `Pair(Number,T)` is a polymorphic pair type denoting all pair types whose first component is Number.

   The value constructor of pair types is `cons`.

   $$\texttt{cons}\colon\ type_1 \times type_2 \rightarrow Pair(type_1, type_2)$$

3. **List types:** The type constructor for list types is `List`. For homogeneous lists It takes a single type argument *type* and returns a list type $List(type)$:

   $$List\colon\ TYPES \rightarrow Homogeneous\_list$$

   where $TYPES$ is the set of all types, and $Homogeneous\_list$ is the set of all homogeneous list types. For heterogeneous lists, there is a single type, denoted `List`: The type constructor `List` does not take any type parameters. The homogeneous list type returned by application of *List* to type *type* is denoted $List(type)$.

   The value constructor of list types is `cons`.

Homogeneous lists:	$\texttt{cons}\colon\ type_1 \times List(type_1) \rightarrow List(type_1)$
Heterogeneous lists:	$\texttt{cons}\colon\ type_1 \times List \rightarrow List$

Type inference systems combine the three aspects of programming language: **syntax, evaluation rules, types**.

The type of a language expression: The evaluation of a language expression yields a value. The type to which this value belongs is the type of the expression:

```
Language-expression  ----evaluation---->  Value ----belongs-to----> Type
type-of(exp) = type-of(value-of(exp))
```

In case that different evaluations of an expression return values in different types we say that the expression is ***polymorphic***. We have introduced ***type variables*** to the type language in order to denote the type of such expressions. type expressions that include type variables are termed polymorphic. Hence, the type of a p polymorphic expression is denoted by a polymorphic type expression.

The type constructors `->, Pair, list` are polymorphic: They can be applied to polymorphic type expressions and can return a polymorphic type expression. The "amount of polymorphism" in the type of an expression is determined by the "amount" of primitives occurring in it. For example,

- The type of `(lambda (x) (+ (* x x) 3))` is `[Number -> Number]`. This is determined from the known types of the primitive procedures `+, *`.
- The type of `(lambda (x y) (not (and x y)))` is `[T1*T2 -> Boolean]`. This is determined from the known types of the primitive procedure `not` and the special operator `and`[1].
- The types of `(lambda (x) x)` (which creates the ***identity*** closure) and `(lambda (f x) (f (f x)))` (which creates the closure ⟨`closure (f x) (f (f x))`⟩ are `[T -> T]` and `[[T -> T]*T -> T]`, respectively.

Summary: Types of computed values

1. **Atomic types:**
   - Number:
     (a) Includes integers and reals.
     (b) Value constructors: All number symbols.
     (c) Identifying predicate: `number?`
     Equality predicate: `=`
     Operations: Primitive arithmetic operations and relations.
   - Boolean:
     (a) A set of two values: #t, #f.
     (b) Value constructors: The symbols `#t, #f`.
     (c) Identifying predicate: `boolean?`
     Equality predicate: `eq?`
     Operations: Primitive propositional connectives (note that `and, or` are special operators in Scheme).
   - Symbol:
     (a) The set of all unbreakable keyboard character sequences. Includes variable names.
     (b) Value constructor: `quote`

[1]The `T1, T2` type parameters are due to the lack of real Boolean type in Scehme. In other languages the type of this procedure would be `[Boolean*Boolean -> Boolean]`.

(c) Identifying predicate: `symbol?`
Equality predicate: `eq?`
Operations: None (a degenerated type).

2. **Composite types:**

   – Procedure:

     (a) A polymorphic type: The collection of all procedure (function) types. Each ***concrete*** procedure type includes all procedures with the same argument and result types.
     (b) Type constructor: `->` written in infix notation. A Procedure type expression has the syntax: `[ <Type-expression-tuple> -> Type-expression]`. Type expressions can be polymorphic, i.e., include ***type variables***.
     (c) Value constructor:
     For primitive procedures: We do not create them, and therefore, we do not have value constructors for them (we just have names to apply them).
     For user defined procedures (closures), `lambda` is the value constructor.
     (d) Identifying predicates:
     For primitive procedures: `primitive?` For User defined procedures (closures): `procedure?`
     Equality predicate: none.(why?)
     Operations: All high order procedures (primitive and user defined).

   – Pair:

     (a) Type constructor: `Pair` – a polymorphic type constructor, that takes 2 type expressions as arguments.
     (b) Value constructor:
     `cons` – Can take any values of a Scheme type.
     `Type of cons:  [T1*T2 -> Pair(T1,T2)]`
     (c) Getters: `car`, `cdr`.
     `Type of car:  [Pair(T1,T2) -> T1]` `Type of cdr:  [Pair(T1,T2) -> T2]`
     (d) Predicates: `pair?, equal?`
     `Type of pair?:  [T -> Boolean]`
     `Type of equal?:  [T1*T2 -> Boolean]`

   – List:

     (a) Type constructors: For homogeneous lists, `List(T)` – a polymorphic type constructor that takes a single type expression as argument; for heterogeneous lists, `List` – that takes no parameters.
     (b) Value constructors: `cons, list`.
       – Type of `list:  [T1*...*Tn -> List]`, and also `[T*...*T -> List(T)]`, $n \geq 0$.

  – Type of `cons`: `[T*List -> List]`, and also `[T*List(T) -> List(T)]`.

(c) **Selectors:** `car, cdr`.
Type of `car`: `[List -> T]` and also `[List(T) -> T]`.
Type of `cdr`: `[List -> List]` and also `[List(T) -> List(T)]`.
Preconditions for `(car list)` and `(cdr list)`: `list != ()`.

(d) **Predicates:** `list?, null?, equal?`
Type of `list?`: `[T -> Boolean]`
Type of `null?`: `[T -> Boolean]`
Type of `equal?`: `[T1*T2 -> Boolean]`

2.3.3 A Type Language for Scheme

Writing procedure contracts and checking or inferring the type of language expressions requires a ***type language***, i.e., a language for specification of types. The Scheme types introduced so far include the atomic types Number, Boolean, Symbol, Void, and the composite types Procedure, Pair and List (with two variants: Homogeneous and Heterogeneous).

Union types: We have seen that conditional expressions that can return values of different types require the addition of the `union` operation on types. For simplicity, the formal type language introduced below does not include union types (which means that it cannot describe the type of conditionals with different type values).

The *Tuple* type: In order to simplify the specification of the type of procedures with multiple parameters, we introduce an additional composite type: Tuples. The set of all pairs of values forms the type of 2-tuples, the set of all triplets of values forms the type of 3-tuple, and the set of all n-tuples of values forms the type of n-tuples. The set of 0-ary tuples is the empty set type Empty.

The type constructor of Tuple types is `*` and it can take any number of type arguments.

$$
\begin{aligned}
&*\text{: } \times_1^n TYPES \rightarrow Tuple \quad n \geq 0\\
&*(type_1, \ldots, type_n) = type_1 * \ldots * type_n \quad n \geq 1\\
&*() = Empty \quad n = 0
\end{aligned}
$$

where $Tuple$ is the set of all tuple types. The type expression `Number*Number*Boolean` describes the type of all triplets of 2 numbers and a boolean, e.g., $\langle 2, 7.4, \#t\rangle$, $\langle 0, -63, \#f\rangle$. The type expression `Symbol*[Number -> Number]` describes the type of all pairs of a symbol and a single variable procedure from numbers to numbers.

Tuple type expressions are used in the formal specification of the input types of procedures. For example the procedure type-expression `[Number*T -> Number]` is considered as consisting from the tuple type-expression `Number*T` and the type `Number`. Under this view, the Procedure type constructor `->` takes 2 arguments: A tuple type-expression for the input parameters and an output type-expression.

We introduce tuple types as ***degenerate*** types: There are no value constructors for tuples, and language expressions do not evaluate to tuples. Procedures do not take tuples as arguments and do not return tuples as their output values. Indeed, Pair types and 2-tuple types describe the same sets of values, but no confusion arises since the Scheme subset that we use does not manipulate tuple values.

For the purpose of static type inference, the Scheme language is restricted for conditional expressions with a single type for all consequences. Therefore, the Union type can be removed.

BNF grammar of the type language for the Scheme subset introduced so far:

```
Type ->  'Void' | Non-void
Non-void -> Atomic | Composite | Type-variable
Atomic -> 'Number' | 'Boolean' | 'Symbol'
Composite -> Procedure | Pair | List
Procedure -> '[' Tuple '->' Type ']'
Tuple -> (Non-void '*' )* Non-void | 'Empty'
Pair -> 'Pair' '(' Non-void ',' Non-void ')'
List -> 'List' '(' Non-void ')' | 'List'
Type-variable -> A symbol starting with an upper case letter
```

Note that:

1. No procedure type expression has `Void` for its parameter types.
2. No procedure type expression has `Empty` as its return type.

Specification of Recursive types

The *Church numbers* type is defined inductively as follows: `Zero` is a Church number, and for a Church number `c`, `S(c)` is also a Church number. Church numbers provide ***symbolic representation*** for the natural numbers, where the numbers `0, 1, 2, 3, ...` are represented by the Church numbers `Zero, S(Zero), S(S(Zero)), S(S(S(Zero))), ... .`

We can represent this set by the set of Scheme expressions `'Zero, (lambda () 'Zero), (lambda () (lambda () 'Zero)), (lambda () (lambda () (lambda () 'Zero))), ...`, which can represent the natural numbers via the following procedures, that provide `1:1` mappings between this set and the set of natural numbers:

```
(define church-num
  (lambda (n)
    (if (= n 0)
        'Zero
        (lambda () (church-num (- n 1))))
    ))

(define church-val
  (lambda (church-num)
    (if (eq? church-num 'Zero)
        0
        (+ 1 (church-val (church-num))))
    ))
```

What are the contracts for these procedures? We can try:

```
Signature: church-num(n)
Type: [Number -> Procedure union Number]

Signature: church-val(church-num)
Type: [Procedure union Number -> Number]
```

But this type misses the main purpose of type specification: ***Type safety***, i.e., preventing runtime type errors. Furthermore, no pre-condition can fix the problem, as it requires an inductive characterization of the input values.

A precise type information requires a recursive definition of the set of Church numbers:
`Church-num = 'Zero union [Empty -> Church-num]`
Then, we could have the types:

```
Signature: church-num(n)
Type: [Number -> Church-num]

Signature: church-val(church-num)
Type: [Church-num -> Number]
```

The type language introduced above does not support ***user-defined types***. In particular, it does not enable definition of ***recursive types***. Therefore, we cannot write correct contracts for procedures that operate only on recursive types. In chapter 3 we introduce the ***Abstract Data Type (ADT)*** technique for information representation, that enables a designer to construct newly defined data types. However, this technique is not supported by the Scheme language, and requires a sever discipline on the designer's part.

Chapter 3

Abstraction on Data

Partial sources: SICP 2.1, 2.2 [1]; Krishnamurthi 27-30 [8].

Management of compound data increases the ***conceptual level*** of the design, adds ***modularity*** and enables better ***maintenance, reuse***, and ***integration***. Processes can be designed, without making commitments about concrete representation of information. For example, we can design a process for, say "student registration to a course", without making any commitment as to what a `Student` entity is, requiring only that a student entity must be capable of providing its updated academic record.
Conceptual level: The problem is manipulated in terms of its conceptual elements, using only conceptually meaningful operations.
Modularity: The ***implementation*** can be separated from the ***usage*** of the data – a method called ***data abstraction***.
Maintenance, reuse, integration: Software can be built in terms of general features, operations and combinations. For example, the notion of a ***linear combination*** can be defined, independently of the exact identity of the combined objects, which can be matrices, polynomials, complex numbers, etc. Algebraic rules, such as commutativity, associativity, etc can be expressed, independently from the data identity – numbers, database relations, database classes etc.

Example 3.1. – ***Linear combinations***

```
(define (linear-combination a b x y)
   (+ (* a x) (* b y)))
```

To express the general concept, abstracted from the exact type of `a, b, x, y`, we need operations `add` and `mul` that correctly operate on different types:

```
 (define (linear-combination a b x y)
   (add (mul a x) (mul b y)))
```

The ***data abstraction*** approach enables operations that are ***identified*** by the arguments.

The concepts of ***abstract class*** and ***interface*** in Object-Oriented programming implement the data abstraction approach: A linear combination in an OO language can be implemented by a method that takes arguments that implement an interface (or an abstract class) with the necessary multiplication and addition operations. The parameters of the method are known to have the interface type, but objects passed as arguments invoke their specific type implementation for `add` and `mul`.

3.1 Data Abstraction: Abstract Data Types

Data abstraction intends to separate ***usage*** from ***implementation*** (also termed ***concrete representation***).

Client level – Usage: The parts of the program that use the data objects, i.e., the ***clients***, access the data objects via ***operations***, such as set union, intersection, selection. They make no assumption about how the data objects are implemented.

Supplier level – Implementation: The parts of the program that implement the data objects provide procedures for ***selectors*** (***getters*** in Object-Oriented programming) and ***constructors***. Constructors are the ***glue*** for constructing compound data objects, and selectors are the means for splitting compound objects. The connection between the abstract conceptual (client) level of usage to the concrete level of implementation is done by implementing the client needs in terms of the constructors and selectors.

The principle of data abstraction is the ***separation*** between these levels:

```
                    2. Client Level
                         ||
                         ||
                        \||/
                         \/
 1. Abstract Data Types (ADTs):
          Data operators: Constructors, selectors, predicates
          Correctness rules (Invariants)
                         /\
                        /||\
                         ||
                         ||
                    3. Supplier (implementation) Level
```

Rules of correctness (invariants): An implementation of data objects is verified (tested to be true) by rules of correctness (invariants). An implementation that satisfies the correctness rules is correct. For example, every implementation of arrays that satisfies

```
get(A[i], set(A[i],val)) = val
```

for an array `A`, and reference index `i`, is correct. Correct implementations can still differ in other criteria, like space and time efficiency.

Software development order: First the Abstract Data Types level is determined. Then the ***client level*** is written, and only then the ***supplier level***. In languages that support the notion of ***interface*** (or ***abstract class***), like Java and C++, the ***client-supplier*** separation is enforced by the language constructs. The ADT formulation should be determined in an early stage of development, namely, before the development of the conceptual and implementation levels.

An abstract data type (ADT) consists of:

1. Signatures of constructors.
2. Operator signatures: Selectors, predicates, and possibly other operations.
3. (a) Specification of types, pre-conditions and post-conditions for the constructor and operator signatures.
   (b) Rules of correctness (invariants).

Type vs. ADT: Type is a ***semantic notion***: A set of values. Usually, there are various operations defined on these values – selectors, identifying predicate, equality, and constructors. ADT is a ***syntactic notion***: A collection of operation signatures and correctness rules.

> An ADT is ***implemented*** by a type (usually having the same name) that implements the signatures declared by the ADT. The type constructors, selectors and operations must obey the type specifications, and satisfy the pre/post-conditions and invariants.

3.1.1 Example: Binary Trees – Management of Hierarchical Information

Hierarchical information requires data structure that enable combination of compound data, in a way that preserves the hierarchical structure. Such structures are needed, for example, for representing business organizations, library structures, operation plans, etc. Management of hierarchical data includes operations like counting the number of atomic data elements, adding or removing data elements, applying operations to the data elements in the structure, etc.

Binary-Tree is a natural ADT for capturing binary hierarchical structures. It provides a constructor for combining structures into a new binary tree, and operations for selecting the components, identification and comparison.

3.1.1.1 The Binary-Tree ADT

The Binary-Tree ADT is implemented by the Binary-Tree type. We use the same name for the ADT and the type.

Constructor signatures:

```
Signature: make-binary-tree(l,r)
Purpose: Returns a binary tree whose left sub-tree is l
         and whose right sub-tree is r
Type: [Binary-Tree*Binary-Tree -> Binary-Tree]
Pre-condition: binary-tree?(l) and binary-tree?(r)

Signature: make-leaf(d)
Purpose: Returns a leaf binary-tree whose data element is d
Type: [T -> Binary-Tree]
```

Selector signatures:

```
Signature: left-tree(r), right-tree(r)
Purpose: (left-tree <t>): Returns the left sub-tree of the binary-tree <t>.
         (right-tree <t>): Returns the right sub-tree of the binary-tree <t>.
Type: [Binary-Tree -> Binary-Tree]
Pre-condition: composite-binary-tree?(t)

Signature: leaf-data(r)
Purpose: Returns the data element of the leaf binary-tree <t>.
Type: [Binary-Tree -> T]
Pre-condition: leaf?(t)
```

Predicate signatures:

```
Signature: leaf?(t)
Type: [T -> Boolean]
Post-condition: true if t is a leaf -- constructed by make-leaf

Signature: composite-binary-tree?(t)
Type: [T -> Boolean]
Post-condition: true if t is a composite binary-tree -- constructed by
                make-binary-tree

Signature: binary-tree?(t)
Type: [T -> Boolean]
Post-condition: result = (leaf?(t) or composite-binary-tree?(t) )
```

```
Signature: equal-binary-tree?(t1, t2)
Type: [Binary-Tree*Binary-Tree -> Boolean]
```

Invariants:

```
leaf-data(make-leaf(d)) = d
left-tree(make-binary-tree(l,r)) = l
right-tree(make-binary-tree(l,r)) = r
leaf?(make-leaf(d)) = true
leaf?(make-binary-tree(l,r)) = false
composite-binary-tree?(make-binary-tree(l,r)) = true
composite-binary-tree?(make-leaf(d)) = false
```

Note that the Binary-Tree operations have the `Binary-Tree` as their type. That is, the ADT defines a new type, that extends the type language. In practice, Scheme does not recognize any of the ADTs that we define. This is a programmers' means for achieving software abstraction. Therefore, our typing rule is:

1. ADT operations are declared as introducing new types.
2. Clients of the ADT are expressed (typed) using the new ADT types.
3. Implementers (suppliers) of an ADT use ***already implemented*** types or ADTs types.

3.1.1.2 Client level: Binary-Tree management

Some ***client operations*** may be:

```
;Signature: count-leaves(tree)
;Purpose: Count the number of leaves of 'tree'
;Type: [Binary-Tree -> number]
(define count-leaves
  (lambda (tree)
    (if (composite-binary-tree? tree)
        (+ (count-leaves (left-tree tree))
           (count-leaves (right-tree tree)))
        1)))

;Signature: has-leaf?(item,tree)
;Purpose: Does 'tree' includes a leaf labaled 'item'
;Type: [T*Binary-Tree -> Boolean]
(define has-leaf?
  (lambda (item tree)
    (if (composite-binary-tree? tree)
        (or (has-leaf? item (left-tree tree))
            (has-leaf? item (right-tree tree)))
        (equal? item (leaf-data tree)))))

;Signature: add-leftmost-leaf(item,tree)
;Purpose: Creates a binary-tree with 'item' added as a leftmost leaf to 'tree',
          such that the left-most leaf turns into a branch with 'item' as a left
          leaf-subtree and the older leaf as a right leaf-subtree
;Type: [T*Binary-Tree -> Binary-Tree]
(define add-leftmost-leaf
  (lambda (item tree)
    (if (composite-binary-tree? tree)
        (make-binary-tree (add-leftmost-leaf item (left-tree tree))
                          (right-tree tree))
        (make-binary-tree (make-leaf item)
                          tree))
    ))
```

Note: The client operations are written in a ***non-defensive*** style. That is, correct type of arguments is not checked. They should be associated with appropriate type-checking procedures, that their clients should apply prior to their calls, or be extended with defensive correct application tests.

3.1.1.3 Supplier (implementation) level

In this level we define the Binary-Tree type that implements the Binary-Tree ADT. The implementation is in terms of the already implemented List (not necessarily homogeneous) type. We use the first representation discussed in subsection 1.3.3.2, where the unlabeled binary-tree in Figure 1.3, is represented by the list `(1 (2 3))`. In every implementation, the Binary-Tree type is replaced by an already known type. We present two type implementations.

Binary-Tree implementation I: A binary-tree is represented as a heterogeneous list.

```
;Signature: make-binary-tree(l,r)
;Type: [T1*T2 -> List]
;Pre-condition: binary-tree?(l) and binary-tree?(r)
(define make-binary-tree
  (lambda (l r)
    (list l r)))
```

```
;Signature: make-leaf(d)
;Type: [T -> T]
(define make-leaf
  (lambda (d) d))

;Signature: left-tree(t)
;Type: [List -> List union T]
;Pre-condition: composite-binary-tree?(t)
(define left-tree
  (lambda (t) (car t)))

;Signature: right-tree(t)
;Type: [List -> List union T]
;Pre-condition: composite-binary-tree?(t)
(define right-tree
  (lambda (t) (cadr t)))

;Signarture: leaf-data(t)
;Type: [T -> T]
;Pre-condition: leaf?(t)
(define leaf-data
  (lambda (t) t))

;Signarture: leaf?(t)
;Type: [T -> Boolean]
(define leaf?
  (lambda (t) #t))

;Signarture: composite-binary-tree?(t)
;Type: [T -> Boolean]
(define composite-binary-tree?
  (lambda (t)
    (and (list? t)
         (list? (cdr t))
         (null? (cddr t))
         (binary-tree? (left-tree t))
         (binary-tree? (right-tree t)))
    ))

;Signarture: binary-tree?(t)
;Type: [T -> Boolean]
```

```
(define binary-tree?
  (lambda (t) (or (leaf? t) (composite-binary-tree? t))))

;Signature: equal-binary-tree?(t1,t2)
;Type: [T1*T2 -> Boolean]
;Pre-condition: binary-tree?(t1) and binary-tree?(t2)
(define equal-binary-tree?
  (lambda (t1 t2)
    (cond ( (and (composite-binary-tree? t1)
                 (composite-binary-tree? t2))
            (and (equal-binary-tree? (left-tree t1)
                                     (left-tree t2))
                 (equal-binary-tree? (right-tree t1)
                                     (right-tree t2))))
          ( (and (leaf? t1) (leaf? t2))
            (equal? (leaf-data t1) (leaf-data t2)))
          (else #f))))
```

Note the difference between typing the ADT operation signatures and typing their implementation. The ADT signatures are typed in terms of the ADT type itself since the signatures are used by the ADT clients, while the implementation uses already defined types.

Once the ADT is implemented, the client level operations can be applied:

```
> (define t (make-binary-tree (make-leaf 1)
                              (make-binary-tree (make-leaf 2)
                                                (make-leaf 3))))

> (define tt (make-binary-tree t t))

> t
(1 (2 3))
> tt
((1 (2 3)) (1 (2 3)))
> (leaf? t)
#t
> (composite-binary-tree? t)
#t
> (binary-tree? t)
#t
> (left-tree t)
1
> (right-tree t)
```

```
(2 3)
> (leaf-data (left-tree t))
1

> (count-leaves t)
3
> (count-leaves tt)
6
> (has-leaf? 2 t)
#t
> (add-leftmost-leaf 0 t)
((0 1) (2 3))
```

Does the implementation satisfy the invariants of the Binary-Tree ADT? The invariants are:

```
leaf-data(make-leaf(d)) = d
left-tree(make-binary-tree(l,r)) = l
right-tree(make-binary-tree(l,r)) = r
leaf?(make-leaf(d)) = true
leaf?(make-binary-tree(l,r)) = false
composite-binary-tree?(make-binary-tree(l,r)) = true
composite-binary-tree?(make-leaf(d)) = false
```

The first three invariants are satisfied. But what about the last one? Consider, for example:

```
> (composite-binary-tree? (make-leaf (list 5 6)))
#t
> (leaf? (make-leaf (list 5 6)))
#t
> (has-leaf? (list 5 6) (make-leaf (list 5 6)))
#f
```

What is the problem? There is no way to distinguish leaves that carry data of 2-element lists from composite-binary trees. The `leaf` predicate just accepts any argument, and the `composite-binary-tree?` tests whether its argument is a 2 element list of data of the binary-tree structure.

The Binary-Tree implementation does not provide means for singling out binary-trees from lists of an appropriate structure, and in particular, cannot distinguish leaves that carry list data from composite-binary-trees. The solution is to ***tag*** the implementation values by their intended type.

Tagged-data ADT: The idea is to introduce an interface of operations that enable identification of data. The identification means can be any kind of tagging. The interface below assumes that tags are symbols. The Tagged-data ADT defines a polymorphic type: Tagged-data(T):

```
Signature: attach-tag(x,tag)
Purpose: Construct a tagged-data value
Type: [T*Symbol -> Tagged-data(T)]

Signature: get-tag(tagged)
Purpose: Select the tag from a tagged-data value
Type: [Tagged-data(T) -> Symbol]

Signature: get-content(tagged)
Purpose: Select the data from a tagged-data value
Type: [Tagged-data(T) -> T]

Signature: tagged-data?(datum)
Purpose: Identify tagged-data values
Type: [T -> Boolean]

Signature: tagged-by?(tagged,tag)
Purpose: Identify tagged-data values
Type: [T*Symbol -> Boolean]
```

Invariants:

```
get-tag(attach-tag(x,tag)) = tag
get-content(attach-tag(x,tag)) = x
tagged-data?(d) <=> exist x and tag such that d=attach-tag(x,tag)
tagged-by?(attach-tag(x,tag),tag')) <=> (tag = tag')
```

Binary-tree implementation II – Tagged-data based implementation using heterogeneous lists:

```
;Signature: make-binary-tree(l,r)
;Type: [T1*T2 -> Tagged-data(List)]
;Pre-condition: binary-tree?(l) and binary-tree?(r)
(define make-binary-tree
  (lambda (l r)
    (attach-tag (list l r) 'composite-binary-tree)))
```

```
;Signature: make-leaf(d)
;Type: [T -> Tagged-data(T)]
(define make-leaf
  (lambda (d)
    (attach-tag d 'leaf)))

;Signature: left-tree(t)
;Type: [Tagged-data(List) -> Tagged-data(List union T)]
;Pre-condition: composite-binary-tree?(t)
(define left-tree
  (lambda (t) (car (get-content t))))

;Signature: right-tree(t)
;Type: [Tagged-data(List) -> Tagged-data(List union T)]
;Pre-condition: composite-binary-tree?(t)
(define right-tree
  (lambda (t) (cadr (get-content t))))

;Signarture: leaf-data)t)
;Type: [Tagged-data(T) -> T]
;Pre-condition: leaf?(t)
(define leaf-data
  (lambda (t) (get-content t)))

;Type: [T -> Boolean]
(define leaf?
  (lambda (t) (tagged-by? t 'leaf)))

;Type: [T -> Boolean]
(define composite-binary-tree?
  (lambda (t)
    (tagged-by? t 'composite-binary-tree)))
```

`binary-tree?` and `equal-binary-tree?` are not modified, as they are implemented in terms of other Binary-Tree ADT operations. Client operations stay intact.

Correctness of the tagged implementation of the Binary-tree ADT: The invariants are:

```
leaf-data(make-leaf(d)) = d
left-tree(make-binary-tree(l,r)) = l
right-tree(make-binary-tree(l,r)) = r
```

```
leaf?(make-leaf(d)) = true
leaf?(make-binary-tree(l,r)) = false
composite-binary-tree?(make-binary-tree(l,r)) = true
composite-binary-tree?(make-leaf(d)) = false
```

The invariant are proved by applying one of the evaluation algorithms. Based on known results, that changing evaluation ordering (as in the case of the `applicative-eval` and `normal-eval` algorithms) might affect termination, but not the returned value upon termination, we allow to change evaluation ordering in proofs.

1. Proving: `leaf-data(make-leaf(d)) = d`. We prove: For an arbitrary expression `d`, `eval[(leaf-data (make-leaf d))] = eval[d]`

   ```
   eval[(leaf-data (make-leaf d))] ==>*
   eval[(leaf-data (attach-tag eval[d] 'leaf))] ==>*
   eval[(get-content (attach-tag eval[d] 'leaf))] ==>
     based on the Tagged-data ADT invariant get-content(attach-tag(x,tag))=x
   eval[d]
   ```

2. Proving: `composite-binary-tree?(make-leaf(d)) = false`. We prove: For an arbitrary expression `d`, `eval[(composite-binary-tree (make-leaf d))] = #f`

   ```
   eval[(composite-binary-tree (make-leaf d))] ==>*
   eval[(composite-binary-tree (attach-tag eval[d] 'leaf))] ==>*
   eval[(tagged-by? (attach-tag eval[d] 'leaf)) 'composite-binary-tree] ==>
     based on the Tagged-data ADT invariant
     tagged-by?(attach-tag(x,tag),tag')) <=> (tag = tag')
   eval[(equal? 'leaf 'composite-binary-tree)] ==> #f
   ```

Implementation for the Tagged-data ADT: In order to apply the Tagged-data based implementation of Binary-tree we still need an implementation for the Tagged-data ADT. Here is one possibility – Pair based:

```
;Signature: attach-tag(x,tag)
;Type: [Symbol*T -> Pair(Symbol, T)]
  (define attach-tag (lambda (x tag) (cons tag x)))

;Signature: get-tag(tagged)
;Type: Pair(Symbol,T) -> Symbol
  (define get-tag (lambda (tagged) (car tagged)))
```

```
;Signature: get-content(tagged)
;Type: [Pair(Symbol,T) -> T]
  (define get-content (lambda (tagged) (cdr tagged)))

;Signature: tagged-data?(datum)
;Type: [T -> Boolean]
  (define tagged-data?
     (lambda (datum)
       (and (pair? datum) (symbol? (car datum))) ))

;Signature: tagged-by?(tagged,tag)
;Type: [T*Symbol -> Boolean]
  (define tagged-by?
     (lambda (tagged tag)
       (and (tagged-data? tagged)
            (eq? (get-tag tagged) tag))))

> (define tagged-1 (attach-tag 1 'number))
> (get-tag tagged-1)
number
> (get-content tagged-1)
1
> (tagged-data? tagged-1)
#t
```

Once the Tagged-data ADT is implemented, its clients – Binary-Tree, and the clients of the clients can be applied:

```
> (define t (make-binary-tree (make-leaf 1)
                              (make-binary-tree (make-leaf 2)
                                                (make-leaf 3))))

> (define tt (make-binary-tree t t))

> t
(composite-binary-tree (leaf . 1) (composite-binary-tree (leaf . 2) (leaf . 3)))
> tt
(composite-binary-tree
  (composite-binary-tree (leaf . 1) (composite-binary-tree (leaf . 2) (leaf . 3)))
  (composite-binary-tree (leaf . 1) (composite-binary-tree (leaf . 2) (leaf . 3))))
> (leaf? t)
```

```
#f
> (composite-binary-tree? t)
#t
> (binary-tree? t)
#t
> (left-tree t)
(leaf . 1)
> (right-tree t)
(composite-binary-tree (leaf . 2) (leaf . 3))
> (leaf-data (left-tree t))
1

> (count-leaves t)
3
> (count-leaves tt)
6
> (has-leaf? 2 t)
#t
> (add-leftmost-leaf 0 t)
(composite-binary-tree
  (composite-binary-tree (leaf . 0) (leaf . 1))
  (composite-binary-tree (leaf . 2) (leaf . 3)))

> (make-leaf (list 5 6))
(leaf 5 6)
> (composite-binary-tree? (make-leaf (list 5 6)))
#f
> (leaf? (make-leaf (list 5 6)))
#t
> (has-leaf? (list 5 6) (make-leaf (list 5 6)))
#t
```

Note the last three calls. Earlier, `(make-leaf (list 5 6))` was recognized both as a leaf and as a composite-binary-tree.

3.1.2 Example: Rational Number Arithmetic (SICP 2.1.1)

Rational number arithmetic supports arithmetic operations like addition, subtraction, multiplication and division of rational numbers. A natural candidate for the ADT level is an ADT that describes rational numbers, and provides operations for selecting their parts, identifying them and comparing them. Therefore, we start with a definition of an ADT ***Rat*** for rational numbers. It assumes that a rational number is constructed from a numerator and a denominator numbers.

3.1.2.1 The Rat ADT

The Rat ADT would be implemented by the Rat type. We use the same name for the ADT and the type.
Constructor signature:

```
Signature: make-rat(n,d)
Purpose: Returns a rational number whose numerator is the
         integer <n>, and whose denominator is the integer <d>
Type: [Number*Number -> Rat]
Pre-condition: d != 0, n and d are integers.
```

Selector signatures:

```
Signature: numer(r), denom(r)
Purpose: (numer <r>): Returns the numerator of the rational number <r>.
         (denom <r>): Returns the denominator of the rational number <r>.
Type: [Rat -> Number]
Post-condition for denom: result != 0.
```

Predicate signatures:

```
Signature: rat?(r)
Type: [T -> Boolean]
Post-condition: result = (Type-of(r) = Rat)

Signature: equal-rat?(x, y)
Type: [Rat*Rat -> Boolean]
```

The ADT invariants will be discussed following the presentation of several alternative implementations.

Note that the Rat operations are declared as have the Rat type. That is, we use the ADT as a new type, that extends the type language. In practice, Scheme does not recognize any of the ADTs that we define. This is a programmers means for achieving software abstraction. Therefore, our typing rule is:

1. ADT operations are declared as introducing new types.
2. Clients of the ADT are expressed (typed) using the new ADT types.
3. Implementers (suppliers) of an ADT use ***already implemented*** types or ADTs.

3.1.2.2 Client level: Rational-number arithmetic

The ***client operations*** for using rationals are: ***Addition, subtraction, multiplication, division***, and ***equality***. In addition, the client level includes a `print-rat` operation, for nice intuitive display of rationals. All operations are implemented in terms of the Rat ADT:

```
Type: [Rat*Rat -> Rat]   for add-rat, sub-rat, mul-rat, div-rat.
(define add-rat
  (lambda (x y)
     (make-rat (+ (* (numer x) (denom y))
                  (* (denom x) (numer y)))
               (* (denom x) (denom y)))))

(define sub-rat
  (lambda (x y)
     (make-rat (- (* (numer x) (denom y))
                  (* (denom x) (numer y)))
               (* (denom x) (denom y)))))

(define mul-rat
  (lambda (x y)
     (make-rat (* (numer x) (numer y))
               (* (denom x) (denom y)))))

(define div-rat
  (lambda (x y)
     (make-rat (* (numer x) (denom y))
               (* (denom x) (numer y)))))

Type: Rat -> Void
 (define print-rat
    (lambda (x)
       (display (numer x))
       (display "/")
       (display (denom x))
       (newline)))
```

Note: In all Rat arithmetic procedures, clients should verify that the arguments are of type Rat (using the `rat?` procedure).

3.1.2.3 Supplier (implementation) level

In this level we define the Rat type that implements the Rat ADT. The implementation is in terms of the already implemented Pair type. The implementation depends on a ***representation*** decision: What are the values of the Rat type. In every implementation, the Rat type is replaced by an already known type.

We present several Rat type implementations. All are based on the Pair type, but differ in the actual integers from which the values of Rat are constructed.

Rat implementation I – Based on an unreduced Pair representation: A rational number is represented by a pair of its numerator and denominator. That is, `Rat=Pair(Number,Number)`:

```
Signature: make-rat(n,d)
Type: [Number *Number -> Pair(Number,Number)]
Pre-condition: d != 0
  (define make-rat (lambda (n d) (cons n d)))

Type: [Pair(Number,Number) -> Number]
  (define numer (lambda (r) (car r)))

Type: [Pair(Number,Number) -> Number]
  (define denom (lambda(r) (cdr r)))

Type: [T --> boolean]
  (define rat? (lambda (r) (pair? r)))

Type: [Pair(Number,Number)*Pair(Number,Number) -> Number]
  (define equal-rat?
     (lambda (x y)
       (= (* (numer x) (denom y))
          (* (numer y) (denom x))))


Pre-condition and argument types tests:
(define make-rat-pre-condition-argument-type-test
   (lambda (n d)
     (and (integer? n) (integer? d) (not (= d 0)))))

(define numer-argument-type-test
   (lambda (r)
     (rat? r))
```

```
(define denom-argument-type-test
    (lambda (r)
      (rat? r)))
```

Once the ADT is implemented, the client level operations can be applied:

```
> (define one-half (make-rat 1 2))
> (define two-sixth (make-rat 2 6))
> (print-rat (add-rat one-half two-sixth))
10/12
> (print-rat (mul-rat one-half two-sixth))
2/12
> (print-rat (div-rat two-sixth one-half))
4/6
> (div-rat two-sixth one-half)
(4 . 6)
> (define x (print-rat (div-rat two-sixth one-half)))
 > x
???????
```

Note on the types of the implemented Rat operations: Note the difference between typing the ADT operation signatures and typing their implementation. The ADT signatures are typed in terms of the ADT itself since the signatures are used by the ADT clients, while the implementation uses already defined ADTs. For the above implementation `Rat=Pair(Number,Number)`, the Rat type in the ADT declaration is replaced by the implementation type `Pair(Number,Number)`. For example, in the ADT declaration the type of `make-rat` is `[Number*Number -> Rat]`, while the type of the implemented `make-rat` is `[Number*Number -> Pair(Number,Number)]`.
Variation on the Rat operator definition: The Rat value constructor and selectors can be defined as new names for the Scheme primitive Pair constructor and selectors:

```
(define make-rat cons)
(define numer car)
(define denom cdr)
(define rat? pair?)

> make-rat
#<primitive:cons>
```

In the original definition, `make-rat` uses cons, and therefore is a compound procedure. In the second definition, `make-rat` is another name for the primitive procedure which is the value of `cons`. In this case there is a single procedure with two names. The second option is more efficient in terms of time and space.

Rat implementation II – Tagged unreduced Pair representation: The intention behind the previous implementation of Rat is to identify the Rat type with the type `Pair(Number,Number)`. But the identifying predicate `rat?` is defined by:
`(define rat? (lambda (r) (pair? r)))`.
Therefore, `rat?` identifies ***any pair*** as a rational number implementation, including for example, the pairs `(a . b)`, and `(1 . 0)`:

```
> (rat? (make-rat 3 2))
#t
>  (rat? (cons (quote a) (quote b)))
#t
```

The problem is that the Rat implementation does not provide any means for singling out pairs that implement Rat values from all other pairs. The solution is to ***tag*** the implementation values by their intended type.

Tagged Pair based implementation: The tagged implementation is
`Rat=Tagged-data(Pair(Number,Number))=Pair(Symbol,Pair(Number,Number))`. `equal-rat?` is not listed as it does not change.

```
Signature: make-rat(n, d)
Type: [Number*Number -> Tagged-data(Pair(Number,Number))]
Pre-condition: d != 0; n and d are integers.
  (define make-rat
     (lambda (n d)
       (attach-tag (cons n d) 'rat)))

Signature: numer(r)
Type: [Tagged-data(Pair(Number,Number)) -> Number]
  (define numer
     (lambda (r)
        (car (get-content r))))

Signature: denom(r)
Type: [Tagged-data(Pair(Number,Number)) -> Number]
Post-condition: result != 0
  (define denom
     (lambda (r)
       (cdr (get-content r))))

Signature: rat?(r)
Type: [T -> Boolean]
```

```
Post-condition: result = (Type-of(r) = Rat)
  (define rat? (lambda (x) (tagged-by? x 'rat)))

> (define one-half (make-rat 1 2))
> (define two-sixth (make-rat 2 6))
> (print-rat (add-rat one-half two-sixth))
10/12
> (print-rat (mul-rat one-half two-sixth))
2/12
> (define x (print-rat (div-rat two-sixth one-half)))
> one-half
(rat 1 . 2)
> (get-tag one-half)
rat
> (content one-half)
(1 . 2)
> (rat? one-half)
#t
> (rat? (content one-half))
#f
> (rat? (div-rat one-half two-sixth))
#t
```

Rat implementation III – Tagged, reduced at construction time Pair representation: The idea behind this implementation is to represent the rational number by a reduced pair of numerator and denominator. The reduction uses the `gcd` procedure from `http://mitpress.mit.edu/sicp/code/ch2support.scm`).

```
(define gcd
  (lambda (a b)
    (if (= b 0)
        a
        (gcd b (remainder a b)))))
```

The implementation is still: `Rat=Pair(Symbol,Pair(Number,Number))`. `rat?, equal-rat?` are not listed as there is no change.

```
Signature: make-rat(n,d)
Type: [Number*Number -> Tagged-data(Pair(Number,Number))]
Pre-condition: d != 0; n and d are integers
  (define make-rat
    (lambda (n d)
       (let ((g (gcd n d)))
          (attach-tag (cons (/ n g) (/ d g)) 'rat))))

Signature: numer(r)
Type: [Tagged-data(Pair(Number,Number)) -> Number]
   (define numer (lambda (r) (car (get-content r))))

Signature: denom(r)
Type: [Tagged-data(Pair(Number,Number)) -> Number]
Post-condition: result != 0
   (define denom (lambda (r) (cdr (get-content r))))

> (print-rat (div-rat two-sixth one-half))
2/3
> (define one-half (make-rat 5 10))
> one-half
(rat 1 . 2)
```

Rat implementation IV: Tagged, reduced at selection time Pair representation: The idea behind this implementation is to represent the rational number by the given numerator and denominator, but reduce them when queried. The implementation is still: `Rat=Pair(Symbol,Pair(Number,Number))`. `rat?, equal-rat?` are not listed as there is no change.

```
Signature: make-rat(n,d)
Type: [Number*Number -> Tagged-data(Pair(Number,Number))]
Pre-condition: d != 0 ; n and d are integers
   (define make-rat
      (lambda (n d)
        (attach-tag (cons n d) 'rat)))

Signature: numer(r)
Type: [Tagged-data(Pair(Number,Number)) -> Number]
  (define numer
     (lambda (r)
        (let ((g (gcd (car (get-content r))
                      (cdr (get-content r)))))
          (/ (car (get-content r)) g))))


Signature: denom(r)
```

```
Type: [Tagged-data(Pair(Number,Number)) -> Number]
Post-condition: result != 0
  (define denom
     (lambda (r)
        (let ((g (gcd (car (get-content r))
                      (cdr (get-content r)))))
           (/ (cdr (get-content r)) g))))
```

Invariants of the Rat ADT: We get back now to the Rat ADT invariants. The role of the invariants is to characterize ***correct*** implementations. Our intuition is that all presented implementations are correct. Hence, we need rules that are satisfied by all implementations above.
First suggestion:

```
numer(make-rat(n,d)) = n
denom(make-rat(n,d)) = d
```

Are these rules satisfied by all implementations above? The answer is ***no!***. Implementations III and IV do not satisfy them. Yet, our intuition is that these implementations are correct. That means that the suggested invariants are too strict, and therefore reject acceptable implementations.
Second suggestion:

```
numer(make-rat(n,d))/denom(make-rat(n,d)) = n/d
```

This invariant is satisfied by all of the above implementations.

Summary of the operations of the Rat ADT:

```
Signature: make-rat(n,d)
Purpose: Returns a rational number whose numerator is the
         integer <n>, and whose denominator is the integer <d>
Type: [Number*Number -> Rat]
Pre-condition: d != 0, n and d are integers.
```

Selector signatures:

```
Signature: numer(r), denom(r)
Purpose: (numer <r>): Returns the numerator of the rational number <r>.
         (denom <r>): Returns the denominator of the rational number <r>.
Type: [Rat -> Number]
Post-condition for denom: result != 0.
```

Predicate signatures:

```
Signature: rat?(r)
Type: [T -> Boolean]
Post-condition: result = (Type-of(r) = Rat)

Signature: equal-rat?(x, y)
Type: [Rat*Rat -> Boolean]

Rule of correctness (invariant):
[ (numer (make-rat n d)) / (denom (make-rat n d)) ] = n/d
```

3.1.3 Example: Church Numbers

The *Church numbers* type (already introduced at the end of Chapter 2) is defined inductively as follows: `Zero` is a Church number, and for a Church number `c`, `S(c)` is also a Church number. These are all Church numbers. This inductive definition can be summarized in the following definition:
`Church-num = 'Zero union S(Church-num)`

Church numbers provide ***symbolic representation*** for the natural numbers, where the numbers `0, 1, 2, 3, ...` are represented by the Church numbers `Zero, S(Zero), S(S(Zero)), S(S(S(Zero))), ...` . Viewing Church numbers as an ADT separates their usage from their implementation. Client view is:

```
Signature: make-church-num(n)
Purpose: Returns a Church number
Type: [Number -> ChurchNum]
Pre-condition: n >= 0, an integer.
```

Selector signature:

```
Signature: church-val(church-num)
Purpose: (church-val <church-num>): Returns the numerical value of church-num
Type: [ChurchNum -> Number]
Post-condition: result >= 0.
```

Predicate signatures:

```
Signature: church?(x)
Type: [T -> Boolean]
Post-condition: result = (Type-of(x) = Church-num)

Signature: equal-rat?(x, y)
Type: [Church-num*Church -> Boolean]
```

invariant:

`church-val(make-church-num(n)) = n`

Client level operations on Church numbers is the standard integer arithmetics. We present here only the supplier level (implementation), without tagging.

Church Numbers implementation I – Based on lambda abstraction: Consider the set of Scheme expressions `0, (lambda () 'Zero), (lambda () (lambda () 'Zero)), (lambda () (lambda () (lambda () 'Zero))), ...`, whose evaluation yields the set `0, <closure () 'Zero>, <closuer () <closure () 'Zero», <closure () <closure () <closure () 'Zero»>, ...`. It can implement Church numbers, by representing `Zero` by `o`, and representing the Church number `S(S...(S(Zero)))`, by `(lambda () (lambda () ... (lambda () 'Zero)))`, with the same number of `S` and `lambda` applications. Under this implementation, the set of Church numbers is implemented as the type:
`Church-num = 'Zero union [Empty -> Church-num]`

We already know that Inductively-defined (recursive) types cannot be described by the type language introduced in Chapter 2. Therefore, the supplier view:

```
Signature: church-num(n)
Type: [Number -> Procedure union Number]

Signature: church-val(church-num)
Type: [Procedure union Number -> Number]
```

is not precise, and cannot guarantee ***type safety***. Therefore, for recursive types, we define the type of the supplier view using the recursive implementation type, named also `Church-num`.

```
Type: [Number -> ChurchNum]
(define make-church-num
  (lambda (n)
    (if (= n 0)
        'Zero
        (lambda () (make-church-num (- n 1))))
    ))

Type: [Number -> ChurchNum]
(define church-val
  (lambda (church-num)
    (if (eq? church-num 'Zero)
        0
        (+ 1 (church-val (church-num))))
    ))
```

Invariant correctness: In order to prove `church-val(make-church-num(n)) = n`, we need to prove, for an arbitrary non-negative integer `n`:

`eval[church-val(make-church-num(n))] = eval[n]`

The proof is obtained by processing the operational semantics algorithms (e.g., `applicative-eval` on both sides.

Church Numbers implementation II – Based on Pair implementation: Consider the set of Scheme expressions `Zero, (S . Zero), (S . (S . Zero)), (S . (S . (S . Zero))), ....` This set is created by evaluating the Scheme expressions: `'Zero, (cons 'S 'Zero), (cons 'S (cons 'S 'Zero)), (cons 'S (cons 'S (cons 'S 'Zero))), ... .` It can implement Church numbers, by representing `Zero` by the symbol `'Zero`, and representing the Church number `S(S...(S(Zero)))`, by `(S . (S ... (S . Zero)))`, with the same number of `S`-s. Under this implementation, the set of Church numbers is implemented as the type:
`Church-num = 'Zero union Pair(Symbol,Church-num)`

```
Signature: make-church-num(x)
Purpose: Computes the nth Church number.
Type: Number -> Church-num
(define church-num
  (lambda (n)
    (if (zero? n)
        'Zero
        (cons 'S (make-church-num (- n 1))))
    ))

Signature: church-val(x)
Purpose: Computes the nth Church number.
Type: Church-num -> Number
(define church-val
  (lambda (church-num)
    (if (eq? church-num 'Zero)
        0
        (+ 1 (church-val (cdr church-num))))
    ))
```

3.1.4 What is Meant by Data? (SICP 2.1.3)

The question that motivates this subsection is: ***What is data?*** The notion of ***data*** is usually understood as something ***consumed by procedures***. But in functional languages, procedures are first class citizens, i.e., handled like values of other types. Therefore in such languages the distinction between data and procedures is especially obscure.

In order to clarify this issue we ask whether procedures can be used as data, i.e., consumed by procedures. Specifically, we consider the previous Binary-Tree management or the Rational Number arithmetic problem, where the implementations use the built-in Pair and List types. We pose the problem:

> Suppose that the Scheme application does not include built-in Pair or List types.
> How can we build an implementation for the Binary-Tree and the Rat ADTs?

We solve the problem by:

1. Defining a Pair ADT: `Pair(T1,T2)`.
2. Defining a Pair type that implements the Pair ADT in terms of the Procedure type. That is: `Pair(T1,T2) = [Symbol -> (T1 union T2)]`.

The Pair ADT:

```
Signature: cons(x,y)
Type: [T1*T2 -> Pair(T1,T2)]

Signature: car(p)
Type: [Pair(T1,T2) -> T1]

Signature: cdr(p)
Type: [Pair(T1,T2) -> T2]

Signature: pair?(p)
Type: [T -> Boolean]

Signature: equal-pair?(p1,p2)
Type: [Pair(T1,T2)*Pair(T1,T2) -> Boolean]

Invariants:
(car (cons x  y)) = x
(cdr (cons x  y)) = y
```

Below are two implementations of the Pair ADT in terms of procedures. A pair is represented by a procedure, that enables selection of the pair components. The implementations differ in their processing of the pair components. The first implementation, termed ***eager***, represents a pair as a procedure built specifically for selection of the pair components. The second implementation, termed ***lazy***, represents a pair as a procedure built for answering any request about the pair components.

3.1.4.1 Pair implementation I: Eager Procedural representation

```
Signature: cons(x,y)
Type: [T1*T2 -> [Symbol -> (T1 union T2)]
(define cons
  (lambda (x y)
    (lambda (m)
       (cond ((eq? m 'car) x)
             ((eq? m 'cdr) y)
             (else (error "Argument not 'car or 'cdr -- CONS" m) ))) ))

Signature: car(pair)
Type: [[Symbol -> (T1 union T2)] -> T1]
 (define car (lambda (pair) (pair 'car)))

Signature: cdr(pair)
Type: [[Symbol -> (T1 union T2)] -> T2]
 (define cdr (lambda (pair) (pair 'cdr)))

Signature: equal-pair?(p1,p2)
Type: [[[(T1 union T2)] -> T2]*[[(T1 union T2)] -> T2] -> Boolean]
(define equal-pair?
  (lambda (p1 p2)
     (and (equal? (car pair1) (car pair2))
          (equal? (cdr pair1) (cdr pair2)) )))
```

A pair data value is a procedure, that stores the information about the pair components. `car` takes a pair data value – a procedure – as an argument, and applies it to the symbol `car`. `cdr` is similar, but applies its argument pair procedure to the symbol texttttcdr. The pair procedure, when applied to the symbol `car`, returns the value of the first parameter of cons. Hence, `(car (cons x y))`, evaluates to the first pair component. The rule for `cdr` holds for similar arguments. Note that the definition of `cons` returns a closure as its value, but does not apply it!

```
applicative-eval[ (cons 1 2) ] ==>*
<closure (m)
       (cond ((eq? m 'car) 1)
             ((eq? m 'cdr) 2)
             (else (error "Argument not 'car or 'cdr -- CONS" m) ))>

applicative-eval[ (car (cons 1 2 )) ] ==>
      applicative-eval[ car ] ==> <closure (pair) (pair 'car)>
```

```
     applicative-eval[ (cons 1 2) ] ==>* <the cons closure as above >
 sub[pair, <cons closure>, (pair 'car) ] ==> (<cons closure> 'car)
 reduce:
 ( (lambda (m)
       (cond ((eq? m 'car) 1)
             ((eq? m 'cdr) 2)
             (else (error "Argument not 'car or 'cdr -- CONS" m) )))
'car) ==>*
 applicative-eval, sub, reduce:
 (cond ((eq? 'car 'car) 1)
      ((eq? 'car 'cdr) 2)
        (else (error "Argument not 'car or 'cdr -- CONS" 'car) )) ==>
 1

 > (define x (cons 1 2))
 > x
 #<Closure (m) (cond ((eq? m 'car) x) ((eq? m 'cdr) y) (else (error
 #"Argument not ... "))))>
 > (define y (car x))
 > y
 1
 > (define z (cdr x))
 > z
 2
 > (define w (cons y z))
 > (car w)
 1
 > (cdr w)
 2
 > cons
 #<Closure (x y) (lambda (m) (cond ((eq? m 'car) x) ((eq? m 'cdr) y)
 (else (error ... ))))>
 > car
 #<Closure (pair) (pair 'car)>
```

Notes:

1. Pairs are represented as procedures that receive messages. A pair is created by application of the `cons` procedure, that generates a new procedure for the defined pair. Therefore, the variables `x, w` above denote different pair objects – different procedures (recall that the Procedure type does not have an equality predicate).

2. The `equal-pair?` implementation uses the built-in primitive predicate `equal?`. Since Pair is a polymorphic ADT, its implementation requires a polymorphic equality predicate, that can be either built-in or written (for example, as a very long conditional of value comparisons).

3. The technique of EAGER procedural abstraction, where data values are implemented as procedures that take a message as input, is called ***message passing***.

An alternative writing of this implementation, using a locally created procedure named `dispatch`:

```
Signature: cons(x,y)
Type: [T1*T2 -> [Symbol -> T1 union T2 union String]
 (define cons
  (lambda (x y)
   (letrec
      ((dispatch (lambda(m)
                        (cond ((eq: m 'car) x)
                                 ((eq? m 'cdr) y)
                                 (else
                              (error "Argument not 'car or 'cdr -- CONS" m))))
         ))
     dispatch)))
```

The Pair implementation does not support the predicate `pair?`. In order to implement `pair?` we need an explicit typing, that should be planed as part of an overall types implementation.

3.1.4.2 Pair implementation II: Lazy Procedural representation

The eager procedural implementation for the Pair ADT represents a Pair value as a procedure that already prepared the computations for all known selectors. The ***lazy*** procedural implementation defers everything: A Pair value is represented as a procedure that "'waits" for just any selector. In selection time, the given selector procedure is applied by the pair components. The constructor does not prepare anything – it is truly lazy!

```
Signature: cons(x,y)
Type: [T1*T2 -> [ [T1*T2 -> T3] -> T3]]
  (define cons
    (lambda (x y)
      (lambda (sel) (sel x y))))

Signature: car(pair)
```

```
Type: [[ [T1*T2 -> T3] -> T3] -> T1]
  (define car
    (lambda (pair)
      (pair (lambda (x y) x))))

Signature: cdr(pair)
Type: [[ [T1*T2 -> T3] -> T3] -> T2]
  (define cdr
    (lambda (pair)
      (pair (lambda (x y) y))))
```

Evaluation examples:

```
applicative-eval[ (cons 1 2) ] ==>
<closure (sel) (sel 1 2)>

applicative-eval[ (car (cons 1 2 )) ] ==>*
   applicative-eval[ car ] ==> <closure (pair) (pair (lambda(x y) x))>
   applicative-eval[ (cons 1 2) ] ==>* <closure (sel) (sel 1 2) >
sub, reduce:
applicative-eval[
   ( <closure (sel) (sel 1 2) > (lambda(x y) x) ) ] ==>*
applicative-eval[ ( (lambda(x y) x) 1 2) ] ==>
applicative-eval, sub, reduce:
1
```

Question: Assume that the Pair ADT has an additional operation `function-value` specified by the following contract:

```
Signature: sum-values(pair,f)
Purpose: Returns the sum of f values on the components of a pair of numbers
Type: [Pair(Number,Number)*[Number -> Number] -> Number]
Example: sum-values((2 . 3),(lambda (x) (- x))) = -5
```

Add this operation to the eager and lazy procedural implementations of the ADT Pair.

Side comment: The lazy procedural implementation implements the *Visitor* design pattern

Software design patterns [4] (`http://www.cs.up.ac.za/cs/aboake/sws780/references/patternstoarchitecture/Gamma-DesignPatternsIntro.pdf`) is an approach that provides solutions for typical problems that accrue in multiple contexts. ***Visitor*** is a well known design pattern, suggested in [4]. Here is a short description taken from wikipedia:

> In ***object-oriented programming*** and software engineering, the ***visitor design pattern*** is a way of separating an ***algorithm*** from an object structure it operates on. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.
>
> In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through ***double dispatch***.

In the ***Visitor*** design pattern, a client holds an operation – the ***visitor***, and an element – the ***object***, where the exact identity of both is not known to the client. The client lets the visitor approach the object (by applying the `accept` method of the object. The object then dispatches itself to the visitor. After this double dispatch – visitor to object and object to visitor, the concrete visitor holds the concrete object and can apply its operation on the object.

The lazy procedural implementation is based on a similar double dispatch: In order to operate, a selector gives itself (***visits***) to an object, and then the object dispatches itself to the operator for applying its operation.

3.1.4.3 Comparison of the eager and the lazy procedural implementations:

1. **Eager:** More work at constructions time. Immediate at selection time.
   Lazy: Immediate at construction time. More work at selection time.

2. **Eager:** Selectors that are not simple getters can have any arity.
   Lazy: selectors can be added freely, but must have the same arity.

3.2 The Sequence Interface (includes examples from SICP 2.2.3)

Object-oriented programming languages support a variety of interfaces and implementation utilities for aggregates, like Set, List, Array. These interfaces declare standard collection services like `has-next()`, `next()`, `item-at(ref)`, `size()`, `is-empty()` and more.

Functional languages provide furthermore, powerful ***sequence operations*** that put an ***abstraction barrier*** (***ADT interface***) between clients running sequence applications to the sequence implementation. The advantage is the separation between usage and implementation: Ability to develop ***abstract level client applications***, without any commitment to the exact sequence implementation. Sequence operations abstract away the element-by-element sequence manipulation. Using sequence operations, client procedures become clearer, cleaner, and their uniformity stands out.

3.2.1 Mapping over Lists

The basic sequence operation is ***sequence procedure*** `map`, that ***applies a transformation to all elements of a list***, and returns a list of the results. It takes a procedure of one argument, and a list and applies the procedure to all elements of the list and returns a list of the results:

```
Signature: map(proc,sequence)
Purpose: Apply 'proc' to all 'sequence'.
Type: [[T1 -> T2]*List(T1) -> List(T2)]
Examples:
    (map abs (list -10 2.5 -11.6 17)) ==> (10 2.5 11.6 17)
    (map (lambda (x) (* x x))
         (list 1 2 3 4))              ==> (1 4 9 16)
```

Post-condition: For all i=1..length(sequence): $result_i$ = proc(sequence$_i$)

Client level of the Sequence Interface:

Example 3.2. *Mapping over homogeneous lists: Scaling a number list by a given factor:*

```
Signature: scale-list(items,factor)
Purpose: Scaling elements of a number list by a factor.
Type: [List(Number)*Number -> List(Number)]
  (define scale-list
     (lambda (items factor)
       (if (null? items)
           (list)
           (cons (* (car items) factor)
                (scale-list (cdr items) factor)))))

> (scale-list (list 1 2 3 4 5) 10)
(10 20 30 40 50)
```

Written as a client of the Sequence Interface:

```
> (define scale-list
     (lambda (items factor)
       (map (lambda (x) (* x factor))
            items))
> (scale-list (list 1 2 3 4 5) 10)
 (10 20 30 40 50)
```

Example 3.3. *Mapping over hierarchical (heterogeneous) lists (viewed as trees): Scaling an unlabeled number-binary-tree:*

```
Signature: scale-tree(tree,factor)
Purpose: Scale an unlabeled tree with number leaves.
Type: [(List*Number) -> Number]
  (define scale-tree
     (lambda (tree factor)
         (cond ((empty? tree) tree)
              ((not (list? tree)) (* tree factor))
              (else (cons (scale-tree (car tree) factor)
                          (scale-tree (cdr tree) factor))))))

> (scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7))
              10)
(10 (20 (30 40) 50) (60 70))
```

Written as a client of the Sequence Interface: An unlabeled tree is a list of trees or leaves. Tree scaling can be obtained by mapping `scale-tree` on all branches that are trees, and multiplying those that are leaves by the factor.

```
Signature: scale-tree(tree,factor)
Purpose: Scale an unlabeled tree with number leaves.
Type: [List*Number -> List]
  (define scale-tree
     (lambda (tree factor)
         (map (lambda (sub-tree)
                 (if (list? sub-tree)
                     (scale-tree sub-tree factor)
                     (* sub-tree factor)))
              tree)))

> (scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7))
              10)
(10 (20 (30 40) 50) (60 70))
> (scale-tree (list) 10)
()
> (scale-tree (list 1) 10)
(10)
>
```

The second version is better since it clearly conceives a tree as a list of branches that are either trees or leaves (numbers). It ignores the detailed tree construction, is simpler, less prone to errors, and does not depend on lower level construction.

Implementation of the `map` Sequence operation:

```
Signature: map(proc,items)
Purpose: Apply 'proc' to all 'items'.
Type: [[T1 -> T2]*List(T1) -> List(T2)]
  (define map
     (lambda (proc items)
        (if (null? items)
            (list)
            (cons (proc (car items))
                  (map proc (cdr items))))))

> (map abs (list -10 2.5 -11.6 17))
(10 2.5 11.6 17)
> (map (lambda (x) (* x x))
       (list 1 2 3 4))
(1 4 9 16)
```

Value and importance of mapping operations: Mapping operations establish a higher level of abstraction in list processing. The element-by-element attention is shifted into a whole-list transformation attention. The two `scale-list` procedures perform exactly the same operations, but the mapping version supports a higher level of abstraction.

Mapping provides an ***abstraction barrier*** for list processing.

More general `map`: The definition of `map` in Scheme is more general, and allows the application of n-ary procedures to n list arguments:

```
> (map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
(741 852 963)
> (map (lambda (x y) (+ x (* 2 y)))
       (list 1 2 3)
       (list 4 5 6))
(9 12 15)
```

Tree `map`: Recursion (induction) on the ***depth of the tree (list)*** rather than on all branches (length of the heterogeneous list).

```
Signature: tree-map(proc tree)
Purpose: Apply .proc to all leaves of the tree (represented as a heterogeneous list)
Type: [List -> List]
(define tree-map
  (lambda (proc tree)
    (map (lambda (sub-tree)
           (if (list? sub-tree)
               (tree-map proc sub-tree)
               (proc sub-tree)))
         tree)))

> (tree-map abs (list -1 (list 2 (list 3 -4) 5) (list -6 7)))
'(1 (2 (3 4) 5) (6 7))  ‘
```

3.2.2 More sequence operations: The Sequence interface as an abstraction barrier

We add more sequence operations, that enable rich management of lists at the Sequence level:

1. **Filtering a homogeneous list:**

   ```
   Signature: filter(predicate, sequence)
   Purpose: return a list of all sequence elements that satisfy the predicate
   Type: [[T-> Boolean]*List(T) -> List(T)]
   Example:  (filter odd? (list 1 2 3 4 5)) ==> (1 3 5)
   Post-condition: result = sequence - {el|el∈sequence and not(predicate(el))}
   ```

2. **Accumulation of procedure application:**

   ```
   Signature: accumulate(op,initial,sequence)
   Purpose: Accumulate by 'op' all sequence elements, starting (ending)
            with 'initial'
   Type: [[T1*T2 -> T2]*T2*List(T1) -> T2]
   Examples: (accumulate + 0 (list 1 2 3 4 5)) ==> 15
             (accumulate * 1 (list 1 2 3 4 5)) ==> 120
             (accumulate cons (list) (list 1 2 3 4 5)) ==> (1 2 3 4 5)
   Post-condition: result =  (sequence_n op (sequence_n-1 ... (sequence_1 op initial)...)
   ```

   There are several primitive procedures for accumulation: `foldr` which is equal to the above `accumulate`, and also `foldl, fold` which apply the operation 'op' in different orderings.

3. **Enumeration of elements:**

   (a) **Interval enumeration:**

```
Signature: enumerate-interval(low, high)
Purpose: List all integers within an interval:
Type: [Number*Number -> List(Number)]
Example: (enumerate-interval 2 7) ==> (2 3 4 5 6 7)
Pre-condition: high ≥ low
Post-condition: result = (low low+1 ... high)
```

(b) **Heterogeneous list (tree) enumeration:**

```
Signature: enumerate-tree(tree)
Purpose: List all leaves of a number tree
Type: [List union T -> List(Number)]
Example: (enumerate-tree (list 1 (list 2 (list 3 4)) 5)) ==> (1 2 3 4 5)
Post-condition: result = flatten(tree)
```

Client level of the Sequence interface: We show an example of two seemingly different procedures that actually share common sequence operations. Nevertheless, the similarity is revealed only when using the Sequence interface. The two procedures are `sum-odd-squares` that sums the squares of odd leaves in an unlabeled number tree, and `even-fibs` that lists the even numbers in a Fibonacci sequence up to some point.

```
Signature: sum-odd-squares(tree)
Purpose: return the sum of the odd squares of the leaves
Type: [List union Number -> Number]
   (define sum-odd-squares
     (lambda (tree)
       (cond ((null? tree) 0)
             ((not (list? tree))
              (if (odd? tree) (square tree) 0))
             (else (+ (sum-odd-squares (car tree))
                      (sum-odd-squares (cdr tree)))))
       ))
```

It does the following:

1. Enumerates the leaves of a tree.
2. Filters them using the odd? filter.
3. Squares the selected leaves.
4. Accumulates the results, using +, starting from 0.

```
Signature: even-fibs(n)
Purpose:  List all even elements in the length n prefix of the
          sequence of Fibonacci numbers
Type: [Number -> List(Number)]
  (define even-fibs
    (lambda (n)
      (letrec ((next (lambda(k)
                        (if (> k n)
                            (list)
                            (let ((f (fib k)))
                               (if (even? f)
                                   (cons f (next (+ k 1)))
                                   (next (+ k 1))))))))
           (next 0))))
```

It does the following:

1. Enumerates the integers from 0 to n.
2. Computes the Fibonacci number of each.
3. Filters them using the even? filter.
4. Accumulates the results, using cons, starting from the empty list.

This analysis can be visualized as:

```
sum-odd-squares:  enumerate: tree leaves ---> filter: odd? --->
                  map: square   ---> accumulate: +, 0.
even-fibs:       enumerate: integers    ---> map: fib     --->
                  filter: even? ---> accumulate: cons, (list).
```

Rewriting the procedures as clients of the Sequence interface:

```
Signature: sum-odd-squares(tree)
Purpose: return the sum of all odd square leaves
Type: [List -> Number]
  (define sum-odd-squares
     (lambda (tree)
        (accumulate +
                    0
                    (map square
                         (filter odd?
                                 (enumerate-tree tree))))))
```

```
Signature: even-fibs(n)
Purpose:  List all even elements in the length n prefix of the
          sequence of Fibonacci numbers
Type: [Number -> List(Number)]
  (define even-fibs
    (lambda (n)
      (accumulate cons
                  (list)
                  (filter even?
                          (map fib
                               (enumerate-interval 0 n))))))
```

Implementation of the additional Sequence operations

1. **Filtering a sequence:**

```
Signature: filter(predicate, sequence)
Purpose: return a list of all sequence elements that satisfy the predicate
Type: [[T-> Boolean]*List(T) -> List(T)]
  (define filter
    (lambda (predicate sequence)
      (cond ((null? sequence) sequence)
            ((predicate (car sequence))
             (cons (car sequence)
                   (filter predicate (cdr sequence))))
            (else (filter predicate (cdr sequence))))))
```

2. **Accumulation:**

```
Signature: accumulate(op,initial,sequence)
Purpose: Accumulate by 'op' all sequence elements, starting (ending)
         with 'initial'
Type: [[T1*T2 -> T2]*T2*List(T1) -> T2]
  (define accumulate
    (lambda (op initial sequence)
      (if (null? sequence)
          initial
          (op (car sequence)
              (accumulate op initial (cdr sequence))))))
```

3. **Enumeration of elements:**

(a)
```
Signature: enumerate-interval(low, high)
Purpose: List all integers within an interval:
Type: [Number*Number -> List(Number)]
  (define enumerate-interval
    (lambda (low high)
      (if (> low high)
          (list)
          (cons low (enumerate-interval (+ low 1) high)))))
```

(b)
```
Signature: enumerate-tree(tree)
Purpose: List all leaves of a number tree
Type: [List union T -> List(Number)]
  (define enumerate-tree
    (lambda (tree)
      (cond ((null? tree) (tree))
            ((not (list? tree)) (list tree))
            (else (append (enumerate-tree (car tree))
                          (enumerate-tree (cdr tree)))))
       ))
```

Reuse – Value of abstraction: More client level applications:

Example 3.4. *list-fib-squares:*

```
Signature: list-fib-squares(n)
Purpose: Compute a list of the squares of the first n+1 Fibonacci numbers:
       Enumerate [0,n] --> map fib --> map square --> accumulate: cons, (list).
Type: [Number -> List(Number)]
  (define list-fib-squares
     (lambda (n)
        (accumulate cons
                    (list)
                    (map square
                         (map fib
                              (enumerate-interval 0 n))))))
```

```
 > (list-fib-squares 10)
 (0 1 1 4 9 25 64 169 441 1156 3025)
```

Example 3.5. *product-of-squares-of-odd-elements:*

```
Signature: product-of-squares-of-odd-elements(sequence)
Purpose: Compute the product of the squares of the odd elements in a
         number sequence.
         Filter: odd? --> map square --> accumulate: *, 1.
Type: [List(Number) -> Number]
  (define product-of-squares-of-odd-elements
     (lambda (sequence)
       (accumulate *
                   1
                   (map square
                        (filter odd? sequence)))))

> (product-of-squares-of-odd-elements (list 1 2 3 4 5))
225
```

Example 3.6. *salary-of-highest-paid-programmer:*

```
Signature: salary-of-highest-paid-programmer(records)
Purpose: Compute the salary of the highest paid programmer:
          Filter: programmer? --> map: salary --> accumulate: max, 0.
Type: [List -> Number]
  (define salary-of-highest-paid-programmer
     (lambda (records)
       (accumulate max
                   0
                   (map salary
                        (filter programmer? records))))
```

3.2.3 Partial evaluation of sequence operations

Example 3.7 (Partial evaluation of a sequence procedure – Curried `map`).

First version – Delaying the list:

```
Naive version:
Signature: c-map-proc(proc)
Purpose: Create a delayed map for 'proc'.
Type: [[T1 -> T2] -> [List(T1) -> List(T2)]]
(define c-map-proc
  (lambda (proc)
    (lambda (lst)
      (if (empty? lst)
          empty
          (cons (proc (car lst))
                ((c-map-proc proc) (cdr lst)))))
    ))

(define c-map-cube (c-map-proc (lambda (x) (* x x x))))
> (c-map-cube '(1 2 3 4))  ***c-map-proc is applied 4 times
'(1 8 27 64)

A Curried version that performs partial evaluation:
(define c-map-proc
  (lambda (proc)
    (letrec ((iter (lambda (lst)
                     (if (empty? lst)
                         empty
                         (cons (proc (car lst))
                               (iter (cdr lst)))))
               ))
        iter)
    ))

(define c-map-cube (c-map-proc (lambda (x) (* x x x))))
> (c-map-cube '(1 2 3 4))  ***c-map-proc is not applied
'(1 8 27 64)
```

Second version – Delaying the procedure; non-naive Currying:

```
Signature: c-map-list(lst)
Purpose: Create a delayed map for 'lst'.
Type: [List(T1) -> [[T1 -> T2] -> List(T2)]]
(define c-map-list
  (lambda (lst)
    (if (empty? lst)
        (lambda (proc) empty)                    ;c-map-list returns a procedure
        (let ((mapped-cdr (c-map-list (cdr lst))))  ;Inductive Currying
          (lambda (proc)
            (cons (proc (car lst))
                  (mapped-cdr proc)))))
    ))

(define c-map-1234 (c-map-list '(1 2 3 4)))
> (c-map-1234 (lambda (x)(* x x x)))
```

```
'(1 8 27 64)
```

3.2.4 Nested mappings

Loops form a conventional control structure. In functional languages, nested loops are implemented nested mappings.

Example 3.8. *Compute all permutations of the elements of a set S::*

In order to systematically solve the problem we need to take the following decisions:

1. An ADT for representing a set.
2. An ADT for representing a permutation.
3. A decision about the collection for all permutations.
4. A method for generating permutations.

The Set interface (an ADT without invariants):
Constructors: `make-set`(el_1, ..., el_n), `extend-set(el,s)`
Selectors: `set-remove-el(el,s)` – which removes `el` from `s`, `select-el(s)`
Predicates: `empty-set?(s), set?(s), equal-set?`(s_1, s_2)
Neutral element: `empty-set`
Permutation representation: A permutation will be represented using a list (heterogeneous), and the collection of permutations will be a list of permutation lists.
Computation method: We use an inductive construction, that builds a list of all permutations from the list of all permutations of a subset in which one set element is removed.

1. If S is empty – the empty list.
2. If S is not empty:
   For every member `x` of S:
   1. compute all permutations of S-x, and
   2. adjoin `x` in front of all permutations.

First we need to decide how the ADT Set is implemented. Assume that a set is implemented as a list (possibly with element repetitions, which should be ignored). First try:

```
(define permutations
  (lambda (s)
    (letrec((extend-permutations (lambda (x pers)
                                   (map (lambda (p) (cons x p))
                                        pers))))
     (if (empty-set? s)
         (list empty)                  ; list of the empty permutation
         (map
           (lambda (x)
             (extend-permutations x (permutations (set-remove-el x s))))
           s)))
    ))
> (permutations (make-set 2 5 7))
'(((2 (5 (7))) (2 (7 (5)))) ((5 (2 (7))) (5 (7 (2)))) ((7 (2 (5))) (7 (5 (2)))))
```

What happened? Every inductive step causes an additional nesting of the permutations.

```
> (permutations (list 5 7))
'(((5 (7))) ((7 (5))))
> (permutations (list 7))
'(((7)))
> (permutations empty)
'(())
```

We need to ***flatten*** the list of lists of permutations for a selected member `x`, before moving to the next element. We can do that by accumulating all lists, using the `append` list operation. The flattening of a list using accumulate with append is popular, and can be abstracted:

```
Type: [[T1 -> List(T2)]*List(T1) -> List(T2)]
(define flatmap
   (lambda (proc seq)
      (foldr append (list) (map proc seq))))
```

Recall that `foldr` is the primitive that acts as the `accumulate` procedure defined earlier. Second try:

```
(define permutations
  (lambda (s)
    (letrec((extend-permutations (lambda (x pers)
                                   (map (lambda (p) (cons x p))
                                        pers))))
     (if (empty-set? s)
         (list empty)                  ; list of the empty permutation
         (flatmap
           (lambda (x)
             (extend-permutations x (permutations (set-remove-el x s))))
           s)))
    ))
> (permutations (make-set 2 5 7))
'((2 5 7) (2 7 5) (5 2 7) (5 7 2) (7 2 5) (7 5 2))
```

Which can even be shortened into:

```
(define permutations
  (lambda (s)
    (if (empty-set? s)
        (list empty)                  ; list of the empty permutation
        (flatmap
           (lambda (x)
             (map (lambda (p) (cons x p))
                  (permutations (set-remove-el x s))))
           s))
    ))
```

In order to apply the `permutations` procedure we used a List implementation of sets:

```
(define make-set list)
(define set-remove-el
  (lambda (item sequence)
    (filter (lambda (x) (not (= x item)))
            sequence)))
(define empty-set? empty?)
```

Abstraction barrier question: Does the above implementation of `permutations` preserve the abstraction barrier of the Set ADT?
Answer: Observe the two applications of sequence operations in the code of `permutations`. The `map` application indeed performs a mapping over a list operation, since the `permutations` procedure returns a list. However, the application of `flatmap` applies a List sequence operation to the set-valued parameter `s`. Suppose that the Set implementation is not based on lists but, say, on numbers. Then every call to `permutations` of a non-empty set causes a runtime type error!

Example 3.9. *Generate a list of all triplets (i, j, i+j), such that:* $1 \leq j < i \leq n$ *(for some natural number n), and* $i + j$ *is prime.*

Approach:

1. Generate a list of pairs (i j).
2. Filter those with prime sum.
3. Create the triplets.

1. Creating the pairs:

```
For i= 1,n
   for j = 1,i-1
      (list i j)

> (map (lambda (i)
          (map (lambda (j) (list i j))
               (enumerate-interval 1 (- i 1))))
       (enumerate-interval 1 n))
```

Note: n is free. For example:

```
> (map (lambda (i)
           (map (lambda (j) (list i j))
                (enumerate-interval 1 (- i 1))))
        (enumerate-interval 1 5))
(()
 ((2 1))
 ((3 1) (3 2))
 ((4 1) (4 2) (4 3))
 ((5 1) (5 2) (5 3) (5 4)))
```

To remove the extra parentheses: Use `flatmap` to accumulate by append, starting from ().

```
 (accumulate append
             (list)
             (map (lambda (i)
                    (map (lambda (j) (list i j))
                         (enumerate-interval 1 (- i 1))))
                  (enumerate-interval 1 n)))
```

Note: n is free. For example:

```
 > (accumulate append
               (list)
               (map (lambda (i)
                      (map (lambda (j) (list i j))
                           (enumerate-interval 1 (- i 1))))
                    (enumerate-interval 1 5)))
 ((2 1)
  (3 1)
  (3 2)
  (4 1)
  (4 2)
  (4 3)
  (5 1)
  (5 2)
  (5 3)
  (5 4))
```

2. Filter the pairs with a prime sum – The filter predicate:

```
(define prime-sum?
  (lambda (pair)
     (prime? (+ (car pair) (cadr pair)))))

> (prime-sum? (list 3 6))
#f
> (prime-sum? (list 3 4))
#t
```

3. Make the triplets:

```
(define make-pair-sum
  (lambda (pair)
    (list (car pair) (cadr pair) (+ (car pair) (cadr pair)))))
```

The overall prime-sum-pairs procedure:

```
(define prime-sum-pairs
   (lambda (n)
     (map make-pair-sum
          (filter prime-sum?
                  (flatmap
                    (lambda (i)
                       (map (lambda (j) (list i j))
                            (enumerate-interval 1 (- i 1))))
                    (enumerate-interval 1 n))
          )))
```

For example:

```
> (prime-sum-pairs 5)
((2 1 3) (3 2 5) (4 1 5) (4 3 7) (5 2 7))
```

Chapter 4

Delayed Evaluation on Data and on Control (Behavior)

This chapter presents two methods of using delayed evaluation: *Lazy lists*, which use a delayed evaluation on possibly infinite data, and *continuation passing style*, which uses delayed evaluation for specification of future computations.

4.1 Lazy Lists (ML Sequences, Scheme Streams)

Based on:

1. Paulson: Chapter 5.12 – 5.16,
2. SICP: 3.5

Lazy lists (streams in Scheme, or sequences in ML), are lists whose elements are not explicitly computed. When working with a lazy operational semantics (normal order substitution or environment model), all lists are lazy. However, when working with an eager operational semantics (applicative order substitution), all lists are not lazy: Whenever a list constructor applies, it computes the full list.
In Scheme:

– `(cons head tail)` – means that both `head` and `tail` are already evaluated.

– `(list el1, ..., el2)` – means that the `eli`-s are already evaluated.

– `(append list1 list2)` – means that `list1` and `list2` are already evaluated.

– `(map f lst)` – means that `lst` is already evaluated.

Therefore, in eager operational semantics, lazy lists must be defined as a new datatype, and be implemented in a way that enforces ***delaying*** the computation of their elements. The ***unique*** delaying mechanism in Scheme is wrapping the delayed computation as a closure: `(lambda () <expression>)`

Lazy lists can support very big and even infinite sequences. Input lazy lists can support high level real-time programming – modeling and applying abstract concepts to input that is being read (produced). They provide a natural way for handling infinite series in mathematics.

Lazy lists are a special feature of functional programming. They are easy to implement in functional languages due to the first class status of high order functions: Creation at run time.

While working with lazy (possibly infinite) lists, we can view, at every moment, only a finite part of the data. Therefore, when designing a recursive function, we are not worried about termination – the function always terminates because the list is not computed. Instead, we should make sure that **every finite part of the result can be produced in finite time**.

Lazy lists remove the space inefficiency that characterizes sequence operations. We have seen that sequence manipulation allows for powerful sequence abstractions using the `Sequence` interface. But, sequence manipulation requires large space due to the creation of intermediate sequences. Sometime, large sequences are built just in order to retrieve few elements.

Compare the two equivalent procedures for summing the primes within a given interval:

1. **The standard iterative style:**

   ```
   (define sum-primes
     (lambda (a b)
       (letrec ((iter (lambda (count accum)
                        (cond ((> count b) accum)
                              ((prime? count) (iter (1+ count) (+ count accum)))
                              (else (iter (1+ count) accum)))))
                )
         (iter a 0))
       ))
   ```

   using a filter procedure `prime?`

2. **Using sequence operations:**

   ```
   (define sum-primes
     (lambda (a b)
   ```

```
         (accumulate +
                   0
                   (filter prime?
                           (enumerate-interval a b)))))
```

The first function interleaves the `prime?` test with the summation, and creates no intermediate sequence. The second procedure first produces the sequence of integers from `a` to `b`, then produces the filtered sequence, and only then accumulates the primes. Consider (do not try!):

```
- head(tail(filter(prime?,
                   enumerate_interval(10000, 1000000))
```

In order to find the second prime that is greater than 10000 we construct: The list of integers between 10000 and 1000000, and the list of all primes between 10000 and 1000000, instead of just finding 2 primes!!!

Lazy lists provide:

– Simplicity of sequence operations.

– Low cost in terms of space.

– Ability to manipulate large and infinite sequences.

4.1.1 The Lazy List (ML Sequence, Scheme Stream) Abstract Data Type

Main idea: The sequence is not fully computed. The tail of the list is wrapped within a closure, and therefore not evaluated. We have seen this idea earlier: Whenever we need to delay a computation, we wrap the delayed expression within a closure, that includes the necessary environment for evaluation, and yet prevents the evaluation. This is a special feature of languages that support run time generated closures.

The lazy list datatype is called ***stream*** in Scheme (***sequence*** in ML). Its values are either the empty list `'()` or a pair of any value and a ***delayed*** lazy list. The delay is created by a "closure wrap":

Definition: The lazy lists type `LzL` is a recursively defined set of the following values

1. `'empty-lzl` ∈ `LzL`
2. for every scheme-type value `v`, and a lazy list `lzl`, ⟨ `v,` ⟨ `closure () lzl` ⟩⟩ ∈ `LzL`

that is the set `LzL` is defined as follows:
`LzL = { 'empty-lzl }` ∪ `Scheme-type*[Empty -> LzL]`
The type expression for defining lazy lists:
Homogeneous lazy lists: `LzL(T) = { empty-lzl } | T*[Empty -> LzL(T)]`
Heterogeneous lazy lists: `LzL = { empty-lzl } | T*[Empty -> LzL]`

Using lambda abstraction for delaying evaluation creates a ***promise:*** the unevaluated part of the list. Some examples for created "promises":

```
> (define l0 (list))
> (define l1 (cons 1 (lambda () l0)) )
> (define l2 (cons 2 (lambda () l1)))
> l0
'()
> l1
'(1 . #<procedure>)
> ((cdr l1))
'()
> l2
'(2 . #<procedure>)
> ((cdr l2))
'(1 . #<procedure>)
```

In general, the values of lazy lists are constructed recursively, by providing a current value and a computation to be delayed:

```
;Signature: interegers-from(n)
;Type: [N -> N*[Empty -> LzL(N)]] (= LzL(N) )
> (define integers-from
    (lambda (n)
      (cons n (lambda () (integers-from (add1 n))))
      ))
> (define ints (integers-from 0))
> ints
'(0 . #<procedure>)
> ((cdr ints))
'(1 . #<procedure>)
> ((cdr ((cdr ints))))
'(2 . #<procedure>)
```

Note: The tail is always wrapped within a function, and the recursion has no basis.

Lazy lists are usually big or infinite, and therefore are not explicitly created. Rather, they are implicitly created, by recursive functions. Since it is hard to explicitly manage these lambda wrappings and exposure, we develop a set of lazy list operations, by analogy with lists.

4.1.1.1 The LzL type interface (operations)

The Lazy-list type (ADT) has the following interface:
`empty-lzl, cons-lzl, head, tail, empty-lzl?`

Implementation of the interface:

Value-constructors:

```
(define empty-lzl empty)

Signature: cons-lzl(x,lzl)
Type: [T*LzL -> LzL]
Pre-condition: lzl=empty-lzl or <closure () Lazy-list>
(define cons-lzl cons)
```

Question: What happens if `cons-lzl` is given a non-LzL 2nd argument?

Head and tail:

```
Signature: head(lz-ist)
Type: [LzL -> T]   ;that is, the type is [T*[Empty -> LzL] -> T]
Pre-condition: lzl-lst is not empty: (not (empty-lzl? lzl-lst)))
(define head car)
```

The `tail` of a lazy list is a parameter less function. Therefore, to inspect the tail, apply the tail function. The application forces evaluation of the tail.

```
Signature: tail(lz-ist)
Type: [LzL -> T]   ;that is, the type is [T*[Empty -> LzL] -> T]
Pre-condition: lzl-lst is not empty: (not (empty-lzl? lzl-lst)))
(define tail
  (lambda (lz-lst)
      ((cdr lz-lst))
    ))

> (head l1)
1
> (tail l1)
'()
> (tail l2)
'(1 . #<procedure>)
> (tail (tail ints))
'(2 . #<procedure>)
> (car (tail (tail ints)))
2
```

Predicate:

```
Signature: empty-lzl?(lz-ist)
Type: [T -> Boolean]
(define empty-lzl? empty?)
```

The first n elements of a lazy-list and the nth element selectors:

```
Signature: take(lz-lst,n)
Type: [LzL*Number -> List]
Comment: If n > length(lz-lst) then the result is lz-lst as a List
(define take
  (lambda (lz-lst n)
    (if (or (= n 0) (empty-lzl? lz-lst))
        empty-lzl
        (cons (head lz-lst)
              (take (tail lz-lst) (sub1 n))))
    ))
> (take ints 5)
'(0 1 2 3 4)
> (take (integers-from 30) 7)
'(30 31 32 33 34 35 36)
> (take (cons-lzl 1
          (lambda () (cons-lzl 2
                       (lambda () (cons-lzl 3
                                    (lambda () empty-lzl))))))
        3)
'(1 2 3)
> (take (cons-lzl 1
          (lambda () (cons-lzl 2
                       (lambda () (cons-lzl 3
                                    (lambda () empty-lzl))))))
        6)
'(1 2 3)
```

```
Signature: nth(lz-lst,n)
Type: [LzL*Number -> T]
Pre-condition: n < length(lz-lst)
(define nth
  (lambda (lz-lst n)
    (if (= n 0)
        (head lz-lst)
        (nth (tail lz-lst) (sub1 n)))
    ))
>(nth ints 25)
25
```

Evaluation of `(take (integers-from 30) 2)` **– using the substitution model:**

```
  applicative-eval[(take (integers-from  30) 2)] ==>
        applicative-eval[(integers-from 30) ] ==>              ; eval step
                applicative- eval[ (cons 30  (lambda () (integers-from (+ 30 1))))]
                                  ==> (30 . (lambda () (integers-from (+ 30 1))))
  applicative-eval
       [(30 . (take ((lambda () (integers-from (+ 30 1))))  (- 2 1))) ] ==>
                                                      ; substitute, reduce
       applicative-eval[ (take ((lambda () (integers-from (+ 30 1)))) (- 2 1))]
                                                      ==>      ; eval step
          applicative-eval[ (lambda () (integers-from (+ 30 1))) ] ==>
                                                               ; eval step
          applicative-eval[ (integers-from (+ 30 1)) ] ==>
                                                              ;reduce step
          applicative-eval[ (cons 31 (lambda () (integers-from (+ 31 1)))) ]
                          ==> ( 31 . (lambda () (integers-from (+ 31 1))))
       applicative-eval
         [ (31 . (take ((lambda ()  (integers-from (+ 31 1)))) (- 1 1))) ] ==>
                                                              ;reduce step
            applicative-eval
                 [ (take ((lambda () (integers-from (+ 31 1)))) (- 1 1)) ] ==>
                                                                  ;eval step
              applicative-eval[ (lambda () (integers-from (+ 31 1))) ]
                                                          ==>   ;eval step
              applicative-eval[ (integers-from (+ 31 1)) ] ==>
                                                              ;reduce step
              applicative-eval[ (32 . (lambda () (integers-from (+ 32 1)))) ]
                          ==> (32 . (lambda () (integers-from (+ 32 1))))
            '() ==>
       (31 . ()) ==>
  (30 . (31 . ())) = (30 31)
```

Notes:

1. The third element of the `(integers-from 30)` list, 32, is computed, although not requested!

2. A repeated computation, say `(take (integers-from 30) 7)`, repeats all the inspection steps of the lazy-list.

4.1.2 Integer lazy lists

The infinite lazy list of ones:

```
(define ones (cons-lzl 1 (lambda () ones)))
>(take ones 7)
'(1 1 1 1 1 1 1)
> (nth ones 10)
1
```

Example 4.1 (The infinite lazy-lists of factorials and Fibonacci numbers).

The infinite lazy-list of integer factorials can be defined using the factorial procedure:

```
(define fact
  (lambda (n) (if (= n 1)
                  1
                  (* n (fact (- n 1))))))

Type: [Number -> LzL(Number)]
(define facts-from
  (lambda (k)
    (letrec ((helper
              (lambda (n fact-n)
                (cons-lzl fact-n (lambda () (helper (add1 n) (* (add1 n) fact-n)))))
             ))
      (helper k (fact k)))))

(define facts-from-3 (facts-from 3))
> (take facts-from-3 6)
'(6 24 120 720 5040 40320)
```

The idea is to define a computation that keeps track of pairs of an integer and its factorial value.

An alternative approach, that internally builds the lazy-list of all integer factorials, does not rely on an external factorial procedure:

```
(define facts
  (letrec ((factgen
            (lambda (n fact-n)
              (cons-lzl fact-n (lambda () (factgen (add1 n) (* (add1 n) fact-n)))))
```

```
          ))
    (factgen 1 1)))
> (take facts 5)
'(1 2 6 24 120)
```

A similar approach can be used to build the lazy-list of all Fibonacci numbers:

```
(define fibs
  (letrec ((fibgen
           (lambda (a b)
             (cons-lzl a (lambda () (fibgen b (+ a b)))))
          ))
    (fibgen 0 1)))
> (take fibs 7)
'(0 1 1 2 3 5 8)
```

Note that the body of the delayed tail of a lazy list must be an application of a function that constructs a lazy list.

4.1.3 Elementary Lazy List Processing

Procedures that construct lazy lists by manipulation of other lazy lists, usually have the form

```
(lambda (lz-lst ...)
   .... (cons (...)
              (lambda () "application of some procedures on the tail(s) of the
                          input lazy-list(s)")))
```

Example 4.2 (Applying `square` to a lazy list).

```
Signature: squares(lz-list)
Type: [LzL(Number) -> LzL(number)]
(define squares
  (lambda (lz-lst)
    (if (empty-lzl? lz-lst)
        lz-lst
        (cons-lzl (let ((h (head lz-lst)))
                    (* h h))
                  (lambda () (squares (tail lz-lst)))))
    ))

> (take (squares ints) 7)
'(0 1 4 9 16 25 36)
```

Example 4.3 (Lazy list addition).

```
Signature: lz-lst-add(lz1,lz2)
Type: [LzL(Number)*LzL(Number) -> LzL(number)]
(define lz-lst-add
  (lambda (lz1 lz2)
    (cond ((empty-lzl? lz1) lz2)
          ((empty-lzl? lz2) lz1)
          (else
           (cons-lzl (+ (head lz1) (head lz2))
                 (lambda () (lz-lst-add (tail lz1) (tail lz2))))))
    ))

; definition of the integers using lazy-list addition:
(define integers (cons-lzl 0 (lambda () (lz-lst-add ones integers))))
> (take integers 7)
'(0 1 2 3 4 5 6)
> (take (lz-lst-add (integers-from 100) (squares integers)) 7)
'(100 102 106 112 120 130 142)

;; definition of the Fibonacci numbers using lazy-list addition
(define fib-numbers
  (cons 0
        (lambda () (cons 1 (lambda () (lz-lst-add (tail fib-numbers) fib-numbers))))
        ))
> (take fib-numbers 7)
'(0 1 1 2 3 5 8)
```

Example 4.4 (Lazy list `append (interleave)`).

Regular lists append is defined by:

```
(define append
  (lambda (l1 l2)
    (if (empty? l1)
      l2
      (cons (car l1)
            (append (cdr l1) l2)))))
```

Trying to write an analogous lazy-list-append yields:

```
Signature: lz-lst-append(lz1,lz2)
Type: [LzL*LzL -> LzL]
```

```
(define lz-lst-append
  (lambda (lz1 lz2)
    (if (empty-lzl? lz1)
        lz2
        (cons-lzl (head lz1)
              (lambda () (lz-lst-append (tail lz1) lz2))))
    ))
```

The problem: Observing the elements of the appended list, we see that all elements of the first lazy-list come before the second lazy-list. What if the first list is infinite? There is no way to reach the second list. So, this version does not satisfy the natural property of lazy-list functions: Every finite part of the lazy-list "depends" on at most a finite part of the lazy-list.

Therefore, when dealing with possibly infinite lists, append is replaced by an interleaving function, that interleaves the elements of lazy-lists in a way that guarantees that every element of the lazy-lists is reached within finite time:

```
Signature: interleave(lz1,lz2)
Type: [LzL*LzL -> LzL]
(define interleave
  (lambda (lz1 lz2)
    (if (empty-lzl? lz1)
        lz2
        (cons-lzl (head lz1)
              (lambda () (interleave lz2 (tail lz1)))))
    ))
> (take (lz-lst-append (integers-from 100) fibs) 7)
'(100 101 102 103 104 105 106)
> (take (interleave (integers-from 100) fibs) 7)
'(100 0 101 1 102 1 103)
```

4.1.4 High Order Lazy-List Functions

High order list functions, like `map` and `filter`, need their duals for lazy-lists (why they cannot directly apply?). When applied to lazy lists, they take and return lazy-lists as parameters and returned values.

```
Signature: lz-lst-map(f,lz)
Type: [[T1 -> T2]*LzL(T1) -> LzL(T2)]
(define lz-lst-map
  (lambda (f lz)
    (if (empty-lzl? lz)
        lz
        (cons-lzl (f (head lz))
              (lambda () (lz-lst-map f (tail lz)))))
    ))
> (take (lz-lst-map (lambda (x) (* x x)) ints) 5)
'(0 1 4 9 16)

Signature: lz-lst-filter(p,lz)
Type: [[T1 -> T2]*LzL(T1) -> LzL(T1)]
(define lz-lst-filter
  (lambda (p lz)
    (cond ((empty-lzl? lz) lz)
          ((p (head lz)) (cons-lzl (head lz) (lambda () (lz-lst-filter p (tail lz)))))
          (else (lz-lst-filter p (tail lz))))
    ))

(define divisible?
  (lambda (x y)
     (= (remainder x y) 0)))
(define no-sevens (lz-lst-filter (lambda (x) (not (divisible? x 7))) ints))
> (nth no-sevens 100)  ;The 100th integer not divisible by 7:
117
```

Concrete lazy-list mappings and filters:

```
; lazy-list scaling:
Signature: lz-lst-scale(c,lz)
Type: [Number*LzL(Number) -> LzL(Number)]
(define lz-lst-scale
  (lambda (c lz)
    (lz-lst-map (lambda (x) (* x c)) lz)
  ))

;The lazy-list of powers of 2:
(define double (cons-lzl 1 (lambda () (lz-lst-scale 2 double))))
> (take double 7)
'(1 2 4 8 16 32 64)

;lz-lists of lazy lists:
> (take (lz-lst-map integers-from ints) 3)
'((0 . #<procedure:...zy-lists-aux.rkt:11:12>)
  (1 . #<procedure:...zy-lists-aux.rkt:11:12>)
  (2 . #<procedure:...zy-lists-aux.rkt:11:12>))
```

Example 4.5 (Lazy-list iteration).

Recall the integers lazy-list creation function:

```
(define integers-from
  (lambda (n)
    (cons-lzl n (lambda () (integers-from (add1 n))))
    ))
```

It can be re-written as:

```
(define integers-from
  (lambda (n)
    (cons-lzl n (lambda () (integers-from (lambda (k) (add1 k)) n)))
    ))
```

A further generalization can replace the concrete function `(lambda (k) (add1 k))` by a function parameter:

```
Signature: integers-iterate(f,n)
Type: [[Number->Number]*Number -> LzL(Number)]
(define integers-iterate
  (lambda (f n)
    (cons-lzl n (lambda () (integers-iterate f (f n))))
    ))
> (take (integers-iterate add1 3) 7)
'(3 4 5 6 7 8 9)
> (take (integers-iterate (lambda (k) (* k 2)) 3) 7)
'(3 6 12 24 48 96 192)
> (take (integers-iterate (lambda (k) k) 3) 7)
'(3 3 3 3 3 3 3)
```

Example 4.6 (Primes again: two ways to define the infinite lazy-list of primes:).

Primes – First definition:

```
(define primes
  (cons-lzl 2 (lambda () (lz-lst-filter prime? (integers-from 3)))))

(define prime?
  (lambda (n)
    (letrec ((iter (lambda (lz)
                     (cond ((> (sqr (head lz)) n) #t)
                          ((divisible? n (head lz)) #f)
                          (else (iter (tail lz)))))
               ))
      (iter primes))
    ))
> (take primes 6)
'(2 3 5 7 11 13)
```

Primes – Second definition: The lazy-list of primes can be created as follows:

1. Start with the integers lazy-list: `[2,3,4,5,....]`.
2. Select the first prime: 2.
   Filter the current lazy-list from all multiples of 2: `[2,3,5,7,9,...]`
3. Select the next element on the list: 3.
   Filter the current lazy-list from all multiples of 3: `[2,3,5,6,11,13,17,...]`.
4. i-th step: Select the next element on the list: k. Surely it is a prime, since it is not a multiplication of any smaller integer.
   Filter the current lazy-list from all multiples of k.
5. All elements of the resulting lazy-list are primes, and all primes are in the resulting lazy-list.

```
Signature: sieve(lz)
Type: [LzL(Number) -> LzL(Number)]
(define sieve
  (lambda (lz)
    (cons-lzl (head lz)
          (lambda ()
            (sieve (lz-lst-filter (lambda (x) (not (divisible? x (head lz))))
                                  (tail lz)))))
    ))

(define primes1 (sieve (integers-from 2)))
> (take primes1 7)
'(2 3 5 7 11 13 17)
```

4.2 Continuation Passing Style (CPS) Programming

Continuation Passing Style is a programming method that assumes that every user defined procedure `f$` carries a ***continuation***, which is a ***future computation specification*** `cont`, in the form of a procedure, that needs to apply once the computation of `f$` ends.

Since a CPS procedure carries a future computation, they are written as iterative procedures: The "bottom" action is directly applied, and all depending actions are postponed to the continuation. A CPS procedure has one of the following formats:

1. `(continuation (primitive-procedure ....))`
2. `(CSP-user-procedure .... continuation)`
3. A conditional with the above alternatives.

Example 4.7. *The procedures*

```
(define square (lambda (x) (* x x)))
(define add1 (lambda (x) (+ x 1)))
```

turn into:

```
(define square$
   (lambda (x cont) (cont (* x x)))

(define add1$
   (lambda (x cont) (cont (+ x 1))))
```

Notation: A CPS version of a procedure `proc` is conventionally named `proc$`.

Example 4.8. *The procedure:*

```
(define h
   (lambda (x) (add1 (+ x 1))))
```

turns into:

```
(define h$
   (lambda (x cont) (add1$ (+ x 1) cont)))
```

The solution used in `square, add1`, of applying the continuation to the body, does not work for `h` because we have to pass a continuation to `add1$`! Note that once in CPS, all user defined procedures are usually written in CPS.

Example 4.9. *Nested applications – The procedure:*

```
(define h1
   (lambda (x) (square (add1 (+ x 1)))))
```

turns into:

```
(define h1$
   (lambda (x cont)
     (add1$ (+ x 1) (lambda (add1-res) (square$ add1-res cont)))
   ))
```

What happened?
Since `(add1 (+ x 1))` is the first computation to occur, we must pass `add1$` the future computation, which is the application of `square$` to the value of `(add1$ (+ x 1))`. Once `square$` is applied, it is only left to apply the given future `cont`.

Example 4.10. *Determining evaluation order – The procedure:*

```
(define h2
   (lambda (x y)(mult (square x) (add1 y)))))
```

where `mult` is:

```
(define mult
  (lambda (x y) (* x y)))
```

turns into:

```
(define h2$
   (lambda (x y cont)
      (square$ x
          (lambda (square-res)
             (add1$ y
                 (lambda (add1-res) (mult$ square-res add1-res cont)))))
   ))
```

or into:

```
(define h2$
   (lambda (x y cont)
      (add1$ y
          (lambda (add1-res)
             (square$ x
                 (lambda (square-res) (mult$ square-res add1-res cont)))))
   ))
```

where `mult$` is:

```
(define mult$
  (lambda (x y cont) (cont (* x y))))
```

Why?
Because we need to split the body of `h2` into a single computation that is given a future continuation. Since Scheme does not specify the order of argument evaluation we can select either `(square x)` or `(add1 y)` as the first computation to happen. The rest is pushed into the continuation.

What is the relationship between `h2` to `h2$`?

Claim 4.1. For every numbers `x, y`, and a continuation procedure `cont`:
`(h2$ x y cont) = (cont (h2 x y))`

Proof.

```
applicative-eval[ (h2$ x y cont) ] ==>*
applicative-eval[ (square$ x
                    (lambda (square-res)
                       (add1$ y
                           (lambda (add1-res) (mult$ square-res add1-res cont))))) ] ==>*
applicative-eval[ ((lambda (square-res)
                    (add1$ y (lambda (add1-res) (mult$ square-res add1-res cont))))
                  (* x x)) ] ==>*
applicative-eval[ (add1$ y (lambda (add1-res) (mult$ x*x add1-res cont))) ] ==>*
applicative-eval[ ((lambda (add1-res) (mult$ x*x add1-res cont)) (+ y 1)) ] ==>*
applicative-eval[ (mult$ x*x y+1 cont) ] ==>*
applicative-eval[ (cont (* x*x y+1)) ] =
     since applicative-eval[ (* x*x y+1) ] = applicative-eval[ (h2 x y) ]
applicative-eval[ (cont (h2 x y)) ]
```

□

The general relationship between a procedure to its CPS version is defined as follows:

Def 4.1. A procedure `f$` is ***CPS equivalent*** to a procedure `f`, if for every input value `x1, x2, ..., xn`, $n \geq 0$, and a continuation `cont`, `(f$ x1 ... xn cont) = (cont (f x1 ... xn))`

CPS is useful for various computation tasks. We concentrate on two such tasks:

1. Turning a recursive process into an iterative one.
2. Controlling multiple alternative future computations: Errors (exceptions), search, and backtracking.

4.2.1 Recursive to Iterative CPS Transformations

Example 4.11. ***Factorial*** – *Consider the recursive factorial procedure:*

```
(define fact
   (lambda (n)
      (if (= n 0)
          1
          (* n (fact (- n 1))))))
```

An evaluation of its application relies on a recursion management mechanism of the evaluation algorithm (e.g., `applicative-eval`), that "knows" to ***delay a computation*** until a recursive call evaluation is finished. The delayed computation is a ***future computation*** for a recursive call. The management of future computations due to recursive procedure calls is usually performed by holding a stack of ***procedure activation frames*** that keep track of the delayed computations. Every procedure call opens a new activation frame. Figure 4.1 show the procedure call stack for `(fact 3)`.

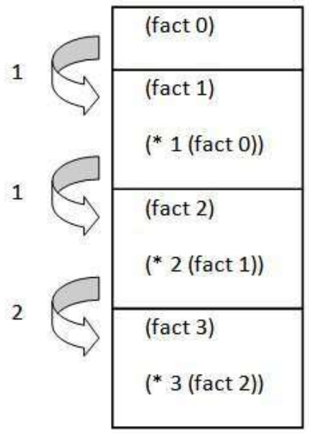


Figure 4.1: Stack of procedure activation frames for (fact 3)

The recursion control mechanism can be replaced by ***continuation management***, where an activation frame is replaced by a future (continuation) computation, provided to the recursive application. The recursive call `(* n (fact (- n 1)))` whose computation opens a new frame for `(fact (- n 1))` with the delayed `(* n result)` computation in the current frame, is replaced by the recursive call:

```
   (fact$ (- n 1)
          (lambda (result) (* n result)))
```

The calling frame is replaced by a continuation. But – the frames are built one of top of the otehr, So – the same should apply for the continuation. Therefore, the continuation to `(fact (- n 1))` should apply the previous continuatiuon:

```
    (fact$ (- n 1)
            (lambda (result) (cont (* n result))))
```

The top frame, that folds back all the open frames is replaced by application of the delayed computation to its result: ((cont 1)). Overall, the `fact` procedure turns into a `fact$` procedure, with body: 1 → turns into `(cont 1)`
or
`(* n (fact (- n 1)))` → turns into

```
    (fact$ (- n 1)
            (lambda (res) (cont (* n res))))
```

and altogether:

```
(define fact$
    (lambda (n cont)
       (if (= n 0)
           (cont 1)
           (fact$ (- n 1) (lambda (res) (cont (* n res)))))
    ))
```

Clearly, a fact$ computation creates an iterative process.

Claim 4.2. `fact$` is CPS equivalent to `fact`. That is, for every $n \geq 0$ and continuation procedure `cont`, `(fact$ n cont) = (cont (fact n))`.

Proof. Since `fact` is a recursive procedure, the proof is by induction (straightforward expansion does not terminate).
Induction base: $n = 0$.

```
applicative-eval[ (fact$ 0 cont) ] ==>*
applicative-eval[ (cont 1) ] =
applicative-eval[ (cont (fact 0)) ]
```

Inductive hypothesis: $n = k \geq 0$ Assume that the hypothesis holds for every integer $i \leq k$.
Inductive step: $n = k + 1 > 0$

```
applicative-eval[ (fact$ n cont) ] ==>*
applicative-eval[ (fact$ (- n 1) (lambda (res) (cont (* n res)))) ] ==>*
    by the inductive hypothesis,
applicative-eval[ ((lambda (res) (cont (* n res))) (fact (- n 1))) ] ==>*
```

```
applicative-eval[ (cont (* n (fact (- n 1)))) ] =
    since applicative-eval[ (* n (fact (- n 1))) ] = applicative-eval[ (fact  n) ]
applicative-eval[ (cont (fact  n)) ]
```

□

What continuations are constructed during the computation and how and when they are applied?
Intuitively we understand that the deeper we get into the recursion, the longer is the continuation. We demonstrate the sequence of procedure calls:

```
(fact$ 3 (lambda (x) x))
==>
(fact$ 2 (lambda (res)
            ( (lambda (x) x)
              (* 3 res))))
==>
(fact$ 1 (lambda (res)
            ( (lambda (res)
                 ( (lambda (x) x) (* 3 res)))
              (* 2 res))))
==>
(fact$ 0 (lambda (res)
            ( (lambda (res)
                 ( (lambda (res)
                      ( (lambda (x) x) (* 3 res)))
                   (* 2 res)))
              (* 1 res))))
==>
( (lambda (res)
     ( (lambda (res)
          ( (lambda (res)
               ( (lambda (x) x) (* 3 res)))
            (* 2 res)))
       (* 1 res)))
  1)
==>
( (lambda (res)
     ( (lambda (res)
          ( (lambda (x) x) (* 3 res)))
       (* 2 res)))
  1)
```

```
==>
( (lambda (res)
     ( (lambda (x) x) (* 3 res)))
  2)
==>
( (lambda (x) x) 6)
==>
6
```

We see that the procedure creates an iterative process – requires constant space on the function call stack.

Guidelines for CPS transformation for turning a recursive process into an iterative one:

1. Look for a ***deepest*** call to a user defined procedure that is not the last evaluation to compute.
2. Turn it into the body of the CPS procedure, and ***fold*** all later computations into the future continuation.
3. If no future computations for a deepest expression: Apply the continuation to the expression.

In many cases providing an iterative procedure is not straightforward. For example in search problems on hierarchical structures:

```
 (define sum-odd-squares
   (lambda (tree)
     (cond ((null? tree) 0)
           ((not (list? tree))
            (if (odd? tree) (square tree) 0))
           (else (+ (sum-odd-squares (car tree))
                    (sum-odd-squares (cdr tree)))))))
```

An iterative version is not immediate because of the deep unbounded hierarchy. In such cases the CPS transformation that creates an iterative version is helpful.
Observation: The continuations grow! While the function call stack keeps a constant size for calls of `fact$`, the size of the closure values grows with the recursive calls! The stack space is traded for the closure value space.

Example 4.12. ***Ackermann*** *function:*

```
(define ackermann
    (lambda (a b)
      (cond ( (zero? a) (+ 1 b))
            ( (zero? b) (ackermann (- a 1) 1))
            (else (ackermann (- a 1) (ackermann a (- b 1)))))
    ))
```

The function creates a tree recursive process. A CPS-equivalent iterative version is constructed along the same guidelines:

1. Identify innermost expression.
2. If it is not an application of a user procedure: Apply the continuation on the expression.
3. If it is an application of a user procedure – pass the remaining computation as a continuation.

```
(define ackermann$
    (lambda (a b cont)
      (cond ( (zero? a) (cont (+ 1 b)))
            ( (zero? b) (ackermann$ (- a 1) 1 cont))
            (else (ackermann$ a (- b 1)
                       (lambda (res) (ackermann$ (- a 1) res cont)))))
    ))
```

Example 4.13. ***map*** *function:*

```
(define map
    (lambda (f list)
       (if (null? list)
           list
           (cons (f (car list))
                 (map f (cdr list))))
    ))
```

This procedure includes two user-procedure calls, nested within a cons application. Therefore, the process is not iterative. In order to transform it into a CPS-equivalent iterative version we need to select an expression that does not include nested user procedure applications, and postpone the rest of the computation to the future continuation. The two nested user procedure calls appear in the arguments of the `cons` application. We can select either of them, thus receiving two CPS versions:

```
 (define map$
    (lambda (f$ list cont)
       (if (null? list)
           (cont list)
```

```
          (f$ (car list)
              (lambda (f-res)
                (map$ f$
                      (cdr list)
                      (lambda (map-res)
                         (cont (cons f-res map-res)))))))
      ))

  (define map$
    (lambda (f$ list cont)
      (if (null? list)
          (cont list)
          (map$ f$
                (cdr list)
                (lambda (map-res)
                   (f$ (car list)
                       (lambda (f-res)
                          (cont (cons f-res map-res)))))))
      ))
```

The ***to-CPS*** transformations above are done intuitively, by ***observing*** a deepest user-procedure call and delaying other computations to the continuation. Based on the ***head and tail position*** analysis, introduced in Chapter **??**, this analysis can be formalized (and automated), so that an expression can be ***proved*** to create iterative processes. Moreover, the transformation itself can be automated. Having an iterative process automated identifier, a compiler (interpreter) can be ***tail recursive*** – can identify iterative expressions and evaluate them using bounded space.

4.2.2 Controlling Multiple Alternative Future Computations: Errors (Exceptions), Search and Backtracking

These are the applications that actually gain from the CPS style: Multiple clients, with different future computations can use an independent CPS-based service. Instead of modifying the code of a service for achieving different future computations, the client hands the future computations to the service. The responsibility of the client is to provide future computations, and the responsibility of the server is to correctly apply the given future continuations. The benefit for the client is that it needs not bother to properly apply the future continuations, while the benefit for the server is that it needs not bother to modify its code for each future continuation.

This usage of CPS reminds the "inversion of control" design pattern of Fowler `http://martinfowler.com/bliki/InversionOfControl.html`: Instead of a client that holds a server and complements its code with application of desirable future computations, we have a client that provides future computations to a server, which is responsible for their proper applications.

Example 4.14. *Replace a call to `error` by a fail continuation:*

An error (exception) marks an alternative, not planned future. Errors and exceptions break the computation (like a `goto` or `break` in an imperative language). A call to the Scheme primitive procedure `error` breaks the computation and returns no value. This is a major problem to the type system.

In the CPS style errors can be implemented by continuations. Such a CPS procedure carries two continuations, one for the planned future – the ***success*** continuation, and one for the error – the ***fail*** continuation.

```
Signature: sumlist(li)
Purpose: Sum the elements of a number list. If the list includes a non
            number element -- produce an error.
Type: [List -> Number union ???]
   (define sumlist
      (lambda (li)
        (cond ((null? li) 0)
              ((not (number? (car li))) (error "non numeric value!"))
              (else
                (+ (car li) (sumlist (cdr li)))))
       ))
```

An iterative CPS version, that uses ***success/fail*** continuations:

```
(define sumlist
   (lambda (li)
     (letrec
       ((sumlist$
          (lambda (li succ-cont fail-cont)
            (cond ((null? li) (succ-cont 0))      ;; end of list
                  ((number? (car li))  ;; non-end, car is numeric
                   (sumlist$ (cdr li)
                             (lambda (sum-cdr)    ;success continuation
                                (succ-cont (+ (car li) sum-cdr)))
                             fail-cont))             ;fail continuation
                  (else (fail-cont))))       ;apply the fail continuation
       ))
       (sumlist$ li
                 (lambda (x) x)
                 (lambda ( ) (display "non numeric value!"))))
   ))
```

Note that while the success continuation is gradually built along the computation – constructing the ***stored*** future actions, the fail continuation is not constructed. When applied, it discards the success continuation.

Example 4.15. *Using a fail continuation in search.*

In this example, the fail continuation is used to direct the search along a tree. If the search on some part of the tree fails, the fail continuation applies the search to another part of the tree.

The tree in this example is an ***unlabeled tree***, that is, a tree whose branches are unlabeled, and whose leaves are labeled by atomic values. This is the standard view of an heterogeneous list. For example, `((3 4) 5 1)` is a list representation of such a tree.
Tree interface:
Constructors: `make-tree(1st,2nd, ...), add-subtree(first,rest), make-leaf(data), empty-tree`;
Selectors: `first-subtree(tree), rest-subtree(tree), leaf-date(leaf)`;
Predicates: `composite-tree?(t), leaf?(t), empty-tree?(t)`;
with the obvious preconditions for the constructors and the selectors. Implementation appears at the end of the example.

The CPS version includes a success and a fail continuations. In the search decision point, when the search is turned to the first sub-tree, the fail continuation that is passed is the search in the rest of the sub-trees. The fail continuation is applied when the search reaches a leaf and fails.

```
Signature: leftmost-even(tree)
Purpose: Find the left most even leaf of an unlabeled tree whose leaves are
          labeled by numbers. If none, return #f.
Type:  [List -> Number union Boolean]
Examples: (leftmost-even '((1 2) (3 4 5))) ==> 2
           (leftmost-even '((1 1) (3 3) 5)) ==> #f
(define leftmost-even
  (lambda (tree)
       (leftmost-even$ tree (lambda (x) x) (lambda ( ) #f))
  ))

(define leftmost-even$
  (lambda (tree succ-cont fail-cont)
     (cond ((empty-tree? tree) (fail-cont))
           ((leaf? tree)
               (if (even? (leaf-data tree))
                   (succ-cont tree)
                   (fail-cont)))
           (else                            ; Composite tree
               (leftmost-even$
                       (first-subtree tree)
                       succ-cont
                       (lambda () (leftmost-even$ (rest-subtrees tree)       ;  (*)
                                                  succ-cont
                                                  fail-cont)))))
  ))
```

The `leftmost-even` procedure performs an exhaustive search on the tree, until an even leaf is found. Whenever the search in the first sub-tree fails, it invokes a recursive search on the rest of the sub-trees. This kind of search can be viewed as a ***backtracking*** search policy: If the decision to search in the first sub-tree appears wrong, a retreat to the decision point occurs, and an alternative route is selected.

Note that the fail continuation that is passed to the fail continuation that is constructed in the decision point (marked by *) is the fail continuation that is passed to `leftmost-even$` as an argument. To understand that think about the decision points:

– If the search in `(first-subtree tree)` succeeds, then the future is `succ-cont`.

– If it fails, then the future is the search in `(rest-subtrees tree)`.

– If the search in `(rest-subtrees tree)` succeeds, the future is `succ-cont`.

– If it fails, then the future is `fail-cont`.

Example of a search trace:

```
(leftmost-even ((1 2) (3 4) 5)) ==>
(leftmost-even$ ((1 2) (3 4) 5) (lambda (x) x) (lambda () #f)) ==>
(leftmost-even$ (1 2) (lambda (x) x)
              (lambda ()
                (iter$ ((3 4))
                       (lambda (x) x)
                       (lambda () #f)))) ==>
(leftmost-even$ 1 (lambda (x) x)
          (lambda ()
            (iter$ (2)
                   (lambda (x) x)
                   (lambda ()
                     (iter$ ((3 4) 5)
                            (lambda (x) x)
                            (lambda () #f)))))) ==>*
```

```
(leftmost-even$ (2)
        (lambda (x) x)
        (lambda ()
          (iter$ ((3 4) 5)
                 (lambda (x) x)
                 (lambda () #f)))) ==>*
( (lambda (x) x) 2) ==>
 2
```

Implementation of the unlabeled-tree interface as a heterogeneous list:

```
(define make-tree list)
(define add-subtree cons)
(define make-leaf (lambda (d) d))
(define empty-tree empty)

(define first-subtree car)
(define rest-subtrees cdr)
(define leaf-data (lambda (x) x))

(define composite-tree? pair?)
(define leaf?
  (lambda (t) (not (list? t))))
(define empty-tree? empty?)

(define tree (make-tree (make-tree (make-leaf 3) (make-leaf 4))
                        (make-leaf 5)
                        (make-leaf 1)))
>tree
'((3 4) 5 1)
```

We use the printed form of heterogeneous lists for describing unlabeled trees as well.

Suppose that a different client wishes to have the original tree, in case that a an even leaf does not exists:

```
(define leftmost-even
  (lambda (tree)
      (leftmost-even$ tree (lambda (x) x) (lambda ( ) tree))
  ))
```

The service (suppolier) code is not affected. Only the calling arguments.

At last, a non-CPS version:

```
(define leftmost-even
  (lambda (tree)
    (letrec
      ((iter (lambda (tree)
               (cond ((empty-tree? tree) #f)
                     ((leaf? tree)
                      (if (even? (leaf-data tree)) (leaf-data tree) #f))
                     (else
                       (let ((res-first (iter (first-subtree tree))))
                         (if res-first
                             res-first
                             (iter (rest-subtrees tree))))))))
      ))
      (iter tree))
   ))
```

Example 4.16. *Using a success continuation for reconstructing a hierarchy.*

In this example, the success/fail continuations are used for reconstructing the original hierarchical structure, after replacing an old leaf by a new one. In this example the CPS style ***simplifies*** the implementation, and enables easy revision of future continuations. Therefore, we ***start with*** a CPS version. Then show the more complex and less readable non-CPS version.

A CPS version:

```
Signature: replace-leftmost(old new tree)
Purpose: Find the left most leaf  whose value is 'old' and replace it
          by new. If none, return #f.
Type:  [T1*T2*List -> List union Boolean]
Examples: (replace-leftmost 3 1 '((2 2) (4 3 2 (2)) 3) ) ==>
                                 ((2 2) (4 1 2 (2)) 3)
          (replace-leftmost 2 1 '((1 1) (3 3))) ==> #f
(define replace-leftmost
  (lambda (old new tree)
     (replace-leftmost$ tree (lambda (x) x) (lambda() #f) )
   ))

(define replace-leftmost$
   (lambda (tree succ-cont fail-cont)
       (cond ((empty? tree) (fail-cont))
             ((leaf? tree)
              (if (eq? (leaf-data tree) old)
                  (succ-cont (make-leaf new))
                  (fail-cont)))
            (else                        ; Composite tree
              (replace-leftmost$
                             (first-subtree tree)
                             (lambda (first-res)
                               (succ-cont (add-subtree first-res (rest-subtrees tree))))
                             (lambda ()
                               (replace-leftmost$
                                 (rest-subtrees tree)
                                 (lambda (rest-res)
                                   (succ-cont
                                      (add-subtree (first-subtree tree) rest-res)))
                                 fail-cont)))) )
  ))
```

Explanation: For a composite tree, apply the search on its left sub-tree:

- The success continuation:
  1. Combines the resulting already replaced sub-tree with the rest sub-trees, and then
  2. Applies the given success continuation.
- The fail continuation:
  1. Applies the search to the rest sub-trees. For this search:
     (a) The success continuation combines the first sub-tree with the resulting already replaced rest sub-trees, and then
     (b) Applies the original success continuation.
  2. The fail continuation is the originally given fail continuation.

Question: Suppose that in case that `old` is not found we are asked to return the original tree. This is a more reasonable request, as the default of “replace the leftmost occurrence of”. How should the `replace-leftmost` procedure be change?
To answer that, we observe that the CPS procedure does not make any assumption about the future, success or fail, continuations. So, the answer lies in the “futures” provided to the CPS procedure: in case of success, do nothing (the identity procedure), but in case of failure, return the given tree:

```
(define replace-leftmost
  (lambda (old new tree)
      (replace-leftmost$ tree (lambda (x) x) (lambda() tree) ))    ;the only change!
   ))
```

A non-CPS version: The non-CPS version searches recursively along the tree.

1. If a replacement in a sub-tree is successful, then the result should be combined with the rest of the tree.
2. If a replacement in a sub-tree fails, then
   (a) If the replacement in the rest of the sub-tree is successful, the sub-trees should be combined.
   (b) Otherwise, the replacement fails.

Therefor, this version faces the problem of marking whether a search was successful ***and*** returns the result of the replacement. That is, the internal procedure has to return two pieces of information:

- The replaced structure
- A sign of whether the replacement was successful.

Therefore, the internal `iter` procedure returns a pair of the new structure and a boolean flag marking success or failure.

```
(define replace-leftmost
  (lambda (old new tree)
    (letrec ((combine-tree-flag cons)
             (get-tree car)
             (get-flag cdr)
             (iter
                (lambda (tree flag)
                  (cond ((empty-tree? tree) (combine-tree-flag tree flag))
                        ((leaf? tree)
                          (if (and (not flag) (eq? (leaf-data tree) old))
                              (combine-tree-flag (make-leaf new) #t)
                              (combine-tree-flag tree flag)))
                        (else
                         (let ((new-first-flagged (iter (first-subtree tree) flag)))
                           (if (get-flag new-first-flagged)
                               (combine-tree-flag
                                   (add-subtree (get-tree new-first-flagged)
                                                (rest-subtrees tree))
```

```
                             #t)
                         (let
                          ((new-rest-flagged (iter (rest-subtrees tree) flag)))
                            (combine-tree-flag
                              (add-subtree (first-subtree tree)
                                    (get-tree new-rest-flagged))
                              (get-flag new-rest-flagged)))))))
      ))
(let ( (replace-result (iter tree #f)) )
  (if (get-flag replace-result)
      (get-tree replace-result)
      #f)) )))
```

Chapter 5

Type Correctness

Based on Krishnamurthi [8] chapters 24-26, Friedman and Wand [3] chapter 7.

Contracts of programs provide specification for their most important properties: Signature, type, preconditions and postconditions. It says nothing about the implementation (such as performance).

Program correctness deals with proving that a program implementation satisfies its contract:

1. ***Type correctness***: Check well-typing of all expressions, and possibly infer missing types.

2. ***Program verification***: Show that if preconditions hold, then the program terminates, and the postconditions hold (the Design by Contract philosophy).

Program correctness can be checked either statically or dynamically. In ***static program correctness*** the program text is analyzed without running it. It is intended to reveal problems that characterize the program independently of specific data. Static type checking verifies that the program will not encounter run time errors due to type mismatch problems. In ***dynamic program correctness***, problems are detected by running the program on specific data. Static correctness methods are considered strong since they analyze the program as a whole, and do not require program application. Dynamic correctness methods, like unit testing, are complementary to the static ones.

This chapter presents two algorithms for static type inference:

1. An inference algorithm based on type inference rules. This algorithm is non-deterministic and its implementation is complex, as it involves cumbersome data structures and non-deterministic decisions.

2. An inference algorithm based on solving type equations. This algorithm is an outcome of the first algorithm. It is deterministic, involves simple data structures, and simple

to implement. Chapter 3 presents an implementation of this algorithm, emphasizing the importance of ***abstract data types*** in software.

5.1 What is Type Checking/Inference?

Most programming languages are ***typed***, i.e., the sets of their computed values are split into subsets, termed ***types***, that collect together values of a similar kind. In the part of Scheme that we study:

$$Computed\_values = \{Numbers,\ Booleans,\ Symbols,\ Procedures,\ Tuples\}$$

where
$Numbers = \{1, 2, 5.1, -3, ...\}$
$Booleans = \{\#t, \#f\}$
$Symbols = \{a,\ ab1,\ moshe,\ ...\}$
$Procedures$ = set of all primitive procedures and closures over values, which is internally split into 1-ary closures from numbers to numbers, 2-ary closures from number pairs to closures, etc.
$Tuples$ = set of all tuples of values, which is internally split into pairs of numbers, pairs of closures, triplets, quadruples of values, etc.

Once a language uses computations in known domains like Arithmetics or Boolean logic, its semantics admits types. Most programming languages admit ***fully typed*** semantics, i.e., every computed value belongs to a known type. Some languages, though, have ***semi-typed*** semantics, i.e., they allow typeless values. Such are, for example, languages of web applications that manage semi-structured data bases. Only theoretical computation languages, like Lambda Calculus and Pure Logic Programming have untyped semantics. These languages do not include built-in domains.

The closures computed by a typed language are not defined over all values. The application of a primitive procedure or a closure to wrong arguments create a computation error, termed ***run-time*** errors. Correct applications are characterized by ***well typing*** rules, that dictate correct combinations of types.
The basic well typing rule – correct closure application: A closure is defined only on values in its domain and returns values only in its range.
If types are inter-related, then well typing includes additional rules. For example, if type ***Integer*** is a subtype of (included in) type ***Real*** which is a subtype of type ***Complex***, then a closure from ***Real*** to ***Integer*** can apply also to ***Integer*** values, and its result values can be the input to a function defined on type ***Real***.

Type checking/inference involves association of program expressions with types. The intention is to associate an expression e with type t, such that evaluation of e yields values in t. This way the evaluator that runs the program can check the well typing conditions before actually performing bad applications.

> The purpose of type checking/inference is to guarantee ***type safety***, i.e., predict well typing problems and prevent computation when well typing is violated.

Specification of type information about language expressions:

Type checking/inference requires that the language syntax includes type information. Many programming languages have ***fully-typed*** syntax, i.e., the language syntax requires full specification of types for all language constructs. In such languages, all constants and variables (including procedure and function variables) must be provided with full typing information. Such are, for example, the languages Java, C, C++, Pascal[1].

Some languages have a ***partially-typed*** syntax, i.e., programs do not necessarily associate their constructs with types. The Scheme and Prolog languages do not include any typing information. The ML language allows for partially-typed specifications. In such languages, typing information might arise from built-in typing of language primitives. For example, in Scheme, number constants have the built-in `Number` typing, and the symbol "`-`" has the built-in Procedure type `[Number*Number -> Number]`.

If a language syntax does not include types, and there are no built-in primitives with built-in types, then the language has an ***untyped*** semantics. Pure Logic Programming is an untyped language.

Static/dynamic type checking/inference:
Type checking/inference is performed by an algorithm that uses type information within the program code for checking well typing conditions. If the type checking algorithm is based only on the program code, then it can be applied off-line, without actually running the program. This is ***static type checking/inference***. Such an algorithm uses the known semantics of language constructs for statically checking well typing conditions. A weaker version of a type checking algorithm requires concrete data for checking well typing, usually based on ***type tagging*** of values. Therefore, they require an actual program run, and the type checking is done at runtime. This is ***dynamic type checking***. Clearly, static type checking is preferable over dynamic type checking.

The programming languages Pascal, C, C++, Java that have a fully typed syntax, have static type checking algorithms. The ML language, that allows for partial type specification, has static ***type inference*** algorithms. That is, based on partial type information provided for primitives and in the code, the algorithm statically infers type information, and checks for well typing. The Scheme and Prolog languages, that have no type information in their syntax, have only dynamic typing.

Properties of type checking/inference algorithms:
The goal of a type checking/inference algorithm is to detect all violations of well typing. A

[1] Java generics and C++ templates provide a restricted form of type variables.

type checking algorithm that detects all such violations is termed ***strong***. Otherwise, it is ***weak***. The difficulty is, of course, to design *efficient, static, strong* type checking algorithms. Static type checkers need to follow the operational semantics of the language, in order to determine types of expressions in a program, and check for well typing. The type checker of the C language is known to be weak, due to pointer arithmetics.

How types are specified? – the Type Language:
Specification of typing information in programs and within type checking algorithms requires a ***language for writing types***. A type specification language has to specify atomic types and composite types. Some languages allow for ***User defined types***, in addition to the types that are built-in (like primitives) in the language. This is possible in all object-oriented languages, where every class defines a new type. ML also allows for user defined types. It is not possible in Scheme (without structs).
Language expressions vs. type expressions: The programming language and the type language are two different languages, and the aim is to assign a type expression to every language expression. Language expressions include ***value constructors*** and type expressions include ***type constructors***. Value constructors create values of types, and type constructors create composite types.
Examples of value constructors in Scheme: Number symbols (the symbol 3 creates the value 3), boolean symbols, `lambda`.
Examples of type constructors in the type language introduced below: `*` that constructs the Tuple type and `->` that constructs the Procedure type.

In object-oriented languages, where every class has an associated type, class constructors act as value constructors for the class type – create objects (instances) of the class, while class declarations act as type constructors.
Type polymorphism: Some programming languages include expressions that evaluate or create values from multiple types. Such expressions are termed ***polymorphic***. For example, in object-oriented languages, Class-hierarchy implies subtyping relationships between class types, and assign multiple types to attributes and methods within a class hierarchy. Functional languages (like Scheme and ML), support polymorphic procedures, i.e., procedures having multiple types.

In order to describe the multiple types of a polymorphic language expression, the type language includes ***type variables***. Type expressions that include type variables describe multiple types, and are termed ***polymorphic type expressions***. The type of a polymorphic language expression is specified by a polymorphic type expressions.

Summary of terms:

1. **Programming language properties:** ***Typed*** language semantics; ***Semi-typed*** language semantics; ***Well typing*** rules; ***Fully/partially*** typed syntax.

2. **Type language properties:** ***Atomic/composite*** types; ***User defined*** types; ***Polymorphic*** type-expressions, ***value/type constructors***.
3. **Type correctness mechanisms:** ***Static/dynamic type checking***; ***Type inference***; ***Strong/weak*** type mechanism; ***Type safety***.

5.2 The Type Language

The formal type language, which was introduced in Chapter 2, includes the atomic types Number, Boolean, Symbol, Void, the empty type Empty, and the composite types Procedure, Pair, List. Union types are not included in the formal specification. Moreover, Void is not an input parameter type for Procedure types, and Empty is not an output parameter type. The type inference methods that are introduced in this chapter are restricted to infer only these types. The restrictions are:

1. Type inference does not apply to language expressions that return union types (like conditionals), procedures defined on the `void` value of the `Void` type, or procedures with no returned value.
2. Type specification does not enable user-defined types. In particular, there are no recursive types (apart for the built-in ones).

Type variables and Type polymorphism:
Scheme expressions that do not include primitives do not have a specified type. Such expressions can yield, at runtime, values of different types, based on the types of their variables. We have already seen that procedures like the identity procedure `(lambda (x) x)` can be applied to values of any type and returns a value of the input type:

```
> ( (lambda (x) x) 3)
3
> ( (lambda (x) x) #t)
#t
> ( (lambda (x) x) (lambda (x) (+ x 1)))
#<procedure:x>
```

Therefore, the identity procedure has multiple types in these applications:
In the first: `[Number -> Number]`
In the 2nd: `[Boolean -> Boolean]`
In the 3rd: `[[Number -> Number] -> [Number -> Number]]`
Type variables provide the necessary abstraction: The well-typing rules enable ***substitution*** (replacement) of type variables by other type expressions. Here are some type expressions that describe the types of procedures:

1. The type expression for the identity procedure is `[T -> T]`.
2. The type expression for `(lambda (f x)(f x))` is `[[T1 -> T2]*T1 -> T2]`.
3. The type expression for `(lambda (f x)((f x) x))` is
   `[[T1 -> [T1 -> T2]]*T1 -> T2]`.

Type expressions that include type variables are called ***polymorphic type-expressions***, and are constructed using ***polymorphic type-constructors***. Language expressions whose type is polymorphic are called ***polymorphic language-expressions***.

BNF grammar of the type language for the inference algorithms:

```
Type ->  'Void' | Non-void
Non-void -> Atomic | Composite | Type-variable
Atomic -> 'Number' | 'Boolean' | 'Symbol'
Composite -> Procedure | Pair | List
Procedure -> '[' Tuple '->' Type ']'
Tuple -> (Non-void '*' )* Non-void | 'Empty'
Pair -> 'Pair' '(' Non-void ',' Non-void ')'
List -> 'List' '(' Non-void ')' | 'List'
Type-variable -> A symbol starting with an upper case letter
```

Value constructors and type constructors: We already know that value constructors are functions that construct values of types, based on given input values. Type constructors are functions that construct types, based on given input types. In our type language, the type constructor for Procedure types is `->`, the type constructor for Pair types is `Pair`, and the type constructor for List types is `List`. For atomic types, the type is a type constructor of a single value – itself.

Note the analogy with value constructors: The value constructor for procedure values is `lambda`, for Pair and List values it is `cons` and the empty list `empty` is a value constructor for itself. Similarly, values of atomic types, like numbers and symbols are constructors of themselves.

5.3 Type Substitutions and Unifiers

Checking well-typing of polymorphic type-expressions requires precise definition of correct replacement of type variables by type expressions. It involves concepts that we have already met in the formal definition of the operational semantics: ***Substitution, Composition of substitutions, the Substitute operation***, and requires ***Renaming***. We also add the notions of ***Type-unification, Type-unifier*** and ***Type-most-general-unifier*** (***Type-mgu***).

Definition: A ***type-substitution*** s is a mapping from a finite set of type variables to a finite set of type expressions, such that $s(T)$ ***does not include*** T. A ***type-binding*** is a pair $\langle T, s(T)\rangle$ such that $T = s(T)$.

Substitutions are written using set notions: `{T1=Number, T2=[[Number->T3]->T3]}`. `{T1=Number, T2=[[Number->T3]->T2]}` is illegal.

Definition: The ***application of a type-substitution*** s to a type expression τ, denoted $\tau \circ s$ (or just τs), consistently replaces all occurrences of type variables T in τ by their mapped type expressions $s(T)$. The replacement is ***simultaneous***.

For example,
`[[T1->T2]->T2]`$\circ$`{T1=Boolean, T2=[T3->T3]} = [[Number->[T3->T3]] -> [T3->T3]]`
We say that a type expression τ' is an ***instance*** of a type expression τ, if there is a type-substitution s such that $\tau \circ s = \tau'$. τ is ***more general*** than τ', if τ' is an instance of τ. The following type expressions are instances of `[T -> T]`:

```
[Number -> Number] = [T->T]∘{T = Number}
[Symbol -> Symbol] = [T->T]∘{T = Symbol}
[[Number->Number] -> [Number->Number]] = [T->T]∘{T = [Number->Number]}
[[Number->T1] -> [Number->T1]] = [T->T]∘{T = [Number->T1]}
```

Definition: ***Combination (composition) of type-substitutions***
The combination of type-substitutions s and s', denoted $s \circ s'$, is an operation that results a type-substitution, or fails. It is defined by:

1. s' is applied to the type-expressions of s, i.e., for every variable T' for which $s'(T')$ is defined, occurrences of T' in type expressions in s are replaced by $s'(T')$.
2. A variable T' in s', for which $s(T)$ is defined, is removed from the domain of s', i.e., $s'(T)$ is not defined on it any more.
3. The modified s' is added to s.
4. Identity bindings, i.e., $s(T) = T$, are removed.
5. If for some variable, $(s \circ s')(T)$ includes T, the combination fails.

For example,
```
{T1=Number, T2=[[Number->T3] -> T3]}∘{T3=Boolean, T1=[T2->T2]} =
{T1 = Number, T2 = [[Number->Boolean]->Boolean], T3=Boolean}
```

Definition: ***Renaming of type variables***
This is the operation of consistent renaming of type variables within a type expression, by new type symbols, that do not occur in the type expression.

Renamed type expressions are equivalent:

```
[[T1 -> T2]*T1 -> T2] ≡  [[S1 -> T2]*S1 -> T2]
[[T1 -> T2]*T1 -> T2] ≡  [[S1 -> S2]*S1 -> S2]
```

The variables in the substituting expressions should be new. For example, the following renamings of `[[T1->T2]*T1 -> T2]` are illegal:

```
[[T1->T2]*S2 -> T2]
[[T2->T2]*T2 -> T2]
[[[T1->T2]->T2]*[T1->T2] -> T2]
```

Unification of type expressions is an operation that makes type expressions identical by application of a type substitution. For example:
`[S*[Number->S]->S]` ∘ `{S=Pair(T1), T2=[Number->S], T3=Pair(T1)}` =
`[Pair(T1)*T2->T3]` ∘ `{S=Pair(T1), T2=[Number->S], T3=Pair(T1)}` =
`[Pair(T1)*[Number->Pair(T1)]->Pair(T1)]`
Therefore, `{S=Pair(T1), T2=[Number->S], T3=Pair(T1)}` is a ***unifier*** for these type expressions.

Definition: *Unifier of type expressions*
A unifier of type expressions τ_1, τ_2 is a type substitution s such that $\tau_1 \circ s = \tau_2 \circ s$.
The type expressions **should not include common type variables**! (Apply renaming, if needed.)

Example 5.1 (Unification of type expressions)**.**

1. The type expressions
   `[S*[Number->S1]->S]` and
   `[Pair(T1)*[T1->T1]->T2]`
   are unifiable by `{S=Pair(Number), T1=Number, S1=Number, T2=Pair(Number)}`
2. The type expressions
   `[S*[Number->S]->S]` and
   `[Pair(T1)*[T1->T1]->T2]`
   are not unifiable!

Unifiable type expressions can be unified by multiple unifiers. For example, the type expressions `[S*S->S]` and `[Pair(T1)*T2->T2]` are unifiable by the unifiers
`{S=Pair(T1), T2=Pair(T1)}`,
`{S=Pair(Number), T2=Pair(Number)}`,
`{S=Pair(Boolean), T2=Pair(Boolean)}`, etc.
The first unifier is the ***most general unifier*** (***mgu***), since it substitutes only the necessary type variables. All other unifiers are obtained from it by application of additional substitutions. The most general unifier is unique, up to consistent renaming.

5.4 Static Type Inference for Scheme

A type checking/inference algorithm checks/infers correctness of types of program expressions. Its goal is to guarantee ***type safety***, i.e., to infer type T for an expression `e`, in case that the application of the operational semantics algorithm to `e` completes its computation, and results a value in type T. If the application of the operational semantics does not complete safely, the type inference algorithm should not infer a type, and moreover, point to the problem. That is, the algorithm should be ***type safe***.

The algorithm proves that *the type of a given expression* `e` *is* `t`*, under assumptions about types of variables in* `e`. For example, type inference for the expression `(+ x 5)` needs to state that provided that the type of `x` is Number, the type of the expression is also Number. It should not prove that under no assumptions, the type of this expression is Number.

The typing system is introduced gradually. First, we introduce a typing system for a restricted language that includes atomic expressions with numbers, booleans, primitive procedures, and variables, and composite expressions with `quote` forms, `lambda` forms and application forms. Then we extend the basic system for typing conditionals and for typing in presence of definitions, including definitions of recursive procedures. At last, we introduce the type-equations based inference algorithm.

5.4.1 Typing Statements

A typing statement is an assertion of the form:

If the type of language variables in a language expression `e` is ⟨type-assumptions⟩
Then the type of `e` is ⟨type-conclusion⟩.

The typing statement terminology consists of the ***type-environment*** concept which states "type-assumptions" and its ***extension*** operation, and the ***typing-statement*** concept and its ***instantiation*** and ***unification*** operations.

Type environment: A ***type environment*** is a substitution of language variables by type expressions, i.e., a mapping of a finite set of variables to type expressions. It is denoted as a set of variable type assumptions. For example,

```
{x:Number, y:[Number --> T]}
```

is a type environment, in which the variable `x` is mapped to the Number type, and the variable `y` is mapped to the polymorphic procedure type `[Number -> T]`.

1. The ***type of a variable*** `v` with respect to a type environment `Tenv` is denoted `Tenv(v)`.
2. The ***empty type environment***, denoted { }, stands for no assumptions about types of variables.

3. **Extending a type environment**: Assume that we wish to extend the above type environment with a type assumption about the type of variable `z`: `{z:Boolean}`. This is performed by the ***substitution composition***:
   `{x:Number, y:[Number->T]}` $\circ$ `{z:Boolean}` $=$ `{x:Number, y:[Number->T], z:Boolean}`.
   Recall that the empty substitution is the neutral element of the substitution-composition operation: `{ }`$\circ$`{x1:T1, ..., xn:Tn} = {x1:T1, ..., xn:Tn}`.

4. **Application of a type-substitution to a type environment:** For a type environment `Tenv` and a type-substitution s, the application of s to `Tenv`, denoted `Tenv` $\circ$ s, is the application of s to all type expressions in `Tenv`. For example:
   `{x:[S1*S2->S1], y:[Number->S2]}` $\circ$ `{S1=Pair(T), S2=[Number->T], S3=Pair(T)}`
   $=$
   `{x:[Pair(T)*[Number->Pair(T)]->Pair(T)], y:[Number->[Number->T]]}`

Typing statement: A ***typing statement*** is a ***true/false formula*** that states a judgment about the type of an expression, given a type environment.

Notation: `Tenv |- e:t`
It states: For every type-substitution s, if `Tenv` $\circ$ s holds then `e` has type `t` $\circ$ s.

For example, the typing statement

```
{x:Number} |- (+ x 5):Number
```

states that under the assumption that the type of `x` is Number, the type of `(+ x 5)` is Number. The typing statement

```
{f:[T1 --> T2], g:T1} |- (f g):T2
```

states that for every consistent replacement of `T1, T2`, under the assumption that the type of `f` is `[T1 -> T2]`, and the type of `g` is `T1`, the type of `(f g)` is `T2`.

The following typing statements are false:

```
{f:[T1 --> T2]} |- (f g):T2
```

This is false because having no type assumption on `g`, `(f g)` might not satisfy the well-typing rules of Scheme, and create a runtime error. The typing statement

```
{f:[Empty --> T2], g:T2} |- (f g):T2
```

is false because based on the operational semantics of Scheme, if `f` is a parameter-less procedure, the expression `(f g)` does not satisfy the well-typing rules of Scheme.

1. **Instantiation of typing statements:** An ***instance of a typing statement*** TS is a typing statement TS' that results from the application of a type-substitution s to all type expressions in TS: $TS' = TS \circ s$.
   A typing statement implies all of its instances. For example, the (false) typing statement
   `{f:[T1 -> T2], g:T2} |- (f g):T2`
   implies the (false) instances:
   `{f:[T1 -> Boolean], g:Boolean} |- (f g):Boolean` and
   `{f:[Number->[Number->Boolean]],g:[Number->Boolean]} |- (f g):[Number->Boolean]`

2. **Unifiable typing statements:** Typing statements TS and TS' are ***unifiable*** if there exists a type substitution s such that $TS \circ s = TS' \circ s$. TS and TS' should not include common type variables: **Unification must be preceded by renaming!**

   Example 5.2 (Unification of typing statements)**.**

   (a) The typing statements
   `{f:[T1 -> T2], g:T1} |- (f g):T2` and
   `{f:[Number -> Number], g:T3} |- (f g):Number`
   are unifiable by `{T1=Number, T2=Number, T3=Number}`

   (b) The typing statements
   `{f:[T1 -> T2], g:T1} |- (f g):T2` and
   `{f:T3, g:Pair(Number)} |- (f g):T4`
   are unifiable by `{T3=[Pair(Number)->T4], T1=Pair(Number), T2=T4}`

   (c) The typing statements
   `{f:[T1 -> T2], g:T1} |- (f g):T2` and
   `{f:T3, g:Pair(T3)} |- (f g):T4`
   are not unifiable!

   Similarly to type expressions, unifiable typing statements can be unified by multiple unifiers. The ***most general unifier*** (***mgu***), is the one that substitutes only the necessary type variables. All other unifiers are obtained from it by application of additional substitutions.

   Unification of typing statements can be extended to sets of typing statements, where the intention is that the sets include pairwise unifiable typing statements.

3. **Extending the type environment in a typing statement:** By definition of its meaning, if a typing statement
   $\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash e : \tau$ holds, then any extension:
   $\{x_1 : \tau_1, \ldots, x_n : \tau_n, y : \rho\} \vdash e : \tau$ also holds.

5.4.2 Static type inference for a restricted Scheme language

We start with a Scheme subset that is defined by the following grammar (no conditionals, no definitions):

```
<scheme-exp>       -> <exp>
<exp>              -> <atomic> | <composite>
<atomic>           -> <number> | <boolean> | <variable>
<composite>        -> <special> | <form>
<number>           -> Numbers
<boolean>          -> '#t' | '#f'
<variable>         -> Restricted sequences of letters, digits, punctuation marks
<special>          -> <lambda> | <quote>
<form>             -> '(' <exp>+ ')'
<lambda>           -> '(' 'lambda' '(' <variable>* ')' <exp>+ ')'
<quote>            -> '(' 'quote' <variable> ')'
```

In order to provide a type checking/inference system, we need to formulate the ***well typing*** rules for the language. We introduce only well typing rules intended to guarantee that procedure can be applied to arguments. The rules depend on the operational semantics. They consist of ***typing axioms***, which take care of language primitives and of variables, and ***typing rules***, which reflect the operational semantics of composite expressions.

Well typing rules for the restricted language:

```
Typing axiom Number:
    For every type environment _Tenv and number _n:
          _Tenv |- _n:Number

Typing axiom Boolean:
    For every type environment _Tenv and boolean _b:
          _Tenv |- _b:Boolean

Typing axiom Variable:
    For every type environment _Tenv and variable _v:
           _Tenv |- _v:Tenv(v)
    i.e., the type statement for _v is the type that _Tenv assigns to it.

Typing axioms Primitive procedure:
    Every primitive procedure has its own function type.
    Examples:
    The + procedure has the typing axiom:
         For every type environment _Tenv:
           _Tenv |- +:[Number* ... *Number -> Number]
    The not procedure has the typing axiom:
         For every type environment _Tenv:
           _Tenv |- not:[_S -> Boolean]
         _S is a type variable. That is, not is a polymorphic
         primitive procedure.
    The display procedure has the typing axiom:
         For every type environment _Tenv:
           _Tenv |- display:[_S -> Void]
         _S is a type variable. That is, display is a polymorphic
         primitive procedure.
```

Typing axiom *Symbol*:

```
    For every type environment _Tenv and a syntactically legal sequence of
    characters _s:
          _Tenv |- (quote _s):Symbol
```

Typing rule *Procedure*:

```
    For every: type environment _Tenv,
               variables _x1, ..., _xn, n ≥ 0
               expressions _e1, ..., _em, m ≥ 1, and
               type expressions _S1, ...,_Sn, _U1, ...,_Um :
  Procedure with parameters (n > 0):
    If     _Tenv∘{_x1:_S1, ..., _xn:_Sn } |- _ei:_Ui for
    all i = 1..m ,
    Then   _Tenv |- (lambda (_x1 ... _xn ) _e1 ... _em) : [_S1*...*_Sn -> _Um]
  Parameter-less Procedure (n = 0):
    If     _Tenv |- _ei:_Ui for all i=1..m,
    Then   _Tenv |- (lambda ( ) _e1 ... _em):[Empty -> _Um]
```

Typing rule *Application*:

```
    For every: type environment _Tenv,
               expressions _f, _e1, ..., _en, n ≥ 0 , and
               type expressions _S1, ..., _Sn, _S:
  Procedure with parameters (n > 0):
    If     _Tenv |- _f:[_S1*...*_Sn -> _S],
           _Tenv |- _e1:_S1, ..., _Tenv |- _en:_Sn
    Then   _Tenv |- (_f _e1 ... _en):_S
  Parameter-less Procedure (n = 0):
    If     _Tenv |- _f:[Empty -> _S]
    Then   _Tenv |- (_f):_S

Typing rule Monotonicity:
    For every: type environments _Tenv, _Tenv',
               expression _e and type expression _S:
    If     _Tenv |- _e:_S,
    Then   _Tenv∘_Tenv' |- e:_S
```

Notes:

1. **Meta-variables:** The typing axioms and rules include ***meta-variables*** for language expressions, type expressions and type environments. When axioms are instantiated or rules are applied, the meta-variables are replaced by real expressions of the same kind. The meta-variables should not be confused with language or type variables. Therefore, they deliberately are preceded with "_", so to distinguish them from non-meta variables.
2. **Axiom and rule independence:** Each typing axiom and typing rule specifies an independent (stand alone), universally quantified typing statement. The meta-variables used in different rules are not related, and can be ***consistently renamed***.
3. **Identifying pattern:** Apart for the ***Application*** rule, each typing axiom or rule has an ***identifying typing statement pattern***. That is, each axiom or rule is characterized by a different typing statement pattern.
   For example, the identifying pattern of the ***Number*** rule is `_Tenv |- _n:Number`; the identifying pattern of the ***Procedure*** rule is `_Tenv |- (lambda (_x1 ... _xn) _e1 ... _em):[_S1*...*_Sn -> _S]`. The ***Application*** rule is the only rule which is applied when all other rules/axioms do not apply.
4. **The monotonicity rule:** This rule states that typing statements stay valid under additional typing information for variables. It is needed since sometimes, different sub-expressions require different typing assumptions. In such cases, there is a need to combine typing statements that rely on different type environments. The monotonicity rule enables extension of type environments, so that the typing of all sub-expressions rely on the same type environment.
5. **Exhaustive sub-expression typing:** Every typing rule requires typing statements for all sub-expressions of the expression for which a typing statement is derived. This property guarantees ***type safety*** – the typing algorithm assign a type to every sub-expression which is evaluated at run-time.

The type inference algorithm:

Type inference is performed by considering language expressions as ***expression trees***. For example, the expression `(+ 2 (+ 5 7))` is viewed as the expression tree in Figure 5.1. The

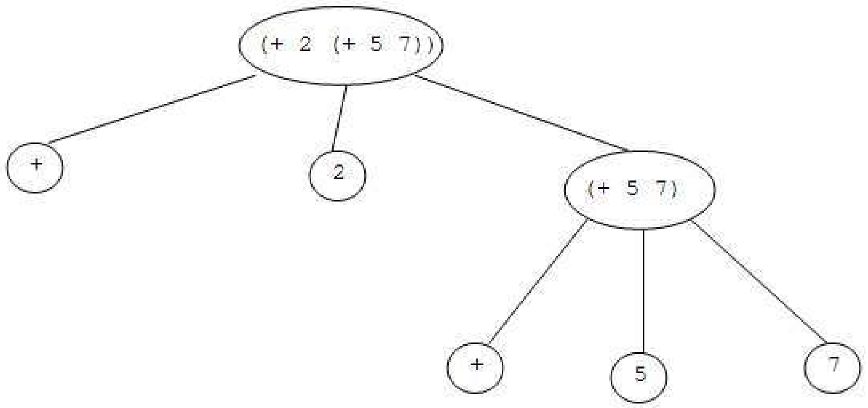


Figure 5.1

tree has the given expression as its root, leaves `7, 5, +, 2, +` and the internal node `(+ 5 7)`. The algorithm assigns a type to every sub-expression, in a bottom-up manner. The algorithm starts with ***derived typing statements*** for the leaves. These typing statements result from ***instantiation of typing axioms***. Next, the algorithm derives a typing statement for the sub-expression `(+ 5 7)`, by ***application of a typing rule*** to already derived typing statements. The algorithm terminates with a derived typing statement for the given expression.

Example 5.3. *Derive a typing statement for* `(+ 2 (+ 5 7))`.

The leaves of the tree are numbers and the primitive variable +. Typing statements for them can be obtained by instantiating typing axiom ***Number*** for the number leaves, and the typing axiom ***Primitive procedure*** for the + leaves:

```
1. { } |- 5:Number
2. { } |- 7:Number
3. { } |- 2:Number
4. { } |- +:[Number*Number -> Number]
```

Application of typing rule ***Application*** to typing statements 4,1,2, with type-substitution `{_S1=Number, _S2=Number, _S=Number}`:

```
5. { } |- (+ 5 7):Number
```

Applying typing rule ***Application*** to typing statements 4,3,5, with type-substitution `{_S1=Number, _S2=Number, _S=Number}`:

```
6. { } |- (+ 2 (+ 5 7)):Number
```

The final typing statement states that under no type assumption for variables, the type of `(+ 2 (+ 5 7))` is `Number`. When such a statement is derived, we say that `(+ 2 (+ 5 7))` is ***well typed***, and its type is `Number`.

Algorithm **Type-derivation** below infers types of Scheme expressions, based on the well typing rules. It uses the procedure **instantiate** for instantiating a typing statement, the procedure **apply-rule**, for instantiating a typing rule, and a local variable `derived-ts-pool` for collecting the intermediate derived typing statements.

Algorithm Type-derivation:
Input: A language expression `e`
Output: A pair of a typing statement for `e` (`Tenv |- e:t`) and its ***derivation*** (a sequence of typing statements), or `FAIL`
Method:

1. Consistently rename all bound variables in `e` with new fresh variables.
2. Let `derived-ts-pool` be a local variable, initialized to the empty sequence.
3. For every leaf sub-expression `l` of `e`, apart from procedure parameters:
   Find: A typing axiom A with typing statement TS, such that:
   (a) Replace the type-environment meta-variable by a type-environment expression, and the language variable by language expressions. Let TS' be the resulting typing statement.
   (b) Find a type substitution s for such that TS' is an instance of A. That is, $A \circ s = TS'$.

   do: Rename TS', number it (new serial number), and add it to `derived-ts-pool`.
4. For every non-leaf sub-expression `e'` of `e` (bottom-up order, including `e`):
   Find: A typing rule R = `If` *condition* `Then` *consequence*, such that:
   (a) Replace the meta-type-environments variables by type-environment expressions, and the language variables by language expressions. Let R'= `If` *condition'* `Then` *consequence'* be the resulting rule.
   (b) Find a set DTS of derived typing statements in `derived-ts-pool`, and a most general unifier s for *condition'* and DTS. That is, *condition'* $\circ s = DTS' \circ s$.

do: Rename the typing statement *consequence*$' \circ s$, number it (new serial number), and add it to `derived-ts-pool`.

5. If `derived-ts-pool` includes a typing statement TS for `e`, i.e., `Tenv |- e:t`,
Output = $\langle TS,$`derived-ts-pool`$\rangle$.
Otherwise, **Output** = `FAIL`.

Definition: (Well typing, type of well typed expression)

– `e` is ***well-typed*** if **Type-derivation**(e) does not fail, i.e., **Type-derivation**(e) = $\langle TS,$`derivation`$\rangle$, where `derivation` is the set of typing statements obtained during the type inference for `e`.

– `e` has type `t` if it is well typed, and **Type-derivation**(e)=$\langle$`{ } |- e:t,derivation`$\rangle$.

Heuristic guidelines: Algorithm **Type-derivation** has points of non-determinism at applications of typing axioms and rules, since there is a need to select substitutions for the meta-variables. But, since well typing requires an empty type environment, our heuristic rule is to always pick a minimal type environment. Note however that ***the type environment must include type assumptions for all free variables in the typed expression!***

1. Applications of the ***Number, Boolean, Symbol*** and ***Primitive-procedure*** axioms always use an empty type environment.

2. Applications of the ***Variable*** axiom use type environments with an assumption to the typed variable.

3. Applications of the ***Procedure*** rule use type environments that includes type assumptions only to the free variables of the lambda expressions (which are extended in the rule condition by type assumptions to the parameters).

4. Applications of the ***Application*** rule use type environments according to already derived typing statements (in `derived-ts-pool`).

Example 5.4.

Find a type for the variable `x`, i.e., apply **Type-derivation**(x): The only axiom that has instances for `x` is ***Variable***.
The axiom: For every type environment `_Tenv` and variable `_v`: `_Tenv |- _v:_Tenv(_v)`.
Replacement for `_Tenv` and `_v`:

variable	**expression**
`_Tenv`	`{x:T}`
`_v`	`x`

No further type substitution is required since the resulting typing statement `{x:T} |- x:T` is a typing statement for `x`. **result:** `x` is well-typed but does not have a type.

Example 5.5. *Application of the **Procedure** rule, with `ts-pool` that includes*

`{x1:[Number -> T], x2:Number} |- (x1 x2):T`

for deriving a typing statement for `(lambda(x2)(x1 x2))`.

The ***Procedure*** rule:

```
Typing rule Procedure:
    For every: type environment _Tenv,
               variables _x1, ..., _xn, n>=0
               expressions _e1, ..., _em, m>=1, and
               type expressions _S1,...,_Sn, _U1,...,_Um:
  Procedure with parameters:
    If     _Tenv∘{_x1:_S1, ..., _xn:_Sn} |- _bi:_Ui for all i=1..m,
    Then   _Tenv |- (lambda (_x1 ... _xn) _e1 ... _em):[_S1*...*_Sn -> _Um]
  Parameter-less Procedure:
    If     _Tenv |- _ei:_Ui for all i=1..m,
    Then   _Tenv |- (lambda ( ) _e1 ... _em):[Empty -> _Um]
```

The replacement of `_Tenv` and the language variables (for `n=m=1`) is:

variable	expression
_Tenv	{x1:[Number -> T]}
_x1	x2
_e1	(x1 x2)

Applying the type-substitution `{_S1 = Number, _U1 = T}` to the condition typing statement yields the typing statement in `ts-pool`, since `{x1:[Number -> T]}∘{x2:Number} = {x1:[Number -> T], x2:Number}`
The derived typing statement:
`{x1:[Number -> T]} |- (lambda(x2)(x1 x2)):[Number -> T]`.

Example 5.6. *Further application of the **Procedure** rule, in order for deriving a typing statement for* `(lambda(x1)(lambda(x2)(x1 x2)))`.

Again, it is the ***Procedure*** rule that applies, with `n=m=1`.
The replacement of `_Tenv` and the language variables is:

variable	expression
_Tenv	{ }
_x1	x1
_e1	(lambda (x2)(x1 x2))

Applying the type-substitution `{_S1 = [Number -> T], _U1 = [Number -> T]}` to the condition typing statement yields the typing statement that is derived in the previous example,

since `{ }∘{x1:[Number -> T]} = {x1:[Number -> T]}`
The derived typing statement:
`{ } |- (lambda (x1)(lambda (x2) (x1 x2))):[[Number -> T] -> [Number -> T]].`

Conventions: In every rule application we note:

1. The support typing statements from the ones already derived (belong to `derived-ts-pool`.
2. The involved type-substitution. We omit specification of the replacement to type environment meta-variables and to language meta variables.

Example 5.7. *Derive the type for* `((lambda(x)(+ x 3)) 5)`. *The tree structure is described in Figure 5.2:*

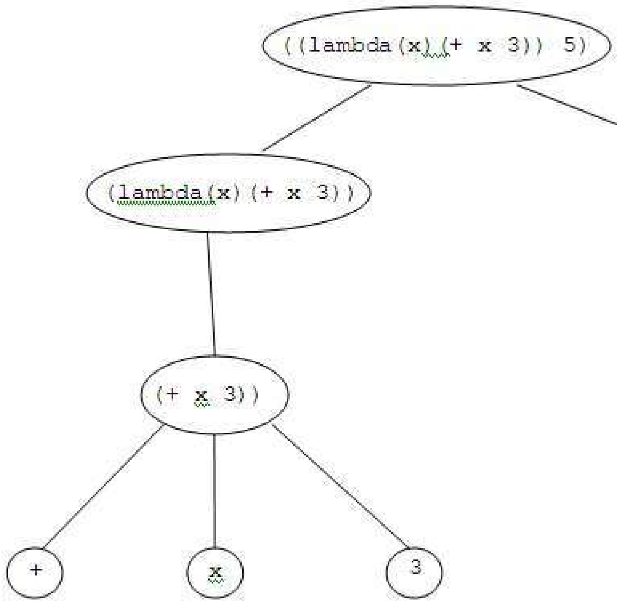


Figure 5.2

The leaves of this expression are numbers, the primitive variable `+` and the variable `x`. Typing statements for the number and `+` are obtained by instantiating the ***Number*** and the ***primitive procedure*** axioms:

```
1. { } |- 5:Number
2. { } |- 3:Number
3. { } |- +:[Number*Number -> Number]
```

A typing statement for the variable x leaf can be obtained only by instantiating the ***Variable*** axiom. Since no type is declared for x, its type is just a type variable:

```
4. {x:T} |- x:T
```

The next sub-expression for typing is `(+ x 3)`. This is an application, and is an instantiation of the identifying pattern of the ***Application*** rule. But, we need to get different instantiations of the ***Number*** and the ***Primitive procedure*** axioms, so that the type environments for all statements can produce a consistent replacement for the type environment meta-variable `_Tenv` in the ***Application*** rule. Therefore, we use the ***Monotonicity*** rule. Applying this rule to typing statements 2, 3:

```
5. {x:T1} |- 3:Number
6. {x:T2} |- +:[Number*Number -> Number]
```

Note the renaming before insertion to the derived typing statements pool. Applying typing rule ***Application*** to typing statements 6, 4, 5, with type-substitution `{_S1=T1=T2=T=number, _S2=Number, _S=Number}`:

```
7. {x:Number} |- (+ x 3):Number
```

The next expression corresponds to the pattern of the ***Procedure*** rule. It is applied to statement 7, with the type-substitution `{_S1=Number, _U1=Number}`:

```
8. { } |- (lambda (x) (+ x 3)):[Number -> Number]
```

The overall expression corresponds to the pattern of the ***Application*** typing rule. Applying this rule to statements 8, 1, with type-substitution `{_S1=Number, _S=Number}`:

```
9. { } |- ((lambda (x) (+ x 3)) 5):Number
```

Therefore, the expression `((lambda(x)(+ x 3)) 5)` is well typed, and its type is `Number`. The above sequence of typing statements is its type derivation.
Convention: We avoid explicit application of the ***monotonicity*** rule, and just mention it in the justification of rule application. For example, in the last proof, we can skip explicit derivation of typing statements (5), (6), and derive statement (7) directly from statements (2), (3), (4), while noting that in addition to ***Application*** it also involves ***Monotonicity***.

Example 5.8 (A failing type derivation). *Derive a type for* `(+ x (lambda(x) x))`*.*
Renaming: *The expression includes two leaves labeled* ***x*** *that reside in two different lexical scopes, and therefore can have different types. Renaming results* `(+ x (lambda(x1) x1))`*. The tree structure is described in Figure 5.3.*

The leaves of this expression are `x1,x,+`. Instantiating the ***Variable*** and the ***primitive procedure*** axioms:

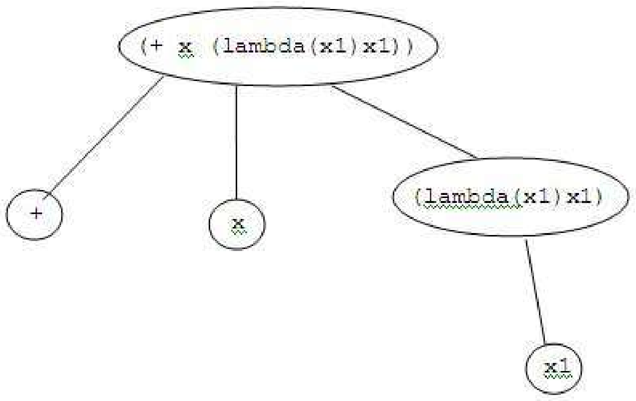


Figure 5.3

```
1. {x1:T1} |- x1:T1
2. { } |- +:[Number*Number -> Number]
3. {x:T2} |- x:T2
```

Applying typing rule ***Procedure*** to statement 1, with the type-substitution `{_S1=T1, _U1=T1}`:

```
4. { } |- (lambda (x1) x1):[T1 -> T1]
```

The only rule identifying pattern that can unify with the given expression `(+ x (lambda (x1) x1))` is that of the ***Application*** rule. But the rule cannot apply to the already derived statements since there is no type-substitution that turns the statements in the rule condition to be instances of the derived statements: For the procedure type we need the type substitution `{_S1=Number, _S2=Number, _S=Number}`; the type `T2` of the first argument can be substituted by `Number`. For the second argument `(lambda(x1) x1)`, we need `_Tenv |- (lambda(x1) x1):Number`, which is not an instance of derived statement no. 4 (no variable substitution can turn statement no. 4 into the necessary typing statement: In statement 4 the type is `[T1 -> T1]`, while the necessary type is `Number`). Therefore, `Type-derivation((+ x (lambda (x1) x1)))=FAIL`.

Example 5.9 (A failing type derivation)**.** *Derive the type for* `(lambda(x) (x x))`. *The tree structure is described in Figure 5.4.*

Note that atomic components with the same label are listed as a single leaf. This is correct since type checking is preceded by renaming, implying that multiple occurrences of a variable have the same binding (declaration), and therefore must have the same type.

Instantiating the ***Variable*** rule:

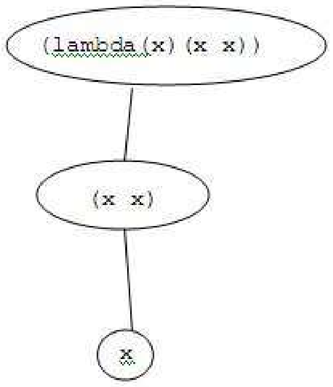


Figure 5.4

```
1. {x:T1} |-  x:T1 (the x procedure)
```

Now we want to get a typing statement for `(x x)`. This expression unifies only with the identifying pattern of the ***Application*** typing rule. In order to apply this rule with typing statements no. 1, 2 as its support, we have to find a replacement to `_Tenv` and the language variables, and a type-substitution that turn the statements in the condition of the ***Application*** rule into instances of 1:

variable	**expression**
`_Tenv`	`{x:T1}`
`_f`	`x`
`_e1`	`x`

The type-substitution is `{_S1=T1, _S=T2}`. The rule requires that if for some `_Tenv`, `_Tenv |- _e1:_S1` and also `_Tenv |- _f:[_S1 -> _S]`, then `_Tenv |- (_f _e1):_S`. But in this case, the condition on `_e1` means `{x:T1} |- x:T1` while the condition on `_f` means `{x:T1} |- x:[T1 -> T2]`, implying `T1 = [T1 -> T2]` which is an illegal type-substitution due to circularity. Therefore, no rule can be applied, and the expression is not well-typed.
Discussion: The last two examples show failed type derivations. The derivation in Example 5.8 fails because the given expression is not well typed. The derivation in Example 5.9 fails because the expression cannot be typed by a finite derivation based on the given system of typing rules. In general, there can be 3 reasons for a failing type derivation:

1. The given system is weak, i.e., some rules are missing.
2. The given expression is erroneous.
3. The given expression cannot be finitely typed.

Example 5.10. *Typing a high order procedure:*

```
(lambda (g dx)
     (lambda (x)
       (/ (- (g (+ x dx)) (g x))
          dx)))
```

The leaves of this expression are `dx`; then `g, x` from sub-expression `(g x)`; `g, +, x, dx` from sub-expression `(g (+ x dx))`; and `-, /`. We note that all repeated occurrences of variables reside in the same lexical scope, and therefore must have a single type (cannot be distinguished by renaming). Therefore, we have a single typing statement for every variable. Instantiations of the ***Variable*** axiom:

```
1. {dx:T1} |- dx:T1
2. {x:T2} |- x:T2
3. {g:T3} |- g:T3
```

Instantiations of the ***Primitive procedure*** axiom:

```
4. { } |- +:[Number*Number -> Number]
5. { } |- -:[Number*Number -> Number]
6. { } |- /:[Number*Number -> Number]
```

Typing `(g x)` – apply typing rule ***Application*** to typing statements 2,3, with type-substitution `{_S1=T2, _S=T4, T3=[T2 -> T4]}`:

```
7. {x:T2, g:[T2 -> T4]} |- (g x):T4
```

Typing `(+ x dx)` – apply typing rule ***Application*** to typing statements 4,2,1, with type-substitution `{_S1=T2=Number, _S2=T1=Number, _S=Number}`:

```
8. {x:Number, dx:Number} |- (+ x dx):Number
```

Typing `(g (+ x dx))` – apply typing rule ***Application*** to typing statements 3,8, with type-substitution `{_S1=Number, _S=T5, T3=[Number -> T5]}`:

```
9. {x:Number, dx:Number, g:[Number -> T5]} |- (g (+ x dx)):T5
```

Typing `(- (g (+ x dx)) (g x))` – apply typing rule ***Application*** to typing statements 5,9,7, with type-substitution `{_S1=T5=Number, _S2=T4=Number, _S=Number}`:

```
10. {x:Number, dx:Number, g:[Number -> Number]} |-
                                        (- (g (+ x dx)) (g x)):Number
```

Typing `(/ (- (g (+ x dx)) (g x)) dx)` – apply typing rule ***Application*** to typing statements 6,10,1, with type-substitution `{_S1=Number, _S2=T1=Number, _S=Number}`:

```
11. {x:Number, dx:Number, g:[Number -> Number]} |-
                                           (/ (- (g (+ x dx)) (g x))
                                              dx):Number
```

Typing `(lambda (x) (/ (- (g (+ x dx)) (g x)) dx))` – apply typing rule ***Procedure*** to typing statement 11, with type-substitution `{_S1=Number, _U1=Number}`:

```
12. {dx:Number, g:[Number -> Number]} |-
                                    (lambda (x)
                                      (/ (- (g (+ x dx)) (g x))
                                         dx)):[Number -> Number]
```

Typing the full expression – apply typing rule ***Procedure*** to typing statement 12, with type-substitution `{_S1=[Number -> Number], _S2=Number, _U1=[Number -> Number]}`:

```
13. { } |- (lambda (g dx)
                 (lambda (x)
                    (/ (- (g (+ x dx)) (g x))
                       dx))):
               [[Number -> Number]*Number -> [Number -> Number]]
```

Which steps use the ***monotonicity*** typing rule?

5.4.3 Adding definitions:

Consider the sequence of expressions:

```
> (define x 1)
> (define y (+ x 1))
> (+ x y)
3
```

What is the type of `x`?
What is the type of `y`?
What is the type of `(+ x y)`?
Clearly, `(+ x y)` should be well-typed, and its type is Number. The typing system described so far cannot support a proof that concludes:

```
{ } |- (+ x y):Number
```

because there is no support for deriving `{ } |- x:Number` and `{ } |- y:Number`. Therefore, we add a rule that states that if the expression in the definition is well-typed and has a type `t`, then the defined variable has this type as well:

```
Typing rule Define:
    For every definition expression (define _x _e) and type expression _S:
    If     { } |- _e:_S,
    Then   { } |- (define _x _e):Void, and
           { } |- _x:_S
Restriction: No repeated variable definitions.
```

Note that there is no derivation for the `define` expression. The reason is twofold: First, `define` expressions are not nested within other expressions, and therefore, no other expression require typing a `define` expression as its component. Second, the type of a `define` expression is known in advance: It's always `void`.

Example 5.11. *Given the definitions:*

```
> (define x 1)
> (define y (+ x 1))
```

derive a type for `(+ x y)`.

1. Deriving a type for `x`:

   ```
   1. { } |- 1:Number
   ```

   Applying the ***Define*** rule:

   ```
   2. { } |- (define x 1):Void
   3. { } |- x:Number
   ```

2. Deriving a type for `y`:

   ```
   4. { } |- +:Number*Number -> Number.
   ```

   Applying the ***Application*** typing rule to statements no. 1, 3, 4, with type-substitution `{_S1=T1=Number, _S2=T2=Number, _S=Number}`:

   ```
   5. { } |- (+ x 1):Number
   ```

   Applying the ***Define*** rule to statement 5:

```
6. { } |- (define y (+ x 1)):Void
7. { } |- y:Number
```

3. Type derivation for `(+ x y)`:
   Applying the ***Application*** typing rule to statements no. 3, 4, 7, with type substitution `{_S1=T1=Number, _S2=T2=Number, _S=Number}`:

   ```
   4.  { } |- (+ x y):Number
   ```

Example 5.12. *Given the definition:*

```
> (define deriv
    (lambda (g dx)
       (lambda (x)
           (/ (- (g (+ x dx)) (g x))
              dx))))
```

derive a type for `(deriv (lambda (x) x) 0.05)`.

1. The definition of `deriv` is well-typed since Example 5.10 presents a type derivation to the `lambda` expression. The inferred type is: `[[Number -> Number]*Number -> [Number -> Number]]`.
2. Applying the ***Define*** rule to this conclusion:

   ```
   1. { } |- (define deriv
                (lambda (g dx)
                   (lambda (x)
                       (/ (- (g (+ x dx)) (g x))
                          dx)))):Void
   2. { } |- deriv:[[Number -> Number]*Number -> [Number -> Number]]
   ```

3. Type derivation for `(deriv (lambda (x) x) 0.05)`:
   By instantiating the ***Number*** and the ***Variable*** axioms:

   ```
   3. { } |- 0.05:Number
   4. {x:T1} |- x:T1
   ```

   Applying the ***Procedure*** rule to statement 4, with type substitution `{_S1=T1, _U1=T1}`:

```
5. { } |- (lambda (x) x):[T1 -> T1]
```

Applying the ***Application*** rule to statements no. 2, 5, 3, with the type-substitution `{_S1=[Number -> Number], T1=Number, _S2=Number, _S=[Number -> Number]}`:

```
7. { } |- (deriv (lambda (x) x) 0.05):[Number -> Number]
```

The expression `(deriv (lambda (x) x) 0.05)` is well-typed and its type is `[Number -> Number]`.

5.4.4 Adding control:

Typing conditionals require addition of well-typing rules whose identifying patterns correspond to the conditional special forms. A reasonable rule might be:

```
For every type environment _Tenv,
          expressions _p, _c, _a, and
          type expression _S:
 If       _Tenv |- _p:Boolean and
          _Tenv |- _c:_S and
          _Tenv |- _a:_S
 Then     _Tenv |- (if _p _c _a):_S
```

This is the right thing to require, since it enables static typing of conditionals that infers a single type, independently of the control flow. This is, indeed, the typing rule in all statically typed languages. However, in Scheme:

1. Conditionals do not expect a boolean predicate expression: Every expression that evaluates not to #f is interpreted as True.
2. The consequence and alternative expressions can have different types.

In order to avoid the ***Union*** type we require that the consequence and the alternative have the same type. The resulting typing rule:

```
Typing rule If:
   For every type environment _Tenv,
             expressions _e1, _e2, _e3, and
             type expressions _S1, _S2:
   If        _Tenv |- e1:_S1,
             _Tenv |- e2:_S2,
             _Tenv |- e3:_S2
   Then      _Tenv |- (if e1 e2 e3):_S2
```

Note that although the rule conclusion does not include any dependency on the predicate type _S1 and the predicate type _S1 is arbitrary, it is still included in the rule. The purpose is to guarantee that the predicate is well-typed. This is necessary for guaranteeing type safety. Note also that while the evaluation of a conditional follows only a single conclusion clause, the type derivation checks all clauses. That is, type derivation and language computation follow different program paths.

Example 5.13. *Derive a type for the expression:*

```
(+ 3
   (if (zero? mystery)
       5
       ( (lambda (x) x) 3)))
```

The tree structure is described in Figure 5.5. The leaves are `3,5,x,zero?,mystery,+`.

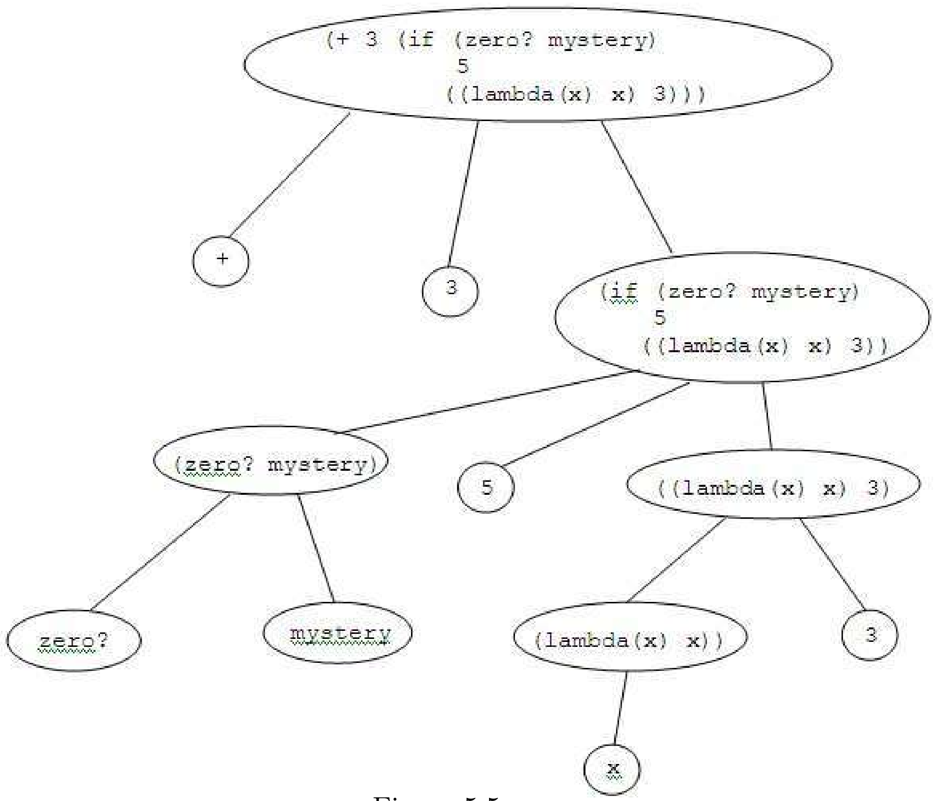

(+ 3 (if (zero? mystery) 5 ((lambda(x) x) 3)))
+
3
(if (zero? mystery) 5 ((lambda(x) x) 3))
(zero? mystery)
5
((lambda(x) x) 3)
zero?
mystery
(lambda(x) x))
3
x

Figure 5.5

Instantiating the ***Number, Variable*** and ***Primitive procedure*** axioms:

```
1. { } |- 3:Number
2. { } |- 5:Number
3. {x:T1} |- x:T1
4. {mystery: T2} |- mystery:T2
5. { } |- zero?:[Number -> Boolean]
6. { } |- +:[Number*Number -> Number]
```

Applying typing rule ***Procedure*** to statement no. 3, with type substitution `{_S1=T1, _U1=T1}`:

```
7. { } |- (lambda (x) x):[T1 -> T1]
```

Applying typing rule ***Application*** to statements no. 7, 1, with type substitution `{_S1=Number, _S=T1=Number}`:

```
8. { } |- ( (lambda (x) x) 3 ):Number
```

Applying typing rule ***Application*** to statements no. 5, 4, with type substitution `{_S1=T2=Number, _S=Boolean}`:

```
9. {mystery:Number} |- (zero? mystery):Boolean
```

Applying typing rule ***If*** to statements no. 9, 2, 8, with type substitution `{_S1=Boolean, _S2=Number}`:

```
10. {mystery:Number} |-
               ( if (zero? mystery)
                    5
                    ( (lambda (x) x) 3 ) ):Number
```

Applying typing rule ***Application*** to statements no. 6, 1, 10, with type substitution `{_S1=Number, _S2=Number, _S=Number}`:

```
11. {mystery:Number} |- (+ 3 ( if (zero? mystery)
                                 5
                                ((lambda (x) x) 3) )):Number
```

If the expression is preceded by a definition of the `mystery` variable as a Number value, the expression is well typed, and its type is number.

5.4.5 Adding recursion:

Recursive definitions require modification of the notion of a well-typed definition. For non recursive definitions, a definition `(define x e)` is well-typed if `e` is well typed. In a recursive definition `(define f e)`, `e` includes ***free*** occurrences of `f`, and therefore cannot be statically typed without an ***inductive environment*** about the type of `f`. Hence, we say that in a recursive definition `(define f e)`, `e` is ***well-typed*** and has type `[S1*...Sn -> S]`, for `n>0` or `[Empty -> S]` for `n=0`, if **Type-derivation**(`e`) outputs a typing statement `{f:[S1*...Sn -> S]} |- e:[S1*...Sn -> S]` (alternatively `{f:[Empty -> S]} |- e:[Empty -> S]`).

Therefore, we add a rule for recursive definitions `(define f e)`, that states that if `e` is well-typed with typing statement `{f:T } |- e:T`, then the definition is well-typed and the defined variable has this type as well:

```
Typing rule Recursive-definition :
    For every: recursive-definition expression (define _f _e) and type
    expression _S:
    If     {_f:_S } |- _e:_S,
    Then   { } |- (define _f _e):Void, and
           { } |- _f:_S
Restriction: No repeated variable definitions.
```

Example 5.14. – *Given the definition:*

```
> (define factorial
     (lambda (n)
       (if (= n 1)
           1
           (* n (factorial (- n 1))))))
```

derive a type for `(fact 3)`.

1. Type derivation for the definition expression of `factorial`:
   The leaves are `=,-,*,n,1,factorial`. Instantiating the ***Number, Variable, Primitive procedure*** typing axioms:

   ```
   1. { } |- =:[Number*Number -> Boolean]
   2. { } |- -:[Number*Number -> Number]
   3. { } |- *:[Number*Number -> Number]
   4. { } |- 1:Number
   5. {n:T1} |- n:T1
   6. {factorial:T2} |- factorial:T2
   ```

   Applying typing rule ***Application*** to statements no. 2, 5, 4, with type-substitution `{_S1=T1=Number, _S2=Number, _S=Number}`

   ```
   7. {n:Number} |- (- n 1):Number
   ```

Applying typing rule ***Application*** to statements no. 6, 7, with type-substitution `{_S1=Number, _S=T3, T2=[Number -> T3]}`:

```
8. {factorial:[Number -> T3], n:Number} |- (factorial (- n 1)):T3
```

Applying typing rule ***Application*** to statements no. 3, 5, 8, with type-substitution `{_S1=Number, _S2=Number, _S=Number, T1=Number, T3=Number}`:

```
9. {factorial:[Number -> Number], n:Number} |- (* n (factorial (- n 1)))):Number
```

Applying typing rule ***Application*** to statements no. 1, 5, 4, with type-substitution `{_S1=T1=Number, _S2=Number, _S=Boolean}`:

```
10. {n:Number} |- (= n 1):Boolean
```

Applying typing rule ***If*** to statements no. 10, 4, 9, with type-substitution `{_S1=Boolean, _S2=Number, _S=Boolean}`:

```
11. {factorial:[Number -> Number], n:Number} |-
                  (if (= n 1)
                      1
                     (* n (factorial (- n 1)) )):Number
```

Applying typing rule ***Procedure*** to statement no. 11, with type-substitution `{_S1=Number, _U1=Number}`:

```
12. {factorial:[Number -> Number]} |-
             (lambda (n)
                (if (= n 1)
                    1
                    (* n (factorial (- n 1)) )) ):[Number -> Number]
```

Therefore, the definition of `factorial` is well typed since this is a recursive definition, and the type of `factorial` is `[Number -> Number]`.
Applying typing rule ***Recursive-definition*** to statement no. 12, with type-substitution `{_S=[Number -> Number]}`:

```
13. { } |- (define factorial (lambda (n)
                               (if (= n 1)
                                   1
                                   (* n (factorial (- n 1)) )) )):Void
14. { } |- factorial:[Number -> Number]
```

2. Type derivation for the application `(factorial 3)`:
   Instantiating the ***Number*** typing axiom:

   ```
   15. { } |- 3:Number
   ```

   Applying typing rule ***Application*** to statements no. 14, 15, with type substitution `{_S1=Number, _S=Number}`:

   ```
   3. { } |- (factorial 3):Number
   ```

Note: ***Inter-related recursive definitions***: Consider

```
      (define f (lambda (...) (... (g ...) ...)
      (define g (lambda (...) (... (f ...) ...)
```

The ***Recursive-definition*** does not account for inter-related recursion since typing each defining expression requires a type environment for the other variable. Indeed, in statically typed languages (like ML), recursion and inter-related recursion require explicit declaration. Question: Why this is not a problem in Scheme?

5.4.6 Type Correctness with the Pair and List Types

The Pair and List Types have operations which are Scheme primitives (no special operators). In addition, there is the `empty` value. Therefore, there is a need for an axiom for `empty`, and the ***Primitive procedure*** typing axiom has to be extended for the new primitives.

Pairs:

```
For every type environment _Tenv and type expressions _S,_S1,_S2:
       _Tenv |- cons:[_S1*_S2 -> Pair(_S1,_S2)]
       _Tenv |- car:[Pair(_S1,_S2) -> _S1]
       _Tenv |- cdr:[Pair(_S1,_S2) -> _S2]
       _Tenv |- pair?:[_S -> Boolean]
       _Tenv |- equal?:[Pair(_S1,_S2)*Pair(_S1,_S2) -> Boolean]
```

Homogeneous lists:

```
 For every type environment _Tenv and type expression _S:
        _Tenv |- empty:List(_S)
        _Tenv |- list:[Empty -> List(_S)]
        _Tenv |- list:[_S* ...*_S -> List(_S)]  n >0
        _Tenv |- cons:[_S*List(_S) -> List(_S)]
        _Tenv |- car:[List(_S) -> _S]
        _Tenv |- cdr:[List(_S) -> List(S)]
        _Tenv |- null?:[List(_S) -> Boolean]
        _Tenv |- list?:[_S -> Boolean]
        _Tenv |- equal?:[List(_S)*List(_S) -> Boolean]
```

Heterogeneous lists:

```
 For every type environment _Tenv and type expression _S:
        _Tenv |- empty:List
        _Tenv |- list:[Empty -> List]
        _Tenv |- cons:[S*List -> List]
        _Tenv |- car:[List -> _S]
        _Tenv |- cdr:[List -> List]
        _Tenv |- null?:[List -> Boolean]
        _Tenv |- list?:[_S -> Boolean]
        _Tenv |- equal?:[List*List -> Boolean]
```

Example 5.15. *Derive the type of the `firstFirst` procedure. Its definition:*

```
(define firstFirst (lambda(pair)
                      (car (car pair))))
```

We derive a type for the `lambda` form: Instantiation of the ***Variable*** axiom and the ***Primitive procedure*** Pair axiom (an arbitrary decision, and with renaming):

```
1. {pair:T1 } |- pair:T1
2. { } |- car:[Pair(T2,T3) -> T2]
```

Typing `(car pair)` – Applying the ***Application*** typing rule to statements 1,2, with type substitution `{S1=T1=Pair(T2,T3), S=T2}`:

```
3. {pair: Pair(T2,T3)} |- (car pair):T2
```

Typing `(car (car pair))` – In order to apply the ***Application*** typing rule to statements 3,2, we first have to rename these statements:

```
2'. { } |- car:[Pair(T21,T31) -> T21]
3'. {pair: Pair(T22,T32)} |- (car pair):T22
```

Applying the ***Application*** typing rule to statements 2',3', with type substitution `{S1=T22=Pair(T21,T31), S=T21}`:

```
4. {pair:Pair(Pair(T21,T31),T32 ) } |- (car (car pair)):T21
```

Application of typing rule ***Procedure*** to statement 4, with type substitution `{S1=Pair(Pair(T21,T31),T32), U1=T21}`:

```
5. { } |- (lambda (pair) (car (car pair))):
                                        [Pair(Pair(T21,T31),T32) -> T21]
```

Following this definition, the type of `firstFirst` is `{[Pair(Pair(T21,T31),T32) -> T21]}`, and well typed derivations can end with the type assignment `{firstFirst<-Pair(Pair(T21,T31),T32) -> T21}`.

Example 5.16. *Type derivation for `(firstFirst (cons (cons 1 2) 3))`:*

Instantiation of the ***Number*** axiom, ***Variable*** axiom, and the ***Primitive procedure*** Pair axiom (an arbitrary decision, and with renaming):

```
1. { } |- 1:Number
2. { } |- 2:Number
3. { } |- 3:Number
4. { } |- cons:[T1*T2 -> Pair(T1,T2)]
5. {firstFirst:T3} |- firstFirst:T3
```

Typing `(cons 1 2)` – Applying the ***Application*** typing rule to statements 1,2,4, with type substitution `{S1=Number, S2=Number, S=Pair(Number,Number), T1=Number, T2=Number}`:

```
6. { } |- (cons 1 2):Pair(Number,Number)
```

Typing `(cons (cons 1 2) 3)` – Applying the ***Application*** typing rule to statements 6,3,4, with type substitution `{S1=Pair(Number,Number), S2=Number, S=Pair(Pair(Number,Number),Number), T1=Pair(Number,Number), T2=Number}`:

```
7. { } |- (cons (cons 1 2) 3):Pair(Pair(Number,Number),Number)
```

Typing `(firstFirst (cons (cons 1 2) 3))` – Applying the ***Application*** typing rule to statements 7,5, with type substitution `{S1=Pair(Pair(Number,Number),Number), S=T4, T3=[Pair(Pair(Number,Number),Number) -> T4]}`:

```
7. {firstFirst: [Pair(Pair(Number,Number),Number) -> T4]} |-
                                        (firstFirst (cons (cons 1 2) 3)):T4
```

The definition of `firstFirst` justifies derivations whose last typing statement has the type environment `{firstFirst:[Pair(Pair(T21,T31),T32) -> T21]}`. The type environment in statement 7 is an instance of this type environment, under the type substitution `{T21=T31=T32=Number}`. Therefore, the type derivation of `(firstFirst (cons (cons 1 2) 3))` is correct.

5.5 Type Checking and Inference using Type Constraints Approach

Algorithm **Type-derivation** involves non-deterministic decisions in applications of typing axioms and rules:

1. Replacement of the type environment meta variables.
2. Selection of type-substitution that can unify the typing statements in the rule condition with supporting typing statements from the pool.
3. Application of the monotonicity rule.

In addition, implementation of this algorithm involves management of multiple data types: type-environment, language variables and expressions, type expressions.

The algorithm can be mechanized by replacing the applications of typing axioms and rules with ***type equations*** that are constructed for every sub-expression of the typed expression. A solution to the equations assigns types to every sub -expression. The type checking/inference procedure turns into a ***type equation solver***. Compilers and interpreters like ML, that support static type checking/inference use the type equation approach.

Example 5.17. – *Typing the procedure* `(lambda (f x) (f x x))` *using type equations.*

There are 4 type variables: $T_f, T_x, T_{(fxx)}, T_{(lambda(fx)(fxx))}$. The type constraints (equations) are:

$$\begin{array}{lcl} T_f & = & [T_x * T_x \to T_{(fxx)}] \\ T_{(lambda(fx)(fxx))} & = & [T_f * T_x \to T_{(fxx)}] \end{array}$$

Solution: $T_{(lambda(fx)(fxx))} = [[T_x * T_x \to T_{(fxx)}] * T_x \to T_{(fxx)}]$.

Type equation solvers use a ***unification algorithm*** for unifying polymorphic type expressions. We will introduce a unification algorithm in the operational semantics of ***Logic programming*** (Chapter 7). We now demonstrate a method for inferring a type for an expression `e`, by constructing and solving type equations. The method has four stages:

1. Rename bound variables in `e`.
2. Assign type variables to all sub-expressions.

3. Construct type equations.
4. Solve the equations.

We demonstrate the method by example. The running example is the expression:

```
(lambda (f g)
    (lambda (x)
      (f (+ x (g 3)))))
```

5.5.1 Creating and solving type constraints

Stage I: Renaming : None needed.

Stage II: Assign type variables: Every sub expression is assigned a type variable:

Expression	*Variable*
`(lambda (f g) (lambda (x) (f (+ x (g 3)))))`	T_0
`(lambda (x) (f (+ x (g 3))))`	T_1
`(f (+ x (g 3)))`	T_2
`f`	T_f
`(+ x (g 3))`	T_3
`+`	T_+
`x`	T_x
`(g 3)`	T_4
`g`	T_g
`3`	T_{num3}

Stage III: Construct type equations – the typing rules of algorithm Type-derivation turn into type equations: The rules are:

1. **Number, Boolean, Symbol, Primitive-procedures:** Construct equations using their types. For example, for the number 3: $T_{num3} = Number$ and for the binary primitive procedure $+$: $T_+ = Number * Number \to Number$.
2. **Procedure (Lambda expressions):** The type inference rule is:

```
For every: type environment _Tenv,
  variables _x1, ..., _xn, n ≥ 0
  expressions _e1, ..., _em, m ≥ 1, and
  type expressions _S1, ...,_Sn, _U1, ...,_Um :
Procedure with parameters (n > 0):
  If     _Tenv∘{_x1:_S1, ..., _xn:_Sn } |- _ei:_Ui for all i = 1..m ,
  Then   _Tenv |- (lambda (_x1 ... _xn ) _e1 ... _em) : [_S1*...*_Sn -> _Um]
Parameter-less Procedure (n = 0):
  If     _Tenv |- _ei:_Ui for all i=1..m,
  Then   _Tenv |- (lambda ( ) _e1 ... _em):[Empty -> _Um]
```

Extracting the type restriction on the involved sub-expressions yields:
For `(lambda (v1 ...vn) e1 ... em)` with $n > 0$, construct the equation:
$T_{(lambda(v1...vn)e1...em)} = [T_{v1} * \ldots * T_{vn} \rightarrow T_{em}]$.
For `(lambda ( ) e1 ... em)`, construct the equation:
$T_{(lambda()e1...em)} = [Empty \rightarrow T_{em}]$.

3. **Application:** The type-inference rule is:

```
For every: type environment _Tenv,
  expressions _f, _e1, ..., _en, n ≥ 0 , and
  type expressions _S1, ..., _Sn, _S:
Procedure with parameters (n > 0):
  If     _Tenv |- _f:[_S1*...*_Sn -> _S],
         _Tenv |- _e1:_S1, ..., _Tenv |- _en:_Sn
  Then   _Tenv |- (_f _e1 ... _en):_S
Parameter-less Procedure (n = 0):
  If     _Tenv |- _f:[Empty -> _S]
  Then   _Tenv |- (_f):_S
```

Extracting the type restriction on the involved sub-expressions yields:
For `(f e1 ... en)` with $n > 0$, construct the equation:
$T_f = [T_{e1} * \ldots * T_{en} \rightarrow T_{(fe1...en)}]$.
For `(f)` construct the equation:
$T_f = [Empty \rightarrow T_{(f)}]$.

Note: The inference rules for `Procedure` and `Application` require type inference for all internal expressions, even though the final inferred type does depend on their type. Why? In the type-equations approach this requirement is achieved by constructing type equations for all sub-expressions, as described below.

The algorithm constructs equations for the primitive sub-expressions and for all composite sub-expressions. For the running example:
The equations for the primitive sub-expressions are:

Expression	*Equation*
`3`	$T_{num3} = Number$
`+`	$T_{+} = Number * Number \rightarrow Number$

The equations for composite sub-expressions are:

Expression	*Equation*
`(lambda (f g) (lambda (x) (f (+ x (g 3)))))`	$T_0 = [T_f * T_g \rightarrow T_1]$
`(lambda (x) (f (+ x (g 3))))`	$T_1 = [T_x \rightarrow T_2]$
`(f (+ x (g 3)))`	$T_f = [T_3 \rightarrow T_2]$
`(+ x (g 3))`	$T_+ = [T_x * T_4 \rightarrow T_3]$
`(g 3)`	$T_g = [T_{num3} \rightarrow T_4]$

Stage IV: Solving the equations: The equations are solved by gradually producing type-substitutions for all type variables. For an expression `e`, the algorithm infers a type `t` if the final type-substitution maps its variable T_e to `t` (includes the binding $\langle T_e = \texttt{t}\rangle$). If an expression has an infer ed type then all of its sub-expressions have types as well. If the procedure fails (output is `FAIL`) then either there is a type error or the constructed type equations are too weak. Circular type-substitution cause failure. The solution is processed by considering the equations one by one.
Input: A set of type equations.
Output: A type substitution of FAIL.
Method:

Initialization:

`substitution := { }`
Order the set of input equations in some sequential order.
`equation :=` $te_1 = te_2$, the first equation.

Loop:

1. Apply the current substitution to the equation:
   `equation :=` $te_1 \circ$ `substitution` $=$ $te_2 \circ$ `substitution`
2. If $te_1 \circ$ `substitution` and te_2`substitution` are atomic types:
   if $te_1 \circ$ `substitution` $\neq$ te_2`substitution`: `substitution := FAIL`
   otherwise: Do nothing.
3. Without loss of generality:
   If $te_1 \circ$ `substitution` $= T$, i.e., a type variable, and $te_1 \circ$ `substitution` $\neq$ $te_2 \circ$ `substitution`:
   `substitution := substitution`$\circ\{T = te_2 \circ$ `substitution`$\}$. That is, apply the equation to `substitution`, and add the equation to the substitution. If the application fails (circular mapping), `substitution := FAIL`.
4. if $te_1 \circ$ `substitution` and $te_2 \circ$ `substitution` are composite types:
   if they have the same type constructor: Split $te_1 \circ$ `substitution` and $te_2 \circ$

`substitution` into component type expressions, create equations for corresponding components, and add the new equations to the pool of equations.
if they have different type constructors: `substitution := FAIL`

5. Without loss of generality:
   if $te_1 \circ$ `substitution` is an atomic type and $te_2 \circ$ `substitution` is a composite type: `substitution := FAIL`
6. if there is a next equation: `equation := next(equation)`

until
`substitution = FAIL` or
there is no next equation.

Output: `substitution`

Continuing the running example:

	Equations	`substitution`
1.	$T_0 = [T_f * T_g \rightarrow T_1]$	$\{\}$
2.	$T_1 = [T_x \rightarrow T_2]$	
3.	$T_f = [T_3 \rightarrow T_2]$	
4.	$T_+ = [T_x * T_4 \rightarrow T_3]$	
5.	$T_g = [T_{num3} \rightarrow T_4]$	
6.	$T_{num3} = Number$	
7.	$T_+ = Number * Number \rightarrow Number$	

Equation 1: The application `equation1` $\circ$ `substitution` leaves `equation1` as a substitution, since one of its sides is still a type variable. Therefore:
`substitution := substitution` $\circ$ `(equiation1` $\circ$ `substitution)` :

	Equations	`substitution`
2.	$T_1 = [T_x \rightarrow T_2]$	$\{T_0 = [T_f * T_g \rightarrow T_1]\}$
3.	$T_f = [T_3 \rightarrow T_2]$	
4.	$T_+ = [T_x * T_4 \rightarrow T_3]$	
5.	$T_g = [T_{num3} \rightarrow T_4]$	
6.	$T_{num3} = Number$	
7.	$T_+ = Number * Number \rightarrow Number$	

Equation 2: Following the same argument:
`substitution := substitution` $\circ$ `(equiation2` $\circ$ `substitution)` :

	Equations	`substitution`
3.	$T_f = [T_3 \to T_2]$	$\{T_0 = [T_f * T_g \to [T_x \to T_2]],$
4.	$T_+ = [T_x * T_4 \to T_3]$	$T_1 = [T_x \to T_2]\}$
5.	$T_g = [T_{num3} \to T_4]$	
6.	$T_{num3} = Number$	
7.	$T_+ = Number * Number \to Number$	

Equation 3: Following the same argument:
`substitution := substitution` $\circ$ `(equiation3` $\circ$ `substitution)` :

	Equations	`substitution`
4.	$T_+ = [T_x * T_4 \to T_3]$	$\{T_0 = [[T_3 \to T_2] * T_g \to [T_x \to T_2]],$
5.	$T_g = [T_{num3} \to T_4]$	$T_1 = [T_x \to T_2],$
6.	$T_{num3} = Number$	$T_f = [T_3 \to T_2]\}$
7.	$T_+ = Number * Number \to Number$	

Equation 4: Following the same argument:
`substitution := substitution` $\circ$ `(equiation4` $\circ$ `substitution)` :

	Equations	`substitution`
5.	$T_g = [T_{num3} \to T_4]$	$\{T_0 = [[T_3 \to T_2] * T_g \to [T_x \to T_2]],$
6.	$T_{num3} = Number$	$T_1 = [T_x \to T_2],$
7.	$T_+ = Number * Number \to Number$	$T_f = [T_3 \to T_2],$
		$T_+ = [T_x * T_4 \to T_3]\}$

Equation 5: Following the same argument:
`substitution := substitution` $\circ$ `(equiation5` $\circ$ `substitution)` :

	Equations	`substitution`
6.	$T_{num3} = Number$	$\{T_0 = [[T_3 \to T_2] * [T_{num3} \to T_4] \to [T_x \to T_2]],$
7.	$T_+ = Number * Number \to Number$	$T_1 = [T_x \to T_2],$
		$T_f = [T_3 \to T_2],$
		$T_+ = [T_x * T_4 \to T_3],$
		$T_g = [T_{num3} \to T_4]\}$

Equation 6: Following the same argument:
`substitution := substitution` $\circ$ `(equiation6` $\circ$ `substitution)`:

	Equations	`substitution`
7.	$T_+ = [Number * Number \rightarrow Number]$	$\{T_0 = [[T_3 \rightarrow T_2] * [Number \rightarrow T_4] \rightarrow [T_x \rightarrow T_2]],$ $T_1 = [T_x \rightarrow T_2],$ $T_f = [T_3 \rightarrow T_2],$ $T_+ = [T_x * T_4 \rightarrow T_3],$ $T_g = [Number \rightarrow T_4],$ $T_{num3} = Number\}$

Equation 7: Applying the substitution to `equation7`, i.e., `equiation7` $\circ$ `substitution`, yields the equation $[T_x * T_4 \rightarrow T_3] = [Number * Number \rightarrow Number]$, which has composite type expressions on both sides. Since these expressions have the same type constructor, the new equation is split into three equations: $T_x = Number$, $T_4 = Number$ and $T_3 = Number$. Processing the three equations yields:

Equations	`substitution`
	$\{T_0 = [[Number \rightarrow T_2] * [Number \rightarrow Number] \rightarrow [Number \rightarrow T_2]],$ $T_1 = [Number \rightarrow T_2],$ $T_f = [Number \rightarrow T_2],$ $T_+ = [Number * Number \rightarrow Number],$ $T_g = [Number \rightarrow T_4],$ $T_{num3} = Number,$ $T_x = Number,$ $T_4 = Number,$ $T_3 = Number\}$

Output: `substitution`

Properties of the type-equation based inference algorithm:

1. **Characterization:** The algorithm is deterministic. It is derived from the **type-derivation** rule-based algorithm by:
   (a) Replacing language expressions by type variables that stand for their types. This steps implies removal of the type environment expression.
   (b) For every composite sub-expression and every primitive symbol: Constructing type equations, that are derived from the inference rules of these expressions. Together with the requirement to process all equations, this step guarantees well-typing of every sub-expression
2. **Well-typing:** If the algorithm terminates with non `Fail substitution` then the input language expression is well-typed. For example, for the running example,

```
(lambda (f g)
    (lambda (x)
      (f (+ x (g 3)))))
```

the algorithm proves that it is well-typed. If the lambda expression includes internal expressions, as in:

```
(lambda (f g)
    (lambda (x)
      (display x) (newline) (* x x) (f (+ x (g 3)))))
```

the algorithm guarantees that the internal expressions are also checked for well-typing.

3. **Type inference:** If the input language expression `e` does not have free variables, then if it is well-typed its inferred type is the output mapping of its type variable: `substitution`(T_e). For the running example expression, its type variable was marked T_0 and its inferred type is
`substitution`(T_0) = $[[Number \rightarrow T_2] * [Number \rightarrow Number] \rightarrow [Number \rightarrow T_2]]$
4. **Type safety:** If the input language expression does not have free variables, The algorithm guarantees type safety, since it enforces well-typing of all sub-expressions.
5. **Comparison with the rule-based inference method:** The inference algorithm holds a small set of inference rules, that are repeatedly used in a non-deterministic way in all type inference applications. In contrast, the equation-based algorithm creates a new set of equations in every application, and process the set in a deterministic way.

Extending the typed language:
The typed language under the type-equations approach can be extended, similarly to the one under the type-inference-rules approach, to apply to conditional expressions, definitions, and other language expressions. This requires addition of equations for the new kinds of language expressions. Such equations are obtained by extracting from the corresponding type-inference rules, the enforced type inter-relationship constraints.

5.5.2 Implementing Type Inference Using the Type Constraints Approach

In order to implement the type-inference method of using type constraints, we adopt the ADT approach:

1. ADTs: Type-Expression (TE), Substitution, Equation, where the Substitution and Equation ADTs are clients (built on top) of the Type-Expression ADT.

2. Client procedures:
   (a) Of Substitution: `substitution-application,substitution-combination`.
   (b) Of Equation and of the client procedures of Substitution: `infer-type, solve`.
3. Implementation:
   (a) Module `type-expression-adt`: Type-Expression is implemented as a tagged-data, using the type constructor as a tag.
   (b) Module `substitution-adt`: Substitution is implemented as a pair-list (2-element list) of variables and type expressions. Includes the client procedures `substitution -application`, `substitution-combination`.
   (c) Module `equation-adt`: Equation is implemented as a pair-list of type expressions.
   (d) Module `solve`: Includes the client procedures `infer-type, solve`.
   (e) Additional modules:
   Module `Scheme-exprs-type-vars` includes the procedure `make-expr-tvars-list` which associates a type variable with every sub-expression of the given expression.
   Module `asp` includes abstract syntax procedures for Scheme expressions.
   Module `auxiliary` includes auxiliary procedures.
   (f) **Testing modules**: `type-expression-adt-tests, substitution-adt-tests, equation-adt-tests, solve-tests, utils`.

The full system appears in the course site.

Type-Expression ADT (partial):

```
Constructors:
Signature: make-proc-te(tuple-te, te)
Type: Client view: [Tuple*Type -> Procedure
Example: (make-proc-te (make-tuple-te (list Number)) 'Number)
         ==> (-> (*Number) Number)

Signature: make-tuple-te(te-list)
Type: Client view:[List(TE)-> Tuple]
Example: (make-tuple-te
           (list 'Number (make-proc-te (make-tuple-te (list 'Number))'T1)))
         ==> (* Number (-> (* Number) T1))

Some getters:
Signature: get-constructor(te)
```

```
Type: Client view: [Composite-Type-Expression -> Symbol]
Example: (get-constructor (make-tuple-te (list 'Number 'Number))) ==> *

Signature: tuple-components(te)
Type: Client view: [Tuple -> List(TE)]
Example: (tuple-components (make-tuple-te (list 'Number 'Number))) ==> (Number Number)

Signature: proc-return-te(te)
Type: Client view:[Procedure -> TE]
Example: (proc-return-te (make-proc-te (make-tuple-te (list 'Number)) 'T1)) ==> T1

Some predicates:
Signature: equal-atomic-te?(te1 te2)
Type: [List union Symbol * List union Symbol -> Boolean]

Signature: type-expr?(te)
Type: [T -> Boolean]
```

Substitution ADT (partial):

```
Constructors:
Signature: make-sub(variables, tes)
Type: Client view: [List(Symbol)*List(Type-expression) -> Substitution]
Example: (make-sub '(x y z)
           (list 'Number 'Boolean
                  (make-proc-te (make-tuple-te (list 'Number)) 'Number)))
         ==> ((x y z) (Number Boolean (-> (* Number) Number)))

Some getters:
Signature: get-variables(sub)
Type: Client view: [Substitution -> List(Var)]
Example: (get-variables
           (make-sub
             '(x y z)
             (list 'Number 'Boolean
                   (make-proc-te (make-tuple-te (list 'Number)) 'Number))) )
         ==> (x y z)

Predicates:
Signature: non-circular?(var, te)
Type: [Symbol*Type-Expression -> Boolean]
Example: (non-circular? 'T (make-proc-te (make-tuple-te (list 'Number)) 'Number)) ==> #t
```

```
         (non-circular? 'T (make-proc-te (make-tuple-te (list 'T)) 'Number)) ==> #f

Signature: sub?(sub)
Type: [T -> Boolean]
```

Equation ADT (partial):

```
Constructors:
Signature: make-equation-from-tes(te1,te2)
Type: Client view: [Type-expression*Type-expression -> Equation]
Example: (make-equation-from-tes
           '((-> (* Number Number) Number)
           '(-> (* Number) Boolean)))
         ==> ((-> (* Number Number) Number)  (-> (* Number) Boolean)))

Signature: make-equations(expression-type-vars-list)
Purpose: Return a set of equations for a given Scheme expression and a list of
         pairs: Sub-expression and its type variable
Type: Client view: [List(Pair-List(Scheme-expression,Symbol)) -> List(Equation)]
Example: (make-equations
          '(((lambda(x)(+ x 1)) var_0) ((+ x 1) var_1) (1 var_4) (x var_3) (+ var_2)))
         ==> '((var_0 (-> (* var_3) var_1)) (var_2 (-> (* var_3 var_4) var_1)))

Signature: make-equation(se,expr-tvars-list)
Purpose: Return a single equation
Type: Client view:
[Scheme-expression*List(Pair-List(Scheme-expression,Symbol)) -> Equation]

Some getters:
Signature: get-left(eq)
Type: Client view: [Equation -> Type-Expression]

Some predicates:
Signature: equation?(eq)
Type: [T -> Boolean]
```

The system is architectured after the ADTs: There are 3 Type-equation, Substitution and Equation ADT modules. The Substitution-ADT module includes the 2 client procedures of Substitution: `substitution-application,substitution-combination`.

The main functionality of type inference is defined in the "solve" module, in which the 2 main client procedures `infer-type, solve` are defined. The `solve` procedure performs the actual task of solving the type constraints, and the `infer-type` procedure performs the overall type inference task, starting from a Scheme expression, via assigning type variables to all sub-expressions, construction of the equations, and solving them.

```
Signature: infer-type(scheme-expr)
Purpose: Infer the type of a scheme expression using the equations method
Type: Client view: [Scheme-expression -> Type-expression]
Example: (infer-type '(lambda (f x) (f (f x)))) ==> (-> (* (-> (* T_4) T_4) T_4) T_4)
(define infer-type
  (lambda (scheme-expr)
    (let ((expr-tvars-list (make-expr-tvars-list scheme-expr)))
                                 ;make-expr-tvars-list is defined in ass2.rkt
      (get-expression-of-variable
        (solve-equations (make-equations expr-tvars-list))
        (val-of-index scheme-expr expr-tvars-list))
           ;get the type expression of the variable that represents scheme-expr,
           ;in the substitution returned by solve-equations.
      )
  ))

(define solve-equations
  (lambda(equations)
    (solve equations (make-sub '() '()))))

Signature: solve(equations,substitution)
Purpose: Solve the equations, starting from a given substitution. returns the
         resulting substitution, or error, if not solvable
(define solve
  (lambda(equations subst)
    (if (null? equations)
        subst
        (let
          ((eq (make-equation-from-tes
                (substitution-application subst
                                 (get-left (get-first-equation equations)))
                (substitution-application subst
                                 (get-right (get-first-equation equations)))))
             )
          (letrec ((solve-var-eq            ;If one side of eq is a variable
                    (lambda(var-part other-part)
                       (solve (cdr equations)
                         (substitution-combination subst
                              (make-sub (list var-part) (list other-part))))))
                   (both-sides-atomic?
                    (lambda(eq) (and (atomic-te? (get-left eq))
                                     (atomic-te? (get-right eq)))))
                   (handle-both-sides-atomic
                    (lambda(eq)
                      (if (equal-atomic-te? (get-left eq) (get-right eq))
                          (solve (get-rest-equations equations) subst)
                          (error 'solve "equation contains unequal atomic types: ~e" eq))))
                  )
            (cond ((variable-te? (get-left eq))
                   (solve-var-eq (get-left eq) (get-right eq)))
                  ((variable-te? (get-right eq))
                   (solve-var-eq (get-right eq) (get-left eq)))
                  ((both-sides-atomic? eq) (handle-both-sides-atomic eq))
                  ((and (composite-te? (get-left eq))
                        (composite-te? (get-right eq)) (unifyable-structure eq))
                   (solve (append (get-rest-equations equations) (split-equation eq))
                          subst))
                  (else (error 'solve "equation contains unknown type expresion: ~s" eq)))
          )))
  ))
```

Notes:

1. The type inference system follows the principles of the ADT approach for software construction.
2. But, the system does not preserves the principles of data abstraction. Try finding where the "abstraction barrier" is broken, and why.
3. What are the conclusions for leaving the role of keeping abstraction barriers in the hands of the software developers?

Chapter 6

Evaluators for Functional Programming

Sources: SICP 4.1. [1] and extensions.

Introduction on Meta-Circular Evaluators

Programming languages are used to ***describe problems*** and specify solutions. They provide means for ***combination*** and ***abstraction*** that enable hiding unnecessary details, and expressing high level concepts. The design of new descriptive languages is a natural need in complex applications. It arises in multiple paradigms, and not restricted to the design of programming languages.

Metalinguistic abstraction is used to ***describe languages***. It involves two major tasks:

1. ***Syntax***, i.e., ***language design***: Language atoms, primitives, combination and abstraction means.
2. ***Semantics (operational)***, i.e., ***language evaluation rules*** — a procedure that when applied to a language expression, performs the actions needed for evaluating that expression.

The method of implementing a language in another language is called ***embedding***. The evaluator that we implement for Scheme, uses Scheme (i.e., some already implemented Scheme evaluator) as an embedding (implementation) language. That is:

1. Interpreted language: Scheme.
2. Implementation (embedding) language: Scheme.

Such evaluators, in which the target language is equal to the implementation language, are called ***meta-circular evaluators***.

The evaluators that we provide are ***meta-circular evaluators*** for Scheme (without `letrec`). We provide two evaluators and a compiler:

1. ***Substitution evaluator***: This evaluator implements the ***applicative-eval*** operational semantics algorithm. Its rules distinguish between atomic to composite expressions. For composite expressions, special forms have their own computation rules. Primitive forms evaluate all sub-expressions, and apply the primitive procedure. Otherwise, the computation rule follows the ***eval-substitute-reduce*** pattern.

2. ***Environment evaluator***: This evaluator implements the ***environment-based*** operational semantics, also introduced in this chapter. This evaluator modifies the substitution evaluator by introducing an ***environment*** data structure, that extends the simple ***global environment*** of the substitution evaluator.

3. ***Environment-based compiler***: A compiler that is based on the environment evaluator: Applies static (compile time) translation of Scheme code to the underlying application (Dr. Racket) code, while delaying run-time dependent computation.

All evaluators have the same architecture, described in Figure 6.1. The course site includes full implementations for the three evaluators. The **Core** package is a client of the

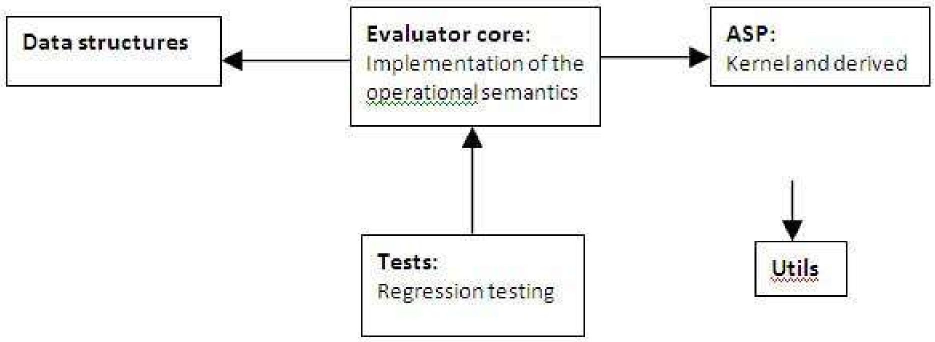


Figure 6.1: Visual representation of environments

ASP and the **Data structures** packages, which are stand alone packages. There is a single **ASP** package, used by all evaluators. But every evaluator has its own **Data structures** package. The **Tests** packages of the evaluators provide ***regression testing*** – i.e., tests that should be repeatedly run following every modification, in order to verify that a newly inserted test does not violate an older test. The **Utils** package provides services used by all otter packages.

The use of an ***abstract syntax parser*** creates an ***abstraction barrier*** between the concrete syntax and its client – the evaluator:

– Concrete syntax can be modified, without affecting the clients.

– Evaluation rules can be modified, without affecting the syntax.

Input to the evaluators: All evaluators receive as input a scheme expression or an already evaluated scheme expression (in case of repeated evaluation). Therefore, there is a question of ***representation***: "How to represent a Scheme expression?" For that purpose, the evaluators exploit the uniformity of Scheme expressions and the ***printed form of lists***:

1. Compound Scheme expressions have the form: `( <exp> <exp> ... <exp> )`, where `<exp>` is any Scheme expression, i.e.: Atomic (Number or Boolean or Variable), or compound.

2. The printed form of Scheme lists is: `( <val> <val> ... <val> )`, where `<val>` is the printed form of a value of a Scheme type, i.e.: Number or Symbol or Boolean or Procedure or Pair or List.

The evaluators treat Scheme expressions as ***constant lists***. This view saves us the need to write an input tokenizer for retrieving the needed tokens from a character string which is a Scheme expression (as needed in other languages, that treat symbolic input as strings – like JAVA). The components of a Scheme expressions are retrieved using the standard List selectors: `car, cdr, list-ref`. For example:

```
> (derive-eval (list '+ 1 2))
3
> (derive-eval (list 'lambda (list 'lst) (list 'car (list 'car 'lst)) ))
(procedure (lst) ((car (car lst))))
```

Note that the input to the evaluators is an ***unevaluated*** list! Otherwise, the evaluator is asked to evaluate an ***already evaluated*** expression, and is either useless or fails:

```
> (derive-eval (lambda (lst) (car (car lst)) ))
. . ASP.scm:247:31: car: expects argument of type <pair>; given #<procedure>
```

Quoted lists: Building Scheme expressions as constant lists using List constructors is quite heavy. In order to relax the writing, we introduce the Scheme syntactic sugar for constant lists: `" ' "` (***yes, it is the same `" ' "` symbol, used to shorten the `quote` constructor of the Symbol type!***). The `" ' "` symbol is a macro character, replaced by construction of the list value whose printed form is quoted. That is,

```
'(lambda (lst) (car (car lst)) ) =
(list 'lambda (list 'lst) (list 'car (list 'car 'lst)) ).
```

Using " ' ", Scheme expressions can be given as constant input lists:

```
> (derive-eval '(lambda (lst) (car (car lst)) ))
(procedure (lst) ((car (car lst))))
```

6.1 Abstract Syntax Parser (ASP) (SICP 4.1.2)

An abstract syntax parser is a tool that can:

1. Determine the ***kind*** of a Scheme expression;
2. Can ***select the components*** of a Scheme expression;
3. Can ***construct*** a language expression, when given its components;

These services result from the ***abstract syntax*** essentials:

– It distinguishes alternatives and components of a category.

– It ignores other syntactic details.

For example, for the `<conditional>` category, it distinguishes

– The `<if>` and `<cond>` alternatives.

– For the `<if>` category, it distinguishes the 3 components: `<predicate>, <consequence>, <alternative>`.

The abstract syntax parser implements an ***interface*** of the abstract syntax of the interpreted language:

– ***Constructors***;

– ***Selectors***, for retrieving the components of an expression;

– ***Predicates***, for identification.

That is:

The abstract syntax is a collection of ADTs that are implemented by the concrete syntax type, using the ASP!

For every Scheme expression, its ADT includes its constructor, selectors and predicates. The ASP ***implements*** the expression ADT.

The abstract syntax parser does not provide information about the concrete syntax of expressions. Therefore, revision of the exact syntax of the `cond` ADT does not modify the API of the abstract syntax. It affects only the implementation of the `cond` ADT! This way the exact syntax of the expressions is separated from the core of any tool that uses the parser.

Derived expressions: Language expressions are classified into:

1. ***Language kernel*** expressions: Form the core of the language – Every implementation must implement them.
2. ***Derived*** expressions: Re-written using the core expressions. They are ***implementation invariants***.

For example, in Scheme, it is reasonable to include only one conditional operator in the kernel, and leave the other as derived. Another natural example is `let` expressions, that can be re-written as applications of anonymous procedures (closures).

Identifying Scheme expressions: The name (identifier) of a special form is used as a ***type tag***, that identifies the type of expressions. This is similar to the type Tagged-data(T) in chapter 3. This approach exploits the prefix syntax of Scheme. This way:

– A `lambda` expression is identified as a list starting with the `lambda` tag.

– An `if` expression is identified as a list starting with the `if` tag.

The Tagged-data interface and an implementation is given below:

```
; Signature: attach-tag(x, tag)
; Type: [LIST*Symbol -> LIST]
  (define attach-tag
     (lambda (x tag) (cons tag x)))
               ; Note that the tagged content MUST be a list!

; Signature: get-tag(tagged)
; Type: LIST -> Symbol
  (define get-tag (lambda (tagged) (car tagged)))

; Signature: get-content(tagged)
; Type: [LIST -> T]
  (define get-content
     (lambda (tagged) (cdr tagged)))

Signature: tagged-data?(datum)
Type: [T -> Boolean]
  (define tagged-by?
     (lambda (datum)
       (and (list? datum) (symbol? (car datum))) ))
```

```
; Signature: tagged-by?(tagged, tag)
; Type: [T*Symbol -> Boolean]
  (define tagged-by?
     (lambda (tagged tag)
       (and (tagged-data? tagged)
            (eq? (get-tag tagged) tag))))
```

6.1.1 The parser procedures:

For each type of expression the abstract syntax parser implements:

– Predicates, to identify Scheme expressions.

– Selectors to select parts of Scheme expressions.

– Constructors.

1. **Atomic expressions:**

   – **Atomic identifier:**

   ```
   (define atomic?
       (lambda (exp)
          (or (number? exp) (boolean? exp) (variable? exp) (null? exp))))
   ```

   – **Numbers:**

   ```
   (define number?
      (lambda (exp)
        (number? exp)))
   ```

   – **Booleans:**

   ```
   (define boolean?
      (lambda (exp)
        (or (eq? exp '#t) (eq? exp '#f))))
   ```

   – **Variables:**

   ```
   (define variable?
      (lambda (exp) (symbol? exp)))
   ```

2. **Quoted expressions:**

```
(define quoted?
   (lambda (exp)
     (tagged-by? exp 'quote)))

(define text-of-quotation
   (lambda (exp) (car (get-content exp))))

(define make-quote
   (lambda (text)
      (attach-tag (list text) 'quote)))
```

3. **Lambda expressions:**

```
(define lambda?
  (lambda (exp)
     (tagged-by? exp 'lambda) ))

(define lambda-parameters
   (lambda (exp)
      (car (get-content exp))))

(define lambda-body
   (lambda (exp)
      (cdr (get-content exp))))

; Type: LIST(Symbol)*LIST -> LIST
(define make-lambda
   (lambda (parameters body)
      (attach-tag (cons parameters body) 'lambda)))
```

4. **Definition expressions – 2 forms:**

   – **Syntax:** (define <var> <val>)

```
(define definition?
  (lambda (exp)
      (tagged-by? exp 'define)))

(define definition-variable
  (lambda (exp)
      (car (get-content exp))))
```

```
(define definition-value
  (lambda (exp)
    (cadr (get-content exp))))

(define make-definition
  (lambda (var value)
    (attach-tag (list var value) 'define)))
```

– **Function (procedure) definition:**

```
(define (<var> <par1> ... <parn>) <body>)

(define function-definition?
   (lambda (exp)
      (and (tagged-by? exp 'define)
           (list? (cadr exp)))))

(define function-definition-variable
   (lambda (exp)
     (caar (get-content exp))))

(define function-definition-parameters
   (lambda (exp)
     (cdar (get-content exp))))

(define function-definition-body
   (lambda (exp)
     (cdr (get-content exp))))
```

Note that we do not provide a constructor for function definition expressions, since they are derived expressions.

5. **Conditional expression – `cond`:**

```
(define cond? (lambda (exp) (tagged-by? exp 'cond)))

(define cond-clauses (lambda (exp) (cdr exp)))
(define cond-predicate (lambda (clause) (car clause)))
(define cond-actions (lambda (clause) (cdr clause)))

(define cond-first-clause (lambda (clauses) (car clauses)))
```

```
(define cond-rest-clauses (lambda (clauses) (cdr clauses)))
(define cond-last-clause? (lambda (clauses) (null? (cdr clauses))))
(define cond-empty-clauses? (lambda (clauses) (null? clauses)))
(define cond-else-clause?
   (lambda (clause) (eq? (cond-predicate clause) 'else)))

; A constructor for cond clauses:
(define make-cond-clause
   (lambda (predicate exps) (cons predicate exps)))

; A constructor for cond:
(define make-cond
   (lambda (cond-clauses)
     (attach-tag cond-clauses 'cond)))
```

6. **Conditional expression –** `if`**:**

```
(define if?
  (lambda (exp) (tagged-by? exp 'if)))

(define if-predicate
  (lambda (exp)
    (car (get-content exp))))

(define if-consequent
  (lambda (exp)
    (cadr (get-content exp))))

(define if-alternative
  (lambda (exp)
    (caddr (get-content exp))))

(define make-if
   (lambda (predicate consequent alternative)
       (attach-tag (list predicate consequent alternative) 'if)))
```

7. `let`:

```
(define let? (lambda (exp) (tagged-by? exp 'let)))
```

```
(define let-bindings
  (lambda (exp)
    (car (get-content exp))))

(define let-body
  (lambda (exp)
    (cdr (get-content exp))))

(define let-variables
   (lambda (exp)
     (map car (let-bindings exp))))

(define let-initial-values
   (lambda (exp)
     (map cadr (let-bindings exp))))

(define make-let
   (lambda (bindings body)
     (attach-tag (cons bindings body) 'let)))
```

8. `letrec`:

```
(define letrec?
   (lambda (exp) (tagged-by? exp 'letrec)))

(define letrec-bindings
  (lambda (exp)
    (car (get-content exp))))

(define letrec-body
  (lambda (exp)
    (cdr (get-content exp))))

(define letrec-variables
   (lambda (exp) (map car (letrec-bindings exp))))

(define letrec-initial-values
   (lambda (exp) (map cadr (letrec-bindings exp))))
```

```
(define make-letrec
   (lambda (bindings body)
     (attach-tag (cons bindings body) 'letrec)))

(define letrec-binding-variable
   (lambda (binding) (car binding)))

(define letrec-binding-value
   (lambda (binding) (cadr binding)))
```

9. **Procedure application expressions – any composite expression that is not one of the above:**

```
(define application? (lambda (exp) (list? exp)))

(define operator (lambda (exp) (car exp)))
(define operands (lambda (exp) (cdr exp)))
(define no-operands? (lambda (ops) (null? ops)))
(define first-operand (lambda (ops) (car ops)))
(define rest-operands (lambda (ops) (cdr ops)))

(define make-application
   (lambda (operator operands) (cons operator operands)))
```

10. **Begin:**

```
(define begin? (lambda (exp) (tagged-by? exp 'begin)))
(define begin-actions (lambda (exp) (get-content exp)))
(define make-begin (lambda (seq) (attach-tag seq 'begin)))
```

11. **Sequence:**

```
(define sequence-last-exp? (lambda (exp) (null? (cdr exp))))

(define sequence-first-exp (lambda (exps) (car exps)))

(define sequence-rest-exps (lambda (exps) (cdr exps))

(define sequence-empty? (lambda (exp) (null? exp)))
```

6.1.2 Derived expressions

Derived expressions are expressions that can be defined in terms of other expressions that the evaluator already can handle. A derived expression is not part of the language kernel: It is not directly evaluated by the evaluator. Instead, it is syntactically translated into another semantically equivalent expression that is part of the language kernel. For example, `if` can be a derived expression, defined in terms of `cond`:

```
(if (> x 0)
    x
    (if (= x 0)
        0
        (- x)))
```

can be reduced to:

```
(cond ((> x 0) x)
      (else (cond ((= x 0) 0)
                  (else (- x)))))
```

which can be optimized into

```
(cond ((> x 0) x)
      ((= x 0) 0)
      (else (- x)))
```

This is a conventional method that provides further abstraction and flexibility to languages – A compiler or interpreter does not handle explicitly the derived expressions. This way, the evaluator provides semantics and implementation only to its ***core*** (***kernel***) expressions. All other (***derived***) expressions are defined in terms of the core expressions, and therefore, are independent of the semantics and the implementation.

Management of derived expressions consists of:

1. **Overall management:**
   (a) A predicate `derived?` that identifies derived expressions.
   (b) A procedure `shallow-derive` that translates a derived expression into a kernel expression without handling of nested derived expressions.
   (c) A procedure `derive` that recursively applies `shallow-derive`.
2. **Concrete translation:** For every derived expression a ***shallow*** translation procedure is provided. For example, if `if` is a derived expression, then there is a procedure `if->cond`.

6.1.2.1 Overall management of derived expressions

```
(define derived?
  (lambda (exp)
    (or (if? exp) (function-definition? exp) (let? exp))))

; Type: [<Scheme-exp> -> <Scheme-exp>]
; Pre-condition: exp is a derived expression.
(define shallow-derive
  (lambda (exp)
    (cond ((if? exp) (if->cond exp))
          ((function-definition? exp) (function-define->define exp))
          ((let? exp) (let->combination exp))
          ((letrec? exp) (letrec->let exp))
          (else (error 'shallow-derive "unhandled derivation: ~s" exp)))))

; Type: [<Scheme-exp> -> <Scheme-exp>]
; Deep derivation -- due to the recursive application
; Handling of multiple (repeated) derivation
(define derive
  (lambda (exp)
    (if (atomic? exp)
        exp
        (let ((derived-exp
               (let ((mapped-derive-exp (map derive exp)))
                 (if (not (derived? exp))
                     mapped-derive-exp
                     (shallow-derive mapped-derive-exp)))
               ))
          (if (equal? exp derived-exp)
              exp
              (derive derived-exp))   ; Repeated derivation
          ))))
```

6.1.2.2 Concrete translations

1. `if` as a derived expression:

```
(define if->cond
  (lambda (exp)
    (let ((predicate  (if-predicate exp))
          (first-actions (list (if-consequent exp)))
          (second-actions (list (if-alternative exp))) )
      (let ((first-clause (make-cond-clause predicate first-actions))
            (second-clause (make-cond-clause 'else second-actions))
           )
        (make-cond (list first-clause second-clause))))
  ))
```

```
 > (if->cond '(if (> x 0)
                  x
                  (- x)))
 (cond ((> x 0) x) (else (- x)))
 > (if->cond '(if (> x 0)
                  x
                  (if (= x 0)
                      0
                      (- x))))
 (cond ((> x 0) x) (else (if (= x 0) 0 (- x))))
```

`if->cond` performs a shallow translation: It does not apply recursively, all the way down to nested sub-expressions. A ***deep*** `if->cond` should produce:

```
(cond ((> x 0) x) (else (cond ((= x 0) 0) (else (- x)))))
```

But, this is not needed since `derive` takes care of applying `shallow-derive` in all nested applications.

Note: The parser provides selectors and predicates for **all** language expressions, including derived ones. Note the usage of the `cond` constructor. This is typical for expressions that are used for defining other derived expressions.

2. `cond` as a derived expression: The cond expression:

```
(cond ((> x 0) x)
      ((= x 0) (display 'zero) 0)
      (else (- x)))
```

is translated into:

```
(if (> x 0)
    x
    (if (= x 0)
        (begin (display 'zero)
               0)
        (- x)))
```

```
(define cond->if
   (lambda (exp)
     (letrec
       ((sequence->exp
          (lambda (seq)
             (cond ((sequence-empty? seq) seq)
                   ((sequence-last-exp? seq) (sequence-first-exp seq))
                   (else (make-begin seq)))))
        (expand-clauses
          (lambda (clauses)
             (if (cond-empty-clauses? clauses)
                 'false             ; no else clause
                 (let
                   ((first (cond-first-clause clauses))
                    (rest (cond-rest-clauses clauses)))
                   (if (cond-else-clause? first)
                       (if (cond-empty-clauses? rest)
                           (sequence->exp (cond-actions first))
                           (error "ELSE clause isn't last -- COND->IF"
                                                      clauses))
                       (make-if (cond-predicate first)
                                (sequence->exp (cond-actions first))
                                (expand-clauses rest)))))))
          )
       (expand-clauses (cond-clauses exp)))
    ))
```

```
> (cond->if '(cond ((> x 0) x) (else (if (= x 0) 0 (- x)))))
(if (> x 0) x (if (= x 0) 0 (- x)))

> (cond->if '(cond ((> x 0) x)
                   (else (cond ((= x 0) 0)
                               (else (- x))))))
(if (> x 0) x (cond ((= x 0) 0) (else (- x))))
```

Again, this is a shallow `cond->if` translation.

3. `let` as a derived expression: The expression

```
(let ((x (+ y 2))
      (y (- x 3)))
   (* x y))
```

is equivalent to

```
((lambda (x y)
 (* x y))
   (+ y 2)
   (- x 3))

(define let->combination
  (lambda (exp)
    (let ((vars (let-variables exp))
          (body (let-body exp))
          (initial-vals (let-initial-values exp)))
    (make-application (make-lambda vars body) initial-vals))))
```

4. Procedure-definition as a derived syntax: The expression

```
(define (f x y)
   (display x) (+ x y))
```

is equivalent to

```
(define f
   (lambda (x y) (display x) (+ x y)))
```

Since the shortened procedure definition syntax provides basic notation relaxation, it makes sense to consider it as a derived expression and not as language special form.

```
(define function-define->define
  (lambda (exp)
    (let ((var (function-definition-variable exp))
          (params (function-definition-parameters exp))
          (body (function-definition-body exp)))
      (make-definition var (make-lambda params body)))))
```

6.2 A Meta-Circular Evaluator for the Substitution Model – Applicative-Eval Operational Semantics

The substitution model manages entities of three kinds:

1. ***Language expressions***.
2. The ***global environment*** mapping for "storing" defined values.

3. ***Values*** that are computed by the evaluation algorithms: Numbers, booleans, symbols, procedures, pairs and lists.

The design of each evaluator starts with the formulation of ADTs (interfaces) for these entities (as established in Chapter 3), and providing an implementation.

Language expressions are already managed by the ASP package, which treats each composite expression as an ADT, and implements constructors, selectors and predicates. All evaluators use the same ASP package. The global environment and the value concepts are formulated as ADTs and implemented in the ***Data Structures*** package. The ***Core*** package implements the operational semantics. The overall architecture of all evaluators is described in figure 6.1, in the beginning of the chapter.
Source code: Substitution-evaluator package in the course site.

6.2.1 Data Structures package

6.2.1.1 Value ADTs and their implementation

The values managed by the evaluator are numbers, booleans, symbols, procedures, pairs and lists. Values should be distinguished from syntactic expressions, since they should not be affected by repeated evaluation. Since Scheme syntax identifies syntactic expressions with computed values (e.g., a quoted symbol is also a syntactic variable; a list is also a genuine Scheme expression), the evaluator must "mark" already computed values. Therefore, **all** commuted values are tagged either as procedures, or as values (including numbers and booleans).

Indeed, algorithm `applicative-eval` introduced in Chapter **??** distinguishes already computed values from syntactic expressions. Consider, for example, the following two evaluations:

```
applicative-eval[((lambda (x)(display x) x) (quote a))] ==>
Eval:
  applicative-eval[(lambda (x)(display x) x)] ==> <Closure (x)(display x) x>
  applicative-eval[(quote a)] ==> a
Substitute: sub[x,a,(display x) x] = (display a) a
Reduce:
applicative-eval[ (display a) ] ==>
Eval:
  applicative-eval[display] ==> Code of display.
  applicative-eval['a'] ==> 'a' , since 'a' is a value of the symbol      (*)
                                        type (and not a variable!).
applicative-eval['a'] ==> 'a'
```

and also:

```
applicative-eval[((lambda (lst)(car lst)) (list 1 2 3))] ==>
Eval:
    applicative-eval[(lambda (lst)(car lst))] ==> <Closure (lst)(car lst) >
    applicative-eval[(list 1 2 3)] ==> List value '(1 2 3)'
Substitute: sub[lst,'(1 2 3)',(car lst)] = (car '(1 2 3)')
Reduce:
applicative-eval[ (car '(1 2 3)') ] ==>
Eval:
    applicative-eval[car] ==> Code of car.
    applicative-eval['(1 2 3)'] ==> '(1 2 3)'                    (*)
==> 1
```

The evaluation correctly completes because `applicative-eval` avoids repetitive evaluations (the lines marked by `(*)`. Otherwise, the first evaluation would have failed with an "unbound variable" error, and the second with "1 is not a procedure". That is, if `e` is a value of Symbol or Pair or List or Procedure (User or primitive), then `applicative-eval[e] = e`.

For that purpose, the evaluator manages ***value types*** with appropriate identification predicates, selectors and constructors. Below we define ADTs that are implemented by evaluator types for primitive procedures, user procedures (closures), and all otter values.

Primitive procedure values: A primitive procedure value is not a syntactic expression, and cannot be repeatedly evaluated, as required for example, in the evaluation of `((lambda (f)(f (list 1 2))) car)`. The evaluator must manage its own ***primitive procedure values***. The management includes the abilities to ***construct***, ***identify***, and ***retrieve the underlying implemented code***.

The `Primitive-procedure` ADT:

1. Constructor `make-primitive-procedure`: Attaches a tag to an implemented code argument.
   Type: `[T -> Primitive-procedure]`.

2. Identification predicate `primitive-procedure?`.
   Type: `[T -> Boolean]`.

3. Selector `primitive-implementation`: It retrieves the implemented code from a primitive procedure value.
   Type: `[Primitive-procedure -> T]`.

Implementation of the Primitive-procedure ADT: Primitive procedures are represented as tagged values, using the tag `primitive`.

```
Type: [T -> LIST]
 (define make-primitive-procedure
      (lambda (proc)
         (attach-tag (list proc) 'primitive)))

Type: [T -> Boolean]
 (define primitive-procedure?
    (lambda (proc)
          (tagged-by? proc 'primitive)))

Type: [LIST -> T]
(define primitive-implementation
      (lambda (proc)
         (car (get-content proc))))
```

For example:

```
> (make-primitive-procedure cons)
(primitive #<primitive:cons>)
> ( (primitive-implementation (make-primitive-procedure cons))
     1 2)
(1 . 2)
```

User procedure (closure) values: Closures should be managed as ***Procedure*** values. The management includes the abilities to ***construct, identify*** and ***select*** (***get***) the parameters and the body of the closure. That is, when the evaluator evaluates a `lambda` expression, it creates its own Procedure value. When a closure is applied, the selectors of the parameters and the body are used.
The `Procedure` ADT:

1. `make-procedure`: Attaches a tag to a list of parameters and body.
   Type: [LIST(Symbol)*LIST –> Procedure]

2. Identification predicate `compound-procedure?`.
   Type: `[T -> Boolean]`

3. Selector `procedure-parameters`.
   Type: `[Procedure -> LIST(Symbol)]`

4. Selector `procedure-body`.
   Type: `[Procedure -> LIST]`

Implementation of the User-procedure ADT: User procedures (closures) are represented as tagged values, using the tag `procedure`.

```
Type: [LIST(Symbol)*LIST -> LIST]
(define make-procedure
   (lambda (parameters body)
         (attach-tag (cons parameters body) 'procedure)))

Type: [T -> Boolean]
(define compound-procedure?
    (lambda (p)
        (tagged-by? p 'procedure)))

Type: [LIST -> LIST(Symbol)]
 (define procedure-parameters
    (lambda (p)
       (car (get-content p))))

Type: [LIST -> LIST]
(define procedure-body
    (lambda (p)
      (cdr (get-content p))))
```

Identifier `evaluator-procedure?` **for either primitive or user procedure:**.

```
Type: [LIST -> Boolean]
Purpose:  An identification predicate for procedures -- closures and primitive:
(define evaluator-procedure?
   (lambda (p)
        (or (primitive-procedure? p) (compound-procedure? p))))
```

Other values: Values result from the evaluation of numbers, booleans or application of primitive procedures. The resulting values should not be repeatedly evaluated. Therefore, the evaluator distinguishes values and evaluate them to themselves.
The Value ADT:

1. Constructor `make-value`.
   Type: `[Scheme-type-value -> Evaluator-value]`.
2. Identification predicate `value?`.
   Type: `[T -> Boolean]`.
3. Selector `value-content`.
   Type: `[Value -> Scheme-type-value]`.

Implementation of the Evaluator-list ADT:

```
Type: [Scheme-type-value -> LIST]
(define make-value
  (lambda (x) (attach-tag (list x) 'value)))

Type: [T -> Boolean]
(define value?
  (lambda (s) (tagged-by? s 'value)))

Type: [LIST -> LIST]
(define value-content
  (lambda (s) (car (get-content s))))
```

6.2.1.2 The Global Environment (GE) ADT and its implementation

The Global Environment (GE) ADT represents the global environment mapping from variables to values, used by the substitution operational semantics algorithms. In addition, in order to use primitive procedures that are already implemented in Scheme, the global environment treats primitive procedures as defined variables.
The `Global Environment` ADT:

1. `make-the-global-environment`: Creates the single value that implements this ADT, including Scheme primitive procedure bindings.
   Type: `[Empty -> GE]`
2. `lookup-variable-value`: For a given variable `var`, returns the value of the global-environment on `var` if defined, and signs an error otherwise.
   Type: `[Symbol -> T]`
3. `add-binding!`: Adds a ***binding***, i.e., a variable-value pair to the global environment mapping. Note that `add-binding` is a ***mutator***: It changes the global environment mapping to include the new binding.
   Type: `[Binding -> Void]`

Implementation of the Global Environment ADT
I. The constructor: We know that the global environment mapping is a substitution. Therefore, the GE ADT is implemented using the Substation ADT, which was already implemented for the static type inference system, using the equations method. The Global Environment type has a single value, named `the-global-environment`. This value is implemented as a substitution that maps primitive-procedure names to their built-in implementations, and other defined names to their values. The Substitution operations, used for the GE implementation are: `make-sub, get-value-of-variable, extend-sub`. The implementation of the Substitution ADT is also included in the `ge-adt.rkt` module.

```
; Type: Client view: [Void -> GE]
;       Supplier view: [Void -> LIST]
; The global environment is implemented as a substitution: A 2-element list of
; symbols and their values
(define make-the-global-environment
  (lambda ()
    (let* ((primitive-procedures
             (list (list 'car car)
                   (list 'cdr cdr)
                   (list 'cons cons)
                   (list 'null? null?)
                   (list '+ +)
                   (list '* *)
                   (list '/ /)
                   (list '> >)
                   (list '< <)
                   (list '- -)
                   (list '= =)
                   (list 'list list)
                   (list 'append append)
                     ;;      more primitives
                   ))
            (prim-variables (map car primitive-procedures))
            (prim-values (map (lambda (x) (make-primitive-procedure (cadr x)))
                              primitive-procedures)) )
      (make-sub prim-variables prim-values))
  ))

(define the-global-environment (make-the-global-environment))
```

II. The selector: The selector of the Global Environment ADT is a procedure that looks for the value of a given variable argument in the global environment:

```
; Type: [Symbol -> T]
(define lookup-variable-value
  (lambda (var)
    (get-value-of-variable the-global-environment var)))
```

If the global environment is not defined on the search variable, an error is triggered.
III. The mutator: The mutator of the Global Environment ADT is `add-binding!`, which adds a binding, i.e., a variable-value pair, to the single GE element `the-global-environment`. `add-binding!` applies the Substitution operation `extend-sub`, to `the-global-environment`

and the new variable-value pair, which creates a new, extended substitution. However, since the interpreter consults `the-global-environment` for values of variables, `add-binding!` ***changes*** the value of `the-global-environment` to the new extended substitution. The mutation is implemented using the Scheme operation `set!`, which changes the value of an already defined variable (assignment).

The implementation of `add-binding!` is not in the realm of functional programming as it is based on mutation. The mutator is used only in the evaluation of `define` forms. If the interpreted language does not include the `define` special operator, the substitution interpreter system is within the functional programming paradigm.

```
; Type: [PAIR(Symbol,T) -> Void]
(define add-binding!
  (lambda (binding)
    (let ((bvar (binding-variable binding))
          (bval (binding-value binding)))
      (set! the-global-environment (extend-sub the-global-environment bvar bval)))
  ))

;;;;;;;;;;; Bindings
; Type: [Symbol*T -> PAIR)Symbol,T)]
(define make-binding
  (lambda (var val)
    (cons var val)))

; Type: [PAIR(Symbol,T) -> Symbol]
(define binding-variable
  (lambda (binding)
    (car binding)))

; Type: [PAIR(Symbol,T) -> T]
(define binding-value
  (lambda (binding)
    (cdr binding)))
```

Once `the-global-environment` is defined, we can look for values of its defined variables:

```
> (lookup-variable-value 'cons)
'(primitive #<procedure:cons>)
> (eq? (primitive-implementation (lookup-variable-value 'cons)) cons)
#t
> ( (primitive-implementation (lookup-variable-value 'cons)) 1 2)
(1 . 2)
```

```
> (lookup-variable-value 'map)
. . get-value-of-variable: sub is an empty substitution or var is not a
variable of sub:
sub is ((car cdr cons null? + * / > < - = list append) ((primitive #<procedure:car>)
(primitive #<procedure:cdr>) (primitive #<procedure:cons>) (primitive #<procedure:null?>)
(primitive #<procedure:+>) (primitive #<procedure:*>) (primitive #<procedure:/>)
(primitive #<procedure:>>) (primitive #<procedure:<>) (primitive #<procedure:->)
(primitive #<procedure:=>) (primitive #<procedure:list>)
(primitive #<procedure:append>)));
var is map
> (add-binding! (make-binding 'map (make-primitive-procedure map)))
> ( (primitive-implementation (lookup-variable-value 'map)) - '(1 2))
(-1 -2)
```

6.2.2 Core Package: Evaluation Rules

The core of the evaluator consists of the `applicative-eval` procedure, that implements the `applicative-eval` algorithm. The main evaluation loop is created by application of closures – user defined procedures. In that case, the evaluation process is an interplay between the procedures `applicative-eval` and `apply-procedure`.

– `(applicative-eval <exp>)` evaluates `<exp>`. It calls the abstract syntax parser for the task of identifying language expressions. The helper procedures `eval-atomic, eval-special-form, eval-list` and `apply-procedure` carry the actual evaluation on the given expression. They use the abstract syntax parser selectors for getting expression components.

– `(apply-procedure <procedure> <arguments>)` applies `<procedure>` to `<arguments>`. It distinguishes between

  – Application of a primitive procedure: By calling `apply-primitive-procedure`.
  – Application of a compound procedure:
    * It substitutes the free occurrences of the procedure parameters in its body, by the argument values;
    * It sequentially evaluates the forms in the procedure body.

6.2.2.1 Main evaluator loop:

The evaluator application is preceded by ***deep replacement*** of all derived expressions by their defining expressions.

```
(define derive-eval
  (lambda (exp)
    (applicative-eval (derive exp))))
```

The input to the evaluator is either a syntactically legal kernel ***Scheme expression*** (the evaluator does not check syntax correctness) or an ***evaluator value***, i.e., an already evaluated Scheme expression. The evaluator does not support the `letrec` special operator. Therefore, the input expression cannot include inner recursive procedures.

```
; Type: [(<Scheme-exp> o "variable-value-substitution" ->
;                            (Evaluator-data-structure-value union Scheme-type)]
;    Evaluator-value is a Scheme-type value, that is marked as a value and its
;    content is any Scheme-type value, including Numbers, Booleans, Symbols, Pairs, Lists.
;    Procedures (user or primnitive) are distinguished from other values.
;    Note that the evaluator does not create closures of the underlying Scheme application.
; Pre-conditions: The given expression is legal according to the concrete syntax.
;                 No derived forms.
;                 Inner 'define' expressions are not legal.
; Post-condition: If the input is an Evaluator-value, then output=input.
(define applicative-eval
  (lambda (exp)
    (cond ((atomic? exp) (eval-atomic exp)) ;Number or Boolean or Symbol or empty
          ((special-form? exp) (eval-special-form exp))
          ((evaluator-value? exp) exp)
          ((application? exp)
           (let ((renamed-exp (rename exp)))
             (apply-procedure (applicative-eval (operator renamed-exp))
                              (list-of-values (operands renamed-exp)))))
          (else (error 'eval "unknown expression type: ~s" exp)))))

(define list-of-values
   (lambda (exps)
      (if (no-operands? exps)
          (list)
          (cons (applicative-eval (first-operand exps))
                (list-of-values (rest-operands exps))))))
```

6.2.2.2 Evaluation of atomic expressions

The identifier of atomic expressions is defined in the ASP:

```
(define atomic?
   (lambda (exp)
     (or (number? exp) (boolean? exp) (variable? exp) (null? exp))))

(define eval-atomic
  (lambda (exp)
    (if (not (variable? exp))
        (make-value exp)
        (lookup-variable-value exp))))
```

6.2.2.3 Evaluation of special forms

```
(define special-form?
   (lambda (exp)
     (or (quoted? exp) (lambda? exp) (definition? exp)
         (if? exp) (begin? exp) )))   ; cond is taken as a derived operator

(define eval-special-form
  (lambda (exp)
    (cond ((quoted? exp) (make-value exp))
          ((lambda? exp) (eval-lambda exp))
          ((definition? exp) (eval-definition exp))
          ((if? exp) (eval-if exp))
          ((begin? exp) (eval-begin exp))
          )))
```

`lambda` **expressions:**

```
(define eval-lambda
   (lambda (exp)
      (make-procedure (lambda-parameters exp)
                      (lambda-body exp))))
```

Definitions: No handling of procedure definitions – they are treated as derived expressions.

```
(define eval-definition
   (lambda (exp)
      (add-binding!
        (make-binding (definition-variable exp)
                      (applicative-eval (definition-value exp))) )
      'ok))
```

`if` expressions:

```
(define (eval-if exp)
   (if (true? (applicative-eval (if-predicate exp)))
       (applicative-eval (if-consequent exp))
       (applicative-eval (if-alternative exp))))
```

sequence evaluation:

```
(define eval-begin
   (lambda (exp)
      (eval-sequence (begin-actions exp))))

(define eval-sequence
  (lambda (exps)
    (let ((vals (map (lambda(e) (applicative-eval e))
                     exps)))
      (last vals))
    ))
```

Auxiliary procedures:

```
(define true?
  (lambda (x)
    (not (false? x))))

(define false?
  (lambda (x)
     (or (eq? x #f) (equal? x '(value #f)))))
```

6.2.2.4 Value identification

```
(define evaluator-value?
  (lambda (val) (or (value? val)
                    (primitive-procedure? val) (compound-procedure? val))))
```

6.2.2.5 Evaluation of applications

`apply-procedure` evaluates a ***form*** (a non-special combination). Its arguments are an Evaluator-procedure, i.e., a tagged procedure value that is created by the evaluator, and ***already evaluated*** arguments (the `applicative-eval` procedure first evaluates the arguments and then calls `apply procedure`). The argument values are either atomic (numbers or booleans) or tagged evaluator values. If the procedure is not primitive, `apply-procedure` carries out the ***substitute-reduce*** steps of the `applicative-eval` algorithm.

```
; Type: [Evaluator-procedure*LIST -> Evaluator-value union Scheme-type]
(define apply-procedure
  (lambda (procedure arguments)
    (cond ((primitive-procedure? procedure)
           (apply-primitive-procedure procedure arguments))
          ((compound-procedure? procedure)
           (let ((parameters (procedure-parameters procedure))
                 (body (rename (procedure-body procedure))))
             (eval-sequence
              (substitute body parameters arguments))))
          (else (error 'apply "Unknown procedure type: ~s" procedure)))))
```

Primitive procedure application: Primitive procedures are tagged data values used by the evaluator. Therefore, their implementations must be retrieved prior to application (using the selector `primitive-implementation`). The arguments are ***values***, evaluated by `applicative-eval`. Therefore, the arguments are ***values*** of the interpreter, i.e., tagged data values – for values, primitive procedures and closures. The primitive procedure implementation should receive only the ***value-content***. The result of the application should be wrapped as an evaluator value.

```
; Type: [Evaluator-primitive-procedure*LIST -> Evaluator-value]
; Retrieve the primitive implementation, retrieve content of the evaluator value
; arguments, apply and create a new evaluator value.
(define apply-primitive-procedure
  (lambda (proc args)
    (make-value
     (apply (primitive-implementation proc)
            (map (lambda (arg)
                   (cond ((evaluator-value? arg) (value-content arg))
                         ((primitive-procedure? arg) (primitive-implementation arg))
                         ((compound-procedure? arg)
                           (error 'apply-primitive-procedure
                                  "primitive-appliled-to-evaluator-procedure: ~s" arg))
                         (else arg)))
                 args)))
    ))
```

`apply` is a Scheme primitive procedure that applies a procedure on its arguments:
`(apply f e1 ... en) ==> (f e1 ... en)`.

Its type is: `[[T1*...*Tn -> T]*LIST -> T]`. For a procedure of n parameters, the list argument must be of length n, with corresponding types.
Problem with applying high order primitive procedures: The `applicative-eval` evaluator cannot apply high order Scheme primitive procedures like `map, apply`. The reason is that such primitives expect a closure as an argument. But in order to create a Scheme closure for a given `lambda` expression, `applicative-eval` has to explicitly call Scheme to evaluate the given expression. Since `applicative-eval` is implemented in Scheme, it means that the Scheme interpreter has to open a new Scheme interpreter process. Things would have been different, if Scheme would have provided a primitive for closure creation. But, since `lambda`, the value constructor of procedures is a special operator, Scheme does not enables us to retrieve its implementation, and therefore we cannot intentionally apply it to given parameters and body.

6.2.2.6 Substitution and renaming

Substitution: The `substitute` procedure substitutes free variable occurrences in an expression by given values. The expression can be either a Scheme expression or a Scheme value. `substitute` is preceded by renaming, and therefore, the substituted variables do not occur as bound in the given expression (all bound variables are already renamed).

The `substitute` operation is part of the Substitution ADT. It is the same procedure as the one used in the type inference system for the equations method, only adapted to the case of Scheme expressions and Scheme values.

```
;Signature: substitute(expr,variables,values)
; Purpose: Consistent replacement of all FREE occurrences of the substitution
;          variables in 'expr' by the values of the substitution, respectively.
;          'expr' is a Scheme expression or an Evaluator value.
;Type: Client view: [Scheme-expression*List(Variable)*List(Evaluator-value) ->
;                              Scheme-expression o "variable-value-substitution]
;Example: (substitute
;            '(lambda (x2)(+ x2 y)) '(x y z) '(3 4 (lambda (x1)(+ x1 1))))
;                                               ==>  (lambda (x2)(+ x2 4))
; Pre-conditions: (1) substitution is not performed on 'define' or 'let'
;                     expressions or on expressions containing such sub-expressions.
;                 (2) 'expr' is already renamed. Therefore, the substitution
;                     variables have no bound occurrences in expr.
(define substitute
  (lambda (expr variables values)
    (if (evaluator-value? expr)
        (substitution-application-to-value (make-sub variables values) expr)
        (substitution-application-to-expr (make-sub variables values) expr))
  ))
```

```
(define substitution-application-to-expr
  (lambda (sub expr)
    (cond ((empty-sub? sub) expr)
          ((or (number? expr) (boolean? expr) (quoted? expr)) expr)
          (else (let ((vars (get-variables sub))
                      (values (get-values sub)))
                  (cond ((variable? expr)
                         (if (member expr vars)
                             (get-value-of-variable sub expr)
                             expr))
                        (else
              ; expression is a list of expressions, lambda, application, cond.
                         (map (lambda (e)
                                (substitute e vars values))
                              expr))))))
    ))

(define substitution-application-to-value
  (lambda (sub val-expr)
    (let ((vars (get-variables sub))
          (values (get-values sub)))
      (cond ((primitive-procedure? val-expr) val-expr)
            ((value? val-expr)
             (if (atomic? (value-content val-expr))
                 val-expr
                 (make-value (map (lambda (e) (substitute e vars values))
                                  (value-content val-expr)))))
            ((compound-procedure? val-expr)
             (make-procedure (procedure-parameters val-expr)
                             (map (lambda (e) (substitute e vars values))
                                  (procedure-body val-expr))))))
  ))
```

Renaming: `rename` performs consistent renaming of bound variables.

```
Signature: rename(exp)
Purpose: Consistently rename bound variables in 'exp'.
Type: [(<Scheme-exp> union Evaluator-value) -> (<Scheme-exp> union Evaluator-value)]
(define rename
  (letrec ((make-new-names
            (lambda (old-names)
              (if (null? old-names)
                  (list)
                  (cons (gensym) (make-new-names (cdr old-names))))))
           (replace
             (lambda (val-exp)
               (cond ((primitive-procedure? val-exp) val-exp)
                     ((value? val-exp)
                      (if (atomic? (value-content val-exp))
                          val-exp
                          (make-value (map rename (value-content val-exp)))))
                     ((compound-procedure? val-exp)
                      (let* ((params (procedure-parameters val-exp))
                             (new-params (make-new-names params))
                             (renamed-subs-body (map rename
                                                     (procedure-body val-exp)))
                             (renamed-body
                           (substitute renamed-subs-body params new-params)))
                        (make-procedure new-params renamed-body))))
               )))
    (lambda (exp)
      (cond ((atomic? exp) exp)
            ((lambda? exp)
             (let* ((params (lambda-parameters exp))
                    (new-params (make-new-names params))
                    (renamed-subs (map rename exp)))
               (substitute renamed-subs params new-params)) )
                                                      ;Replace free occurrences
            ((evaluator-value?  exp) (replace exp))
            (else (map rename exp))
            ))
    ))
```

6.3 The Environment Based Operational Semantics

The major operations of a programming language evaluator are the instantiation (concretization) of abstractions:

1. ***Procedure application*** – for procedure abstractions.
2. ***Class (type) instantiation*** – for data abstractions.

In both cases, the interest of the evaluator is to minimize the instantiation overload. That is, to ***maximize reuse*** among all instantiations of an abstraction object. For that purpose, evaluators try to:

1. ***Separate*** the abstraction object from the ***concrete instantiation*** information;
2. ***Maximize*** the evaluation operation on the single ***abstraction object***, and ***minimize*** the evaluation operation in a ***concrete instantiation***. That is, ***reuse*** a single ***partially*** (maximally) ***evaluated*** abstraction object in all concrete instantiations!

For procedure abstraction it means that evaluators try to:

1. Separate the procedure from the concrete input arguments;
2. Maximize evaluation operation on the procedure object and minimize evaluation actions in every application.

In the substitution evaluator, procedure application involves the following operations:

1. Argument evaluation.
2. ***Renaming***: Repeated in every procedure application.
3. ***Substitution***: Repeated in every procedure application.
4. Reduction (requires syntax analysis).

The substitution operation applies the pairing of procedure parameters with the corresponding arguments. Renaming is an annoying by product of substitution.
The problem: Substitution requires repeated analysis of procedure bodies. In every application, the entire procedure body is repeatedly:

- Renamed
- Substituted
- Analyzed by the ASP (reduce)

The environment based operational semantics replaces substitution and renaming by a data structure – ***environment*** – that is associated with every procedure application. The environment is a finite mapping from variables (the parameters) to values (the argument values). That is, actual substitution is replaced by information needed for substitution, but is not applied (a ***lazy*** approach!).

- The ***environment based evaluator*** saves repeated renaming and substitution.
- The ***environment based compiler*** saves repeated syntax analysis of code.

We present an environment based evaluator `env-eval` that modifies the substitution based evaluator `applicative-eval`. The modifications are:

1. Data structures:

   – The simple ***static*** global environment mapping is modified into a more complex ***dynamic*** (run-time created) mapping structure, termed ***environment***.
   – The closure data structure is modified to carry an environment.
   – `env-eval` processes pure syntactic expressions, since there is no substitution step. There is clear distinction between syntactic expressions to their semantic values. Therefore, there is no need to mark already evaluated expressions as values, implying that the Value data structure is not needed. The evaluator computes plain Scheme values.

2. Evaluation rules:

   – Expressions are evaluated with respect to an environment (replaces the former substitution). The environment plays the role of a ***context*** for the evaluation.
   – The evaluation rule for procedure application is modified, so to replace substitution (and renaming) by environment creation.

6.3.1 Data Structures

6.3.1.1 The environment data structure

– **Environment terminology:**

1. An ***environment*** is a finite sequence of ***frames***: $\langle f_1, f_2, \ldots, f_n\rangle$.
2. A ***frame*** is a substitution of variables by Scheme-type values. It is also called ***symbol table***. A ***variable-value*** pair in a frame is called ***binding***.
3. Environments can ***overlap***. An environment $\langle f_1, f_2, \ldots, f_n\rangle$ includes n embedded environments: $\langle f_1, f_2, \ldots, f_n\rangle$, $\langle f_2, \ldots, f_n\rangle$, $\ldots$, $\langle f_n\rangle$, $\langle\rangle$.
   The empty sequence is called ***the empty environment***.
   The environment $\langle f_{i+1}, f_{i+2}, \ldots, f_n\rangle$ is ***the enclosing environment*** of the frame f_i in $\langle f_1, f_2, \ldots, f_n\rangle$, and f_i extends the environment $\langle f_{i+1}, f_{i+2}, \ldots, f_n\rangle$.
   Another form of environment overlapping is ***tail sharing***, i.e., environments that share their ancestor environment, as in $\langle k, f_1, f_2, \ldots, f_n\rangle$ and $\langle l, f_1, f_2, \ldots, f_n\rangle$.

– **Variable value definitions:**

1. The ***value of a variable*** x ***in a frame*** f is given by $f(x)$.
2. The ***value of a variable*** x ***in an environment*** E is the value of x in the first frame of E in which it is defined. If x is not defined in any frame of E it is ***unbound*** in E.

– **Environment structure:** Environments are dynamically created during computation: Every procedure application creates a new frame that acts as a replacement for substitution. The environment structure that is created during computation is ***a tree structure***. The ***root*** of the environment structure is a single frame environment, called ***the global environment***. The global environment is the only environment that statically exists, and provides the ***starting context*** for every computation. It holds variable bindings that are defined on top level – using the `define` special operator. Environments created during computation can only extend existing environments in the environment tree. Therefore, the global environment – the root – is the last frame in every environment. When a computation ends, only the global environment, and environments held by closures which are values of global variables, are left. All other environments are gone.

Visual notation:

– **Frames:** We draw frames as bounding boxes, with bindings written within the boxes.

– ***Environments***: We draw environments as box pointer diagrams of frames.

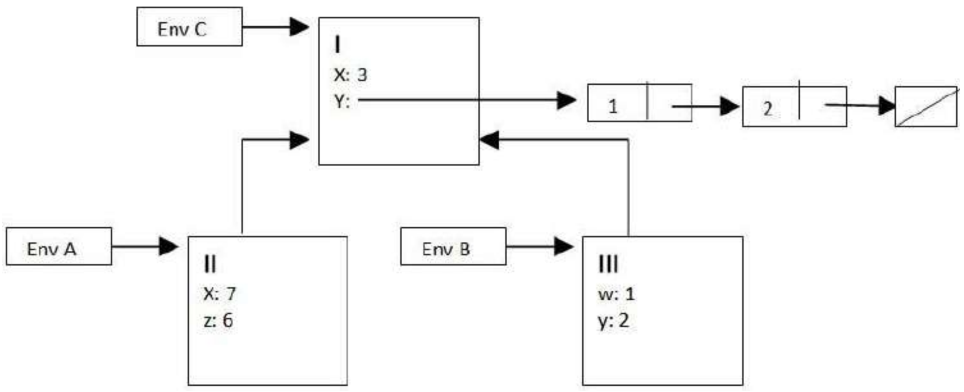


Figure 6.2: Visual representation of environments

Figure 6.2 shows 3 environments: `A, B, C`.

– Environment `C` is the ***global environment***. It consists of a single frame (global) labeled `I`.

– Environment `A` consists of the sequence of frames: `II, I`.

– Environment `B` consists of the sequence of frames: `III, I`.

- The variables `x, z` are bound in frame `II` to 7 and 6, respectively.
- The variables `x, y` are bound in frame `I` to 3 and (1 2), respectively.
- The value of `x` with respect to `A` is 7, and with respect to `B` and to `C` is 3.
- The value of `y` with respect to `A` and to `C` is (1 2), and with respect to `B` it is 2.
- With respect to environment `A`, the binding of `x` to 7 in frame `II` is said to ***shadow*** its binding to 3 in frame `I`.

Operations on environments and frames:

1. **Environment operations:**

   Constructor:
   Environment extension: Env*Frame -> Env
   $< f_1,\ f_2,\ ...,\ f_n > *f\ = < f,\ f_1, f_2,\ ...,\ f_n >$

   Selectors:
   Find variable value: <Variable>*Env -> Scheme-type
   $E(x)\ =\ f_i(x)$, where for $1\ \leq\ j\ <\ i,\ f_j(x)$ is undefined
   (= unbound)

   First-frame: Env -> Frame
   $< f_1,\ f_2,\ ...,\ f_n >_1\ =\ f_1$

   Enclosing-environment: Env -> Env
   $< f_1,\ f_2,\ ...,\ f_n >_{enclosing}\ = < f_2,\ ...,\ f_n >$

   Operation:
   Add a binding: Env*Binding -> Env
   Pre-condition: f1(x)=unbound
   $< f_1,\ f_2,\ ...,\ f_n > * < x,\ val > \ = < f_2,\ ..., f_n > *(f_1 * < x,\ val >)$

2. **Frame operations:**

   Constructor: A frame is constructed from variable and value sequences:
   $(< var_1, ..., var_n >\ ,\ < val_1, ..., val_n >)\ \Rightarrow\ [< var_1, ..., var_n > \rightarrow\ < val_1, ..., val_n >]$

   Selector: Variable value in a frame:
   f(x) or UNBOUND, if x is not defined in f.

6.3.1.2 The closure data structure

A ***closure*** in the environment model is a pair of a procedure code, i.e., parameters and body, and an environment. It is denoted `<Closure parameters body, environment>`. The components of a closure *cl* are denoted $cl_{parameters}$, $cl_{body}, cl_{environment}$.

6.3.2 The Environment Model Evaluation Algorithm

```
Signature: env-eval(e,env)
Purpose: Evaluate Scheme expressions using an Environment data structure
         for holding parameter bindings in procedure applications..
Type: <Scheme-exp>*Env -> Scheme type

env-eval[e,env] =
I. atomic?(e):
    1. number?(e) or boolean?(e): env-eval[e,env] = e.
    2. variable?(e):
       a. If env(e) is defined, env-eval[e,env] = env(e).
       b. Otherwise: e must be a variable denoting a Primitive procedure:
          env-eval[e,env] = built-in code for e.
II. composite?(e): e = ($e_0$ $e_1$ ... $e_n$) (n >= 0):
    1. $e_0$ is a Special Operator:
       env-eval[e,env] is defined by the special evaluation rule of
       e0 (see below).
    2. a. Evaluate: Compute env-eval[$e_i$,env] = $e_i$' for all $e_i$.
       b. primitive-procedure?($e_0$'):
          env-eval[e,env] = system application $e_0$'($e_1$',...,$e_n$')
       c. procedure?($e_0$'): $e_0$' = <Closure (x1 ... xn) $b_1$,...,$b_m$, env' >
          i. Environment-extension:
             new-env = env' $\circ$ [{x1, ..., xn} $\rightarrow$ {$e_1$',...,$e_n$'}]
          ii. Reduce:
              env-eval[$b_1$,new-env],...,env-eval[$b_{m-1}$,new-env]
              env-eval[e,env] = env-eval[$b_m$,new-env]

Special operator evaluation rules:
1. e = (define x $e_1$):
   GE = GE $\circ$ <x,env-eval[$e_1$,GE]>

2. e = (lambda ($x_1$ $x_2$ ... $x_n$) $b_1$ ... $b_m$) at least one $b_i$ is required:
   env-eval[e,env] = <Closure ($x_1$,...,$x_n$) ($b_1$,...,$b_m$), env)
```

```
3. e = (quote $e_1$):
   env-eval[e,env] = $e_1$

4. e = (cond ($p_1$ $e_{11}$ ... $e_{1k_1}$) ... (else $e_{n1}$ ... $e_{nk_n}$))
   If true?(env-eval[$p_1$,env]) (!= #f in Scheme):
      env-eval[$e_{11}$,env],env-eval[$e_{12}$,env],...
      env-eval[e,env] = env-eval[$e_{1k_1}$,env]
   otherwise, continue with $p_2$ in the same way.
   If for all $p_i$-s env-eval[$p_i$,env] = #f:
      env-eval[$e_{n1}$,env],env-eval[$e_{n2}$,env],...
      env-eval[e,env] = env-eval[$e_{nk_n}$,env]

5. e = (if p con alt)
   If true?(env-eval[p,env]):
       then env-eval[e,env] = env-eval[con,env]
       else env-eval[e,env] = env-eval[alt,env]

6. e = (begin $e_1$,...,$e_n$)
   env-eval[$e_1$,env],...,env-eval[$e_{n-1}$,env].
   env-eval[e,env] = env-eval[$e_n$,env].
```

Notes:

1. A new environment is created only when the computation reduces to the evaluation of a closure body. Therefore, there is a 1:many correspondence between environments and lexical scopes.
   - An environment corresponds to a lexical scope (i.e., a procedure definition body).
   - A lexical scope can correspond to multiple environments!
2. `env-eval` consults or modify the environment structure in the following steps:
   (a) Creation of a compound procedure (closure): Evaluation of a `lambda` form (and of a `let` form).
   (b) Application of a compound procedure (closure) – the only way to add a frame (also in the evaluation of a `let` form).
   (c) Evaluation of `define` form – adds a binding to the global environment.
   (d) Evaluation of a variable.
3. **De-allocation of frames:** Is not handled by the environment evaluator (left to the storage-allocation strategy of an interpreter). When the evaluation of a procedure

body is completed, if the new frame is not included within a value of some variable of another environment, the new frame turns into ***garbage*** and disappears from the environments structure.

6.3.2.1 Substitution model vs. environment model

It can be proved:
For scheme expressions in the functional programming paradigm (no destructive operations):

`applicative-eval[e] = env-eval[e,GE]`.

The two evaluation algorithms differ in the application step II.2.c.

– The ***substitute-reduce*** steps in `applicative-eval`: Rename the procedure body and the evaluated arguments, substitute the parameters by the evaluated arguments, and evaluate the substituted body,

is replaced by

– the ***environment extension*** step in `env-eval`: Bind the parameters to the evaluated arguments, extend the environment of the procedure, and evaluate the procedure body with respect to this new environment.

The effect is that explicit substitution in `applicative-eval` is replaced by variable bindings in an environment structure that tracks the scope structure in the code:

> The environment structure in a computation of env-eval is always isomorphic to
> the scope structure in the code. Therefore, the computations are equivalent.

Scheme applications are implemented using the environment model. This result enables us to run functional Scheme code as if it runs under `applicative-eval`.

6.3.2.2 Environment structures

An ***environment*** diagram is a data structure (type) that is associated with a computation of (or part of) `env-eval`. Its values carry the following data:

1. All environments constructed during the computation. Each environment is identified by its (unique) name, and its serial number in the order of environment construction. Usually, the names are `E1, E2, ...`, where the environment named `Ei` is the i-th in the construction order. The global environment is named `GE`.
2. Each environment is associated with a ***lexical block*** in the evaluated code, and its code.

3. Each environment has a ***return control link*** and a ***return value***. The return control link points to the environment where computation continues, once the evaluation of the code of the environment is over. The return value is the value of the evaluation of the code of the environment.

Environment structures are visualized, using the visual notation of environments. This notation is enriched with:

1. Environment names.
2. The lexical blocks of the environments (optional) and their codes. The global scope is marked `B0` and inner scopes are marked `B1, B2, ...`. For example:
$\boxed{\texttt{(define square (lambda(x) } \boxed{\texttt{(* x x)}}^{B1}\texttt{))}}^{B0}$
3. Return control links and return values. These links are marked with dashed arrows. Each dashed arrow points to the name of the environment that the return control link points to, and to the return value of the environment.

The dual structure of environments – Scoping and control: The tree structured lexical scopes are reflected in the tree structure of frames in an environment diagram. The sequential computation is reflected in the linear order of frames. Therefore, the environment links – also called ***access links*** – create a **tree structure**, while the return ***control links*** create a **linear order**.

Example 6.1.

```
env-eval[(define square (lambda(x)(* x x))),GE] ==>
    GE(square) =  env-eval[(lambda(x)(* x x)),GE]> =
                      = <Closure (x)(* x x),GE>
```

Lexical blocks: $\boxed{\texttt{(define square (lambda(x) } \boxed{\texttt{(* x x)}}^{B1}\texttt{))}}^{B0}$
Evaluating this form with respect to the global environment includes evaluation of the lambda form, and ***binding*** `square` to the ***created closure***, in the global environment. The resulting environment diagram is described in Figure 6.3. Note that the closure carries the Global Environment.
Note: Since the Global Environment is static, and always corresponds to the global scope `B0`, the `B0` is usually omitted from the diagram.

Example 6.2. *Continuing Example 6.1:*

```
env-eval[(square 5), GE] ==>
   env-eval[square, GE] ==> <Closure (x)(* x x), GE>
   env-eval[5, GE] ==> 5
```

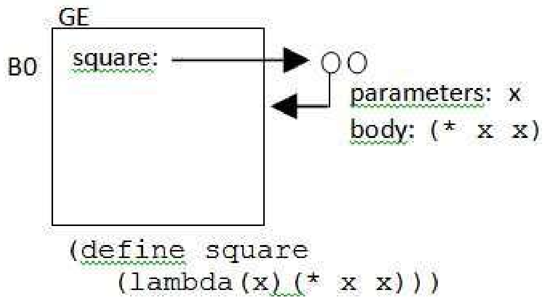


Figure 6.3

```
   Denote: E1 = GE * make-frame([x],[5])
env-eval[ (* x x), E1 ] ==>
   env-eval[*, E1] ==> <Primitive procedure *>
   env-eval[x, E1] ==> 5
   env-eval[x, E1] ==> 5
25
```

The resulting environment diagram is described in Figure 6.4.

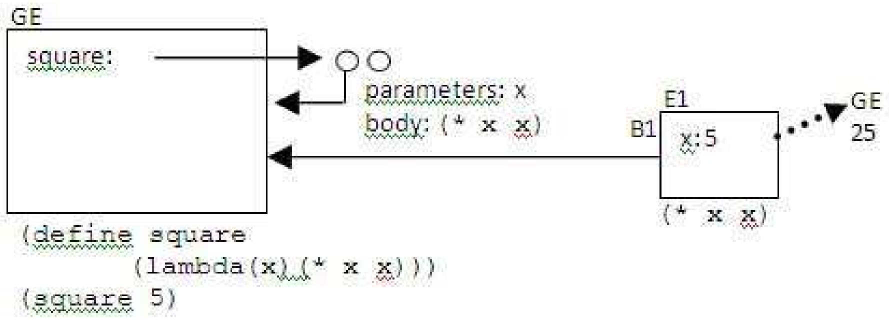


Figure 6.4

Example 6.3. *Continuing Example 6.1, assume that the following definitions are already evaluated (with respect to the Global Environment):*

```
(define sum-of-squares
   (lambda (x y)
      (+ (square x) (square y))))
```

```
(define f
   (lambda (a)
      (sum-of-squares (+ a 1) (* a 2))))
```

Lexical blocks:

$\texttt{(define sum-of-squares (lambda (x y)} \boxed{\texttt{(+ (square x) (square y))}}^{B2}\texttt{))}$

$\texttt{(define f (lambda (a)} \boxed{\texttt{(sum-of-squares (+ a 1) (* a 2))}}^{B3}\texttt{))}$

```
env-eval[(f 5), GE] ==>
   env-eval[f, GE] ==>
        <Closure (a) (sum-of-squares (+ a 1) (* a 2)), GE>
   env-eval[5, GE] ==> 5
   Denote: E1 = GE * make-frame([a],[5])
env-eval[ (sum-of-squares (+ a 1) (* a 2)), E1 ] ==>
   env-eval[sun-of-squares, E1] ==>
        <Closure (x y) (+ (square x) (square y)), GE>
   env-eval[(+ a 1), E1] ==>* 6
   env-eval[(* a 2), E1] ==>* 10
   Denote: E2 = GE * make-frame([x,y],[6,10])
env-eval[ (+ (square x) (square y)), E2 ] ==>
   env-eval[+, E2] ==> <Primitive procedure +>
   env-eval[(square x), E2] ==>
         env-eval[square, E2] ==> <Closure (x) (* x x), GE>
         env-eval[x, E2] ==> 6
         Denote: E3 = GE * make-frame([x],[6])
      env-eval[(* x x), E3] ==>* 36
   env-eval[(square y), E2] ==>
         env-eval[square, E2] ==> <Closure (x) (* x x), GE>
         env-eval[y, E2] ==> 10
         Denote: E4 = GE * make-frame([x],[10])
      env-eval[(* x x), E4] ==>* 100
136
```

The resulting environment diagram is described in Figure 6.5.

Example 6.4. *Assume that the definitions below are already evaluated. Lexical blocks are marked on the definitions.*

```
(define sum
   (lambda (term a next b)
```

$\quad\quad\boxed{\texttt{(if (> a b) 0 (+ (term a) (sum term (next a) next b)))}}^{B1}\texttt{))}$

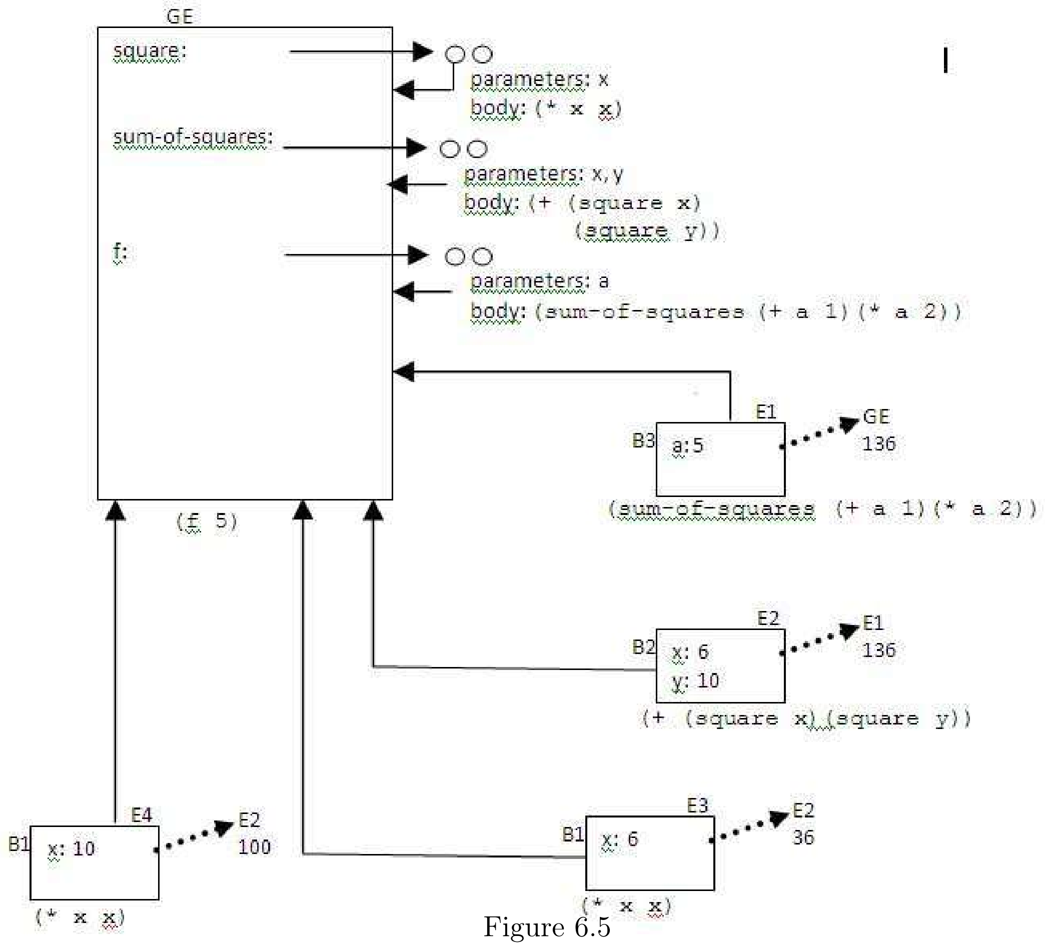


Figure 6.5

```
(define sum-integers (lambda (a b) (sum identity a add1 b)^B2))
(define identity (lambda (x) x^B3))
(define sum-cubes (lambda (a b) (sum cube a add1 b)^B4))
(define cube (lambda (x) (* x x x)^B5))
```

Describe the computation of the following expressions by an environment diagram:

```
(sum-cubes 3 5)
(sum-integers 2 4)
```

Example 6.5. *Assume that the following definitions are already evaluated:*

```
(define make-adder
    (lambda (increment)
      (lambda (x) (+ x increment) B2) B1))
(define add3 (make-adder 3))
(define add4 (make-adder 4))
```

Describe the computation of the following expressions by an environment diagram:

```
(add3 4)
(add4 4)
```

Note that the closures denoted by `add3` and `add4` are created during computations with respect to environments that correspond to lexical block $B1$. All applications of these closures create environments that correspond to block $B2$.

In the substitution model the `increment` parameter would have been substituted by 3 and 4, respectively.

Example 6.6. *Suppose that* `make-adder` *is not needed globally, and is needed only once, for the definition of* `add3`*. Describe the computation of the following expressions by an environment diagram:*

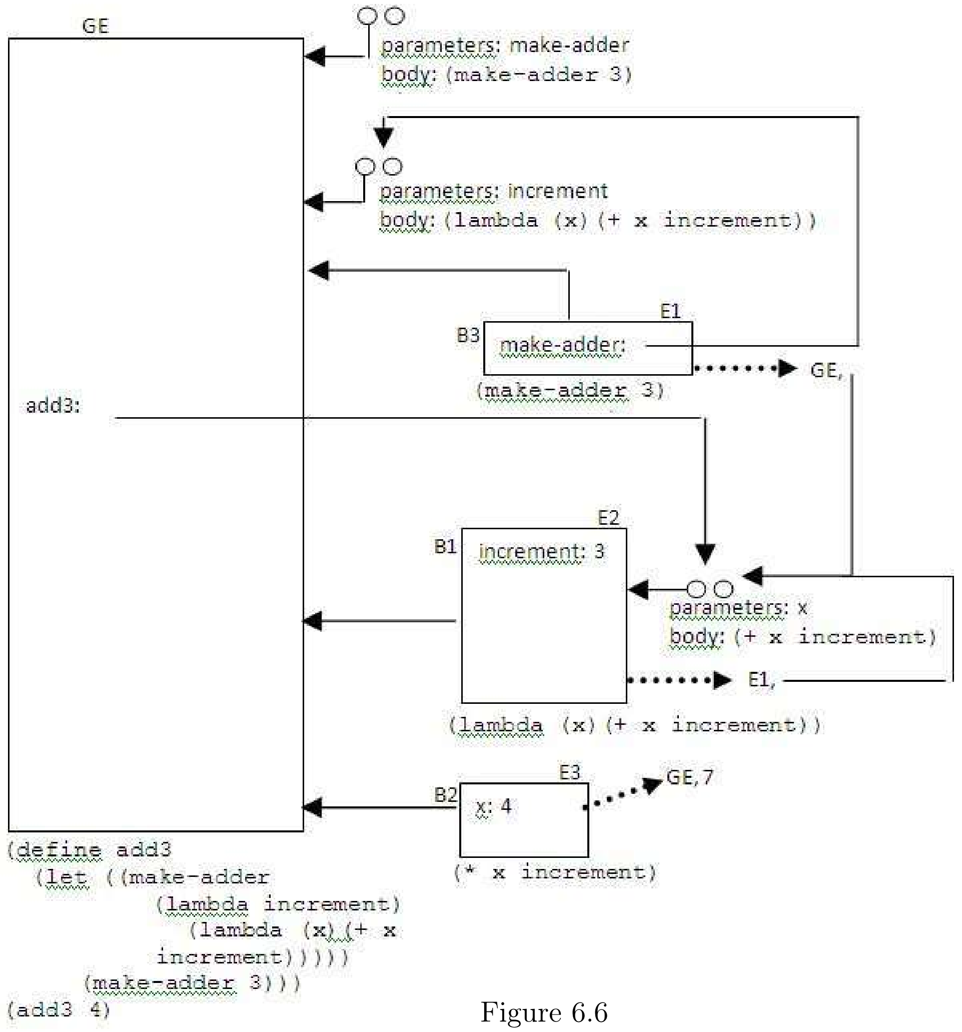
```
(define add3
    (let ((make-adder (lambda (increment)
                        (lambda (x) (+ x increment) B2) B1)) )
      (make-adder 3) B3))

(add3 4)
```

Note that prior to the definition of `add3`, two anonymous closures, `(lambda (make-adder)(make-adder 3))` and `(lambda (increment)(lambda (x)(+ x increment)))` are created, with respect to the Global Environment. Then, the application of the first closure to the second creates an environment that corresponds to lexical block $B3$. The evaluation of the expression of block $B3$ creates another environment that corresponds to lexical block $B1$. The closure denoted by `add3` is created with respect to this last environment. All applications of `add3` create environments that correspond to block $B2$.
The resulting environment diagram is described in Figure 6.6.

Example 6.7. *Describe the computation of the following expressions by an environment diagram:*

```
(define member
   (lambda (el lst)
       (cond ((null? lst) (lst))
```



```
(define add3
  (let ((make-adder
          (lambda increment)
            (lambda (x) (+ x
          increment)))))
     (make-adder 3)))
(add3 4)
```

Figure 6.6

```
             ((eq? el (car lst)) lst)
             (else (member el (cdr lst)))))
(member 'b (list 'a 'b 'c))
```

Note how all recursive calls to `member` are evaluated with respect to environments that correspond to the single lexical block defined by `member`, while the return control links follow the order of recursive calls. Indeed, since `member` is iterative, a tail recursive interpreter would point the return control link directly to the initial calling environment.

Example 6.8. *Try a* ***curried*** *version of* `member`*:*

Suppose that there are several important known lists such as `student-list`, `course-list`, to which multiple searches are applied. Then, it is reasonable to use a ***curried*** version, that defers the element binding, but enables early binding to the searched list (***partial evaluation***):

```
(define c_member
   (lambda (lst)
     (lambda (el)
       (cond ((null? lst) (lst))
             ((eq? el (car lst)) lst)
             (else
                ((c_member (cdr lst)) el)))
     )))
(define search-student-list
    (c_member (get-student-list) ))
(define search-course-list
    (c_member (get-course-list) ))
```

Partial evaluation enables evaluation with respect to a known argument, yielding a single definition and compilation of these procedures, used by all applications.
Describe the computation of several applications of `search-student-list` and of `search-course-list` by an environment diagram. Note how all recursive calls are evaluated with respect to environments that correspond to the lexical blocks defined by these procedures, while the return control links follow the order of recursive calls.

6.3.3 Static (Lexical) and Dynamic Scoping Evaluation Policies

There are two approaches for interpreting variables in a program: ***Static*** (also called ***lexical***) and ***dynamic***. The static approach is now prevailing. The dynamic approach is taken as a ***historical accident***.

The main issue of ***static scoping*** is to provide a policy for distinguishing variables in a program (as we tend to repeatedly use names). It is done by determining the correlation between a ***variable declaration*** (***binding instance***) and its ***occurrences***, based on the static code and not based on the computation order. That is, the declaration that binds a variable occurrence can be determined based on the program text (scoping structure), without any need for running the program. This property enables better compilation and management tools. This is the scoping policy used in the substitution and in the environment operational semantics.

In ***dynamic scoping***, variable occurrences are bound by the ***most recent declarations*** of that variable. Therefore, variable identifications depend on the history of the computation, i.e., on dynamic runs of a program. In different runs, a variable occurrence can be bound by different declarations, based on the computation.

The `env-eval` algorithm can be adapted into `dynamic-env-eval`, that operates under the dynamic evaluation policy, by unifying the enclosing-environment and the return control links. The modifications are:

1. A closure does not carry an environment (that corresponds to the lexical scope of the definition).

```
If e = (lambda (x1 x2 ... xn) b1 ... bm):
   dynamic-env-eval[e,env] = make-procedure((x1,...,xn),(b1,...,bm))
```

2. A closure application is evaluated with respect to its calling environment:

```
Step II.2.c of dynamic-env-eval[e, env]:
II. composite?(e): e = ($e_0$ $e_1$ ... $e_n$) (n >= 0):
    1. $e_0$ is a Special Operator:
       env-eval[e,env] is defined by the special evaluation rule of
       e0 (see below).
    2. a. Evaluate: Compute env-eval[$e_i$,env] = $e_i$' for all $e_i$.
       b. primitive-procedure?($e_0$'):
          env-eval[e,env] = system application $e_0$'($e_1$',...,$e_n$')
       c. procedure?($e_0$'): $e_0$' = <Closure (x1 ... xn) $b_1$,...,$b_m$>
          i. Environment-extension:
             new-env = env ∘ [{x1, ..., xn} → {$e_1$',...,$e_n$'}]    (***)
          ii. Reduce:
              env-eval[$b_1$,new-env],...,env-eval[$b_{m-1}$,new-env]
              env-eval[e,env] = env-eval[$b_m$,new-env]
```

Note that in dynamic scoping evaluation, the tree-like environment structure disappears, leaving only the sequential frames structure of the return-control-links. Lexical scope does not exist, and free variable occurrences in a procedure are bound by the most recent declarations, depending on the computation history. That is, bindings of free variable occurrences are determined by the ***calling*** scopes. Procedure calls that reside in different scopes, might have different declarations that bind free occurrences. Clearly, this cannot be done at compile time (because calling environments exist only at runtime), which is why this evaluation policy is called dynamic scoping evaluation. The impact is that in dynamic evaluation free variables are not used. All necessary scope information is passed as procedure parameters, yielding long parameter sequences.

Example 6.9. *Consider Example 6.6:*

```
(define add3
    (let ((make-adder (lambda (increment)
                        [(lambda (x) [(+ x increment)]B2) ]B1)) )
      [(make-adder 3)]B3))

(add3 4)
```

The computation according to the dynamic evaluation policy is described in the environment diagram in Figure 6.7. Since the `(add3 4)` expression is evaluated with respect to the Global Environment, the free valuable occurrence of `increment` in the definition of the `add3` procedure has no corresponding declaration, and therefore the result of the computation is an "unbound variable" error. Note that since under the dynamic evaluation policy environment structure is useless, only the return-control-links stay.

We see that unlike the applicative and the normal order evaluation algorithms, the static and the dynamic scoping evaluation algorithms yield different results. Therefore, **a programmer must know in advance the evaluation semantics**.

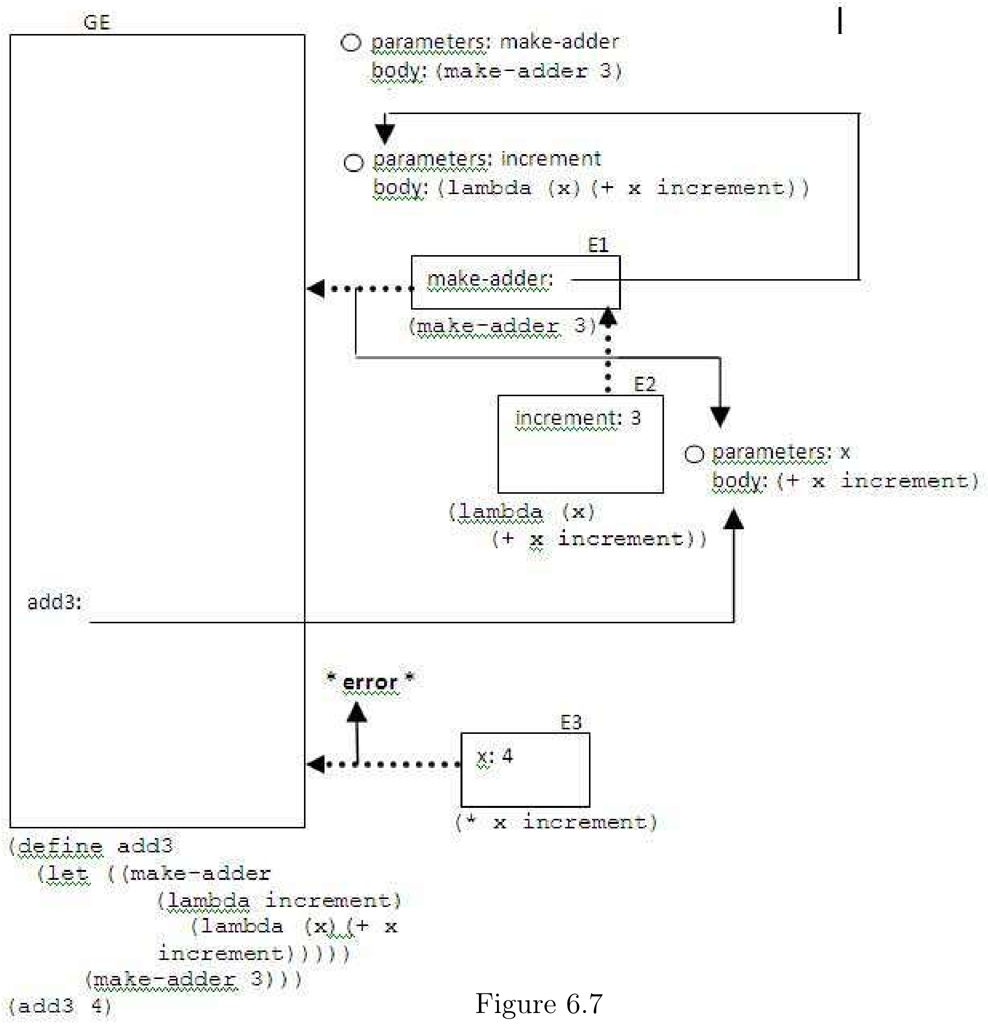

GE
parameters: make-adder
body: (make-adder 3)
parameters: increment
body: (lambda (x) (+ x increment))
E1
make-adder:
(make-adder 3)
E2
increment: 3
parameters: x
body: (+ x increment)
(lambda (x)
(+ x increment))
add3:
* error *
E3
x: 4
(* x increment)
(define add3
(let ((make-adder
(lambda increment)
(lambda (x) (+ x
increment)))))
(make-adder 3)))
(add3 4)

Figure 6.7

Example 6.10.

```
> (define f
      (lambda (x) (a x x)))
          ; 'x' is bound by the parameter of 'f', while 'a' is bound by
          ; declarations in the global scope (the entire program)
> (define g
     (lambda (a x) (f x)))
> (define a +)
          ; An 'a' declaration in the global scope.
> (g * 3)
6
```

In lexical scoping, the `a` occurrence in the body of `f` is bound by the `a` declaration in the global scope, whose value is the primitive procedure `+`. Every application of `f` is done with respect to the global environment, and therefore, `a` is evaluated to `<primitive +>`. However, in a dynamic scoping discipline, the `a` is evaluated with respect to the most recent frame where it is defined – the first frame of g's application, where it is bound to `<primitive *>`. Therefore,
`env-eval[(g * 3),GE] ==> 6`
while
`dynamic-env-eval[(g * 3),GE] ==> 9`

Example 6.11.

Assume the definitions of Example 6.10.

```
> (let ( (a 3))
    (f a))
6
```

`env-eval[(let ((a 3)) (f a)),GE] ==> 6`
while
`dynamic-env-eval[(let ( (a 3)) (f a)),GE] ==> runtime error:  3 is not a procedure.`

Example 6.12.

```
(define init 0)
(define 1+ (lambda(x)(+ x 1)))
(define f
   (lambda (f1)
      (let ((f2 (lambda () (f1 init))))
         (let ((f1 1+)
               (init 1))
          (f2) ))
   ))
```

Which is identical to:

```
(define f
   (lambda (f1)
     ( (lambda (f2)
          ( (lambda (f1 init) (f2) )
             1+ 1))
       (lambda () (f1 init)))
   ))
```

Now evaluate:

```
> (f (lambda (x) (* x x)))
```

```
env-eval[(f (lambda (x) (* x x)))] ==> 0
dynamic-env-eval[(f (lambda (x) (* x x)))] ==> 2
```

Why?

Summary of the evaluation policies discussed so far:

1. The substitution model (applicative/normal) and the environment model implement the static scoping approach;
   (a) Applicative order and environment model – eager evaluation approach.
   (b) Normal order – lazy evaluation approach.
2. (a) The (applicative) `env-eval` algorithm has a normal version (not presented). It extends the applicative version, i.e., no contradiction.
   (b) The `applicative-eval` and the `env-eval` are equivalent.
   (c) All algorithms are equivalent on the intersection of their domains.
   (d) Cost of the normal wider domain: Lower efficiency, and complexity of implementation of the normal policy.
3. The static-dynamic scoping policies: Contradicting results. The algorithm `dynamic-env-eval` is not equivalent to the other 3 algorithms.
   (a) The major drawback of the dynamic evaluation semantics is that programs cannot use free variables, since it is not known to which declarations they will be bound during computation. Indeed, in this discipline, procedures usually have long parameter lists. Almost no modern language uses dynamic evaluation. Logo and Emacs Lisp are some of the few languages that use dynamic evaluation.

(b) The implementation of dynamic evaluation is simple. Indeed, traditional LISPs used dynamic binding.

Conclusion: Programs can be indifferent to whether the operational semantics is one of the applicative, normal or environment models. But, programs cannot be switched between the above algorithms and the dynamic scoping policy.

6.4 A Meta-Circular Evaluator for the Environment Based Operational Semantics

(Environment-evaluator package in the course site.)

Recall the meta-circular evaluator that implements the substitution model for functional programming. It has the following packages:

1. Evaluation rules.
2. Abstract Syntax Parser (ASP) (for kernel and derived expressions).
3. Data structure package, for handling procedures and the Global environment.

The evaluator for the environment based operational semantics implements the `env-eval` algorithm. Therefore, the main modification to the substitution model interpreter involves the management of the data structures: Environment and Closure. The ASP package is the same for all evaluators. We first present the evaluation rules package, and then the data structures package.

6.4.1 Core Package: Evaluation Rules

The `env-eval` procedure takes an additional argument of type `Env`, which is consulted when bindings are defined, and used when a closure is created or applied.

As in the substitution evaluator, there is a single environment that statically exist: The global environment. It includes bindings to the built-in primitive procedures.

6.4.1.1 Main evaluator loop:

(SICP 4.1.1).

The core of the evaluator, as in the substitution model evaluator, consists of the `env-eval` procedure, that implements the environment model `env-eval` algorithm. Evaluation is preceded by deep replacement of derived expressions.

```
; Type: [<Scheme-exp> -> Scheme-type]
(define derive-eval
  (lambda (exp)
    (env-eval (derive exp) the-global-environment)))
```

The input to the environment based evaluator is a syntactically legal ***Scheme expression*** (the evaluator does not check syntax correctness), and an ***environment value***. The evaluator does not support the `letrec` special operator. Therefore, the input expression cannot include inner recursive procedures.

```
; Type: [<Scheme-exp>*Env -> Scheme-type]
;                      (Number, Boolean, Pair, List, Evaluator-procedure)
;       Note that the evaluator does not create closures of the
;       underlying Scheme application.
; Pre-conditions: The given expression is legal according to the concrete syntax.
;                 Inner 'define' expressions are not legal.
(define env-eval
  (lambda (exp env)
    (cond ((atomic? exp) (eval-atomic exp env))
          ((special-form? exp) (eval-special-form exp env))
          ((application? exp)
           (apply-procedure (env-eval (operator exp) env)
                            (list-of-values (operands exp) env)))
          (else (error 'eval "unknown expression type: ~s" exp)))))

; Type: [LIST -> LIST]
(define list-of-values
  (lambda (exps env)
    (if (no-operands? exps)
        '()
        (cons (env-eval (first-operand exps) env)
              (list-of-values (rest-operands exps) env)))))
```

6.4.1.2 Evaluation of atomic expressions

```
(define atomic?
   (lambda (exp)
     (or (number? exp) (boolean? exp) (variable? exp) (null? exp))))

(define eval-atomic
   (lambda (exp env)
         (if (or (number? exp) (boolean? exp) (null? exp))
             exp
             (lookup-variable-value env exp))))
```

6.4.1.3 Evaluation of special forms

```
(define special-form?
   (lambda (exp)
     (or (quoted? exp) (lambda? exp) (definition? exp)
         (if? exp) (begin? exp) )))   ; cond is taken as a derived operator

(define eval-special-form
  (lambda (exp env)
    (cond ((quoted? exp) (text-of-quotation exp))
          ((lambda? exp) (eval-lambda exp env))
          ((definition? exp)
           (if (not (eq? env the-global-environment))
               (error 'eval "non global definition: ~s" exp)
               (eval-definition exp)))
          ((if? exp) (eval-if exp env))
          ((begin? exp) (eval-begin exp env))
          )))
```

`lambda` **expressions:**

```
(define eval-lambda
   (lambda (exp env)
     (make-procedure (lambda-parameters exp)
                     (lambda-body exp)
                     env)))
```

Definition expressions: No handling of procedure definitions: They are treated as derived expressions.

```
(define eval-definition
   (lambda (exp)
      (add-binding!
         (make-binding (definition-variable exp)
                       (env-eval (definition-value exp)
                                 the-global-environment)))
     'ok))
```

`if` **expressions:**

```
(define eval-if
   (lambda (exp env)
     (if (true? (env-eval (if-predicate exp) env))
         (env-eval (if-consequent exp) env)
         (env-eval (if-alternative exp) env))))
```

Sequence evaluation:

```
(define eval-begin
   (lambda (exp env)
     (eval-sequence (begin-actions exp) env)))

(define eval-sequence
  (lambda (exps env)
    (let ((vals (map (lambda (e)(env-eval e env)) exps)))
      (last vals))))
```

Auxiliary procedures:

```
(define true?
   (lambda (x) (not (eq? x #f))))

(define false?
   (lambda (x) (eq? x #f)))
```

6.4.1.4 Evaluation of applications

`apply-procedure` evaluates a ***form*** (a non-special combination). Its arguments are an Evaluator-procedure, i.e., a tagged procedure value that is created by the evaluator, and ***already evaluated*** arguments (the `env-eval` procedure first evaluates the arguments and then calls `apply procedure`). The argument values are either Scheme values (numbers, booleans, pairs, lists, primitive procedure implementations) or tagged evaluator values of procedures or of primitive procedures. If the procedure is not primitive, `apply-procedure` carries out the ***environment-extension-reduce*** steps of the `env-eval` algorithm.

```
; Type: [Evaluator-procedure*LIST -> Scheme-type]
(define apply-procedure
  (lambda (procedure arguments)
    (cond ((primitive-procedure? procedure)
           (apply-primitive-procedure procedure arguments))
          ((compound-procedure? procedure)
           (let* ((parameters (procedure-parameters procedure))
                  (body (procedure-body procedure))
                  (env (procedure-environment procedure))
```

```
                 (new-env (extend-env (make-frame parameters arguments) env)))
            (eval-sequence body new-env)))
         (else (error 'apply "unknown procedure type: ~s" procedure)))))
```

Primitive procedure application: Primitive procedures are tagged data values used by the evaluator. Therefore, their implementations must be retrieved prior to application (using the selector `primitive-implementation`). The arguments are ***values***, evaluated by `env-eval`. Therefore, the arguments are either Scheme numbers, booleans, pairs, lists, primitive procedure implementations, or tagged data values of procedures or of primitive procedures.

```
; Type: [Evaluator-primitive-procedure*LIST -> Scheme-type]
; Purpose: Retrieve the primitive implementation, and apply to args.
(define apply-primitive-procedure
   (lambda (proc args)
      (apply (primitive-implementation proc) args)))
```

6.4.2 Data Structures Package

6.4.2.1 Procedure ADTs and their implementation

The environment based evaluator manages values for ***primitive procedure***, and for ***user procedure***. User procedures are managed since the application mechanism must retrieve their parameters, body and environment. Primitive procedures are managed as values since the evaluator has to ***distinguish*** them from user procedures. For example, when evaluating `(car (list 1 2 3))`, after the ***Evaluate*** step, the evaluator must identify whether the value of `car` is a primitive implementation or a user procedure[1].

Primitive procedure values: The ADT for primitive procedures is the same as in the substitution evaluator.
The ADT:

1. Constructor `make-primitive-procedure`: Attaches a tag to an implemented code argument.
   Type: `[T -> Primitive-procedure]`.

2. Identification predicate `primitive-procedure?`.
   Type: `[T -> Boolean]`.

3. Selector `primitive-implementation`: It retrieves the implemented code from a primitive procedure value.
   Type: `[Primitive-procedure -> T]`.

[1]In the substitution evaluator there was also the problem of preventing repeated evaluations.

Implementation of the Primitive-procedure ADT: Primitive procedures are represented as tagged values, using the tag `primitive`.

```
Type: [T --> LIST]
 (define make-primitive-procedure
      (lambda (proc)
         (attach-tag (list proc) 'primitive)))

Type: [T -> Boolean]
 (define primitive-procedure?
    (lambda (proc)
          (tagged-by? proc 'primitive)))

Type: [LIST -> T]
(define primitive-implementation
      (lambda (proc)
         (car (get-content proc))))
```

User procedure (closure) values: The tagged ***Procedure*** values of `env-eval` are similar to those of `applicative-eval`. The only difference involves the ***environment*** component, used both in construction and in selection.
The ADT:

1. `make-procedure`: Attaches a tag to a list of parameters and body.
   Type: [LIST(Symbol)*LIST*Env –> Procedure]
2. Identification predicate `compound-procedure?`.
   Type: `[T -> Boolean]`
3. Selector `procedure-parameters`.
   Type: `[Procedure -> LIST(Symbol)]`
4. Selector `procedure-body`.
   Type: `[Procedure -> LIST]`
5. Selector `procedure-environment`.
   Type: `[Procedure -> Env]`

Implementation of the User-procedure ADT: User procedures (closures) are represented as tagged values, using the tag `procedure`.

```
Type: [LIST(Symbol)*LIST*Env -> LIST]
(define make-procedure
   (lambda (parameters body env)
            (attach-tag (list parameters body env) 'procedure)))

Type: [T -> Boolean]
(define compound-procedure?
    (lambda (p)
        (tagged-by? p 'procedure)))

Type: [LIST -> LIST(Symbol)]
 (define procedure-parameters
    (lambda (p)
       (car (get-content p))))

Type: [LIST -> LIST]
(define procedure-body
    (lambda (p)
      (cadr (get-content p))))

Type: [LIST -> Env]
(define procedure-environment
   (lambda (p)
     (caddr (get-content  p))))

Type: [T -> Boolean]
Purpose:  An identification predicate for procedures -- closures and primitive:
(define procedure?
   (lambda (p)
        (or (primitive-procedure? p) (compound-procedure? p))))
```

6.4.2.2 Environment related ADTs and their implementations:

The environment based operational semantics has a rich environment structure. Therefore, the interface to environments includes three ADTs: Env, Frame, Binding. The Env ADT is implemented on top of the Frame ADT, and both are implemented on top of the Binding ADT. Frame is implemented on top of the Substitution ADT.

The Env ADT and its implementation:
The ADT:

1. `make-the-global-environment()`: Creates the single value that implements this ADT, including Scheme primitive procedure bindings.
   Type: `[Void -> GE]`

2. `extend-env(frame,base-env)`: Creates a new environment which is an extension of `base-env` by `frame`.
   Type: `[Frame*Env -> Env]`

3. `lookup-variable-value(env,var)`: For a given variable `var`, returns the value of `env` on `var` if defined, and signs an error otherwise.
   Type: `[Symbol*Env -> T]`

4. `first-frame(env)`: Retrieves the first frame.
   Type: `[Env -> Frame]`

5. `enclosing-env(env)`: Retrieves the enclosing environment.
   Type: `[Env -> Env]`

6. `empty-env?(env)`: checks whether `env` is empty.
   Type: `[Env -> Boolean]`

7. `add-binding!(binding)`: Adds a ***binding***, i.e., a variable-value pair to the global environment mapping. Note that `add-binding` is a ***mutator***: It changes the global environment mapping to include the new binding.
   Type: `[PAIR(Symbol,T) -> Void]`

Implementation of the Env ADT: An environment is a sequence of frames, which are substitutions of Scheme variables by Scheme values. Environments are implemented as lists of frames. The end of the list is `the-empty-environment`.

The global environment requires mutation, in order to implement the effect of the `define` special operator. Since the global environment is the last enclosing environment in every environment, it means that **every environment must allow changes in its frames** (actually only in its last non-empty frame). Therefore, The list of frames which implements an environment must consist of ***mutable values***. In DrRacket, mutable values are implemented using the `Box` data structure, whose values are created using the constructor `box`, retrieved using the selector `unbox`, and changed using the mutator `set-box!`.

```
;;;  Global environment construction:
(define the-empty-environment '())

; Type Client view: [Void -> GE]
;      Supplier view: [Void -> LIST(Box(LIST)) union {empty}
(define make-the-global-environment
  (lambda ()
    (let* ((primitive-procedures
             (list (list 'car car)
                   (list 'cdr cdr)
                   (list 'cons cons)
                   (list 'null? null?)
                   (list '+ +)
                   (list '* *)
                   (list '/ /)
                   (list '> >)
                   (list '< <)
                   (list '- -)
                   (list '= =)
                   (list 'list list)
                   (list 'box box)
                   (list 'unbox unbox)
                   (list 'set-box! set-box!)
                   ;; more primitives
                   ))
           (prim-variables (map car primitive-procedures))
           (prim-values (map (lambda (x) (make-primitive-procedure (cadr x)))
                               primitive-procedures))
           (frame (make-frame prim-variables prim-values)))
    (extend-env frame the-empty-environment))))

(define the-global-environment (make-the-global-environment))

;;;  Environment operations:
; Environment constructor: ADT type is [Frame*Env -> Env]
; An environment is implemented as a list of boxed frames. The box is
; needed because the first frame, i.e., the global environment, is
; changed following a variable definition.
; Type: [[Symbol -> PAIR(Symbol,T) union {empty}]*
;       LIST(Box([Symbol -> PAIR(Symbol,T) union {empty}])) ->
;                          LIST(Box([Symbol -> PAIR(Symbol,T) union {empty}]))]
(define extend-env
  (lambda (frame base-env)
    (cons (box frame) base-env)))

; Environment selectors
; Input type is an environment, i.e.,
;                           LIST(Box([Symbol -> PAIR(Symbol,T) union {empty}]))
(define enclosing-env (lambda (env) (cdr env)))
(define first-boxed-frame (lambda(env) (car env)))
(define first-frame (lambda(env) (unbox (first-boxed-frame env))))
```

```
; Environment selector: ADT type is [Var*Env -> T]
; Purpose: If the environment is defined on the given variable, selects its value
; Type: Client view: [Env*Variable -> T]
;     Supplier side: [LIST(Box(Frame))*Symbol -> T]
;Implementation: Uses the frame->lookup-variable-value getter of frames.
(define lookup-variable-value
  (lambda (env var)
    (letrec ((defined-in-env       ; ADT type is [Var*Env -> Binding union {empty}]
               (lambda (var env)
                 (if (empty-env? env)
                     (error 'lookup "variable not found: ~s\n  env = ~s" var env)
                     (frame->lookup-variable-value
                       (first-frame env)
                       var
                       (lambda () (defined-in-env var (enclosing-env env))))))
               ))
      (defined-in-env var env))
    ))

; Environment identification predicate
; Type: [T -> Boolean]
(define empty-env?
  (lambda (env)
    (eq? env the-empty-environment)))

; Global environment mutator: ADT type is [Binding -> Void]
; Type: [PAIR(Symbol,T) -> Void]
; Note: Mutation is enabled only for the global environment
(define add-binding!
  (lambda (binding)
    (let ((frame (first-frame the-global-environment)))
      (set-box! (first-boxed-frame the-global-environment)
                (extend-frame binding frame)))
    ))
```

Note the delicate handling of boxed frames: The (single) element of `the-global-environment` is a boxed frame. Evaluation of a `define` expression requires changing the value that is stored in this frame. As noted already for the substitution interpreter, the `add-binding!` mutator turns the interpreter itself into a **non-Function Programming** application, since it supports. changing the value (state) of the `the-global-environment` variable.

The Frame ADT and its implementation:
The ADT:

1. `make-frame(variables,values)`: Creates a new frame from the given variables and values.
   Type: `[LIST(Symbol)*LIST -> Frame]`
   Pre-condition: number of variables = number of values

2. `extend-frame`: Creates a new frame that includes the new binding and all previous bindings in the given frame.
   Type: `[Binding*Frame -> Frame]`

3. `frame->lookup-variable-value`: Gets the value of a given variable, if defined. If undefined, applies the given fail continuation.
   Type: `[Frame*Variable*Procedure -> T]`

Implementation of the Frame ADT: A frame is implemented as a substitution from Scheme variables to Scheme values:

```
; Frame constructor: A frame is a mapping from Scheme variables to Scheme values.
; Type: Client view: [[LIST(Symbol)*LIST -> Frame]
; Implementation: A substitution from Scheme variables to Scheme values.
; Precondition: All preconditions for corerct Frame implementation are already
;               checked in the Sunstitution constructor.
(define make-frame
  (lambda (variables values)
    (make-sub variables values)
    ))

; Frame constructor:
; Purpose: Produces a new frame that extends the given frame with the new binding.
; Type: Client view: [Binding*Frame -> Frame]
(define extend-frame
  (lambda (binding frame)
    (let ((bvar (binding-variable binding))
          (bval (binding-value binding)))
      (extend-sub frame bvar bval))
 ))

;Frame getter:
; Purpose: Get the value of a given variable. If undefined, apply given fail continuation
; Type: Client view:  [Frame*Variable*Procedure -> T]
(define frame->lookup-variable-value get-value-of-variable)
```

```
         ;The substitution getter (lambda (sub var fail-cont) ...}
```

Side note: Alternatively, a frame (and a substitution) can be implemented as a ***lookup procedure*** for a variable value, in the lists of variables and values that construct the frame:

```
(define make-frame
  (lambda (variables values)
    (letrec ((lookout
              (lambda (vars vals)
                (lambda (var)
                  (cond ((empty? vars) empty)
                        ((eq? var (car vars)) (make-binding (car vars) (car vals)))
                        (else ((lookout (cdr vars) (cdr vals))
                               var)))))))
      (lookout variables values))
  ))
```

The Binding ADT and its implementation:
The ADT:

1. `make-binding(var,val)`: Creates a binding.
   Type: `[Symbol*T -> Binding]`
2. Two selectors for the value and the value: `binding-variable, binding-value`, with types `[Binding -> Symbol]` and `[Binding -> T]`, respectively.

Implementation of the Binding ADT: Bindings are implemented as pairs.

```
Type: [Symbol*T --> PAIR(Symbol,T)]
(define make-binding
   (lambda (var val)
      (cons var val)))

Type: [PAIR(Symbol,T) -> Symbol]
(define binding-variable
   (lambda (binding)
      (car binding)))

Type: [PAIR(Symbol,T) -> T]
(define binding-value
   (lambda (binding)
      (cdr binding)))
```

6.5 A Meta-Circular Compiler for Functional Programming (SICP 4.1.7)

The `env-eval` evaluator improves the `applicative-eval`, by replacing environment association for renaming and substitution. Yet, it does not handle the repetition of code analysis in every procedure application. The problem is that syntax analysis is mixed within evaluation. There is no separation between:

– ***Static analysis***, to

– ***Run time*** evaluation.

A major role of a compiler is static (compile time) syntax analysis, that is separated from run time execution.

Consider a recursive procedure:

```
(define (factorial n)
  (if (= n 1)
      1
     (* (factorial (- n 1)) n)))
```

Its application on a number `n` applies itself additional `n-1` times. In each application the procedure code is repeatedly analyzed. That is, `eval-sequence` is applied to `factorial` body, just to find out that there is a single `if` expression. Then, the predicate of that `if` expression is repeatedly retrieved, implying a repeated analysis of `(= n 1)`. Then, again, `(* (factorial (- n 1)) n)` is repeatedly analyzed, going through the case analysis in `env-eval` over and over. In every application of `factorial`, its body is repeatedly retrieved from the closure data structure.

Example 6.13. *Trace the evaluator execution.*

```
> (require-library "trace.ss")
> (trace eval)
(eval)
> (trace apply-procedure)
(apply-procedure)

 *** No analysis of procedure (closure bodies): ***
> (eval
        '(define (factorial n)
          (if (= n 1)
      1
      (* (factorial (- n 1)) n))) )
```

```
|(eval (define (factorial n) (if (= n 1) 1
                                         (* (factorial (- n 1)) n)))
       (((false true car cdr cons null? = * -)
         #f
         #t
         (primitive #<primitive:car>)
         (primitive #<primitive:cdr>)
         (primitive #<primitive:cons>)
         (primitive #<primitive:null?>)
         (primitive #<primitive:=>)
         (primitive #<primitive:*>)
         (primitive #<primitive:->))))
| (eval (lambda (n) (if (= n 1) 1 (* (factorial (- n 1)) n)))
        <<the-global-environment>>)
| (procedure
    (n)
    ((if (= n 1) 1 (* (factorial (- n 1)) n)))
    <<the-global-environment>>)

(factorial 3)
 *** |(EVAL (FACTORIAL 3)
       <<THE-GLOBAL-ENVIRONMENT>>)
| (eval factorial
       <<the-global-environment>>)
| #1=(procedure
       (n)
       ((if (= n 1) 1 (* (factorial (- n 1)) n)))
       <<the-global-environment>>)
| (eval 3
       <<the-global-environment>>)
| 3
 *** | (apply-procedure #2=(procedure
              (n)
              ((if (= n 1) 1 (* (factorial (- n 1)) n)))
              <<the-global-environment>>))
         (3))
 *** | |(EVAL #3=(IF (= N 1) 1 (* (FACTORIAL (- N 1)) N))
         ((#6=(n) 3)
          .
          #8= <<the-global-environment>>))
```

```
| | (eval #3=(= n 1)
        ((#6=(n) 3)
         .
         #8= <<the-global-environment>>))
| | |(eval =
         ((#4=(n) 3)
          .
          #6= <<the-global-environment>>))
| | |(primitive #<primitive:=>)
| | |(eval n
         ((#4=(n) 3)
          .
          #6= <<the-global-environment>>))
| | |3
| | |(eval 1
         ((#4=(n) 3)
          .
          #6= <<the-global-environment>>))
| | |1
| | |(apply-procedure (primitive #<primitive:=>) (3 1))
| | |#f
| | #f
| | (eval #3=(* (factorial (- n 1)) n)
        ((#6=(n) 3)
         .
         #8= <<the-global-environment>>))
| | |(eval *
         ((#4=(n) 3)
          .
          #6= <<the-global-environment>>))
| | |(primitive #<primitive:*>)
| | |(eval #3=(factorial (- n 1))
         ((#6=(n) 3)
          .
          #8= <<the-global-environment>>))
| | | (eval factorial
          ((#4=(n) 3)
           .
           #6= <<the-global-environment>>))
| | | #1=(procedure
         (n)
```

```
           ((if (= n 1) 1 (* (factorial (- n 1)) n)))
           <<the-global-environment>>)
| | | (eval #3=(- n 1)
            ((#6=(n) 3)
             .
             #8= <<the-global-environment>>))
| | | |(eval -
             ((#4=(n) 3)
              .
              #6= <<the-global-environment>>))
| | | |(primitive #<primitive:->)
| | | |(eval n
             ((#4=(n) 3)
              .
              #6= <<the-global-environment>>))
| | | |3
| | | |(eval 1
             ((#4=(n) 3)
              .
              #6= <<the-global-environment>>))
| | | |1
| | | |(apply-procedure (primitive #<primitive:->) (3 1))
| | | |2
| | | 2
 *** | | | (apply-procedure #2=(procedure
                    (n)
                    ((if (= n 1) 1 (* (factorial (- n 1)) n)))
                    <<the-global-environment>>)
             (2))
 *** | | | |(EVAL #3=(IF (= N 1) 1 (* (FACTORIAL (- N 1)) N))
             ((#6=(N) 2)
              .
              #8= <<THE-GLOBAL-ENVIRONMENT>>))
| | | | (eval #3=(= n 1)
              ((#6=(n) 2)
               .
               #8= <<the-global-environment>>))
| | | | |(eval =
               ((#4=(n) 2)
                .
                #6= <<the-global-environment>>))
```

```
| | | | |(primitive #<primitive:=>)
| | | | |(eval n
              ((#4=(n) 2)
               .
               #6= <<the-global-environment>>))
| | | | |2
| | | | |(eval 1
              ((#4=(n) 2)
               .
               #6= <<the-global-environment>>))
| | | | |1
| | | | |(apply-procedure (primitive #<primitive:=>) (2 1))
| | | | |#f
| | | | #f
| | | | (eval #3=(* (factorial (- n 1)) n)
             ((#6=(n) 2)
              .
              #8= <<the-global-environment>>))
| | | | |(eval *
              ((#4=(n) 2)
               .
               #6= <<the-global-environment>>))
| | | | |(primitive #<primitive:*>)
| | | | |(eval #3=(factorial (- n 1))
              ((#6=(n) 2)
               .
               #8= <<the-global-environment>>))
| | | | | (eval factorial
               ((#4=(n) 2)
                .
                #6= <<the-global-environment>>))
| | | | | #1=(procedure
              (n)
              ((if (= n 1) 1 (* (factorial (- n 1)) n)))
              <<the-global-environment>>)
| | | | | (eval #3=(- n 1)
               ((#6=(n) 2)
                .
                #8= <<the-global-environment>>))
| | | |[10](eval -
               ((#4=(n) 2)
                  .
                  #6= <<the-global-environment>>))
| | | |[10](primitive #<primitive:->)
| | | |[10](eval n
                 ((#4=(n) 2)
                  .
                  #6= <<the-global-environment>>))
| | | |[10]2
| | | |[10](eval 1
                 ((#4=(n) 2)
                  .
                  #6= <<the-global-environment>>))
| | | |[10]1
| | | |[10](apply-procedure (primitive #<primitive:->) (2 1))
| | | |[10]1
| | | | | 1
 *** | | | | | (apply #2=(procedure
                        (n)
                        ((if (= n 1) 1 (* (factorial (- n 1)) n)))
                        <<the-global-environment>>))
                 (1))
 *** | | | |[10](EVAL #3=(IF (= N 1) 1 (* (FACTORIAL (- N 1)) N))
                 ((#6=(N) 1)
                  .
                  #8= <<THE-GLOBAL-ENVIRONMENT>>))
| | | |[11](eval #3=(= n 1)
                 ((#6=(n) 1)
                  .
                  #8= <<the-global-environment>>))
| | | |[12](eval =
                 ((#4=(n) 1)
                  .
                  #6= <<the-global-environment>>))
| | | |[12](primitive #<primitive:=>)
| | | |[12](eval n
                 ((#4=(n) 1)
                  .
                  #6= <<the-global-environment>>))
| | | |[12]1
| | | |[12](eval 1
                 ((#4=(n) 1)
                  .
                  #6= <<the-global-environment>>))
| | | |[12]1
| | | |[12](apply-procedure (primitive #<primitive:=>) (1 1))
| | | |[12]#t
| | | |[11]#t
| | | |[11](eval 1
                 ((#4=(n) 1)
                  .
                  #6= <<the-global-environment>>))
| | | |[11]1
| | | |[10]1
| | | | | 1
| | | | |1
| | | | |(eval n
               ((#4=(n) 2)
                .
                #6= <<the-global-environment>>))
| | | | |2
| | | | |(apply-procedure (primitive #<primitive:*>) (1 2))
| | | | |2
| | | | 2
| | | |2
| | | 2
| | |2
| | |(eval n
           ((#4=(n) 3)
            .
            #6= <<the-global-environment>>))
| | |3
| | |(apply (primitive #<primitive:*>) (2 3))
| | |6
| | 6
| |6
| 6
|6
```

The body of the `factorial` procedure was analyzed 4 times. Assume now that we compute

```
> (factorial 4)
24
```

The code of `factorial` body is again analyzed 5 times. The problem:

> `env-eval` performs code analysis and evaluation simultaneously, which leads to ***major inefficiency*** due to repeated analysis.

Evaluation tools distinguish between

– **Compile time (static time)**: Things performed ***before*** evaluation, to

– ***Run time (dynamic time)***: Things performed ***during*** evaluation.

Clearly: Compile time is ***less expensive*** than run time. Analyzing a procedure body once, independently from its application, means compiling its code into something more efficient/optimal, which is ready for evaluation. This way: The major syntactic analysis is done just once!

6.5.1 The Analyzer

Recall that the environment evaluation model improves the substitution model by replacing renaming and substitution in procedure application by environment generation (and environment lookup for finding the not substituted value of a variable). The environment model does not handle the problem of repeated analyses of procedure bodies. This is the contribution of the analyzer: A single analysis in static time, for every procedure.

The analyzing `env-eval` improves env-eval by preparing a procedure that is ***ready for execution***, once an environment is given. It produces a true compilation product.

1. **Input to the syntax analyzer:** Expression in the analyzed language (Scheme).
2. **Output of the syntax analyzer:** A procedure of the target language (Scheme).

Analysis considerations:

1. Determine which parts of the `env-eval` work can be performed statically:
   – Syntax analysis;
   – Translation of the evaluation code that is produced by `env-eval` into an implementation code that is ready for evaluation, in the target language.
2. Determine which parts of the `env-eval` work cannot be performed statically – generation of data structure that implement evaluator values, and environment consultation:
   – Environment construction.
   – Variable lookup.
   – Actual procedure construction, since it is environment dependent.

Since all run-time dependent information is kept in the environments, the compile-time – run-time separation can be obtained by performing ***abstraction on the environment***: The `env-eval` definition

```
(define env-eval
   (lambda (exp env)
    <body>))
```

turns into:

```
(define env-eval
  (lambda (exp)
     (lambda (env)
       <analyzed-body>))
```

That is: `analyze: <Scheme-exp> -> <Closure (env) ...>` The analyzer env-eval can be viewed as a ***Curried*** version of `env-eval`.

Therefore, the derive-analyze-eval is defined by:

```
; Type: [<Scheme-exp> -> [(Env -> Scheme-type)]]
 (define (derive-analyze-eval exp)
   ((analyze (derive exp)) the-global-environment))
```

where, the analysis of `exp` and every sub-expression of `exp` is performed only once, and creates already compiled Scheme procedures. When run time input is supplied (the `env` argument), these procedures are applied and evaluation is completed.

`analyze` is a compiler that performs ***partial evaluation*** of the `env-eval` computation. We can even separate analysis from evaluation by saving the compiled code:

```
 > (define exp1 '<some-Scheme-expression>)
 > (define compiled-exp1 (analyze (derive exp1)))
 > (compiled-exp1 the-global-environment)
```

Compiled-exp1 is a compiled program (a Scheme expression) that can be evaluated by applying it to `the-global-environment` variable.

There are two principles for switching from `env-eval` code to a Curried analyze code.

1. Curry the `env` parameter.

2. Inductive application of `analyze` on all sub-expressions.

The `Env-eval -> analyzer` transformation is explained separately for every kind of Scheme expressions.

6.5.1.1 Atomic expressions:

In the `env-eval`:

```
(define eval-atomic
  (lambda (exp env)
      (if (or (number? exp) (boolean? exp) (null? exp))
          exp
          (lookup-variable-value env exp))))
```

Here we wish to strip the environment evaluation from the static analysis:

```
(define analyze-atomic
  (lambda (exp)
      (if (or (number? exp) (boolean? exp) (null? exp))
          (lambda (env) exp)
          (lambda (env) (lookup-variable-value env exp))
      )))
```

Discussion: What is the difference between the above and:

```
(define analyze-atomic
  (lambda (exp)
      (lambda (env)
        (if (or (number? exp) (boolean? exp) (null? exp))
            exp
            (lookup-variable-value exp env)
      ))))
```

Analyzing a variable expression produces a procedure that at ***run time*** needs to scan the given environment. This is still – a run time excessive overhead. More optimal compilers prepare at compile time code for construction of a ***symbol table***, and replace the above run time lookup by an instruction for direct access to the table.

6.5.1.2 Composite expressions:

Analysis of composite expressions requires inductive thinking. Before Currying, the analyzer is applied to the sub-expressions! Therefore, there are two steps:

1. Apply syntax analysis to sub-expressions.
2. Curry.

In the `env-eval`:

```
(define eval-special-form
  (lambda (exp env)
      (cond ((quoted? exp) (text-of-quotation exp))
            ((lambda? exp) (eval-lambda exp env))
            ((definition? exp)
             (if (not (eq? env the-global-environment))
                 (error "Non global definition" exp)
                 (eval-definition exp)))
            ((if? exp) (eval-if exp env))
            ((begin? exp) (eval-begin exp env))
      )))
```

In the syntax analyzer:

```
(define analyze-special-form
  (lambda (exp)
    (cond ((quoted? exp) (analyze-quoted exp))
          ((lambda? exp) (analyze-lambda exp))
          ((definition? exp) (analyze-definition exp))
          ((if? exp) (analyze-if exp))
          ((begin? exp) (analyze-begin exp)))
        ))
```

`Quote` **expressions:**

```
(define analyze-quoted
  (lambda (exp)
      (let ((text (text-of-quotation exp)))   ; Inductive step
        (lambda (env) text))))                ; Currying
```

Discussion: What is the difference between the above `analyze-quoted` and

```
(define analyze-quoted
  (lambda (exp)
       (lambda (env) (text-of-quotation exp))))
```

`Lambda` **expressions:** In the `env-eval`:

```
(define eval-lambda
  (lambda (exp env)
      (make-procedure (lambda-parameters exp)
                      (lambda-body exp)
                      env)))
```

In the syntax analyzer:

```
(define analyze-lambda
  (lambda (exp)
      (let ((parameters (lambda-parameters exp))
            (body (analyze-sequence (lambda-body exp))))
                                                ; Inductive step
        (lambda (env)                           ; Currying
          (make-procedure parameters body env))))
```

In analyzing a lambda expression, the body is analyzed only once! The body component of a procedure (an already evaluated object) is a Scheme object (closure), not an expression. In `env-eval`, the body of the computed procedures are texts – Scheme expressions.

Definition expressions: In the env-eval:

```
(define eval-definition
  (lambda (exp)
      (add-binding!
        (make-binding (definition-variable exp)
                      (env-eval (definition-value exp)
                                the-global-environment)))
      'ok))
```

In the syntax analyzer:

```
(define (analyze-definition
  (lambda (exp)
    (let ((var (definition-variable exp))
          (val (analyze (definition-value exp)))) ; Inductive step
      (lambda (env)                               ; Currying
       (if (not (eq? env the-global-environment))
           (error 'eval "non global definition: ~s" exp)
           (begin
            (add-binding! (make-binding var (val the-global-environment)))
            'ok))))))
```

Note the redundant `env` parameter in the result procedure! Why?

Analyzing a definition still leaves the load of variable search to run-time, but saves repeated analyses of the value.

`if` **expressions:** In the `env-eval`:

```
(define eval-if
  (lambda (exp env)
      (if (true? (eval (if-predicate exp) env))
  (eval (if-consequent exp) env)
  (eval (if-alternative exp) env))))
```

In the syntax analyzer:

```
(define analyze-if
  (lambda (exp)                          ; Inductive step
      (let ((pred (analyze (if-predicate exp)))
            (consequent (analyze (if-consequent exp)))
            (alternative (analyze (if-alternative exp))))
        (lambda (env)                    ; Currying
          (if (true? (pred env))
              (consequent env)
              (alternative env))))))
```

Sequence expressions: In the `env-eval`:

```
(define eval-begin
  (lambda (exp env)
      (eval-sequence (begin-actions exp) env)))
```

In the syntax analyzer:

```
(define analyze-begin
  (lambda (exp)
      (let ((actions (analyze-sequence (begin-actions exp))))
        (lambda (env) (actions env)))))
```

In the env-eval:

```
; Pre-condition: Sequence of expressions is not empty
(define eval-sequence
  (lambda (exps env)
    (let ((vals (map (lambda (e)(env-eval e env)) exps)))
      (last vals))))
```

In the syntax analyzer:

```
; Pre-condition: Sequence of expressions is not empty
(define analyze-sequence
  (lambda (exps)
    (let ((procs (map analyze exps)))  ; Inductive step
      (lambda (env)                    ; Currying
        (let ((vals (map (lambda (proc) (proc env)) procs)))
          (last vals))))))
```

Application expressions: In the env-eval:

```
(define apply-procedure
  (lambda (procedure arguments)
    (cond ((primitive-procedure? procedure)
           (apply-primitive-procedure procedure arguments))
          ((compound-procedure? procedure)
           (let* ((parameters (procedure-parameters procedure))
                  (body (procedure-body procedure))
                  (env (procedure-environment procedure))
                  (new-env (extend-env (make-frame parameters arguments) env)))
             (eval-sequence body new-env)))
          (else (error 'apply "unknown procedure type: ~s" procedure)))))
```

In the syntax analyzer:

```
(define analyze-application
  (lambda (exp)
     (let ((application-operator (analyze (operator exp)))
           (application-operands (map analyze (operands exp))))
                                                          ; Inductive step
      (lambda (env)
        (apply-procedure
          (application-operator env)
           (map (lambda (operand) (operand env))
              application-operands))))))
```

The analysis of general application first extracts the operator and operands of the expression and analyzes them, resulting Curried Scheme procedures: Environment dependent execution procedures. At run time, these procedures are applied, resulting (hopefully) an evaluator procedure and its operands – Scheme values. These are passed to `apply-procedure`, which is the equivalent of `apply-procedure` in `env-eval`.

```
; Type: [Analyzed-procedure*LIST -> Scheme-type]
```

```
(define apply-procedure
  (lambda (procedure arguments)
    (cond ((primitive-procedure? procedure)
           (apply-primitive-procedure procedure arguments))
          ((compound-procedure? procedure)
           (let* ((parameters (procedure-parameters procedure))
                  (body (procedure-body procedure))
                  (env (procedure-environment procedure))
                  (new-env (extend-env (make-frame parameters arguments) env)))
             (body new-env)))
          (else (error 'apply "unknown procedure type: ~s" procedure)))))
```

If the procedure argument is a compound procedure of the analyzer, then its body is already analyzed, i.e., it is an Env-Curried Scheme closure (of the target Scheme language) that expects a single `env` argument.
Note: No recursive calls for further analysis; just direct application of the already analyzed closure on the newly constructed extended environment.

6.5.1.3 Main analyzer loop:

Modifying the evaluation execution does not touch the two auxiliary packages:

– Abstract Syntax Parser package.

– Data Structures package.

The evaluator of the analyzed code just applies the result of the syntax analyzer:

```
(define derive-analyze-eval
  (lambda (exp)
   ((analyze (derive exp)) the-global-environment)))
```

The main load is put on the syntax analyzer. It performs the case analysis, and dispatches to procedures that perform analysis alone. All auxiliary analysis procedures return `env` Curried execution Scheme closures.

```
; Type: [<Scheme-exp> -> [(Env -> Scheme-type)]]
;                        (Number, Boolean, Pair, List, Evaluator-procedure)
; Pre-conditions: The given expression is legal according to the concrete syntax.
;                 Inner 'define' expressions are not legal.
(define analyze
  (lambda (exp)
    (cond ((atomic? exp) (analyze-atomic exp))
          ((special-form? exp) (analyze-special-form exp))
          ((application? exp) (analyze-application exp))
          (else (error 'eval "unknown expression type: ~s" exp)))))
```

The full code of the analyzer is in the course site.

Example 6.14. *`(analyze 3)` returns the Scheme closure: `<Closure (env) 3>`*

```
> (analyze 3)
#<procedure>       ;;; A procedure of the underlying Scheme.
> ((analyze 3) the-global-environment)
3
```

Example 6.15.

```
 > (analyze 'car)
 #<procedure>       ;;; A procedure of the underlying scheme.
> ((analyze 'car) the-global-environment)
(primitive #<primitive:car>) ;;; An evaluator primitive procedure.

 > (eq? car (cadr  ((analyze 'car) the-global-environment)))
 #t
 > ((cadr  ((analyze 'car) the-global-environment)) (cons 1 2))
 1
```

Example 6.16.

```
> (analyze '(quote (cons 1 2)))
#<procedure>;;; A procedure of the underlying Scheme.
> ((analyze '(quote (cons 1 2))) the-global-environment)
(cons 1 2)
```

Example 6.17.

```
> (analyze '(define three 3))
#<procedure>;;; A procedure of the underlying Scheme.
> ((analyze '(define three 3)) the-global-environment)
ok
> ((analyze 'three) the-global-environment)
3
> (let ((an-three (analyze 'three) ))
     (cons (an-three the-global-environment)
   (an-three the-global-environment)))
(3 . 3)
```

No repeated analysis for evaluating three.

Example 6.18.

```
> (analyze '(cons 1 three))
#<procedure>;;; A procedure of the underlying Scheme.
> ((analyze '(cons 1 three)) the-global-environment)
(1 . 3)
```

Example 6.19.

```
> (analyze '(lambda (x) (cons x three)))
#<procedure>;;; A procedure of the underlying Scheme.
> ((analyze '(lambda (x) (cons x three))) the-global-environment)
(procedure (x) #<procedure> <<the-global-environment>>)
```

Example 6.20.

```
> (analyze '(if (= n 1) 1 (- n 1)))
#<procedure>;;; A procedure of the underlying Scheme.
> ((analyze '(if (= n 1) 1 (- n 1))) the-global-environment)
Unbound variable n
```

Why???????

Example 6.21.

```
> (analyze '(define (factorial n)
                (if (= n 1)
            1
            (* (factorial (- n 1)) n))))
#<procedure>;;; A procedure of the underlying Scheme.
> ((analyze '(define (factorial n)
        (if (= n 1)
            1
            (* (factorial (- n 1)) n)))) the-global-environment)
ok
> ((analyze 'factorial) the-global-environment)
#0=(procedure (n) #<procedure> <<the-global-environment>>)
> (trace analyze)
> ((analyze '(factorial 4)) the-global-environment)
 |(analyze (factorial 4))
 | (analyze factorial)
 | #<procedure>
 | (analyze 4)
 | #<procedure>
 |#<procedure>
 24
```

No repeated analysis for evaluating the recursive calls of factorial.

```
> (trace analyze)
(analyze)

> (derive-analyze-eval
       ' (define (factorial n)
              (if (= n 1)
                  1
                 (* (factorial (- n 1)) n))))
 | (analyze (define (factorial n) (if (= n 1) 1
                                     (* (factorial (- n 1)) n))))
 | |(analyze (lambda (n) (if (= n 1) 1 (* (factorial (- n 1)) n))))
 | | (analyze (if (= n 1) 1 (* (factorial (- n 1)) n)))
 | | |(analyze (= n 1))
 | | | (analyze =)
 | | | #<procedure>
 | | | (analyze n)
 | | | #<procedure>
 | | | (analyze 1)
 | | | #<procedure>
 | | |#<procedure>
 | | |(analyze 1)
 | | |#<procedure>
 | | |(analyze (* (factorial (- n 1)) n))
 | | | (analyze *)
 | | | #<procedure>
 | | | (analyze (factorial (- n 1)))
 | | | |(analyze factorial)
 | | | |#<procedure>
 | | | |(analyze (- n 1))
 | | | | (analyze -)
 | | | | #<procedure>
 | | | | (analyze n)
 | | | | #<procedure>
 | | | | (analyze 1)
 | | | | #<procedure>
 | | | |#<procedure>
 | | | #<procedure>
 | | | (analyze n)
 | | | #<procedure>
 | | |#<procedure>
 | | #<procedure>
 | |#<procedure>
 | #<procedure>
 |ok


> (derive-analyze-eval '(factorial 4))
 |(eval (factorial 4) <<the-global-environment>>)
 | (analyze (factorial 4))
 | |(analyze factorial)
 | |#<procedure>
 | |(analyze 4)
 | |#<procedure>
 | #<procedure>
 |24

> (derive-analyze-eval '(factorial 3))
 | (analyze (factorial 3))
 | |(analyze factorial)
 | |#<procedure>
 | |(analyze 3)
 | |#<procedure>
 | #<procedure>
 |6
```

Chapter 7

Logic Programming - in a Nutshell

Sources:

1. Sterling and Shapiro [11]: The Art of Prolog.

   Topics:

1. Relational logic programming: Programs specify relations among entities.
2. Logic programming: Programs with data structures: Lists, binary trees, symbolic expressions, natural numbers.
3. Meta tools: Backtracking optimization (cuts), unify, meta-circular interpreters.
4. Prolog and more advanced programming: Arithmetic, cuts, negation.

Introduction

The origin of ***Logic Programming*** is in constructive approaches in Automated Theorem Proving, where logic proofs answer queries and construct instantiation to requested variables. The idea behind logic programming suggests a switch in mode of thinking:

1. Structured logic formulas are viewed as ***relationship (procedure) specifications***.
2. A query about logic implication is viewed as a ***relationship (procedure) call***.
3. A constructive logic proof of a query is viewed as a ***query computation***, dictated by an operational semantics algorithm.

Logic programming, like functional languages (e.g., ML, Scheme), departs radically from the mainstream of computer languages. Its operational semantics is not based on the Von-Neumann machine model (like a Turing machine), but is derived from an abstract model of constructive logic proofs (resolution proofs). In comparison, the operational semantics of functional languages is based on the Lambda-calculus reduction rules.

In the early 70s, Kowalski [7] observed that an axiom:

```
A if B1 and B2 ... and Bn
```

can be read as a procedure of a recursive programming language:

– `A` is the procedure head and the `Bi`s are its body.

– An attempt to solve `A` is understood as its execution: To solve (execute) `A`, solve (execute) `B1 and B2 ... and Bn`.

The ***Prolog*** (***Programming in Logic***) language was developed by Colmerauer and his group, as a theorem prover, embodying the above procedural interpretation.

Prolog has developed beyond the original logic basis. Therefore, there is a distinction between (***pure***) ***logic programming*** to ***full Prolog***. The Prolog language includes practical programming constructs, like primitives for arithmetics, and meta-programming and optimization constructs, that are beyond the pure logic operational semantics.

A programming language has three fundamental aspects:

1. **Syntax** - concrete and abstract grammars, that define correct (symbolic) combinations.

2. **Semantics** - the "things" (values) computed by the programs of the language.

3. **Operational semantics** - an evaluation algorithm for computing the semantics of a program.

For logic programs:

1. **Syntax** - a restricted subset of predicate calculus: A logic program is a set of formulas (classified into ***rules*** and ***facts***), defining known relationships in the problem domain.

2. **Semantics** - A set of answers to ***queries***:

   – A program is applied to (or triggered by) a goal (query) logic statement. The goal might include ***variables***.
   – The semantics is a set of answers to goal queries. If a goal includes variables, the answers provide substitutions (instantiations) for the variables in a query.

3. **Operational semantics** - Program execution is an attempt to prove a goal statement.

   – The proof tries to ***instantiate the variables*** (provide values for the variables), such that the goal becomes true.

- A ***computation*** of a logic program is a deduction of consequences of the program. It is triggered by a given goal.
- The operational semantics is the proof algorithm. It is based on two essential mechanisms:
  - ***Unification***: The mechanism for parameter passing. A powerful pattern matcher.
  - ***Backtracking***: The mechanism for searching for a proof.

4. **Language characteristics:**
   - Logic programming has no primitives (apart from the polymorphic unification operator `=`, and `true`).
   - There are no types (since there are no primitives).
   - Logic programming is based on unification: No value computation.
   - Prolog extends pure logic programming with domain primitives (e.g., arithmetics) and rich meta-level features. Prolog is dynamically typed (like Scheme).

Logic programming shows that ***a logic language can be turned into a programming language***, once it is assigned operational semantics.

7.1 Relational Logic Programming

Relational logic programming is a language of relations. It includes explicit relation specification, and rules for reasoning about relations. It is the source for the ***Logic Database*** language ***Datalog***.

7.1.1 Syntax Basics

1. **Atomic symbols:** ***Constant*** symbols and ***variable*** symbols.
   (a) Constant symbols are:
   - ***Individuals*** - describe specific entities, like `computer_Science_Department`, `israel`, etc.
   - ***Predicates*** - describe relations. Some relations are already built-in as **language primitives: `=, true`**.
   - Constant symbols start with lower case letters.

   (b) Variable symbols start with upper case letters or with `_`. Example variables: `X, Y, _Foo, _`. `_` is ***anonymous*** variable.

   (c) Individual constant symbols and variables are collectively called ***terms***.

2. The basic combination means in logic programming is the ***atomic formula***. Atomic formulas include the individual constant `true`, and formulas of the form:

   $$predicate\_symbol(term_1, ..., term_n)$$

   Examples of atomic formulas:

   (a) `father(abraham, isaac)` – In this atomic formula, `father` is a predicate symbol, and `abraham` and `isaac` are individual symbols.

   (b) `father(Abraham, isaac)` – Here, `Abraham` is a variable. Note that `Father(Abraham, isaac)` is syntactically incorrect. Why?

3. Predicate symbols have arity - number of arguments. The arity of `father` in `father(abraham, isaac)` is 2. Since predicate symbols can be overloaded with respect to arity, we denote the arity next to the predicate symbol. The above is `father/2`. There can be `father/3, father/1`, etc.

4. **Abstraction means: *Procedures*.**

   – Procedures are defined using ***facts*** and ***rules***.

   – The collection of facts and rules for a predicate p is considered as the ***definition of*** p.

   – Procedures are triggered using ***Queries***.

7.1.2 Facts

The simplest statements are ***facts***: A fact consists of a single atomic formula, followed by ".". Facts state relationships between entities. For example, the fact

```
father(abraham, isaac).
```

states that the binary relation `father` holds between the individual constants `abraham` and `isaac`. More precisely, `father` is a predicate symbol, denoting a binary relation that holds between the two individuals denoted by the constant symbols `abraham` and by `isaac`.

– A **fact** is a statement consisting of a single atomic formula: It is an ***assertion*** of an atomic formula.

– The simplest fact is:

```
true.
```

It is a language primitive. `true/0` is a zero-ary predicate. It cannot be redefined.

Example 7.1. *Following is a three relationship (procedure) program:*

```
% Signature: parent(Parent, Child)/2
% Purpose:   Parent is a parent of Child
parent(rina, moshe).
parent(rina, rachel).
parent(rachel, yossi).
parent(reuven, moshe).

% Signature: male(Person)/1
% Purpose:   Person is a male.
male(moshe).
male(yossi).
male(reuven).

% Signature: female(Person)/1
% Purpose: Person is a female.
female(rina).
female(rachel).
```

A computation is triggered by posing a ***query*** to a program. A query has the syntax:

$$?\text{-}\ af_1, af_2, \ldots, af_n.$$

where the af_i-s are atomic formulas, and $n \geq 1$. It has the meaning: Assuming that the program facts (and rules) hold, do af_1 and af_2 and ... af_n hold as well. For example, the query:

```
?- parent(rina, moshe).
```

means: "Is rina a parent of moshe?". A computation is a proof of a query. For the above query, the answer is:

```
true ;
fail.
```

That is, it is true and no more alternative answers.
Query: "Is there an X which is a child of rina?":

```
?- parent(rina,X).
X = moshe ;
X = rachel.
```

The “;” stands for a request for additional answers. In this case, there are two options for satisfying the query. We see that ***variables in queries are existentially quantified***. That is, the query “`?- parent(rina,X).`” stands for the logic formula ∃ `X, parent(rina,X).` The constructive proof not only returns `true`, but finds the substitutions for which the query holds. The proof searches the program by order of the facts. For each fact, the computation tries to unify the query with the fact. If the unification succeeds, the resulting substitution for the query variables is the answer (or `true` if there are no variables).

The main mechanism in computing answers to queries is ***unification***, which is a generalization of the pattern matching operation of ML: Unify two expressions by applying a consistent substitution to the variables in the expressions. The only restriction is that the two expressions should not include shared variables.
Query: "Is there an X which is a parent of moshe?":

```
?- parent(X,moshe).
X = rina ;
X = reuven.
```

A complex query: "Is there an X which is a child of rina, and is also a parent of some Y?":

```
?- parent(rina,X),parent(X,Y).
X = rachel,
Y = yossi.
```

A single answer is obtained. The first answer to “`?- parent(rina,X).`”, i.e., `X = moshe` fails. The Prolog interpreter performs ***backtracking***, i.e., goes backwards, and tries to find another answer to the first query, following the rest of the facts, by order.
A complex query: "Find two parents of moshe?":

```
 ?- parent(X,moshe),parent(Y,moshe).
X = rina,
Y = rina ;
X = rina,
Y = reuven ;
X = reuven,
Y = rina ;
X = reuven,
Y = reuven.
```

A complex query: "Find two different parents of moshe?":

```
 ?- parent(X,moshe),parent(Y,moshe),X \= Y.
X = rina,
Y = reuven ;
```

```
X = reuven,
Y = rina ;
fail.
```

Facts can include variables: ***Variables in facts are universally quantified***. The fact "`loves(rina,X).`" stands for the logic formula ∀ `X, loves(rina,X)`, that is, "rina loves everyone".

Example 7.2. *A `loves` procedure:*

```
% Signature: loves(Someone, Somebody)/2
% Purpose: Someone loves Somebody
loves(rina,Y).    /* rina loves everybody. */
loves(moshe, rachel).
loves(moshe, rina).
loves(Y,Y).       /* everybody loves himself (herself). */
```

Queries:

```
 ?- loves(rina,moshe).
true ;
fail.
?- loves(rina,X).
X = rina;
true .
?- loves(X,rina).
X = rina ;
X = moshe ;
X = rina.
?- loves(X,X).
X = rina ;
true.
```

The first query is answered as `true`, based on the first fact, where the fact variable `Y` is substituted by `moshe`. There is no substitution to query variables, and no alternative answers. The second query is also answered as `true`, based on the first fact. In this case, the query variable X is substituted by the fact variable `Y`, but this is not reported by Prolog. There is an alternative answer `X=rina`, based on the forth fact. The third query has three answers, based on the first, third and forth facts, respectively. The forth query has two answers, based on the first and the forth rules, respectively.
Note: Variables in facts are rare. Usually facts state relations among individual constant, not general "life" facts.

7.1.3 Rules

Rules are formulas that state conditioned relationships between entities. Their syntactical form is:

$$H:-B_1,\ldots,B_n.$$

where $H, B_1, \ldots, B_n$ are atomic formulas. H is the ***rule head*** and $B_1, \ldots, B_n$ is the ***rule body***. The intended meaning is that if all atomic formulas in the rule body hold (when presented as queries), then the head atomic formula is also true as a query. The symbol ":-" stands for ***logic implication*** (directed from the body to the head, i.e., `if body then head`), and the symbol "," stands for ***logic and*** (conjunction).
For example, the rule

```
father(Dad, Child) :-  parent(Dad, Child), male(Dad).
```

states that `Dad`(a variable) is a `father` of `Child` (variable) if `Dad` is a `parent` of `Child` and is a `male.` The rule

```
mother(Mum, Child) :-  parent(Mum, Child), female(Mum).
```

states that `Mum` (a variable) is a `mother` of a `Child` (variable) if `Mum` is a `parent` of `Child` and is a `female`. In these rules:

- `father(Dad, Child), mother(Mum, Child)` are the rule heads.
- `parent(Dad, Child), male(Dad)` is the body of the first rule.
- The symbols `Dad, Mum, Child` are variables. They are universally quantified over the rule.

Variable quantification in rules is the same as for facts: ***Variables occurring in rules are universally quantified. The lexical scope of the quantification is the rule***. Variables in different rules are unrelated: Variables are bound only within a rule. Therefore, ***variables in a rule can be consistently renamed***. The `father` rule above, can be equivalently written:

```
father(X, Y):- parent(X, Y), male(X).
```

Consider the following rule, defining a sibling relationship:

```
%Signature: sibling(Person1, Person2)/2
% Purpose:  Person1 is a sibling of Person2.
sibling(X,Y) :- parent(P,X), parent(P,Y).
```

The variable `P` appears only in the rule body. Such variables can be read as ***existentially quantified over the rule body*** (simple logic rewrite of the universally quantified `P`). Therefore, the above rule is read: For all `X,Y`, `X` is a sibling of `Y` if ***there exists*** a `P` which is a `parent` of both `X, Y`.

A ***procedure*** is an ordered collection of rules and facts, sharing a single predicate and arity for the rule heads, and the facts. The collection of rules and facts that make a single procedure is conventionally written consecutively.

Example 7.3. *An eight procedure program:*

```
% Signature: parent(Parent, Child)/2
% Purpose:   Parent is a parent of Child
parent(rina, moshe).
parent(rina, rachel).
parent(rachel, yossi).
parent(reuven, moshe).

% Signature: male(Person)/1
% Purpose:   Person is a male.
male(moshe).
male(yossi).
male(reuven).

% Signature: female(Person)/1
% Purpose: Person is a female.
female(rina).
female(rachel).

% Signature: father(Dad, Child),
% Purpose:    Dad is a father of Child
father(Dad, Child) :-  parent(Dad, Child), male(Dad).

% Signature: mother(Mum, Child),
% Purpose:    "Mum is a mother of Child
mother(Mum, Child) :-  parent(Mum, Child), female(Mum).

% Signature: sibling(Person1, Person2)/2
% Purpose:  Person1 is a sibling of Person2.
sibling(X,Y) :- parent(P,X), parent(P,Y).

% Signature: cousin(Person1, Person2)/2
% Purpose: Person1 is a cousin of Person2.
```

```
cousin(X,Y) :- parent(PX,X), parent(PY,Y), sibling(PX,PY).

% Signature:  grandmother(Person1, Person2)/2
% Purpose:  Person1 is a grandmother of Person2.
grandmother(X,Y) :- mother(X,Z), parent(Z,Y).
```

Queries and answers:

```
 ?- father(D,C).
D = reuven,
C = moshe.
?- mother(M,C).
M = rina,
C = moshe ;
M = rina,
C = rachel ;
M = rachel,
C = yossi ;
fail.
```

Query: "Find a two kids mother":

```
?- mother(M,C1),mother(M,C2).
M = rina,
C1 = moshe,
C2 = moshe ;
M = rina,
C1 = moshe,
C2 = rachel ;
M = rina,
C1 = rachel,
C2 = moshe ;
M = rina,
C1 = rachel,
C2 = rachel ;
M = rachel,
C1 = yossi,
C2 = yossi ;
fail.
```

Query: "Find a two different kids mother":

```
?- mother(M,C1),mother(M,C2),C1\=C2.
```

```
M = rina,
C1 = moshe,
C2 = rachel ;
M = rina,
C1 = rachel,
C2 = moshe ;
fail.
```

Query: "Find a grandmother of yossi":

```
 ?- grandmother(G,yossi).
G = rina ;
fail.
```

In order to compute the `ancestor` relationship we need to insert a ***recursive rule*** that computes the ***transitive closure*** of the `parent` relationships:

```
% Signature:  ancestor(Ancestor, Descendant)/2
% Purpose:  Ancestor is an ancestor of Descendant.
ancestor(Ancestor, Descendant) :-  parent(Ancestor, Descendant).
ancestor(Ancestor, Descendant) :-  parent(Ancestor, Person),
                                   ancestor(Person, Descendant).

?- ancestor(rina,D).
D = moshe ;
D = rachel ;
D = yossi ;
fail.
?- ancestor(A,yossi).
A = rachel ;
A = rina ;
fail.
```

Let us try a different version of the recursive rule:

```
ancestor(Ancestor, Descendant) :-  parent(Ancestor, Descendant).
ancestor(Ancestor, Descendant) :-  ancestor(Person, Descendant),
                                    parent(Ancestor, Person).
?- ancestor(A,yossi).
A = rachel ;
A = rina ;
ERROR: Out of local stack
?- ancestor(rina,D).
```

```
D = moshe ;
D = rachel ;
D = yossi ;
ERROR: Out of local stack
?- ancestor(rina,yossi).
true ;
ERROR: Out of local stack
```

What happened? The recursive rule first introduces a new query for the same recursive procedure `ancestor`. Since this query cannot be answered using the base case, new similar queries are infinitely created. The first version of the `ancestor` procedure does not have this problem since the recursive rule first introduces a base query `parent(Ancestor, Person)`, that enforces a concrete binding to the variables. Then the next query `ancestor(Person, Descendant)`, just checks that the variable values satisfy the `ancestor` procedure. If not, backtracking is triggered and a different option for the first query is tried. The first version of `ancestor` is called ***tail recursive***. It is always recommended to write recursive rules in a tail form.

Summary:

1. A ***rule*** is a conditional formula of the form "`H :- B1,....,Qn.`". `H` is called the ***head*** and `B1,...,Bn` the ***body*** of the rule. `H, B1,...,Bn` are atomic formulas (denote relations).
2. The symbol ":-" stands for "if" and the symbol "," stands for "and".
3. The ***primitive predicate symbols*** `true, =` cannot be defined by rules (cannot appear in rule heads). `true` is a primitive proposition, and `=` is the binary unification predicate.
4. A rule is a ***lexical scope***: The variables in a rule are ***bound procedure variables*** and therefore can be freely renamed.
   - They are universally quantified (∀) over the entire rule.
   - Variables that appear only in rule bodies can be considered as existentially quantified (∃) over the rule body.
5. ***Variables in different rules*** reside in different lexical scopes.
6. ***Rules cannot be nested***.
   - Logic Programming does not support ***procedure nesting***. Compare with the power provided by procedure nesting in functional programming (Scheme, ML). There is no way to define an auxiliary nested procedure (as usually needed in iterative processes).

  - There is no notion of ***nested scopes***.
  - No notion of ***free variables***: All variables are ***bound*** within a rule.
  - The variables occurring in rule heads can be viewed as variable declarations.

7. The operation of consistently renaming the variables in a rule is called ***variable renaming***. It is used ***every time a rule is applied*** (like renaming prior to substitution in the substitution model operational semantics for functional programming - Scheme, ML).

8. ***Programming conventions***:

  - Procedure definitions are singled out. All facts and rules for a predicate are written as a contiguous block, separated from other procedure definitions.
  - Every procedure definition is preceded with a ***contract***, that includes, at least: ***signature specification*** and ***purpose declaration***.

Problem modeling in Relational Logic Programming (RLP): Logic Programming languages express knowledge using predications (atomic formulas) and terms. Predications represent relations in the problem domain, and terms represent entities in the problem domain. Therefore, modeling issues involve decisions about the identity of ***entities*** and ***relations***. In RLP the entities are atomic elements, while in general, they can be complex. In the above "family relationships" problems, the decision to model people using entities and family relationships as relations, is the natural one. However, this is not always the case.

For example, in the domain of *Components and Connectors*, a problem consists of answering (or computing answers to) queries about components that are interconnected by connectors. This domain includes complex software systems whose components are interrelated and depend on each other in multiple manners, data-flow diagrams, digital circuits, and more. All problems have associated visual drawing conventions, where components are visualized using complex geometrical shapes and connectors are visualized using various kinds of lines.

For modeling a Components and Connectors problem there are two possible approaches:

1. Take components as entities and connectors as relationships.

2. Take connectors as entities and components as relationships.

The decision is not straightforward, since both options seem reasonable. However, in general, the second option is the better one, since while their is a wide variety of components that characterize different relations among connectors, connectors are more uniform.

7.1.4 Syntax

Concrete syntax of Relational Logic Programming: A program is a non empty set of procedures, each consisting of an ordered set of rules and facts, having the same predicate and arity.

```
<program>          ->   <procedure>+
<procedure         ->   (<rule> | <fact>)+    with identical predicate and arity
<rule>             ->   <head> ': -' <body>'.'
<fact>             ->   <head>'.'
<head>             ->   <atomic-formula>
<body>             ->   (<atomic-formula>',')* <atomic-formula>
<atomic-formula>   ->  <constant> | <predicate>'('(<term>',')* <term>')'
<predicate>        ->    <constant>
<term>             ->    <constant>  |  <variable>
<constant>         -> A string starting with a lower case letter.
<variable>         -> A string starting with an upper case letter.
<query>            ->   '?-' (<atomic-formula>',')* <atomic-formula> '.'
```

Abstract syntax of Relational Logic Programming:

```
<program>:
   Components: <procedure>
<procedure>:
   Components: Rule: <rule>
               Fact: <atomic-formula>
                       Overall amount of rules and facts: >=1. Ordered.
<rule>:
   Components: Head: <atomic-formula>
               Body: <atomic-formula>  Amount: >=1. Ordered.
<atomic-formula>:
   Kinds: <predication>, constant.
<predication>:
   Components: Predicate: <constant>
               Term: <term>. Amount: >=1. Ordered.
<term>:
   Kinds: <constant>,<variable>
<constant>:
   Kinds: Restricted sequences of letters, digits, punctuation marks,
          starting with a lower case letter.
<variable>:
   Kinds: Restricted sequences of letters, digits, punctuation marks,
          starting with an upper case letter.
<query>:
   Components: Goal: <atomic-formula>. Amount: >=1. Ordered.
```

7.1.5 Operational Semantics

The operational semantics of logic programming is based on two mechanisms: ***Unification*** and ***Search and backtracking***.

7.1.5.1 Unification

Unification is the operation of identifying atomic formulas by substituting expressions for variables. For example, the atomic formulas
$p(3,X)$, $p(Y,4)$ can be unified by the substitution: $\{X = 4, Y = 3\}$, and
$p(X,3,X)$, $p(Y,Z,4)$ can be unified by the substitution: $\{X = 4, Z = 3, Y = 4\}$.

Formal definition of unification:

Definition: A ***substitution*** s in logic programming involves logic variables as variables, and logic terms as values, such that $s(X) \neq X$. A pair $\langle X, s(X)\rangle$ is called a ***binding***, and written $X = s(X)$.

For example,

– $\{X = 4, Z = 3, U = X\}$, $\{X = 4, Z = 3, U = V\}$ are substitutions, while

– $\{X = 4, Z = 3, Y = Y\}$, or $\{X = 4, Z = 3, X = Y\}$ are not substitutions.

Definition: The ***application of a substitution*** s to an atomic formula A, denoted $A \circ s$ (or just As) replaces the terms for their variables in A. The replacement is ***simultaneous***.

For example,

– $p(X,3,X,W) \circ \{X = 4, Y = 4\} = p(4,3,4,W)$

– $p(X,3,X,W) \circ \{X = 4, W = 5\} = p(4,3,4,5)$

– $p(X,3,X,W) \circ \{X = W, W = X\} = p(W,3,W,X)$.

Definition: An atomic formula A' is an ***instance*** of an atomic formula A, if there is a substitution s such that $A \circ s = A'$. A is ***more general*** than A', if A' is an instance of A.

For example,

– $p(X,3,X,W)$ is more general than $p(4,3,4,W)$, which is more general than $p(4,3,4,5)$.

– $p(X,3,X,W)$ is more general than $p(W,3,W,W)$, which is more general than $p(5,3,5,5)$.

– $p(X,3,X,W)$ is more general than $p(W,3,W,X)$, which is more general than $p(X,3,X,W)$.

Definition: A **unifier** of atomic formulas A and B is a substitution s, such that $A \circ s = B \circ s$.

For example, the following substitutions are unifiers of $p(X,3,X,W)$ and $p(Y,Z,4,W)$:

– $\{X = 4, Z = 3, Y = 4\}$

– $\{X = 4, Z = 3, Y = 4, W = 5\}$

– $\{X = 4, Z = 3, Y = 4, W = 0\}$

Definition: A ***most general unifier*** (***mgu***) of atomic formulas A and B is a unifier s of A and B, such that $A \circ s = B \circ s$ is more general than all other instances of A and B that are obtained by applying a unifier. That is, for every unifier s' of A and B, there exists a substitution s'' such that $A \circ s \circ s'' = A \circ s'$.
If A and B are unifiable they have an mgu (unique up to renaming).

For example, $\{X = 4, Z = 3, Y = 4\}$ is an mgu of $p(X,3,X,W)$ and $p(Y,Z,4,W)$.

Definition: ***Combination of substitutions***
The combination of substitutions s and s', denoted $s \circ s'$, is defined by:

1. s' is applied to the terms of s, i.e., for every variable X for which $s(X)$ is defined, occurrences of variables X' in $s(X)$ are replaced by $s'(X')$.
2. A variable X for which $s(X)$ is defined, is removed from the domain of s', i.e., $s'(X)$ is not defined on it any more.
3. The modified s' is added to s.
4. Identity bindings, i.e., $s(X) = X$, are removed.

For example,
$\{X = Y, Z = 3, U = V\} \circ \{Y = 4, W = 5, V = U, Z = X\} =$
$\{X = 4, Z = 3, Y = 4, W = 5, V = U\}$.

Unification algorithms: The method introduced in Chapter 5 for solving type equations is in fact a unification algorithm for any kind of expressions, that computes an mgu. Given atomic formulas A, B, they can be unified by following this method steps:

1. Start with an empty substitution $sub = \{\}$, and an equation sequence that includes a single equation $equations = (A = B)$.

2. Apply *sub* to the first equation in *equations*: Denote $eq'_1 = equation_1 \circ sub$.
3. If one side in eq'_1 is a variable, and the other side is not the same variable, apply eq'_1 to *sub*, and combine the substitutions.
4. **Atomic equation sides:** If both sides in eq'_1 are atomic, then if both sides are the same constant (or predicate) symbol, do nothing, and if the sides are different predicate symbols: return FAIL.
5. **Composite equation sides:** If the predicate symbols and the number of arguments are the same, then split the equation into equations of corresponding arguments, add to *equations*, and repeat the unification process. Otherwise, return FAIL.

We present a different, more efficient ***unification algorithm*** that computes an mgu of two atomic formulas, if they are unifiable. It is based on the notion of disagreement set of atomic formulas.

Definition: The ***disagreement set*** of atomic formulas is the set of left most symbols on which the formulas ***disagree***, i.e., are different.

For example,

– `disagreement-set`$(p(X,3,X,W), p(Y,Z,4,W)) = \{X,Y\}$.

– `disagreement-set`$(p(5,3,X,W), p(5,3,4,W)) = \{X,4\}$.

A unification algorithm:
Signature: ***unify***(A,B)
Type: [Atomic-formula*Atomic-formula -> Substitution or FAIL]
Post-condition: $result = mgu(A,B)$ if A and B are unifiable or $FAIL$, otherwise

```
unify(A,B) =
   let   help(s) =
      if A ∘ s = B ∘ s then s
         else let D = disagreement-set(A ∘ s, B ∘ s)
              in  if D = {X, t}     /* X is a variable; t is a term
                  then help(s ∘ {X = t})
                  else FAIL
              end
   in  help( {} )
   end
```

Example 7.4.

1.
   ```
   unify[  p(X, 3, X, W),   p(Y, Y, Z, Z)  ] ==>
   help[ {} ]  ==>
                D = {X, Y}
   help[  {X = Y} ] ==>
                D = {Y, 3}
   help[  {X = 3, Y= 3} ] ==>
                D = {Z, 3} ]
   help[  {X = 3, Y= 3, Z = 3} ] ==>
                D = {W, 3}
   help[  {X = 3, Y= 3, Z:= 3, W = 3} ] ==>
   {X = 3, Y = 3, Z = 3, W = 3}
   ```

2.
   ```
   unify[  p(X, 3, X, 5),   p(Y, Y, Z, Z)  ] ==>
   FAIL
   ```

Properties of this unification algorithm:

1. The algorithm terminates, because in every recursive call a variable is substituted, and there is a finite number of variables.
2. The algorithm's complexity is quadratic in the length of the input formulas (the exact complexity depends on the implementation, whether it is incremental or not). In real applications variables are not actually substituted. Instead, bindings are kept.
3. There are more efficient algorithms!
4. **Pattern matching:** Unification of atomic formulas where only one includes variables (as in ML function application) is called ***pattern matching***. In ***unify***(A, B):
   (a) If `B` does not include variables: The application `B ∘ s` in the computation of the disagreement set can be saved: No need to scan `B`.
   (b) If, in addition, `A` does not include repeated variable occurrences (as in ML patterns), the application `A ∘ s` can be saved as well: No need to scan `A`. The disagreement set can be found just by keeping parallel running pointers on the two atomic formulas.

   Therefore: If `B` does not include variables, and `A` does not include variable repetitions, the complexity of the algorithm reduces to linear! This argument explains the limitations that ML puts on patterns.

7.1.5.2 *answer-query*: An abstract interpreter for Logic Programming

Description of the computation: The computation of a Prolog program is triggered by a ***query***:

$Q =$ `?- G1, ..., Gn.`
The query components are called ***goals***. The interpreter tries all possible proofs for the query, and computes a ***set of answers***, i.e., substitutions to the variables in the query. Each answer corresponds to a ***proof of the query***. If the query cannot be proved, then the set is ***the empty set***.

Facts as rules: Facts are treated as rules, whose body is the single atomic formula true. For example, the fact

```
r(baz,bar).
is written as the rule
r(baz,bar) :- true.
```

Goal and rule selection: Each proof is a repeated effort to prove:

- The ***selected goal***.
- Using the ***selected rule***.

If the selected rule does not lead to a proof, the next selected rule is tried. This is the ***backtracking mechanism*** of the interpreter. If no rule leads to a proof, the computation fails. Rule selection is performed by ***unification*** between the selected goal and the head of the candidate rule. Therefore, the algorithm has two points of ***non deterministic choice***: The goal and the rule selections.

Selections of goals and rules are directed by a ***goal selection rule*** *Gsel* and a ***rule selection rule*** *Rsel*.

1. **Goal selection:** *Gsel* is a selection function defined on tuples of goals. That is, $Gsel(G_1, \ldots, G_n) = G_i, n \geq 1,\ 1 \leq i \leq n$, such that $G_i \neq true$.

2. **Rule selection:** *Rsel* is a function that for a given goal and a program (a sequence of rules) creates a sequence of all rule-substitution pairs such that:
$Rsel(G, P) = (\langle R, \sigma\rangle)_{R \in P,\ G \circ \sigma = head(R) \circ \sigma}$. That is, for every rule $R \in P$, if R is A :- $B_1, \ldots, B_n$., $n \geq 1$, and ***unify***$(A, G) = \sigma$ succeeds with the unifying substitution (mgu) σ, then $\langle R, \sigma\rangle \in Rsel(G, P)$.

Prolog solves this duplicate non-determinism by selecting

- Goal selection as the ***left most goal*** which is different from *true*. That is, $Gsel_{Prolog}(G_1, \ldots, G_n) = G_1$, provided that $G_1 \neq true$.
- Rule selection by the ***procedure rule ordering***. That is, for Prolog, if the rules in P are ordered as $R_1, \ldots, R_n$, then $Rsel_{Prolog}(G, P) = (\langle R_{i_1}, \sigma_{i_1}\rangle, \ldots, \langle R_{i_m}, \sigma_{i_m}\rangle)$, such that $i_j < i_{j+1}$, for $1 \leq j \leq m$.

The ***answer-query*** algorithm calls a ***proof-tree*** algorithm that builds a ***proof tree*** and collects the answers from its leaves. The proof tree is a tree with labeled nodes and edges. It is defined as follows:

1. The nodes are labeled by queries, with a marked goal in the query (the selected goal).
2. The edges are labeled by substitutions and rule numbers.
3. The root node is labeled by the input query and its selected goal.
4. The child nodes of a node labeled Q with a marked goal G represent all possible successive queries, obtained by applying all possible rules to G. The child nodes are ordered by the rule selection ordering (dictated by $Rsel$), where the left most child corresponds to the first selected rule.

The **Tree operations** used in the ***proof-tree*** algorithm are:

1. **Constructors:**
   - $make\_node(label)$: Creates a node labeled $label$.
   - $add\_branch(tree, edge\_label, branch)$: Adds $branch$ as a right branch to the root node of $tree$, with edge labeled by $edge\_label$.
2. **Selector:** $label(node)$ selects the label of $node$.

The *answer-query* algorithm:

Signature:
`answer-query(Q,P,Gsel,Rsel)`

Input:
A query: $Q =$ `?-` $G_1, \ldots, G_n$., where $G_i, i = 1..n$ is an atomic formula
A program P, with numbered rules (denoted by $number(R)$)
A goal selection rule $Gsel$
A rule selection rule $Rsel$

Output: A set of substitutions for variables of `Q` (not necessarily for all variables).

Method:

1. $answers = \{\})$
2. $PT :=$ ***proof-tree***$(make\_node(Q))$. Answers are collected in the $answers$ variable.

3. Return $\{s \mid s \in answers_{/Q}\}$
   where $answers_{/Q}$ is the restriction of the substitutions in *answers* to the variables of Q.

What are the possible answers of *answer-query*?

1. An empty set answer (no substitutions) marks a failure of the interpreter to find a proof.
2. An answer with empty substitutions marks success proofs with no variables to substitute (e.g., when the query is ground).
3. An answer with non-empty substitutions marks success proofs that require substitutions to some (or all) query variables.

***proof-tree* :**

Signature: ***proof-tree***$(node)$. *node* is a tree node with label *query*, which is a tuple of atomic formulas, called ***goals***.

Input: A tree node *node*

Output: A proof tree rooted in *node*.

Method:

```
if $label(node)$ is ?- $true, \ldots, true$.
then  1. Mark $node$ as a $Success$ node (leaf). A success backtracking point.
      2. $answers := answers \cup \{s_1 \circ s_2 \circ \ldots \circ s_n\}$, where $s_1, \ldots, s_n$ are the substitution
         labels of the path from the tree-root to $node$. Mark $node$ with this substitution.
else  1. Goal selection: $G = Gsel(label(node))$. $G \neq true$ since $Gsel$ does not select a
         $true$ goal.
      2. Variables renaming: Rename variables in every rule and fact of $P$ (justified
         since all variables are universally quantified, i.e., bound).
      3. Rule selection: $rules = Rsel(G, P) = (\langle R, \sigma \rangle)_{R \in P,\ G \circ \sigma = head(R) \circ \sigma}$ apply-rules$(node, G, rules)$
```

***apply-rules*:**

Signature: ***apply-rules***$(node, G, rules)$

Input: *node* is the root-node of a proof-tree, with label *query*, which is a tuple of atomic formulas, called ***goals***; G is an atomic formula ($\neq true$) included in *query*; *rules* is a sequence of rule-substitution pairs $(\langle R, \sigma \rangle)_{R \in P,\ G \circ \sigma = head(R) \circ \sigma}$.

Output: The proof tree rooted in *node*, extended with proof-trees created by the application of the rules in *rules* to the selected goal G.

Method: Rule application: (reduce step)

```
    if $empty?(rules)$
    then output= $node$, i.e., the tree rooted at $node$. A failure backtracking point.
    else assume that the first element in $rules$ is $\langle R, \sigma \rangle$.
        1. New query construction:
           $new\_query = (\mathbf{replace}(label(node), G, body(R))) \circ \sigma$. That is, if $label(node) = G_1, \ldots, G_n, n \geq 1$, $G = G_i$, and $body(R) = B_1, \ldots, B_m, m \geq 1$, then $new\_query = (G_1, \ldots, G_{i-1}, B_1, \ldots, B_m, G_{i+1}, \ldots, G_n) \circ \sigma$.
        2. New query expansion:
           $\mathbf{add\_branch}(node, \langle \sigma,\ number(R) \rangle, \mathbf{proof\text{-}tree}(make\_node(new\_query)), )$
        3. Application of other selected rules: $\mathbf{apply\text{-}rules}(node, G, tail(rules))$
    end
```

Comments:

1. In the **rename** step, the variables in the rules are renamed by new names. This way the program variables in every binding step are different from previous steps. Since variables in rules are bound procedure variables they can be freely renamed.
   Renaming convention: In every recursive call, increase some auxiliary counter, such that variables `X, Y,...` are renamed as `X1, Y1,...` at the first level, `X2, Y2,...` at the second level, etc.

2. Advice for substitution in the **rule selection** step: Let ***unify*** produce a substitution to the rule head variables, rather than to the goal variables. This way the query variables are kept in the substitutions produced in a proof branch. Since the mgu is unique up to renaming, the proof is indifferent to whether it includes $var_{rule\_head} := var_{goal}$ or $var_{goal} := var_{rule\_head}$.

3. The goal and rule selection decisions can affect the performance of the interpreter.

Example 7.5. *The proof tree for the biblical family database:*

```
% Signature: father(F,C)/2
father(abraham.isaac).
father(haran,lot).
father(haran,yiscah).
father(haran,milcah).
```

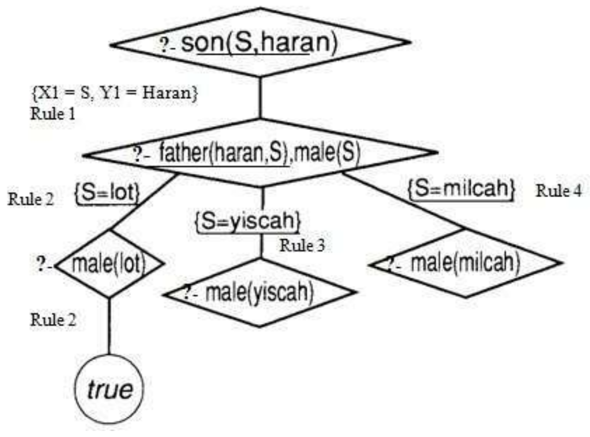


Figure 7.1: The proof tree - a finite success tree

```
% Signature: male(P)/1
male(isaac).
male(lot).

% Signature: female(P)/1
female(milcah).
female(yiscah).

% Signature: son(C, P)/2
son(X, Y) - father(Y, X),  male(X).

% Signature: daughter(C, P)/2
daughter(X, Y) - father(Y, X), female(X).
```

Query: "Find a son of haran":

```
?-  son(S, haran).
```

Analysis of *answer-query*:

Paths in the proof tree:

– A ***path*** from the root in the proof tree corresponds to a **computation** of ***answer-query***.

– A finite root-to-leaf path with a *Success* marked leaf is a ***successful computation path***. A tree with a successful computation path is a ***success tree***. A successful computation path corresponds to a successful computation of ***answer-query***. The ***answer*** of a successful path is its substitution label.
Definition: [*Provable query*]
A query Q ***is provable*** from a program P, denoted $P \vdash Q$, iff for **some** goal and rule selection rules $Gsel$ and $Rsel$, the proof tree algorithm for ***answer-query***$(Q, P, Gsel, Rsel)$ computes a ***success tree***.

– A finite root-to-leaf path with a non *Success* marked leaf is a ***finite-failure computation path***. A proof tree that all of its paths are failed computation paths is a (finite) ***failure tree***.

– An ***infinite computation path*** is an infinite path. Infinite computations can be created by recursive rules (direct or indirect recursion).

Significant kinds of proof trees:

1. **Finite success proof tree:** A finite tree with a successful path.
2. **(Finite) Failure proof tree:** A finite tree with no successful path.
3. **Infinite success proof tree:** An infinite tree with a successful path. In this case it is important not to explore an infinite path. For Prolog: Tail recursion is safe, while left recursion is dangerous.
4. **Infinite failure proof tree:** An infinite tree with no successful path. Dangerous to explore.

The proof tree in Example 7.5 is a finite success tree. The resulting substitution on the successful computation path is: `{X1=S, Y1=haran}` ∘ `{S=lot}`, which is the substitution `{X1=lot, Y1=haran, S=lot}`. The restriction to the query variables yields the single substitution: `{S=lot}`.

Properties of *answer-query*:

1. **Proof tree uniqueness:** The proof tree for a given query and a given program is unique, for all goal and rule selection procedures (up to sibling ordering).
Conclusion: The set of answers is independent of the concrete selection procedures for goals and rules.

2. **Performance:** Goal and rule selection decisions have impact on performance.
   (a) The rules of a procedure should be ordered according to the rule selection procedure. Otherwise, the computation might get stuck in an infinite path, or try multiple failed computation paths.
   (b) The atomic formulas in a rule body should be ordered according to the goal selection procedure.
3. **Soundness and completeness:**
   - **Definition:** [*logical implication*]
     A query `Q` is logically implied from a program `P`, denoted $P \vDash Q$, if it is true whenever the program is true.
   - **Completeness:** $P \vDash Q$ implies $P \vdash Q$. That is, if a query is logically implied from a program, then it is also provable by ***answer-query***.
   - **Soundness:** $P \vdash Q$ implies $P \vDash Q$. That is, if a query is provable by ***answer-query***, then it is logically implied from the program.

7.1.5.3 Properties of Relational Logic Programming

Decidability:

Proposition 7.1.1. Given a program P and a query Q, the problem "Is Q provable from P", denoted $P \vdash Q$, is decidable.

Proof. The proof tree consists of nodes that are labeled by queries, i.e., sequences of atomic formulas. The atomic formulas consist of predicate and individual constant symbols that occur in the program and the query, and from variables. Therefore, (1) the number of atomic formulas is finite, up to variable renaming, and (2) the number of different selected goals in queries on a path is finite (up to variable renaming). Consequently, every path in the proof tree can be decided to be a success, a failure or an infinite computation path. □

Note that all general purpose programming languages are only partially decidable (the halting problem). Therefore, relational logic programming is less expressive than a general purpose programming language.
Question: If relational logic programming is decidable, does it mean that all relational logic programming proof trees are finite?

Types: Logic programming is typeless. That is, the semantics of the language does not recognize the notion of types. The computed values are not clustered into types, and the abstract interpreter algorithm cannot fail at run time due to type mismatch between procedures and arguments.

Comparison:
Logic programming: Typeless. No runtime errors due to non-well-typing.
Scheme: Dynamically typed. Syntax does not specify types. Run time errors.
ML: Statically typed. Type information inferred. No run time errors due to non-well-typing.

7.1.6 Relational logic programs and SQL operations

Relational logic programming is the basis for the ***DataLog language***, which is a logic based language for database processing. DataLog is relational logic programming + arithmetic + negation + some database related restrictions. The operational semantics of DataLog is defined in a different way (bottom up semantics).

DataLog is more expressive than SQL. The relational algebra operations: Union, Cartesian product, diff, projection, selection, and join can be implemented in relational logic programming. Yet, recursive rules (like computing the transitive closure of a relation) cannot be expressed in SQL (at least not in the traditional SQL).

```
Union:
r_union_s(X1, ..., Xn) :- r(X1, ..., Xn).
r_union_s(X1, ..., Xn) :- s(X1, ..., Xn).

Cartesian production:
r_X_s(X1, ..., Xn, Y1, ..., Ym) :- r(X1, ..., Xn ), s(Y1, ..., Ym).

Projection:
r1(X1, X3) :- r(X1, X2, X3).

Selection:
r1(X1,X2, X3) :- r(X1, X2, X3), X2 \= X3.

Natural Join:
r_join_s(X1, ..., Xn, X, Y1, ..., Ym) :-
      r(X1, ..., Xn, X ), s(X, Y1, ..., Ym).

Intersection:
r_meets_s(X1, ..., Xn) :- r(X1, ..., Xn ), s(X1, ..., Xm).

Transitive closure of a binary relation r:
tr_r(X, Y) :- r(X, Y).
tr_r(X, Y) :- r(X, Z), tr_r(Z, Y).
```

For example, if `r` is the `parent` relation, then `tr-parent` is the ancestor relation.
Compare the SQL embedding in relational logic programming, with the SQL embedding in

Scheme (as in the homework for Chapter 3).

7.2 Logic Programming

Logic programming adds an additional syntactic symbol - ***functor*** - that can represent data structures, and is more expressive than relational logic programming. However, this addition is not priceless:

1. The computation algorithm requires a more complex unification operation.
2. The language becomes partially decidable. That is, while the answer to a query in relational logic programming can always be decided to be a success or a failure, logic programming is partially decidable, like all other general purpose programming languages.
3. Logic programming is still a typeless language: No runtime errors.

7.2.1 Syntax

The only difference between the syntax of Logic Programming and the syntax of Relational Logic Programming is the addition of a new kind of a constant symbol: ***Functor*** (***function symbol***). It enriches the set of terms so that they can describe ***structured data***.

Definition: Terms in Logic Programming. The syntax of ***terms*** is more complicated, and requires an inductive definition:

1. **Basis:** Individual constant symbols and variables are terms.
2. **Inductive step:** For terms $t_1, \ldots, t_n$, and a functor f, $f(t_1, \ldots, t_n)$ is a term.

Example 7.6. *Terms and atomic formulas in Logic Programming:*

Terms:

– `cons(a,[ ])` – describes the list `[a]`. `[ ]` is an individual constant, standing for the ***empty list***. The `cons` functor has a syntactic sugar notation as an infix operator `|`: `cons(a,[ ])` is written: `[a|[ ]]`.

– `cons(b,cons(a,[ ]))` – the list `[b,a]`, or `[b|[a|[ ]]]`. The syntax `[b,a]` uses the printed form of lists in Prolog.

– `cons(cons(a,[ ]), cons(b,cons(a,[ ])))` – the list `[[a],[b,a]]`, or `[[a|[ ]]|[b|[a,|[ ]]]]`.

– `time(monday,12,14)`

- `street(alon,32)`
- `tree(Element,Left,Right)` – a binary tree, with `Element` as the root, and `Left` and `Right` as its sub-trees.
- `tree(5,tree(8,void,void),tree(9,void,tree(3,void,void)))`

Atomic formulas: The arguments to the predicate symbols in an atomic formula are terms:

- `father(abraham,isaac)`
- `p(f(f(f(g(a,g(b,c))))))`
- `ancestor(mary,sister_of(friend_of(john)))`
- `append(cons(a,cons(b,[ ])),cons(c,cons(d,[ ])))`
- `cons(a,cons(b,cons(c,cons(d,[ ]))))`
- `append([a,b],[c,d],[a,b,c,d])`

Notes:

1. Every functor has an ***arity***: Number of arguments. In Example 7.6:
   - The arity of `cons` is 2.
   - The arity of `sister_of` is 1.
   - The arity of `time` is 3.
   - The arity of `street` is 2.
2. ***Functors can be nested***: Terms might have unbound depth: `f(f(f(g(a,g(b,c))))).`

   Therefore: The number of different atomic formulas that can be constructed from a given set of predicate, functor and individual constant symbols is unbounded - in contrast to the situation in Relational Logic Programming!
3. ***Predicate symbols cannot be nested***:
   - `p(f(f(f(g(a,g(b,c))))))` – `p` is a predicate symbol, while `f, g,` are functors.
   - `ancestor(mary,sister_of(friend_of(john)))` – `ancestor` is a predicate symbol, and `sister_of,friend_of` are functors.
   - `course(ppl,time(monday,12,14),location(building34,201))` – `course` is a predicate symbol, and `time,location` are functors.
   - `address(street(alon,32),shikun_M,tel_aviv,israel)` – `address` is a predicate symbol, and `street,shikun_M` are functors.

4. The syntax of terms and of atomic formulas is identical. They differ in the position (context) in statements:
   - Terms are arguments to both terms and to predicates.
   - Atomic formulas are the building blocks of rules and facts.

7.2.1.1 Formalizing the syntax extension

New concrete syntax rules:

```
<term>           -> <constant> | <variable> | <composite-term>
<composite-term> -> <functor> '(' (<term>',')* <term>')'
<functor>        -> <constant>
```

New abstract syntax rules:

```
<term>:
  Kinds: <constant>, <variable>, <composite-term>
<composite-term>:
  Components: Functor: <constant>
              Term:  <term>. Amount: >=1. Ordered.
```

7.2.2 Operational semantics

The ***answer-query*** abstract interpreter, presented for Relational Logic Programming, applies to the Logic Programming as well. The only difference is that the unification algorithm has to be extended to handle the richer term structure, which includes functors, and has an unbounded depth.

7.2.2.1 Unification for terms that include functors (composite terms)

The presence of function symbols complicates the unification step in the abstract interpreter. Recall that the rule selection procedure tries to ***unify*** a query goal (an atomic formula) with the head of the selected rule (an atomic formula). The unification operation, if successful, produces a ***substitution*** (***most general unifier***) for the variables in the atomic formulas.

The notion of ***substitution*** is modified as follows:

Definition: A ***substitution*** s is a finite mapping from variables to terms, such that $s(X)$ ***does not include*** X.

All other substitution and unification terminology stays unchanged.

- `unify(member(X,tree(X,Left,Right)) ,`
  `      member(Y,tree(9,void,tree(3,void,void))))`

  yields the mgu substitution: `{Y:=9, X:=9, Left:=void, Right=tree(3,void,void)}`

- `unify(member(X, tree(9,void,tree(E1,L1,R1)) ,`
  `      member(Y,tree(Y,Z,tree(3,void,void))))`

  yields the mgu substitution: `{Y:=9, X:=9, Z:=void, E1:=3, L1:=void, R1:=void}`

- `unify(t(X,f(a),X),t(g(U),U,W))`

  yields the mgu substitution: `{X:=g(f(a)), U:=f(a), W:=g(f(a))}`

- `unify(t(X,f(X),X),t(g(U),U,W))`

  fails!

- `unify(append([1,2,3],[3,4],List),`
  `      append([X|Xs],Ys,[X|Zs]))`

  yields the mgu substitution: `{X:=1, Xs:=[2,3], Ys:=[3,4], List:=[1|Zs]}`

- `unify(append([1,2,3],[3,4],[3,3,4]),`
  `      append([X|Xs],Ys,[Xs|Zs]))`

  fails!

The unification algorithm presented for Relational Logic Programming applies also to Logic Programming. The only difference appears in the kind of terms that populate the ***disagreement sets*** that are computed, and an ***occur check*** restriction that prevents infinite unification efforts.

1. **Disagreement sets:**

   - `disagreement-set`$(t(X, f(a), X),$
     $t(g(U), U, W)) = \{X, g(U)\}$
   - `disagreement-set`$(append([1,2,3],[3,4], List),$
     $append([1|Xs], Ys, [X|Zs])) = \{[2,3], Xs\}$
   - `disagreement-set`$(append([1,2,3],[3,4],[3,3,4]),$
     $append([1,2,3],[3,4],[[1,2]|Zs])) = \{3, [1,2]\}$

2. **Occur check:**
   `disagreement-set`$(t(g(U), f(g(U)), g(U)),$
   $t(g(U), U, W)) = \{f(g(U)), U\}$
   The disagreement set in the `unify` algorithm is used for constructing the mgu substitution, in case that one of its components is a variable. Otherwise, the unification fails. But in case that the disagreement set includes a binding $\langle X, s(X)\rangle$, such that $s(X)$ includes X as in the above example, the mgu cannot be constructed, and the unification fails.
   Therefore, the unification algorithm is extended with the occur check constraint.

A unification algorithm:

```
Signature: unify(A,B)
Type: [Atomic-formula*Atomic-formula -> Substitution union FAIL]
Post-condition: result = mgu(A,B) if A and B are unifiable
                         or FAIL, otherwise
unify(A,B) =
   let   help(s) =
      if A ∘ s = B ∘ s then s
         else let D = disagreement-set(A ∘ s, B ∘ s)
              in  if [D = {X, t} and X does not occur in t]
                            /* The occur check constraint
                  then help(s ∘ {X = t})
                  else FAIL
              end
   in  help( {} )
   end
```

Comparison of the logic programming unification with the ML pattern matching:

1. In ML patterns appear only in function definitions. Expressions in function calls are first evaluated, and do not include variables when matched.

2. Patterns in ML do not allow repeated variables. Compare:

```
  Logic programming                      ML
append([], Xs, XS).                 val  rec append  =
append([X|Xs], Ys, [X|Zs] :-          fn ( [],lst ) => lst
              append(X, Ys, Zs).       | (h::tail, lst) =>
                                               h::append(tail,lst)


member(X, [X|Xs).                   val rec member =
member(X, [Y|Ys]) :-                fn (el, []) => false
            member(X, Ys).           | (el, [h::tail] =>
                                        el = h orelse member(el,tail)
```

3. ML restricts pattern matching to equality types. In logic programming, since there is no evaluation, and comparison is by unification rather than equality, there are no restrictions.

Expressivity and decidability of Logic Programming:

1. Logic Programming has the expressive power of Turing machines. That is, every computable program can be written in Logic Programming. In particular, every Scheme or ML program can be written in Prolog, and vice versa.

2. Logic Programming is only partially decidable - unlike Relational Logic Programming. That is, the problem "Is `Q` provable from `P`", denoted `P |- Q`, is partially decidable. The finiteness argument of Relational Logic Programming does not apply here since in presence of functors, the number of different atomic formulas is unbounded (since terms can be nested up to unbounded depth). Therefore, terminating proofs can have an unbounded length.

7.2.3 Data Structures

7.2.3.1 Trees

1. Defining a tree

```
% Signature: binary_tree(T)/1
% Purpose:  T is a binary tree.
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :-
             binary_tree(Left),binary_tree(Right).
```

2. Tree membership:

```
% Signature: tree_member(X, T)/2
% Purpose:    X is a member of T.
tree_member(X, tree(X, _, _)).
tree_member(X, tree(Y,Left, _)):- tree_member(X,Left).
tree_member(X, tree(Y, _, Right)):- tree_member(X,Right).
```

Note: X might be equal to Y in the second and third clauses. That means that different proof paths provide repeated answers.

Queries:

```
?- tree_member(g(X),
               tree(g(a),
                    tree(g(b), void, void),
                    tree(f(a), void, void))).
?- tree_member(a, Tree).
```

– Draw the proof trees.

– Are the trees finite? Infinite? Success? Failure?

– What are the answers?

7.2.3.2 Natural number arithmetic

Logic programming does not support ***values*** of any kind. Therefore, there is no arithmetic, unless explicitly defined. Natural numbers can be represented by terms constructed from the symbol 0 and the functor s, as follows:

`0 -` denotes zero
`s(0)-` denotes 1
`s(...s(s(0))...),` n times - denotes n

1. Definition of natural numbers:

```
% Signature:  natural_number(N)/1
% Purpose:  N is a natural number.
natural_number(0).
natural_number(s(X)) :- natural_number(X).
```

2. Natural number addition:

```
% Signature:  Plus(X,Y,Z)/3
% Purpose:   Z is the sum of X and Y.
plus(X, 0, X) :- natural_number(X).
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).

?- plus(s(0), 0, s(0)).       /* checks 1+0=1
```

```
Yes.

?- plus(X, s(0), s(s(0))).       /* checks X+1=2, e.g., minus
X=s(0).

?- plus(X, Y, s(s(0))).   /* checks X+Y=2, e.g., all pairs of natural
                             numbers, whose sum equals 2
X=0, Y=s(s(0));
X=s(0), Y=s(0);
X=s(s(0)), Y=0.
```

3. Natural number binary relation - Less than or equal:

```
% Signature:  le(X,Y)/2
% Purpose: X is less or equal Y.
le(0, X) :- natural_number(X).
le(s(X), s(Z)) :- le(X, Z).
```

4. Natural numbers multiplication:

```
% Signature:  Times(X,Y,Z)/2
% Purpose: Z = X*Y
times(0, X, 0) :- natural_number(X).
times(s(X), Y, Z) :-  times(X, Y, XY), plus(XY, Y, Z).
```

7.2.3.3 Lists

1. **Syntax:**
   `[ ]` is the ***empty list***.
   `[Head|Tail]` is a syntactic sugar for `cons(Head, Tail)`, where `Tail` is a list term.
   Simple syntax for bounded length lists:
   `[a|[ ]] = [a]`
   `[a|[ b|[ ]]] = [a,b]`
   `[rina]`
   `[sister_of(rina),moshe|[yossi,reuven]] = [sister_of(rina),moshe,yossi,reuven]`
   Defining a list:

```
list([]).      /* defines the basis
list([X|Xs]) :- list(Xs).     /* defines the recursion
```

2. List membership:

   ```
   % Signature: member(X, List)/2
   % Purpose: X is a member of List.
   member(X, [X|Xs]).
   member(X, [Y|Ys]) :- member(X, Ys).

   ?- member(a, [b,c,a,d]).    /* checks membership
   ?- member(X, [b,c,a,d]).    /* takes an element from a list
   ?- member(b, Z).            /* generates a list containing b
   ```

3. List concatenation:

   ```
   % Signature: append(List1, List2, List3)/3
   % Purpose: List3 is the concatenation of List1 and List2.
   append([], Xs, Xs).
   append([X|Xs], Y, [X|Zs] ) :- append(Xs, Y, Zs).

   ?- append([a,b], [c], X).           /* addition of  two lists
   ?- append(Xs, [a,d], [b,c,a,d]).    /* finds a difference between lists
   ?- append(Xs, Ys, [a,b,c,d]).       /* divides a list into two lists
   ```

4. List selection using `append`:

   (a) List prefix and suffix:

   ```
   prefix(Xs, Ys) :- append(Xs, Zs, Ys).
   suffix(Xs, Ys) :- append(Zs, Xs, Ys).
   ```

   Compare the power of this one step unification with the equivalent Scheme code, that requires "climbing" the list until the prefix is found, and "guessing" the suffix.

   (b) Redefine `member`:

   ```
   member(X, Ys) :- append(Zs, [X|Xs], Ys).
   ```

   (c) Adjacent list elements:

   ```
   adjacent(X, Y, Zs) :- append(Ws, [X,Y|Ys], Zs).
   ```

(d) Last element of a list:

```
last(X, Ys) :- append(Xs, [X], Ys).
```

5. List Reverse:

(a) A recursive version:

```
% Signature:  reverse(List1, List2)/2
% Purpose: List2 is the reverse of List1.
reverse([], []).
reverse([H|T], R) :- reverse(T, S), append(S, [H], R).

?- reverse([a,b,c,d],R).
 R=[d,c,b,a]
```

But, what about:

```
?- reverse(R,[a,b,c,d]).
```

Starting to build the proof tree, we see that the second query is
`?- reverse(T1,S1), append(S1, [H1], [a,b,c,d]).`
This query fails on the first rule, and needs the second. The second rule is applied four times, until four elements are unified with the four elements of the input list. We can try reversing the rule body:

```
reverse([H|T], R) :- append(S, [H], R), reverse(T, S).
```

The new version gives a good performance on the last direction, but poor performance on the former direction.
Conclusion: Rule body ordering impacts the performance in various directions. What about ***reversing rule ordering***? In the reversed direction - ***an infinite loop***.
Typical error: Wrong "assembly" of resulting lists:

```
wrong_reverse([H|T], R):-
   wrong_reverse(T, S), append(S, H, R).
```

(b) An iterative version:

```
% Signature:  reverse(List1, List2)/2
% Purpose: List2 is the reverse of List1. This version uses an additional
```

```
              reverse helper procedure, that uses an accumulator.
reverse(Xs, Ys):- reverse_help(Xs,[],Ys).

reverse_help([X|Xs], Acc, Ys ) :-
               Reverse_help(Xs,[X|Acc],Ys).
reverse_help([ ],Ys,Ys ).

?- reverse([a,b,c,d],R).
R=[d,c,b,a]
```

The length of the single success path is linear in the list length, while in the former version it is quadratic.

Note: The `reverse_help` procedure is an helper procedure that should not reside in the global name space. Unfortunately, Logic Programming does not support nesting of name spaces (like Scheme, ML, Java). ***All names reside in the global space***.

Summary of Logic Programming:

1. Identical syntax to terms and atomic formulas. Distinction is made by syntactical context (recall, in analogy, the uniform syntax of Scheme composite expressions and Scheme lists).
2. No language primitives - besides `true, =`.
3. No computation direction: Procedures (predicates) define ***relations***, not ***functions***.
4. No run time errors.
5. Relational logic programming is decidable - although, there can be infinite branches in search trees.
6. No nesting of name spaces! No local procedures!

7.3 Meta-tools: Backtracking optimization (cuts); Unify; Meta-circular interpreters

7.3.1 Backtracking optimization – The `cut` operator

Backtracking along the proof tree is very expensive. Therefore, there is an obvious interest to avoid needless search. Such cases are:

1. **Exclusive rules:** The proof tree is deterministic, i.e., for every query, there is at most a single success path. Once a success path is scanned, no point to continue the search for alternative solutions.
2. **Deterministic domain rules:** It is known to the designer that once a proof path is taken, there are no other solutions.
3. **Erroneous alternatives:** Alternative proofs yield erroneous answers, i.e., enable skipping mandatory requirements.

In these cases, the proof tree can be pruned, so that the interpreter does not try alternative solutions (and either fails or make mistakes).

Example 7.7. *Deterministic domain rules and erroneous alternatives:*

Assume a domain of colored pieces, with two domain rule:

1. For every color there is at most a single piece.
2. Every piece has a single color.

. A modeler can describe the domain with the following relations:

```
part(a).  % 1
part(b).  % 2
part(c).  % 3

red(a).     % 1

black(b).   % 1

% color(P,C)/2
color(P,red) :- part(P), red(P).      % 1
color(P,black) :- part(P), black(P).  % 2
color(P,unknown) :- part(P).          % 3
```

but testing using queries reveals two problems:

```
?- color(a,C).
C = red ;
C = unknown.

 ?- color(P,red).
P = a ;
false.
```

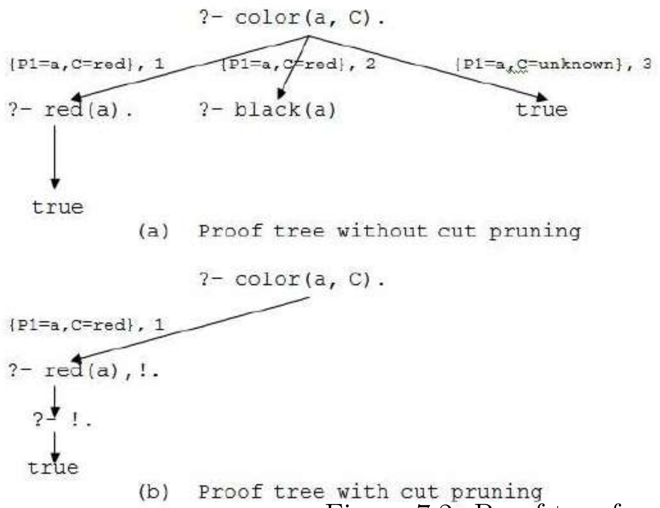


Figure 7.2: Proof tree for colors example

The first query should have a single solution. Rule (3) of predicate `color` is a ***default rule***, based on the second domain rule: Once a color of a part is successfully proved, no other alternative should be tried. The `unknown` color is designed for non-red and non-black colors, and not as an alternative color for a piece. Therefore, the tree shown in Figure 7.2(a), wrongly finds the `unknown` color for `a`.

Moreover, the second query, once finding the answer `P=a`, can terminate, based on the first domain rule. But, the program still searches in vain for a possible alternative solution.

The ***cut system predicate***, denoted `!` , is a Prolog built-in predicate, for pruning proof trees. If used after the color has been identified, it cuts the proof tree so that no alternative colors are searched for a piece, and no alternative pieces are searched for a color:

```
color(P,red) :- part(P), red(P),!.  % 1
color(P,black) :- part(P), black(P),!.   % 2
color(P,unknown) :- part(P).  % 3
```

The new proof tree is shown in Figure 7.2(b). Note that if the first domain rule does not hold, the cuts remove correct solutions, i.e., wrongly prune the proof tree!

The cut goal succeeds whenever it is the current goal, and the proof tree is trimmed of all alternatives on the way back to and **including the point in the derivation tree where the cut was introduced into the sequence of goals (the head goal in the ! rule)**.

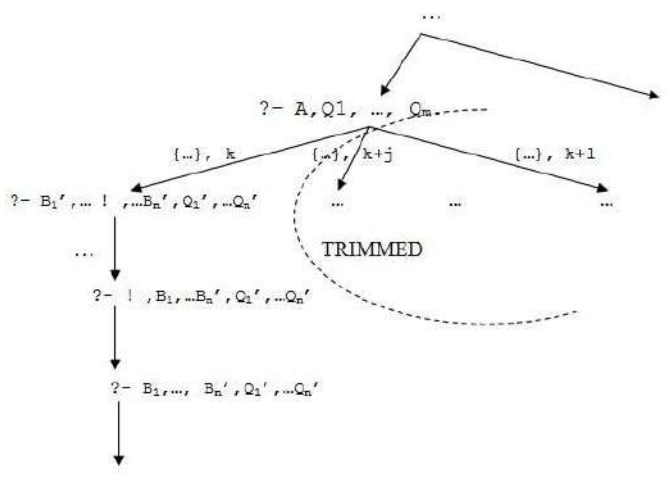

?- A,Q1, ..., Qm
{...}, k
{...}, k+j
{...}, k+1
?- B1',... ! ,...Bn',Q1',...Qn'
TRIMMED
?- ! ,B1,...Bn',Q1',...Qn'
?- B1,..., Bn',Q1',...Qn'

Figure 7.3: Proof tree pruning by the `Cut` operator

Proof tree pruning: For the rule

```
Rule k:    A :- B1, ...Bi,  !, Bi+1, ..., Bn.
```

all alternatives to the node where `A` was selected are trimmed. Figure 7.3 demonstrates the pruning caused by `cut`:

Example 7.8. *Erroneous alternatives:* Consider the following erroneous program that intends to check whether every list element of a list includes some key:

```
% Signature: badAllListsHave(List,Key)/2
% Purpose:   Check whether Key is a member of every element of List
%            which is a list.
% Type: List is a list.
badAllListsHave( [First|Rest],Key):-
                 is_list(First), member(Key,First),badAllListsHave( Rest,Key).
badAllListsHave( [_|Rest],Key):- badAllListsHave( Rest,Key).
badAllListsHave( [ ],_).
```

The query

```
?- badAllListsHave( [ [2], [3] ],3).
```

succeeds, since the second rule enables skipping the first element of the list. The point is that once the `is_list(First)` goal in the first rule succeeds, the second rule cannot function as an alternative. Inserting a cut after the `is_list(First)` goal solves the problem, since it prunes the erroneous alternative from the tree:

```
% Signature: allListsHave(List,Key)/2
% Purpose:   Check whether Key is a member of every element of List
%            which is a list.
% Type: List is a list.
allListsHave( [First|Rest],Key):-
    is_list(First),  !, member(Key,First),allListsHave( Rest,Key).
allListsHave( [_|Rest],Key):-
    badAllListsHave( Rest,Key).
allListsHave( [ ],_).
```

Example 7.9. *Exclusive rules (using arithmetics):*

```
% Signature: minimum(X,Y,Min)/3
% Purpose:   Min is the minimum of the numbers X and Y.
% Type: X,Y are Numbers.
% Pre-condition: X and Y are instantiated.
minimum(X,Y,X) :- X =< Y,!.
minimum(X,Y,Y) :- X>Y.
```

The cut prevents useless scanning of the proof tree.
But:

```
minimum(X,Y,X) :- X =< Y,!.
minimum(X,Y,Y).
```

is wrong. For example, the query `?- minimum(1,2,2)` succeeds.
The problem here is that the cut is not used only as a pruning means, but as part of the program specification! That is, if the cut is removed, the program does not compute the intended minimum relation. Such cuts are called ***red cuts***, and are not recommended. The ***green cuts*** are those that do not change the meaning of the program, only optimize the search.

Example 7.10. *Exclusive rules (using arithmetics):*

A program that defines a relation `polynomial(Term, X)` that states that `Term` is a polynomial in `X`. The polynomials are treated as symbolic expressions. The program is deterministic: A single answer for every query. Therefore, once a success answer is found, there is no point to continue the search.

```
% Signature: polynomial(Term,X)/2
% Purpose:   Term is a polynomial in X.
polynomial(X,X) :-!.
polynomial (Term,X) :-
             constant(Term), !.
polynomial(Terml+Term2,X) :-
             !, polynomial(Terml,X), polynomial(Term2,X).
polynomial(Terml-Term2,X) :-
             !, polynomial(Terml,X), polynomial(Term2,X).
polynomial(Terml*Term2,X) :-
             !, polynomial(Terml,X), polynomial(Term2,X).
polynomial(Terml/Term2,X) :-
             !, polynomial(Terml,X), constant(Term2).
polynomial(TermTN,X) :-
             !, integer(N), N > 0, polynomial(Term,X).

% Signature: constant(X)12
% Purpose:   X is a constant symbol (possibly also a number).
             Atomic is a Prolog type identification built-in predicate.
constant(X) :- atomic(X).
```

Using the cut, once a goal is unified with a rule head, the proof tree is pruned such that no other alternative to the rule can be tried.

7.3.2 Unify

Recall the unification algorithm:

```
Signature: unify(A,B)
Type: [Atomic-formula*Atomic-formula -> Substitution union FAIL]
Post-condition: result = mgu(A,B) if A and B are unifiable
                         or FAIL, otherwise
unify(A,B) =
   let   help(s) =
      if A ∘ s = B ∘ s then s
         else let D = disagreement-set(A ∘ s, B ∘ s)
              in  if [D = {X, t} and X does not occur in t]
                                /* The occur check constraint
                  then help(s ∘ {X = t})
                  else FAIL
              end
   in  help( {} )
```

```
end
```

The algorithm can be implemented in Logic Programming, but it requires some extra logical tools for inspection of term structure. The algorithm is based on the `apply-substitution` operation, and requires the ability to

1. Distinguish between different variables.

2. Distinguish between variables to constants.

The implementation below assumes that the given atomic formulas have been preprocessed into ***ground terms*** (not including variables), such that constants denoting variables are distinguished from true constants. For example, the atomic formulas `c(f(X), G, X), c(G, Y, f(d))` turn into: `c(f(x), g, x), c(g, y, f(d))`. The implementation uses the meta-programming predicate `univ`, denoted `=..`, which turns a term into a list of the functor followed by the arguments. For example:

```
1 ?- f(a,X,g(Y))=..Term_list.
Term_list = [f, a, X, g(Y)].
2 ?- Term=..[f, a, X, g(Y)].
Term = f(a, X, g(Y)).
```

The distinction between variables to constants is implemented using a variable and a constant predicates.

We present first the `substitute` procedure, which implements the `apply-substitution` operation.

```
% Signature: substitute(Exp, S, ExpS)/3
% Purpose: ExpS is the application of a substitution S on a ground term Exp.
% Examples:
% constant_list([ a, b, c, d, [], 5, 6]).
% variable_list([x, y, z]).
% ?-  substitute(p(a, b(x), c(d(y)), x), [[z, 5], [x, 6]], Ans).
% Ans = p(a, b(6), c(d(y)), 6).
% Pre-condition: All logic programming constant symbols in a query must be
%                declared either as expression constants  or as expression
%                variables. That is, belong to one of the constant or variable
%                lists. Otherwise, the program enters an infinite loop.
%
substitute(Exp, [], Exp) :- !.                                            %1
                        % green cut to avoid backtracking in (2)-(6)
substitute(X, S, XS) :- variable(X), member([X, XS], S),                  %2
                        !.             % red cut to avoid backtracking in (3)
substitute(X, S, X) :- variable(X),                                      %3
```

```
                        !.          % green cut to avoid univ of atom in (6)
substitute(X, S, X) :- constant(X),                                     %4
                        !.          % green cut to avoid univ of atom in (6)
substitute([E|Exp],S,[ES|ExpS]) :-                                      %5
                            !,    % red cut to avoid [E|Exp] unification in (6)
                            substitute(E, S, ES), substitute(Exp, S, ExpS),
                            !.          % green cut to avoid rerun of the code
substitute(Exp, S, ExpS) :- Exp =.. List,                               %6
                            substitute(List, S, ListS), ExpS =.. ListS.
```

unify is a three-ary predicate, with two parameters that are the terms to be unified, and a third 'mgu parameter.

```
% Signature: unify(A, B, Mgu)/3
% Examples:
% ?- unify(a(x, 5), a(6, y), Mgu).
% Mgu = [[y, 5], [x, 6]].
% ?- unify(c(a(x), z, x), c(z, y, a(d)), Mgu).
% Mgu = [[x, a(d)], [y, a(a(d))], [z, a(a(d))]]
% ?- unify(c(a(x), d, x), c(z, y, a(z)), Mgu).
% false
%
unify(A, B, Mgu) :- help(A, B, [], Mgu).                            %1

help(A, B, S, Mgu) :- substitute(A, S, AS),                         %1
                      substitute(B, S, BS),
                      help_helper(A, B, S, AS, BS, Mgu).

help_helper(A, B, S, AS, AS, S) :- !.                               %2
                                % green cut to avoid disagreement run in (3)
help_helper(A, B, S, AS, BS, Mgu) :- disagreement(AS, BS, [X, T]),  %3
                            not_occurs(T, X),
                            substitute(S, [[X, T]], SS),
                            help(A, B, [[X, T]|SS], Mgu).
```

The disagreement and the not_occur procedures:

```
% Signature: disagreement(A, B, Set)/3
disagreement(AS, BS, [AS, BS]) :- variable(AS),                     %1
                                    % green cut to avoid backtracking in (5)
disagreement(AS, BS, [BS, AS]) :- variable(BS),                     %2
                                !.  % green cut to avoid backtracking in (6)
```

```
disagreement([A|AS], [A|BS], [X, T]) :- !,  %3 red cut to avoid backtracking in (7)
                                        disagreement(AS, BS, [X, T]).
disagreement([A|_], [B|_], [X, T]) :- !, %4  red cut to avoid backtracking in (7)
                                      disagreement(A, B, [X, T]).
disagreement(AS, _, _) :- constant(AS),                                 %5
                          !,        % red cut to avoid univ of constant in (7)
                          fail.
disagreement(_, BS, _) :- constant(BS),                                 %6
                          !,        % red cut to avoid univ of constant in (7)
                          fail.
disagreement(A, B, [X, T]) :- A =.. A_list,                             %7
                              B =.. B_list,
                              disagreement(A_list, B_list, [X, T]).

% Signature: not_occurs(T, X)/2
%
not_occurs(X, X) :- !,                  %1 red cut to avoid backtracking in (2)
                    fail.
not_occurs(T, _X) :- variable(T),                                       %2
                     !.                % green cut to avoid univ of atom in (5)
not_occurs(T, _X) :- constant(T),                                       %3
                     !.                % green cut to avoid univ of atom in (5)
not_occurs([T|TT], X) :- !,            %4  red cut to avoid univ of list in (5)
                         not_occurs(T, X), not_occurs(TT, X),
                         !.            % green cut to avoid rerun of the code
not_occurs(T, X) :- T =.. T_list,                                       %5
                    not_occurs(T_list, X).

% Signature: variable(Name)/1
variable(X) :- variable_list(Vars), member(X,Vars).
variable_list([x,y,z]).

% Signature: constant(Name)/1
constant(C) :- constant_list(Consts), member(C, Consts).
constant_list([p,a,b,c,d,[],5,6]).
```

7.3.3 Meta-circular interpreters for Logic Programming

Recall the abstract interpreter for logic programs.

1. It is based on ***unification***, ***search***, and ***backtracking***.

   (a) Goal selection - left most for Prolog.

   (b) Rule selection - first for Prolog, with backtracking to the following rules, in case of a failure.

2. It has two points of non-deterministic selection.

This behavior can be encoded into a logic programming procedure `solve` that implements the abstract interpreter algorithm.

We present several `solve` procedures. The interpreters exploit the uniformity of the syntax of terms and of atomic formulas: The atomic formulas of the program are read as terms, for the interpreter[1]. All interpreters work in two stages:

1. **Pre-processing:** The given program `P` (facts and rules) is translated into a new program `P'`, with a single predicate `rule`, with facts alone.

2. Queries are presented to the procedure `solve`, and to the transformed program `P'`.

The first interpreter is a "vanilla" one: A basic interpreter that fully uses Prolog's rule and goal selection rules. The rest of the interpreters vary either in providing also a proof tree in addition to the proof, or in the amount of control they provide to goal selection order.

Pre-processing – Program transformation: The rules and facts of the logic program are transformed into an all facts procedure `rule`. The rule:

```
A :- B1, B2, ..., Bn
```

Is transformed into the fact:

```
rule(A, [B1, B2, ..., Bn] ).
```

A fact `A.` is transformed into `rule(A, true)`. For example, the program:

```
% father(abraham, isaac).
% father(haran, lot).
% father(haran, milcah).
% father(haran, yiscah).
% male(isaac).
% male(lot).
% female(milcah).
% female(yiscah).
% son(X, Y) :- father(Y, X),
%              male(X).
% daughter(X, Y) :- father(Y, X),
%                   female(X).
```

[1]Recall the Scheme meta-circular interpreter, which exploits the uniform syntax of Scheme expressions and the printed form of lists.

is transformed into the program:

```
rule(father(abraham, isaac), true).             %1
rule(father(haran, lot), true).                 %2
rule(father(haran, milcah), true).              %3
rule(father(haran, yiscah), true).              %4
rule(male(isaac), true).                        %5
rule(male(lot), true).                          %6
rule(female(milcah), true).                     %7
rule(female(yiscah), true).                     %8
rule(son(X, Y), [father(Y, X), male(X)]).       %9
rule(daughter(X, Y), [father(Y, X), female(X)]).   %10
```

The new program consists of facts alone.

Vanilla (basic) interpreter

The vanilla interpreter simulates Prolog's goal selection rule, and relies on Prolog's rule selection.

```
% Signature: solve(Exp)/1
% Purpose:  Vanilla interpreter.
% Examples:
% ?- solve(son(lot, haran)).
% true.
%
% ?- solve(son(X, Y)).
% X = isaac,
% Y = abraham ;
% X = lot,
% Y = haran ;
% false.
%
solve(true).                          %1
solve([]).                            %2
solve([A|B]) :- solve(A), solve(B).   %3
solve(A) :- rule(A,B), solve(B).      %4
```

Interpreter operation: The first fact ends a proof of a fact, the second fact ends the proof of a list of goals (the body of a program rule), the third rule selects a goal to prove, and the fourth rule finds a program rule whose head unifies with the given goal. This interpreter relies on Prolog for doing almost everything:

1. Reading and parsing the input.
2. Unification and rule selection (following the order of the `rule` procedure).
3. Goal selection and recursive computation (reduction).

Interpreter with associated proofs for answers:

Successful computations can be associated with a parse tree of the proof:

```
% Signature: solve(Exp, -Proof)/2
% Purpose: Interpreter and associate proofs with success answers
% Examples:
% ?- solve(son(lot,haran), Proof).
% Proof = node(son(lot,haran),
%              [node(father(haran,lot),true),node(male(lot),true)]);
% false.
% ?- solve(son(X,Y), Proof).
% X = isaac,
% Y = abraham,
% Proof = node(son(isaac,abraham),
%              [node(father(abraham,isaac),true),node(male(isaac),true)]) ;
% X = lot,
% Y = haran,
% Proof = node(son(lot,haran),
%              [node(father(haran,lot),true),node(male(lot),true)]) ;
% false.
%
solve(true, true).                                                  %1
solve([], []).                                                      %2
solve([A|B],[ProofA|ProofB]) :- solve(A,ProofA), solve(B, ProofB).   %3
solve(A, node(A,Proof)) :- rule(A,B), solve(B,Proof).               %4
```

Interpreter operation:

1. The first two facts provide proofs to `true` and to the end of a proof.
2. The third rule structures the proof of list of goals as a list of proofs.
3. The fourth rule specifies the proof of an atomic goal `A` that unifies with the head of a program rule with body `B`, to be a tree-node labeled by the goal `A`, and with the proof of `B` as child trees.

To further clarify, given the propositional program

```
a :- b,c,d,e.          %1
b :- f,g.              %2
c.                    %3
d.                    %4
c.                    %5
e.                    %6
f.                    %7
g.                    %8
```

whose pre-processing yields:

```
rule(a, [b,c,d,e]).          %1
rule(b, [f,g]).              %2
rule(c, true).               %3
rule(d, true).               %4
rule(c, true).               %5
rule(e, true).               %6
rule(f, true).               %7
rule(g, true).               %8

6 ?- solve(a, Proof).
Proof = node(a, [node(b, [node(f, true), node(g, true)]),
%                node(c, true),
%                node(d, true),
%                node(e, true)]) ;
Proof = node(a, [node(b, [node(f, true), node(g, true)]),
%                node(c, true),
%                node(d, true),
%                node(e, true)]) ;
false.
```

Addition of failed proofs: Finite failure proofs can be added by detecting goals that could not be computed by the former `solve`, and adding them as a failure node to the proof. This can done by addition of two default rules at the end of the `solve`. One rule, number 5, eliminates failed goals that unify with some `rule` facts, and the last one, number 6, adds the failure node.

```
% Signature: solve(Exp, Proof)/1
% Purpose: Interpreter and associate proofs with success answers.
% Examples:
% ?- solve(son(lot, haran), Proof).
% Proof = node(son(lot, haran),
%              [node(father(haran, lot), true), node(male(lot), true)]);
% false.
% ?- solve(son(X, Y), Proof).
% X = isaac,
% Y = abraham,
% Proof = node(son(isaac, abraham),
%              [node(father(abraham, isaac), true), node(male(isaac), true)]) ;
% X = lot,
% Y = haran,
% Proof = node(son(lot, haran),
%              [node(father(haran, lot), true), node(male(lot), true)]) ;
% X = milcah,
% Y = haran,
% P = node(son(milcah, haran),
%          [node(father(haran, milcah), true), node(male(milcah), fail)]) ;
% X = yiscah,
% Y = haran,
% P = node(son(yiscah, haran),
%          [node(father(haran, yiscah), true), node(male(yiscah), fail)]) ;
% false.
%
solve(true, true) :- !.          %1  red cut to avoid Proof unification in (6)
solve([], []) :- !.              %2  red cut to avoid Proof unification in (6)
solve([A|B], [ProofA|ProofB]) :- !, %3  red cut to avoid Proof unification in (6)
                                 solve(A, ProofA), solve(B, ProofB).
solve(A, node(A, Proof)) :- rule(A, B), solve(B, Proof).              %4
solve(A, _) :- rule(A, _),                                            %5
               !,                    % red cut to avoid backtracking in (7)
               fail.
solve(A, node(A, fail)).                                              %6
```

Interpreter with management of goals

This interpreter uses an explicit control of the goal selection order, using a ***waiting list*** of goals (reminds the CPS approach).

```
% Signature: solve(Exp)/1
% Purpose: Interpreter with explicit management of a waiting list of goals.
% Examples:
% ?- solve(son(lot, haran)).
% true.
% ?- solve(son(X, Y)).
% X = isaac,
% Y = abraham ;
% X = lot,
% Y = haran ;
% false.
%
solve(Goal) :- solve(Goal, []).                                     %1

solve(true, Goals) :- solve([], Goals).                             %1
solve([], []).                                                      %2
solve([], [G|Goals]) :- solve(G, Goals).                            %3
solve([A|B], Goals) :- extend_waiting_list([A|B], Goals, NewGoals),  %4
                       solve([], NewGoals).
solve(A, Goals) :- rule(A, B), solve(B, Goals).                     %5

% Signature: extend_waiting_list(Goals, WaitingList, NewWaitingList)/3
% Purpose: Extend the waiting list of goals.
%
extend_waiting_list(Goals, WaitingList, NewWaitingList) :-          %1
%                                  append(Goals, WaitingList, NewWaitingList).
```

Interpreter operation: The interpreter `solve/2` keeps the goals to be proved in a waiting list - its second argument. The first argument includes the current goals to be proved. The first four rules are waiting list management.

1. Rule (1) refers to a situation where a fact was proved, and there might be goals in the waiting list.
2. Rule (2) is the end of processing: No goal to prove and empty waiting list.
3. Rule (3) is the goal selection rule.
4. Rule (4) refers to a situation where there is a list of current goals to prove. The current goals are added to the waiting list.
5. Rule (5) is the core of the interpreter. By default, the current goal is an atomic formula, and not a list. First, there is a search for a rule or fact of the original program `P` whose head matches the current goal. Then, the body of this fact or rule is solved, in the context of the waiting goals.

Interpreter that combines proofs and management of goals

```
% Signature: solve(Exp, Proof)/2
% Purpose: Interpreter with explicit management of a waiting list of goals., and
%          associate proofs with success answers.
% Examples:
% ?- solve(son(lot, haran), Proof).
% Proof = node(son(lot, haran),
%              [node(father(haran, lot), true), node(male(lot), true)]);
% false.
% ?- solve(son(X, Y), Proof).
% X = isaac,
% Y = abraham,
% Proof = node(son(isaac, abraham),
%              [node(father(abraham, isaac), true), node(male(isaac), true)]) ;
% X = lot,
% Y = haran,
% Proof = node(son(lot, haran),
%              [node(father(haran, lot), true), node(male(lot), true)])
% false.
%
solve(Goal, Proof) :- solve(Goal, [], Proof).                          %1

% Signature: solve(Exp, WaitingList, Proof)/3
%
solve(true, [], true) :- !.       %1 red cut to avoid Goals unification in (3)
solve(true, Goals, Proof) :- solve([], Goals, Proof).                   %2
solve([], [], []).                                                      %3
solve([], [G|Goals], [Proof1|Proof2]) :- solve(G, [], Proof1),          %4
                                         solve([], Goals, Proof2).
solve([A|B], Goals, Proof) :- extend_waiting_list([A|B], Goals, NewGoals),  %5
                              solve([], NewGoals, Proof).
solve(A, Goals, node(A, Proof)) :- rule(A, B), solve(B, Goals, Proof).     %6
```

7.4 Prolog

Pure Prolog: Logic programming with the Prolog specific selection rules:

1. Left most goal.
2. First rule whose head unifies with the selected goal.

Prolog: Extension with Arithmetic, system predicates, primitives, meta-logic (reflection) predicates, extra-logic predicates, high order predicates. The two main features of pure logic programming are lost:

1. Unidirectional definitions.
2. No run time errors.

7.4.1 Arithmetics

The system predicates for arithmetic provide interface to the underlying arithmetic capabilities of the computer. Prolog provides:

1. An arithmetic evaluator: `is`.
2. Arithmetic operations: `+, -, *, /`
3. Primitive arithmetic predicates: `=, !=, <, >`.

All arithmetic predicates pose ***instantiation requirements*** on their arguments. They cause ***runtime errors*** if their arguments cannot be evaluated to numbers.

The `is` arithmetic evaluator is written as an infix operator:

`<value> is <expression>.`

`<expression>` is ***evaluated*** and unified with `<value>`. `<expression>` must be fully instantiated to a number value.

`V is 3 + 6.`	succeeds with V=9.
`V is 3 + X.`	fails since X cannot be evaluated.
`9 is 3 + 6.`	succeeds.
`3+6 is 3 + 6.`	fails.
`V is V + 1.`	fails.

Examples:

1. Factorial - recursive:

```
% Signature: factorial(N, F)/2
% Purpose: F is the factorial of N.
% Type:  N,F: Type is Integer.
% Pre-condition:  N must be instantiated. N>=0.
factorial(0, 1).
factorial(N, F) :-
    N > 0,    /* Defensive programming! Should belong to the  pre-condition.
    N1 is N -1,
    factorial(N1, F1),
    F is N*F1.
```

2. Factorial - iterative.

```
% Signature: factorial(N,F)/2
% Purpose: F is the factorial of N.
% Type:  N,F: Type is Integer.
% Pre-condition:  N must be instantiated. N>=0.
factorial(N, F) :- factorial(N, 1, F).

% Signature:  factorial(N, Acc, F)/3
factorial(0, F, F).
factorial(N, Acc, F) :-
    N > 0,
    N1 is N -1,
    Acc1 is N*Acc,
    factorial(N1, Acc1, F).
```

3. Factorial - another iterative version.

```
% Signature:  factorial(N,F)/2
% Purpose:  F is the factorial of N.
% Type:  N,F: Type is Integer.
% Pre-condition:  N must be instantiated. N>=0.
factorial(N, F) :- factorial(0, N, 1, F).

% Signature: factorial(I, N, Acc, F)/3
factorial(N, N, F, F).
factorial(I, N, Acc, F) :-
    I < N,
    I1 is I +1,
    Acc1 is Acc * I1,
    factorial(I1, N, Acc1, F).
```

4. Computing the sum of members of an integer-list – recursion.

```
% Signature:  sumlist(List, Sum)/2.
% Purpose:  Sum is the sum of List's members.
% Type: List: type is list. Its members are integers.
% Sum: Type is Number.
sumlist( [], 0).
sumlist( [I|Is], Sum) :-
             sumlist(Is, Sum1),
             Sum is Sum1 + I.
```

5. Computing the sum of members of an integer-list - iteration (with accumulator).

```
% Signature:  sumlist(List, Sum)/2.
% Purpose: Sum is the sum of List's members.
% Type: List: type is list. Its members are integers.
        Sum: Type is Number.
sumlist(List, Sum) :- sumlist(List, 0, Sum).

% Signature: sumlist(List, Acc, Sum)/3.
sumlist([], Sum, Sum).
sumlist([I|Is], Sum1, Sum) :-
                     Sum2 is Sum1 + I,
                     sumlist(Is, Sum2, Sum).
```

Restrictions on language primitive predicate symbols:

– They ***cannot be defined*** - appear in rule/fact heads. Because they are already defined.

– They denote infinite relations. Therefore, when they are selected for proving - their arguments must be already instantiated (substituted). Otherwise - the computation will explore an infinite number of facts. That is, the proof of

```
?- 8 < 10.
```

immediately succeeds. But the proof of

```
?- X < 10.
```

has an infinite number of answers. Therefore, it causes a ***run time error!***

Prolog includes, besides arithmetic, a rich collection of system predicates, primitives, meta-logic (reflection) predicates, extra-logic predicates, high order predicates. They are not discussed in this introduction.

7.4.2 Negation in Logic Programming

Logic programming allows a restricted form of negation: ***Negation by failure***: The goal `not(G)` succeeds if the goal `G` fails, and vice versa.

1.
   ```
   male(abraham).
   married(rina).
   bachelor(X) :- male(X), not(married(X)).

   ?- bachelor(abraham).
   Yes.
   ```

2.
   ```
   unmarried_student(X) :- student(X), not(married(X)).
   student(abraham).

   ?- unmarried_student(X).
   X = anraham.
   ```

3. Define a relation that just verifies truth of goals, without instantiation:

   ```
   verify(Goal) :- not(not(Goal)).
   ```

   Opens two search trees - one for each negation. Result is success or fail without any substitution.

Restrictions:

1. **Negated relations cannot be defined:** Negation appears only in rule bodies or queries.
2. Negation is applied to ***goals without variables***.

Therefore:

```
unmarried_student(X) :- not(married(X)), student(X).
```

is dangerous! The query

```
?- unmarried_student(X).
```

is wrong: Check the search tree!

Chapter 8

Imperative Programming

Mutation and Local State

Real world software like:

– Management of a bank account.

– Student registration system.

– Course grading system.

are characterized as ***state based*** systems. The state is created and related to ***change along time***. Systems deal with the ***current balance*** of an account, the ***current student registration database***, and the ***current average grade***. The bank account balance, the registration database and the average grade are changed over time, following management actions.

Such systems differ from the mathematical problems handled in previous chapters, like sequence summation or writing a language interpreter. The theory problems require computation of results, but have no notion of changing problem elements.

One way to model system states is by using ***lazy lists***, that function as time sequences: The lazy list holds the states of the system. For example, assume that the withdrawals from a bank account are kept as a lazy list:

```
(define make-withdrawals
   (lambda (withdraw)
     (cons withdraw
           (lambda () (make-withdrawals (read))))
    ))
```

The states of a bank account to which withdrawals are applied can be modeled as a lazy list of the account balances:

```
(define account-states
   (lambda (balance withdrawals)
     (cons balance
           (lambda ()
             (account-states (- balance (car withdrawals))
                             ((cdr withdrawals)))))
     ))

> (take (account-states 100 (make-withdrawals 10)) 3)
5
6
7
(100 90 85)
```

With the help of a few additional auxiliary procedures that can smooth the handling of the account and the withdrawals, the account can look as if it has a really changing state – modeled as a sequence.

The more conventional, and possibly natural, way to model a changing state is by capturing an ***object*** with a concrete ***store*** that holds the ***changing state***. The Von-Neumann architecture, which is the basis for Turing machines, is based on the idea of stored and changing states. The operational semantics of a program is given by a sequence of ***system-states*** (***configurations***). A system-state is a snapshot of a system. It includes the current content of the machine tape + a pointer to the current position in the tape.
Imperative programming is characterized by:

1. Understanding ***variables*** as ***storage*** places (addresses).
2. ***Mutation operations***, like assignment, for changing the content of variables.
3. ***Operational semantics*** – a ***computation*** is a sequence of ***system-states***. A system-state is given by a pair:
   `<variable contents (values), pointer to a place in the program>`.

Mutation in Racket: Racket supports mutation in three ways:

1. ***Variable assignment*** using the special operator (mutator) `set!`.
2. A ***Box*** type, whose values are ***wrappers*** (or ***pointers***) to values that can be changed during computation.
3. ***Mutable data types***.

The assignment special operator enables modification of a value of an ***already defined variable***. The Box type has the constructor `box`, the selector `unbox` and the mutator `set-box!`. All are primitive procedures, i.e., no special evaluation rule. Mutable data types are not discussed here. If there is a need to modify components of a composite value, e.g., a list, boxes should be involved.

```
> (define lst (list 1 2 3))
> (set! lst (list 1 2 4))
'(1 2 4)
> (set! a 4)
. . set!: assignment disallowed;
 cannot set variable before its definition
 variable: a

; Boxes:
(define a (box 7))
> (* 6 (unbox a))
42
> (set-box! a (+ (unbox a) 3))
> (* 6 (unbox a))
60
; Setting the last element of a list -- must be a list of boxes:
> (define increase-last!
    (lambda (lst)
      (let ((last-el (last lst)))
        (set-box! last-el (add1 (unbox last-el))))
      ))
> (define l (map box '(1 2 3 4)))
> l
'(#&1 #&2 #&3 #&4)
> (increase-last! l)
> l
'(#&1 #&2 #&3 #&5)
> (unbox (last l))
5
;A procedure that returns a Box value:
(define wrap-with-box
  (lambda (x) (box x)))
> (define b (wrap-with-box 5))
> b
'#&5
> (set-box! b 3)
> (* 3 (unbox b))
```

```
9
(define c-add
  (lambda (x)
    (lambda (y) (+ x y))))
(define boxed-c-add (box (c-add 7)))
> ((unbox boxed-c-add) 3)
10
;A procedure with a Box valued input:
;Type: [Box(T) -> T1]
;Purpose: A setter for boxes.
;         Note how Box typed parameters simulate "call by references".
> (define box-setter
    (lambda (b)
      (set-box! b (read))
      (unbox b)))
> (define b (box 4))
> (unbox b)
4
> (box-setter b)
2
2
> (unbox b)
2
```

Example 8.1 (Circular lists).

```
> (define circ-l (list (box 1) (box 2) (box 3)))
> (unbox (car circ-l))
1
> (unbox (cadr circ-l))
2
> (unbox (caddr circ-l))
3
> circ-l
'(#&1 #&2 #&3)
> (set-box! (caddr circ-l) circ-l)
> circ-l
#0='(#&1 #&2 #&#0#)
> (unbox (caddr circ-l))
#0='(#&1 #&2 #&#0#)    ; Note how circular values are printed.
> (eq? circ-l (unbox (caddr circ-l)))
#t
```

The *Box* type:

Type constructor:	$Box(T)$.	
Value constructor:	`box:`	$[T \to Box(T)]$.
Selector:	`unbox:`	$[Box(T) \to T]$.
Mutator:	`set-box!:`	$[Box(T) \to Void]$.
Identifier:	`box?:`	$[T \to Boolean]$.
Equality:		
Value equality:	`equal?:`	$[T1 * T2 \to Boolean]$.
Identity equality:	`eq?:`	$[T1 * T2 \to Boolean]$.

Notes:

– The assignment operator `set!` and the Box type setter `set-box!` introduce new kinds of ***side effects***: variable and structure change. We have already seen two kinds of side effect procedures: `define`, which changes the global environment mapping, and I/O procedures, of which we have seen only the “O” – the `display` procedure. Side effect procedures are applied for the sake of their side effects, and not for their returned value. Therefore, the returned value of side effect procedures is `void`.

– In imperative programming, where procedures can be applied for the sake of their side effects, there is a point to use sequences of expressions, as allowed in bodies of `lambda, let` and `letrec` expressions, or in clauses of `cond` expressions. Therefore, in this paradigm, the `begin` special operator, that wraps a sequence of expressions as a single expression is useful. For example, within `if` expressions.

– The Box type was already used in the evaluators code, in Chapter 6, for implementing environments. An environment was implemented as a sequence of boxed frames. This was done for the purpose of enabling changes of the global frame, following the evaluation of a `define` special form. The `set!` special operator enables changes of non-global frames.

8.1 State Based Modeling

We want to model states in a functional language. How can we modify the given operational semantics for that purpose? We will use ***frames*** as a repository for states:

1. A “state-frame” holds state variables
2. A state-frame is held by a closure that functions as an object, whose state is given by the frame
3. A state-frame is held by ***setter-closures***, that set the value of state variables.

Therefore, a standard template for state management in Scheme is having a closure created within the scope of a `let` expression:

```
(define make-state-manager
   (let ((var1 val1)
         ...
         (varn vlan))
     (lambda (p1 ...) (.... (set! vari expi) ...)))
```

Example 8.2 (Random number generation). *Generate a sequence of random numbers, assuming that a* ***`rand-init`*** *number, and a procedure* ***`rand-update`****, that computes the next number in the sequence, are given:*

```
(define rand
   (let ((x random-init))
     (lambda () (set! x (rand-update x)) x)))
```

In this modeling, `rand` is an object whose local state is kept in `x`, and changes with successive calls to `rand`. The alternative to a local state based `rand`, is to pass the current random value to all client procedures, and explicitly call `rand-update`.

8.1.1 Object modeling

Example 8.3 (Modeling a bank account withdrawal).

```
Signature: make-withdraw(amount)
Purpose: Withdraw 'amount' from a private 'balance'.
Type: [Number -> Number]
Post-condition: result = if balance@pre >= amount
                          then balance@pre - amount
                          else balance
(define make-withdraw
   (let ((balance 100))
     (lambda (amount)
       (if (>= balance amount)
           (begin (set! balance (- balance amount))
                  balance)
           (begin (display "Insufficient funds")
                 (newline)
                  balance))
      )))
```

Note that the procedure is created within the scope of the declaration of `balance`. Therefore:

1. All (free) occurrences of `balance` within the procedure are bound by this declaration. In every application of `withdraw`, all these occurrences reference the single `balance` declaration, created by the `let` application.

2. The `balance` declaration is ***local*** (private) to the withdraw procedure. The `balance` variable is the ***local state*** of the bank account.

```
> withdraw
#<procedure:withdraw>
> (withdraw 25)
75
> (withdraw 25)
50
> (withdraw 60)
Insufficient funds
50
> (withdraw 15)
35
> balance
. . reference to an identifier before its definition: balance
```

– Draw an environment diagram in order to clarify how the `balance` local variable functions as a repository for the ***local state*** of the bank account.

– `begin` is a scheme's special operator. The syntax of a begin form is:
`(begin <exp1> <exp2> ...<expn>)`
where all `<expi>`-s are Scheme expressions. The evaluation of a `begin` form causes the expressions `<exp1>, ..., <expn>` to be evaluated in sequence. The value of the `begin` form is the value of the last argument.

  The `begin` special operator is relevant only in imperative programming – where the computation model supports side effects.

– The post condition includes the keyword `@pre`, as a suffix to the `balance` variable. The `@pre` suffix (taken from the *Object Constraint Language* (*OCL*)) designates the input value of the suffixed parameter.

  The `@pre` suffix is relevant only in imperative programming (programming with change). Why?

Example 8.4 (Make-withdraw – an Account generator)**.**

A withdrawal constructor that takes the initial balance as an argument, and produces an account:

```
Signature: make-withdraw(balance)
Purpose: Create an Account object with 'balance' as an initial amount,
         and withdrawal capabilities.
Type: [Number -> [Number -> Number]]
Contract of the created Account object:
     Signature: parameter: amount
     Purpose: Withdraw 'amount' from a private 'balance'.
     Type: [Number -> Number]
     Post-condition: result = if balance@pre >= amount
                                 then balance@pre - amount
                                 else balance
(define make-withdraw
   (lambda (balance)
     (let ((balance balance))
       (lambda (amount)
         (if (>= balance amount)
             (begin (set! balance (- balance amount))
                    balance)
             (begin (display "Insufficient funds")
                  (newline)
                  balance))
         ))))
```

Each evaluation of a `make-withdraw` form creates a withdraw procedural object, with the `make-withdraw` argument as the initial `balance`. The procedures are distinct objects and do not affect each other. Their `balance` arguments keep the ***local states*** of the accounts. Draw an environment diagram that clarifies how the local states of different account objects are managed.

```
>  (define W1 (make-withdraw 100))
>  (define W2 (make-withdraw 100))
> W1
#<procedure:...pter7\letrec.rkt:52:7>
> (W1 25)
75
> (W2 25)
75
> (W1 80)
Insufficient funds
75
> (W1 30)
45
```

```
> (W1 50)
Insufficient funds
45
> (W2 50)
25
```

Example 8.5 (A bank account object – that supports withdrawals and deposits)**.**

`make-withdraw` is extended with an internal procedure for deposits, and a getter for the balance. Its value is an internal `dispatch` procedure, that based on the given message, invokes either one of the internal procedures.

`make-account` evaluates, for each call, into a distinct `dispatch` procedure that resides in a local environment with `balance` as a local state variable. Within this environment, there are four distinct method procedures: `get-balance, set-balance!, withdraw, deposit`, and a `dispatch` procedure. Each `dispatch` procedure created by `make-account` is a procedural bank account object that handles deposits into and withdrawals from the account. Each evaluation of a `make-account` form creates a new account.

```
Signature: make-account(balance)
Type: [Number -> [Symbol -> [Number -> Number]]]
(define make-account
  (lambda (balance)
     (let ((balance balance))
       (letrec
           ((get-balance (lambda (amount) balance))
            (set-balance! (lambda (amount) (set! balance amount)))
            (withdraw (lambda (amount)
                         (if (>= balance amount)
                             (begin (set-balance! (- balance amount))
                                    balance)
                             (begin (display "Insufficient funds")
                                    (newline)
                                    balance))))
            (deposit (lambda (amount)
                        (set-balance! (+ balance amount))
                        balance))
            (dispatch (lambda (m)
                         (cond ((eq? m 'balance) get-balance)
                               ((eq? m 'withdraw) withdraw)
                               ((eq? m 'deposit) deposit)
                               (else (error 'make-account "Unknown request: ~s" m)))))
       )
```

```
        dispatch))
    ))

> (define acc (make-account 100))
> acc
#<procedure:dispatch>
> ((acc 'withdraw) 50)
50
> ((acc 'withdraw) 60)
Insufficient funds
50
> ((acc 'deposit) 30)
20
> ((acc 'withdraw) 60)
Insufficient funds
20
> (define acc2 (make-account 100))
> ((acc2 'withdraw) 30)
70
> ((acc 'withdraw) 30)
Insufficient funds
20
```

Each call to `make-account` creates a new bank account object (`dispatch` procedure). Each call to a bank account object evaluates into its local `get-balance,withdrawal` or `deposit` procedures. Draw an environment diagram that clarifies how local states of objects are kept. This example uses the ***message passing*** programming style and the local state variables approach.

> The programming languages terminology for this approach is ***encapsulation***.
> The ***hiding principle*** for system design: Protect parts of the system from each other by providing information access only to parts that ***need*** to know, and hide information from all other.
> The encapsulation approach improves ***modularity***.

The syntactic format of operating on an account procedural object may be improved by introducing ***interface*** procedures. Such procedures can act as further ***abstraction*** on the actual representation of objects:

```
Signature: deposit(account amount)
Type: [ [Symbol -> [Number -> Number]]*Number -> Number]
Post-condition: result = (account 'balance)@pre + amount
```

```
(define deposit
  (lambda (account amount)
       ((account 'deposit) amount)))

Signature: withdraw(account amount)
Type: [ [Symbol -> [Number -> Number]]*Number -> Number]
Post-condition: result = if (account 'balance)@pre >= amount
                            then (account 'balance)@pre - amount
                            else (account 'balance)
(define withdraw
  (lambda (account amount)
       ((account 'withdraw) amount)))

> (define acc (make-account 100))
> (withdraw acc 50)
50
> (deposit acc 60)
110
```

Further generalization – a single procedure for the transactions of `account`:

```
Signature: transact(account transaction-type amount)
Type: [ [Symbol -> [Number -> Number]]*Symbol*Number -> Number]
(define transact
  (lambda (account transaction-type amount)
       ((account transaction-type) amount)))

> (transact acc 'withdraw 40)
70
> (transact acc 'deposit 30)
100
```

8.1.2 Procedural implementation of mutable data structures

The modeling of objects with local state uses the approach of ***message passing***, which is the eager procedural implementation of structured data, as introduced in Chapter 3. This approach can be used also for implementing ***mutable data structures***, i.e., data structures having mutation operations. Mutators are operations that modify the local state of objects. The following example provides an implementation for the ***Mutable Pair*** ADT:

Example 8.6 (Mutable Pair ADT).

```
; Type: [T1*T2 -> [Symbol -> Procedure]]
;       The dispatch procedure returns procedures with different arities
;       (Currying could help here).
(define mpair
  (lambda (x y)
    (let ((x x)  ; No sssignment to input parameters
          (y y))
      (letrec ((get-x (lambda () x))
               (get-y (lambda () y))
               (set-x! (lambda (v)
                         (set! x v)))
               (set-y! (lambda (v)
                         (set! y v)))
               (dispatch
                 (lambda (m)
                    (cond ((eq? m 'car) get-x)
                          ((eq? m 'cdr) get-y)
                          ((eq? m 'set-car!) set-x!)
                          ((eq? m 'set-cdr!) set-y!)
                          (else (error 'mpair "Undefined operation ~s" m))))))
       dispatch))
    ))
(define mutable-car (lambda (z) ((z 'car))))
(define mutable-cdr (lambda (z) ((z 'cdr))))
(define set-car!
  (lambda (z new-value)
     ((z 'set-car!) new-value)
     ))
(define set-cdr!
  (lambda (z new-value)
     ((z 'set-cdr!) new-value)
     ))

(define p1 (mpair 1 2))
> (mutable-car p1)
1
> (mutable-cdr p1)
2
> (set-car! p1 3)
> (mutable-car p1)
3
> (set-cdr! p1 4)
> (mutable-cdr p1)
4
> (mutable-cdr (set-cdr! p1 5))
. . procedure application: expected procedure, given: #<void>; arguments were: 'cdr
```

Example 8.7 (Mutable Pair ADT with self count on setters).

The Pair ADT can be extended with self counting of the number of setter application to the pair components:

```
; Type: [T1*T2 -> [Symbol -> Procedure]]
(define mpair
  (lambda (x y)
    (let ((x x)  ; No sssignment to input parameters
          (y y)
          (setx-number 0)
          (sety-number 0))
      (letrec ((get-x (lambda () x))
               (get-y (lambda () y))
               (set-x! (lambda (v)
                         (set! x v)
                         (set! setx-number (add1 setx-number))))
               (set-y! (lambda (v)
                         (set! y v)
                         (set! sety-number (add1 sety-number))))
               (how-many-setx (lambda () setx-number))
               (how-many-sety (lambda () sety-number))
               (dispatch
                 (lambda (m)
                    (cond ((eq? m 'car) get-x)
                          ((eq? m 'cdr) get-y)
                          ((eq? m 'set-car!) set-x!)
                          ((eq? m 'set-cdr!) set-y!)
                          ((eq? m 'how-many-setx) how-many-setx)
                          ((eq? m 'how-many-sety) how-many-sety)
                          (else (error 'mpair "Undefined operation ~s" m))))))
       dispatch))
    ))
(define how-many-setx (lambda (z) ((z 'how-many-setx))))
(define how-many-sety (lambda (z) ((z 'how-many-sety))))

> (define p1 (mpair 1 2))
> (how-many-setx p1)
0
> (set-car! p1 3)
> (mutable-car p1)
3
>(how-many-setx p1)
1
> (set-car! p1 4)
> (how-many-setx p1)
2
```

Example 8.8 (Lazy procedural implementation for the Mutable Pair ADT)**.**

The lazy procedural implementation separates the object, i.e., the mutable pair, from its operations. The application of an operation to an object (i.e., "visit"), the object handles the operation its argument **values** (i.e., "accept"), but it cannot handle it its full state. In order to pass over the local state itself to the mutators, the local state values must be wrapped with boxes, i.e., have an identity that can be passed over to the mutators. Otherwise, the mutators are unable to change the local state of the object.

```
; Type: [T1*T2 -> [[Box(T1)*Box(T2)->T3] -> T3]]
(define lazy-mpair
  (lambda (x y)
    (let ((x (box x))             ;the local x and y are boxed valued
          (y (box y)))
      (lambda (sel) (sel x y)))  ;the pair object passes boxes to the operations
    ))
```

The operations have boxed value parameters. The getters unbox the relevant box, while the setters set the relevant box:

```
; Type: [[[Box(T1)*Box(T2)->T3] -> T3] -> T1]
;       [[[Box(T1)*Box(T2)->T3] -> T3] -> T2]
(define lazy-mcar (lambda (z) (z (lambda (x y) (unbox x)))))
(define lazy-mcdr (lambda (z) (z (lambda (x y) (unbox y)))))

; Type: [[[Box(T1)*Box(T2)->T3] -> T3]*T1-> Void]
;       [[[Box(T1)*Box(T2)->T3] -> T3]*T1-> Void]
(define lazy-set-car!
  (lambda (z new-value)
    (z (lambda (x y) (set-box! x new-value)))))
(define lazy-set-cdr!
  (lambda (z new-value)
    (z (lambda (x y) (set-box! y new-value)))))

> (define p2 (lazy-mpair 1 2))
> (lazy-mcar p2)
1
> (lazy-mcdr p2)
2
> (lazy-set-car! p2 3)
> (lazy-mcar p2)
3
```

It is recommended to draw an environment diagram that clarifies the roll of the boxes in passing over the local state of the lazy mutable pair to the mutators, although they are defined in a different lexical scope. Also, try the above definition without boxes:

1. If the local state variables happen to have the same names as the mutators' variables – no mutation takes place.
2. If the local state variables have different names, unbound-variable errors occur. Try!

Example 8.9 (**Bad** lazy procedural implementations for the Mutable Pair ADT)**.**

```
;;; First implementation:
(define lazy-mpair
  (lambda (x y)
    (let ((x x)
          (y y))
      (lambda (sel) (sel x y)))
    ))

(define lazy-mcar (lambda (z) (z (lambda (x y) x))))
(define lazy-mcdr (lambda (z) (z (lambda (x y) y))))

(define lazy-set-car!
  (lambda (z new-value)
    (z (lambda (x y) (set! x new-value)))))
(define lazy-set-cdr!
  (lambda (z new-value)
    (z (lambda (x y) (set! y new-value)))))

> (define p2 (lazy-mpair 1 2))
> (lazy-mcar p2)
1
```

```
> (lazy-mcdr p2)
2
> (lazy-set-car! p2 3)
> (lazy-mcar p2)
1   ; No change!

;;; Second implementation:
(define lazy-mpair
  (lambda (x y)
    (let ((x-state x)
          (y-state y))
      (lambda (sel) (sel x y)))
    ))

(define lazy-mcar (lambda (z) (z (lambda (x y) x))))
(define lazy-mcdr (lambda (z) (z (lambda (x y) y))))

(define lazy-set-car!
  (lambda (z new-value)
    (z (lambda (x y) (set! x-state new-value)))))
(define lazy-set-cdr!
  (lambda (z new-value)
    (z (lambda (x y) (set! y-state new-value)))))
Error: set!: unbound identifier in module in: x-state
```

Query methods and command operations: In Object-Oriented modeling, it is recommended to distinguish in every class, the query methods that do not create side effects, from the command operations that usually are based on side effects.

8.1.3 Sameness and sharing – value and identity equality

Mutation in data structures raises problems of ***identity*** and ***sharing***. Different objects might have equal states but still have different identity. Therefore, it becomes important to distinguish between object identity to object equality.

Example 8.10 (Sameness and sharing)**.**

```
> (define x (list (box 'a) (box 'b)))
> (define y (list (car x) (box 'c) (box 'd)))
> (define z (list (box 'a) (box 'c) (box 'd)) )
> (define w y)
> x
```

```
'(#&a #&b)
> y
'(#&a #&c #&d)
> z
'(#&a #&c #&d)
> w
'(#&a #&c #&d)

(set-box! (car y) 'aa)
> x
'(#&aa #&b)
> y
'(#&aa #&c #&d)
> z
'(#&a #&c #&d)
> w
'(#&aa #&c #&d)

> (set-box! (car y) x)
> x
#0='(#&#0# #&b)
> y
'(#0=#&(#0# #&b) #&c #&d)
> z
'(#&a #&c #&d)
> w
'(#0=#&(#0# #&b) #&c #&d)
>
```

Note how changing `y` affects both `x` and `w`, although they are not defined using `y`, while it does not affect `z`. The reason is:

– `x` and `y` **share common parts**.

– `y` and `w` are **identical**.

Therefore mutations to `y` affect the seemingly unrelated variables `x` and `w`.

In imperative programming, when object mutations are allowed, there are two kinds of object equality: ***value equality*** and ***identity equality***. Scheme provides two predicates for equality: `equal?` for value equality `eq?` for identity equality:

– `equal?` – State (value) equality (overloading the Pair and List equality).

- `eq?` – Object equality (overloading the Symbol equality).

For the last example:

```
Value and identity equality:
> (eq? y z)
#f
> (eq? y w)
#t
> (equal? y z)
#t

Structure sharing:
> (eq? (car y) (car x))
#t
Before (car y) is set:
> (equal? (car z) (car x))
#t
But after the mutation:
> (set-box! (car y) 'aa)
> (equal? (car z) (car x))
#f
```

Example 8.11 (Sameness and sharing in bank accounts)**.**

```
> (define peter-acc (make-account 100))
> (define fred-acc (make-account 100))
> ((peter-acc 'withdraw) 70)
30
> ((peter-acc 'withdraw) 20)
10
> ((fred-acc 'withdraw) 20)
80
```

Compare with:

```
> (define peter-acc (make-account 100))
> (define fred-acc peter-acc)
> ((peter-acc 'withdraw) 70)
30
> ((peter-acc 'withdraw) 20)
10
> ((fred-acc 'withdraw) 20)
```

```
"Insufficient funds"
10
```

- In the first case the two bank accounts are ***distinct***.
- In the second, there is a ***single*** bank account, with two names (***aliasing***, in the programming languages terminology): The same bank account is ***shared*** by peter and fred. Changes to peter's account affect fred's account!

This example demonstrates the difference between two objects that happen to carry the ***same local state information***, to a single object that is shared by different variables. The latter is a dangerous situation that might cause hard to detect bugs (side-effect bugs). Note that as long as there is ***no mutation***, aliasing causes no problems!

Example 8.12 (List membership using identity equality)**.**

```
(define memq?
  (lambda (x list)
     (cond ((null? list) nil)
           ((eq? x (car list)) list)
           (else (memq x (cdr list))))))
WARNING: redefining built-in memq
> (define x (list 'a 'b))
> (define y (cons x x))
> (memq? x y)
((a b) a b)
> (memq? (list 'a 'b) y)
#f
```

8.2 Operational Semantics: Extending the Environment Model with Local State Management

The standard operational semantics of imperative programming is given as a ***computation***:

- A ***computation*** is a sequence of ***system-states***.
- A system-state is a pair:
  - the set of values of the variables in the current ***scope*** (including variables in its enclosing scopes);
  - a pointer to the current position in the program code.

In the scope of this course we cannot develop a new state-based operational semantics. Instead, we extend the environment model to support state-based modeling. A state in this framework is:

– The set of values of the variables in the current ***environment***;

– A pointer to the current position in the program code.

There are three additions: Additional operations for management of assignment in environments, addition of the `set!` assignment special operator and the `Box` type, and providing explicit support to the `letrec` special operator.

1. In the ***environment*** data structure:
   Addition of an operation (command) for setting the value of an already defined variable:
   `set-binding!:  [<Variable>*Scheme-type*Env -> Void]`
   The semantics of *set-binding!*$(x, val, \langle f_1, f_2, ..., f_n \rangle)$ is defined as follows:

   ```
   if $E(x)$ is defined
     then for the frame $f_i$, such that:
        $f_i$(x) is defined while $f_j$(x) is undefined for $1 \le j < i$,
               set $f_i$(x) = val
     else Error
   ```

2. Addition of the `set!` special operator and the `Box` type:
   `env-eval[(set!  x e),env] = set-binding!(x,env-eval[e,env],env)`
   Addition of the primitive procedures of the `Box` type.

3. Evaluation of a `letrec` form, as described below.

Evaluation of `letrec`: In functional programming (substitution or environment model), recursion is computed by rewriting the code (for example, by adding the function closure as a parameter, or introducing a fix-point parameter). For that reason, the operational semantics described in Chapter **??** does not handle recursion in inner scopes. For the global scope we explicitly added the global-environment mapping, which is defined by application of `define` forms.

For recursive global procedures, the `define` special operator extends the global-environment mapping with a binding of a variable `f` and a closure that carries the global-environment (which is defined on `f`). Therefore, occurrences of `f` in the body of this closure are bound by the declaration of `f` in the global-environment mapping. But, for locally defined recursive procedures we had no similar provision.
Consider:

```
1. (define f (lambda ....))
2. (let ((f (lambda (...) (... (f...)...))))
3.     (f ...)
```

Mark the declaration on line (1) as f_1, and the declaration on line (2) as f_2. Then, the binding of a `let` form is:

```
1. (define $f_1$ (lambda ....))
2. (let (($f_2$ (lambda (...) (... ($f_1$...)...))))
3.     ($f_2$ ...)
```

while the intended semantics of recursion requires:

```
1. (define $f_1$ (lambda ....))
2. (let (($f_2$ (lambda (...) (... ($f_2$...)...))))
3.     ($f_2$ ...)
```

In order to enable local recursive procedures, we introduced the `letrec` special operator, whose semantics cannot be explained, within functional programming, without rewriting the code of the recursive procedure. Its effect is to define the local closures within the scope of the local declarations.

The introduction of local state enables a simple definition for `letrec` semantics, within the environment model.

Example 8.13. *The `letrec` form in*

```
(define fact
  (lambda (n)
    (letrec ((iter (lambda (c p)
                       (if (= c 0)
                           p
                           (iter (- c 1) (* c p))))))
      (iter n 1))
    ))
```

is defined as equivalent to:

```
(define fact
  (lambda (n)
    (let ((iter 'u))
      (set! iter (lambda (c p)
                   (if (= c 0)
                       p
                       (iter (- c 1) (* c p)))))
      (iter n 1))
    ))
```

That is, the `letrec` special operator is defined as a derived operator. The semantics of a `letrec` form

```
(letrec ((f1 lambda-exp1) ...(fn lambda-expn)) e1 ... em)
```

is defined as that of the `let` form:

```
 (let ((f1 'unassigned) ... (fn 'unassigned))
    (set! f1 lambda-exp1)
        ...
    (set! fn lambda-expn)
    e1 ... em))
```

Static (lexical) observation: The lambda expressions that create the closures for the local procedures reside within the `let` scope, and therefore, occurrences of the `let` variables in these expressions are bound by the `let` declarations:

```
(define f1 (lambda ....))
(let ((f2 'unassigned))
  (set! f2 (lambda (...) (... (f2...)...)))
    (f2 ...))
```

Dynamic (evaluation) observation: The created closures carry the environment that corresponds to the external `let` scope, in which the local variable `f2` is initialized to a Box value.

```
env-eval[(letrec ((f1 lambda-exp1) ...(fn lambda-expn)) e1 ... em), env] =
  env-eval[(let ((f1 'unassigned) ... (fn 'unassigned))
             (set! f1 lambda-exp1)
              ...
             (set! fn lambda-expn)
             e1 ... em),
           env ]
```

That is, `letrec` is defined as a ***derived operator***. It is translated into a `let` form, in which the internal function names are not initialized. The function definitions are evaluated within the let body, i.e., with respect to an environment that includes all the `letrec` declared variables.
Draw an environment diagram!

8.3 Extending the Environment Model Interpreter and Compiler

8.3.1 Imperative evaluator

The changes are:

1. **Adding support for assignment in the environment data structure:** Add support for changing the value of a variable in an environment, and add the procedures of the `Box` type to the external variable `the-global-environment`.

   The modification is in the data structures module `env-ds.rkt`. Recall that an environment is implemented as a list of ***boxed*** frames (the last being the global environment). Therefore, it is possible to change frames within this list, without affecting ***clients*** of the environment

   Variable setting in an environment is implemented by the mutating operation `set-binding-in-env` which searches for the first frame in which the given variable is defined, and changes the binding of this variable to the given value.

```
; Environment mutator: ADT type is [Env*Var -> Void]
; Purpose: If the environment is defined on the given variable, set its value
; Type: [Symbol*LIST(Box(Frame)*Var*Scheme-type) -> Void]
; Implementation: Uses the get-value-of-variable selector of Substitution,
;                 which is changed to take also a success continuation.
(define set-binding-in-env!
  (lambda (env var val)
    (letrec ((defined-in-env  ;ADT type is [Var*Env -> Box(Frame) union {empty}]
               (lambda (var env)
                 (if (empty-env? env)
                     (error 'set! "variable not found: ~s\n  env = ~s" var env)
                     (get-value-of-variable
                       (first-frame env)
                       var
                       (lambda (val) (first-boxed-frame env))
                       (lambda () (defined-in-env var (enclosing-env env))))))
               ))
      (let* ((boxed-frame (defined-in-env var env))
             (frame (unbox boxed-frame)))
        (set-box! boxed-frame (change-frame (make-binding var val) frame))))
    ))
```

   This operation calls the new `change-frame` constructor, which creates a new frame, in which the value of the variable is changed to the given value. `change-frame` calls a new substitution mutator `change-sub`, for creating a new substitution with a modified value for the given variable. In practice, the value is not changed, but the new binding is added at the head of the new substitution, so that the lookup selector returns the new value.

2. **Modifying `env-eval` to support the `set!` special operator:** In the core module `interpreter-core.rkt`: Evaluation of assignments calls the new environment mutator:

```
(define eval-assignment
  (lambda (exp env)
    (set-binding-in-env! env
                         (assignment-variable exp)
                         (env-eval (assignment-value exp) env))
    'ok))
```

3. **Adding `letrec` as a derived expression in the ASP:**

```
(define letrec->let
  (lambda (exp)
    (letrec ((make-body (lambda (vars vals)
                          (if (null? vars)
                              (letrec-body exp)
                              (make-sequence (make-assignment (car vars)
                                                              (car vals))
                                             (make-body (cdr vars)
                                                        (cdr vals))))))
             (make-bindings (lambda (vars)
                              (map (lambda (var) (list var 'unassigned))
                                   vars)))
             )
      (let* ((vars (letrec-variables exp))
             (vals (letrec-initial-values exp))
             )
        (make-let (make-bindings vars) (make-body vars vals))))
    ))
```

8.3.2 Imperative compiler

The compiler uses the same data structure and ASP packages. The only modification is to added syntax analysis and translation of assignment expressions:

```
(define analyze-assignment
  (lambda (exp)
    (let ((var (assignment-variable exp))
          (val (analyze (assignment-value exp))))
      (lambda (env)
        (set-binding-in-env! env var (val env))
        'ok))))
```

8.4 Type checking/Inference with assignments:

Typing `begin` expressions: In an imperative programming paradigm the special operator `begin` gains importance. Its typing rule:

```
Typing rule begin:
For every type environment _Tenv,
          expressions _e1, ..., _en, and
          type expressions _S1, ..., _Sn:
    If     _Tenv |- _e1:_S1, ..., _Tenv |- _en:_Sn
    Then   _Tenv |- (begin _e1 ... _en):_Sn
```

Typing `set!` expressions: An assignment `(set!  x e)` is ***well typed*** if `e` is well typed. We can further require that the type of `e` is the type of the variable `x`, although this requirement is not enforced by Scheme.

The type of an assignment expression is always `Void`, but this is conditioned by being well-typed. We adopt the more restricting version, that enforces a common type for the variable and the expression. The rule requires Existence of a type for the assigned variable, and a common type for the variable and the assignment expression. The conclusion infers the `Void` type of assignment expressions. It is necessary since assignment expressions can be components of other expressions.

```
Typing rule set!:
    For every: type environment _Tenv,
               expression _e, variable _x, and
               type expression _S:
    If     _Tenv |- _e:_S,
           _Tenv |- _x:_S,
    Then   _Tenv |- (set! _x _e):Void
```

For typing using the type-constraints method, we need to derive type equations from the inference rule. We see that the rule enforces two constraints: A common type for the variable and the assignment expression, and a `Void` type of assignment expressions.

Type equations for an assignment expression `(set! _x _e)`:
$T_{\_x}$ = $T_{\_e}$
$T_{\texttt{(set! \_x \_e)}}$ = Void

Note that the equations do not check whether the assigned variable is previously defined.
Comparison between typing of definitions and of assignments: A definition expressions cannot be nested, and therefore cannot block typing of expressions that do not include occurrences of the defined variable. Therefore, the `Define` typing rule does not conclude only a type for the defined variable, and not for the `define` expression. In contrast, assignment expressions can be nested, and typing of an enclosing expression depends on typing of the nested assignment expression.

Example 8.14. *Derive a type for:*

```
(lambda (balance)
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
        'Insufficient- funds)))
```

Typing statements for the leaves `balance, amount, -` are obtained by instantiating the ***Variable, Symbol*** and ***Primitive procedure*** axioms:

```
1. {balance:T1} |- balance:T1
2. {amount:T2} |- balance:T2
3. {} |-  -:[Number*Number -> Number]
4. {} |- 'Insufficient-funds:Symbol
5. {} |-  >=:[Number*Number -> Number]
```

Applying typing rule ***Application*** to typing statements 1,2,3, with type substitution `{T1=T2=Number}`:

```
6. {balance:Number, amount:Number} |- (- balance amount):Number
```

Applying typing rule ***set!*** to typing statement 6:

```
7. {balance:Number, amount:Number} |-
                      (set! balance (- balance amount)):unit
```

Applying typing rule ***begin*** to typing statements 1,7, with type substitution `{T1=Number}`:

```
8. {balance:Number, amount:Number} |-
              (begin (set! balance (- balance amount))
                      balance):Number
```

Applying typing rule ***Application*** to typing statements 1,2,7, with type substitution `{T1=T2=Number}`:

```
9. {balance:Number, amount:Number} |- (>= balance amount):Number
```

Applying typing rule ***If*** to typing statements 6,7,9:

```
10. {balance:Number, amount:Number} |-
                 (if (>= balance amount)
                     (begin (set! balance (- balance amount))
                            balance)
                    'Insufficient- funds):Number union Symbol
```

Applying typing rule ***Procedure*** to typing statement 10:

```
11.  {balance:Number} |-
           (lambda (amount)
              (if (>= balance amount)
                  (begin (set! balance (- balance amount))
                         balance)
                 'Insufficient- funds)):[Number -> Number union Symbol]
```

Applying typing rule ***Procedure*** to typing statement 11:

```
12.  {} |-
        (lambda (balance)
            (lambda (amount)
               (if (>= balance amount)
                   (begin (set! balance (- balance amount))
                          balance)
                  'Insufficient- funds))) :
                       [Number -> [Number -> Number union Symbol]]
```

Type checking/inference with the Box type: The Box operations are Racket primitives. Therefore, the ***Primitive procedure*** typing axiom has to be extended for the new primitives.

```
 For every type environment Tenv and type expression S:
        Tenv |- box:[S -> Box(S)]
        Tenv |- unbox:[Box(S) -> S]
        Tenv |- set-box!:[Box(S)*S -> Void]
        Tenv |- box?:[S -> Boolean]
        Tenv |- equal?:[Box(S)*Box(S) -> Boolean]
        Tenv |- eq?:[Box(S)*Box(S) -> Boolean]
```

8.5 Costs of Introducing Mutation

Languages that are characterized as ***functional languages*** (like Scheme and ML) have operational semantics that is based on successive procedure applications (reductions). The main syntactic components are ***expression*** and ***function***. The main semantic notion is ***reduction***. Introducing mutation to such languages sharpens the fundamental difference between ***expressions*** that compute values, to ***side effect*** language constructs that are evaluated for the sake of their side effects and not for their returned values.

Languages that are characterized as ***imperative languages*** (like Pascal, C, C++, Java) have operational semantics that is based on successive evaluation of program ***statements*** (or ***commands***). The main syntactic component is the ***statement***, whose evaluation carries a side effect. The distinction between value returning expressions to side effect commands is usually blurred. The main semantic element is the ***state***, which is given by the variable values.

Function call in functional languages is, usually characterized as ***call-by-value***: The argument expressions are evaluated, and the resulting values are bound to the corresponding function parameters. Function call in imperative languages is, often, characterized as ***call-by-reference***: A function receives implicit references to arguments, rather than copy of their values, and can modify the arguments held by its caller.

8.5.1 Losing Referential Transparency

Languages where ***equals can be substituted for equals*** are called ***referentially transparent***. The static analysis support for such language is much simpler and reliable than for languages that are not referentially transparent.

- In referentially transparent languages programs can be compared and simplified by substituting complex expressions by simpler equal expressions.
- In non referentially transparent languages reasoning about programs is hard: Sameness of objects means sameness of changes, which is hard to define and hard to observe without actually running the program on sample data.

Consider:

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
> (W1 30)
70
> (W1 70)
0
> (W2 70)
30
```

Are `W1` and `W2` the same?
For `W1` and `W2`, we see that they have different effects, although they were created by the same expression. Surely they cannot be substituted one for the other in other expressions.

Value classes in Object-Oriented modeling: Classes of objects that should not have local states are termed ***value classes*** and their objects are ***value objects***. Value classes should provide an `equal` predicate that equates objects with equal state. Sometimes their constructor is restricted not to duplicate objects with equal state.

8.5.2 Dependency on Mutation Ordering

Mutation introduces sensitivity to order of operations, which might cause difficulties, especially in concurrent environment. Without side effects, sequences of expressions are meaningless.
Compare the iterative `factorial`:

```
(define factorial
   (lambda (n)
     (letrec ((iter (lambda(product counter)
                         (if (> counter n)
                             product
                             (iter (* counter product) (+ counter 1))))
     ))
(iter 1 1))))
```

with an alternative imperative style:

```
(define factorial
  (lambda (n)
    (let ((product (box 1))
          (counter (box 1)))
      (letrec ((iter (lambda()
                         (if (> (unbox counter) n)
                             (unbox product)
                             (begin
                               (set-box! product
                                         (* (unbox counter) (unbox product)))
                               (set-box! counter (+ (unbox counter) 1))
                               (iter))))
               ))
         (iter)))
    ))
```

Chapter 9

Conclusion: Techniques, Principles and Language Comparison

What did we learn in this course?

Subjects:

1. **Elements of programming languages:**
   (a) **Language building blocks:** Atomic elements; composition means; abstraction means.
   (b) **Language syntax:** Concrete syntax, abstract syntax.
2. **Meaning of a programming language:**
   (a) **Operational semantics:** Applicative (eager), normal (lazy); lexical scoping, dynamic scoping.
   (b) **Types:** Type inference, type checking; dynamic typing, static typing techniques; polymorphic types; type specification language.
3. **Using programming languages for problem solving:**
   (a) **Procedure abstraction:** As a parameter/argument, returned value, data structure component.
   (b) **Abstract data types (ADT):** Abstraction – separation between application to implementation; invariants, operation contracts, ADT embedding, Design by Contract; hierarchical data types; lazy lists.
4. **Meta-programming tools** Interpreter, compiler; static (compile) time evaluation, run-time evaluation.

5. **Programming styles**:
   (a) Iteration vs. recursion.
   (b) Continuation Passing Style.
   (c) Lazy lists.
6. **Programming considerations and conventions:** Contracts, Tests, Asymptotic complexity, Compile-time vs. Run-time.
7. **Imperative programming:** State, mutation and change in programming.

Techniques:

1. **Substitution operation:** Used in the operational semantics algorithms and in type inference algorithms.
2. **Renaming operation:** Used in the operational semantics algorithms and in type inference algorithms.
3. **Pattern matching, unification operations:** Used in the operational semantics algorithms and in type inference algorithms.
4. **Lazy/eager approaches:** Used in various contexts – semantics, ADT implementation, partial evaluation.
5. **Delayed evaluation**.
6. **Partial evaluation:** Currying.

Programming paradigms:

1. ***Functional programming*** (***Scheme, Lisp, ML***): Its origins are in the ***lambda calculus***.
2. ***Logic programming*** (***Prolog):*** Its origins are in mathematical logic.
3. ***Imperative programming*** (***ALGOL-60, Pascal, C):*** Its origins are in the Von-Neumann computer architecture.

Table 9.1 presents a comparison of the languages discussed in this course, with the addition of the Java language. The comparison covers a variety of parameters, theoretical and practical.

Table 9.1: Language comparison

	Scheme	**ML**	**LP**	**Prolog**	**Java**
Syntax	simple	rich	simple	rich	rich
Uniform Syntax	yes	no	yes	yes	no
Main syntax component	expression	expression	term, formula	term, formula	statement
Typing	dynamic	static	–	dynamic	static
Type inference	no	yes	–	–	no
Polymorphism	yes	yes	–	–	yes
Runtime errors	yes	no	no	yes	yes
Computation direction	yes	yes	no	yes	yes
Runtime procedures	closure	closure	no	term	anonymous classe
Formal operational semantics	yes	yes	yes	no	no
Procedure application mechanism	evaluation	evaluation, pattern matching	unification	unification	evaluation
Programming paradigm	functional	functional	logic	logic	object-oriented
State support	yes	yes	no	no	yes
Expressive power	TM	TM	RLP: Decidable, FLP: TM	TM	TM
User defined types	no	yes	–	–	yes
Scope definition	`lambda`	`fn, =>`	rule	rule	class, statement
Nested scopes	yes	yes	no	no	yes
Delay mechanism	`lambda`	`fn, =>`	–	–	anonymous class

Notes:

1. **Uniform syntax:** A major syntax component (like expressions or statements) and a printed form of a major data structure share a common syntax. In Prolog, terms and atomic formulas share a common syntax.
2. **Name space support:** Support for nested, co-existing scopes.
3. **Runtime procedures:** Support for procedures that are created at run time. In Prolog, terms can be turned into atomic formulas (queries) during computation.

References

[1] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs, 2nd edition.* The MIT Press, 1996.

[2] M. Felleisen, R.B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs.* The MIT Press, 2001.

[3] D.P. Friedman and M. Wand. *Essentials of Programming Languages, 3nd edition.* The MIT Press, 2008.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-wesley Reading, MA, 1995.

[5] S. Gilmore. Programming in Standard ML'97: A Tutorial Introduction. *Laboratory for Foundations of Computer Science, The University of Edinburgh*, 1997.

[6] R. Harper. *Programming in Standard ML.* Carnegie Mellon University, 2011.

[7] R. Kowalski. Predicate Logic as a Programming Language, Information Processing 74. In *Proceedings of the IFIP Congress*, pages 569–574, 1974.

[8] S. Krishnamurthi. *Programming Languages: Application and Interpretation.* Version 26, 2007.

[9] OMG. The UML 2.0 Superstructure Specification. Specification Version 2, Object Management Group, 2007.

[10] L.C. Paulson. *ML for the Working Programmer, 2nd edition.* Cambridge University Press, 1996.

[11] L. Sterling and E.Y. Shapiro. *The Art of Prolog: Advanced Programming Techniques, 2nd edition.* The MIT Press, 1994.

[12] Wikipedia. UML. `http://en.wikipedia.org/wiki/Unified_Modeling_Language`, 2011.