LECTURE NOTES

COMP 3007
PROGRAMMING PARADIGMS

Weixan Li edited by Tony White

Published 2001, Edited Winter 2010

TABLE OF CONTENTS

Chapter 1. Programming Paradigms 1
§ 1. Programming Languages and Programs 1
§ 2. Criteria for Good Programming Languages 4
§ 3. Language Paradigms 7
§ 4. Studying Programming Languages 10

Chapter 2. Introduction to Scheme 11
§ 1. Building Blocks 11
§ 2. Evaluation of an Expression - The Substitution Model 22
§ 3. Conditional Expression 24
§ 4. Our First Scheme Program - Find the square root of a real number *x* using Newton's method 26
§ 5. Recursion and Iteration 28
§ 6. Character and String Manipulation 33
§ 7. Simple Input / Output 36

Chapter 3. Lists in Scheme 39
§ 1. Pairs 39
§ 2. Lists 41
§ 3. Manipulating Lists 46
§ 4. Using Lists to Implement Abstract Data Types 51
(1) Rational Numbers as Pairs 51
(2) Binary Search Trees as Lists 54
(3) Symbolic Algebra 57

Chapter 4. Mutable Data, Objects, and Streams 63
§ 1. Assignment 63
§ 2. Using Assignments to Write Explicit Iterative Procedure 66
§ 3. The Environment Model 67
§ 4. Using Assignment with Data Structure to Implement Objects 75
§ 5. Streams 77

Chapter 5. A Metacircular Evaluator 80
§ 1. Metacircular Evaluator 80
§ 2. Representing the Environment 81
§ 3. Evaluate Expressions 84
§ 4. Running the Evaluator 91

Chapter 6. Introduction to Prolog 94
§ 1. Introduction 94
§ 2. Basic Concepts and Syntax Rules 97
§ 3. Some Built-in Facilities 99
(1) Arithmetic 99
(2) Relational Operators 99

(3) Simple I/O 100
(4) Declarative and procedural meanings of Prolog programs 101
§ 4. The Goals 102
§ 5. Scanning and Backtracking 104
§ 6. Examples 105

Chapter 7. Structures and Lists 110
§ 1. Introduction to Structures 110
§ 2. Binary Trees as Structures 113
§ 3. Lists 115
§ 4. List Processing 117
§ 5. String Manipulation 123

Chapter 8. Controlling Backtracking 124
§ 1. Predicate Cut 124
§ 2. The Cut/Fail Combination and Predicate Not 128
§ 3. Predicate Repeat 130

Chapter 9. Input/Output, Built-in Predicates and Operators 132
§ 1. Reading and Writing Terms 132
§ 2. Communicating with Files 134
§ 3. Characters and Strings I / O 136
§ 4. More Built-in Predicates 138
(1) Testing the Type of Terms 138
(2) Manipulating Database 138
(3) Manipulating Terms 139
(4) Examples 141
§ 5. Operators 143
(1) Precedence and Associativity of Operators 143
(2) Defining Operators 144
(3) More Arithmetic and Logical Operators 144

Chapter 10. Problem Solving by Prolog 147
§ 1. Sorting Algorithms 147
§ 2. The Hanoi Tower Problem 149
§ 3. The Eight-Queen Problem 150
§ 4. Representing Grammar Rules by Prolog 153
(1) Languages and Grammars 153
(2) Representing Grammar Rules by Clauses 154
(3) Adding Parameters into Nonterminals 157
(4) Showing Parsing Trees 157

CHAPTER 1. PROGRAMMING PARADIGMS

§ 1. PROGRAMMING LANGUAGES AND PROGRAMS

A programming language *is a language intended to be used by people to write a* program *to express a* process *by which a computer can solve a problem.*

A program is used to solve a *problem*. A *problem* is a general request of a result (a value, an action, etc.), called the *output*, to be produced:

- from a set of known facts (values, actions, etc.) (the *input*)
- according to the *specification* of the problem.

The *specification* of a problem consists of:

- a detailed description of what the problem is (the *application domain*), and
- a set of given rules to relate the output to the input (including un-expected input).

The specification specifies WHAT to output, and the rules to specify HOW to output.

A problem is to be solved by a *process*. This process is to be given by a sequence of step-by-step instructions, which, when followed, will generate the output from the input. This sequence of instruction is called an *algorithm*.

An algorithm is expressed by a *language*. This language must be easy to express, easy to understand (by people) and un-ambiguous. A language can be of general purpose, or it is designed to express a certain class of algorithms. A language consists of a specified set of *symbols*, the *syntax* (what is a legitimate combination of symbols), and the *semantics* (what is the meaning of a legitimate combination of symbols).

A *programming language* is a language that can *also* be made understood by a computer. As a language, a programming language must be, *first of all*, easy to write and understand by people.

An algorithm written in a programming language is a *program*. If a program is not

written directly in the language that a computer can understand (*machine language*), it has to be translated. There are two ways to translate a program to the machine language: *compilation* and *interpretation*.

Compilation: Translate the entire program (the *source code*) into equivalence form in a lower-level language (the *object code*) and execute the object code (maybe in an interpretated manner). A program used to compile a program is a *compiler*.

Interpretation: Break the entire program into fragments, and translate and execute these fragments one by one. A program used to interpret a program is an *interpreter* (also called an *evaluator*).

More often, the execution of a program using an interpreter is in an *interactive mode*. That is, the program halts from time to time waiting for the next input from the user.

§ 2. Criteria for Good Programming Languages

- Simplicity:

  Easy to learn, easy to use, easy to understand and easy to maintain (some redundancy may be necessary to detect possible errors).

- Abstraction:

  Capability of using data abstractions and procedure abstractions to facilitate modularity in program design.

  A *data abstraction* is a way to treat a set of data items as a single data entry, called an *aggregated data object*, and a *procedure abstraction* is to treat a sequence of instructions as a separated algorithm, called a *subroutine* or a *procedure*.

- Expressiveness:

  With all the facility to express data objects and algorithms in the area that this language is designed for.

- Orthogonality:

  The degree to which different concepts can be combined with each other in a consistent manner.

- Naturalness:

  Represent the design of the algorithm naturally.

- Reliability and verifiability:

  Easy to find errors, handle exceptions, and test.

- Efficiency:

  Use as little time and space as possible when a program is executed.

- Portability:

The machine-independence of programs.

A good programming language should also have a good development environment, editor, compiler, debugger, library packages, etc.

§ 3. LANGUAGE PARADIGMS

(1) *Imperative* (or *procedural*) languages - specify the sequence of the commands to be executed: e.g., FORTRAN, Pascal, C/C++, Ada, etc.

In a procedural language, a program consists of a sequence of statements to be executed in a specified order. The execution of the program usually changes the value stored in memory locations.

(2) *Logical* (or *declarative*) languages - express the problem in a logical format and the problem is solved by a built-in mechanism of the language: e.g., Prolog.

A program written in a logical language specifies a set of facts and rules. The rules relate the truth of facts and the action to be taken. When a program is executed, it checks the truth of the facts by the data provided by the user in a predefined order, and takes the action according to the rules.

(3) *Functional* languages - describe the operations used to solve the problem: e.g., LISP, Scheme.

A functional language organizes a program into functions. The process of problem solving is modeled by evaluating a sequence of functions.

A function consists of

- a *parameter list*: the values to be used to evaluate the function
- a *domain*: the values that the parameters may take
- a *range*: the set of possible output values
- a *definition*: a rule to specify how to evaluate a function
- a *name* (optional).

When a function is evaluated, a set of values, called *arguments*, is provided. An argument may also be the output value of another function.

(4) *Object-oriented* languages - represent the problem as a set of interacting objects or object classes: e.g., Smalltalk, Java.

An object-oriented language combines the features of functional and procedural languages. A *class* specifies the format of *objects*, which are instances of the class. In a class, a collection of data, called *state variables*, associated with objects in this class, and a limited set of *methods* used to manipulate objects in this class are *encapsulated* to

achieve modularity of the program. Complex objects can be built from simpler ones (called *composition*). Classes are related to each other by *inheritance*. Special classes may be *derived* from a more general class. An object in a derived class may be used as an object in the parent class, but it keeps its own identity (*polymorphism*).

There are also hybrid languages, such as C++, which is a procedural language enhanced to support object-oriented programming.

§ 4. STUDYING PROGRAMMING LANGUAGES

The advantage of studying the principle and paradigms of programming languages are:

- To improve your ability in developing effective algorithms
- To improve your ability in using existing languages
- To increase your vocabulary of useful programming constructs
- To allow better choice of programming languages
- To have a better understanding of the significance of implementation
- To make it easier to learn a new language
- To improve your ability in designing a new language

CHAPTER 2. INTRODUCTION TO SCHEME

§ 1. BUILDING BLOCKS

Scheme is a dialect of a programming language called LISP (LISt Processing). It is a functional-oriented language used mainly in Artificial Intelligence. Scheme usually works interactively with an interpreter. The interpreter reads an expression from the command line at the *Scheme prompt*, evaluates the value of the expression, and prints the result as an output. This is called the *read-eval-print loop*, or *repl* for short.

A Scheme program is made up by *form*s (i.e., formulas). A form consists of a number of components included between parentheses. The components of a form may be an *identifier*, a constant data item (such as a number, a string, a character, etc.), or another form.

Simple examples of Scheme programs:

(i)

```
> ( + 1 2 )
3
```

The first line is a form that the user enters at the prompt, which means to evaluate an expression 1 + 2. The second line is what Scheme prints, which is the value of this expression. Note that, Scheme used *prefix expressions*, i.e., the operator is placed before the operands. An expression is entered as a form between a pair of parentheses.

You need the space (one or more) to separate the operator and the operands, but the space between the parentheses and the identifiers is not necessary.

(ii)

```
> ( define size 3 )
> size
3
```

The first 2 lines are entered by the user at the prompt. The first line is a form that *define*s an *identifier* called *size* to represent the value 3, the second line asks Scheme for the

value of size (without parentheses!), and the third line is the response of the Scheme interpreter.

(iii)

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

The first line is a form that defines a *function* (square x), where *square* is an identifier, representing the name of the function, and x is the *parameter*, which is another identifier representing the value to be used by this function. (* x x) is the *definition* of the function (square x) and is also referred to as the *body* of the function. In other words, (square x) is defined to be the same as (* x x). The second line is another form that asks Scheme to evaluate (square x) with x set to the value 3. We also say that the function *square* is *called*, and (square 3) is a *function call*. The value 3 is an *argument*, which is *passed* to function *square*. When an argument is passed to a function, the formal parameter of this function takes the **value** of the argument. This is not significant if the value of a variable does not change. However, it will become significant later when we introduces assignment that changes the value of a variable.

If a function has more than one parameter, the arguments of a function call have to match the formal parameter in the definition of the function in the same order. The third line is the response of Scheme. You may write the first line as a *program* by an editor, and save it in a *file*, and then *load* it into the evaluator. When you type (square 3) at the prompt, Scheme searches the loaded program to find the definition, and evaluates the expression.

Identifiers

Identifiers in Scheme can use digits, letters, and any of the following characters:

! ? . + - * / > = < : $ % ^ & _ ~

Avoid using a character that can start a number to be the first character of an identifier. Identifiers are **case-insensitive**.

Comments

A comment in Scheme starts with a semicolon ';' up to the end of a line.

Quotes

An identifier represents a data value or a procedure. If we want to use an identifier just as a *symbol*, then use the built-in operator *quote*. For instance,

```
( define size 3 )
```

```
> size
3
> ( quote size )
size
> ( + 3 4 )
7
> ( quote ( + 3 4 ) )
( + 3 4 )
```

We may use a single quote mark to replace the operator quote: 'size is the same as (quote size), '(+ 3 4) is the same as (quote (+ 3 4)).

Scheme provides a predicate eqv? to test the equality of data items. Scheme uses #t to mean true, and #f to mean false.

```
> (eqv? 'x 'x)
#t
> (eqv? 'x 'y)
#f
> ( define x 2 )
> ( define y 2)
> ( eqv? x y )
#t
```

Constant values

A *constant* may be a symbol, a number, a character, a Boolean value (#t or #f), or a string. (There are other types of constants).

A symbol is an identifier used to represent a variable, a procedure, etc.

A number may be an integer or a real (some interpreters do not distinguish between integers and real numbers).

A Boolean can take the value *true* or *false*. The value true is represented by #t, and false is represented by #f.

A string is specified between double quotes, such as "a string", and a character is specified by #\, such as #\x, #\space.

Forms

There are two kinds of forms: *definitions* and *expressions*.

Definitions

A definition is used to define an identifier. The first component of a definition is the keyword *define*. The second component is the identifier to be defined, and third component is the definition of the identifier.

```
> ( define size 2 )
> size
2
> ( define pi 3.14159 )
> pi
3.14159
```

An identifier representing a value is called a *variable*. However, in Scheme, the value of a "variable" is rarely changed.

We may also define an identifier by another identifier.

```
> ( define x 2 )
> ( define y x )
> y
2
```

In this case, y is a new identifier that takes the **value** of x. After this definition, x and y are not related to each other. *This is a very important distinction*.

Expressions

An expression represents a value. The first component of an expression is an *operator*, and the other components are *operands*. The operator represents a *procedure*, which is a function that calculates a value from the operands, called the *arguments* (or *actual parameters*) of the procedure. *Primitive procedures* are procedures defined by the language. Arithmetic operations are primitive procedures. Scheme uses the ordinary operators +, -, *, /, for addition, subtraction, multiplication, and division. (Note that, different implementation treats integer division differently. The expression (/ 3 2) may be evaluated as a fraction 3/2, or as a real number 1.5. *Compound procedures* are defined by the user using *lambda forms* to be introduced later.

When we enter an expression at the prompt of a Scheme interpreter, it responds with the value of the expression. The evaluation of an expression generates a *process*.

Examples of expressions

```
> ( +  2  3 )
5
> ( *  4  5  6 )
120
; operator remainder returns the remainder when
```

```
; the first operand divided by the second operand
> ( remainder 11  3 )
2
>   ( *  ( +  1  2 )
         ( -  3  5 )
    )
-6
```

Lambda Expressions

An expression may also return a procedure. The operator used to create a procedure is called *lambda*. The operator lambda has two operands, the first operand is a list of *formal parameters*, and the second is an expression to specify how the function is to be evaluated.

Example. The following expression returns a procedure that adds the square of first parameter with the square of the second parameter:

```
( lambda    ( x y )
            ( + ( * x x ) ( * y y ) )
)
```

We may use this expression to be the operator in another expression:

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

We may also use the lambda expression to define an identifier to represent a procedure (a *named procedure*):

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

This definition has a simplified (semantically equivalent) form:

```
> ( define ( sum-of-squares x y )  ( + ( * x x ) ( * y y ) ) )
```

In this definition, the lambda operator is used implicitly.

The following expression returns a procedure when a parameter y is provided:

```
( lambda ( y ) ( lambda ( x ) ( + x ( * 2 y ) ) ) )
```

```
>  ( ( ( lambda ( y ) ( lambda ( x ) ( + x ( * 2 y ) ) ) ) 3 ) 5 )
11
```

Using an identifier, we may define:

```
( define add-2y  ( lambda ( y ) ( lambda ( x ) ( + x ( * 2 y ) ) )
```

This is the same as:

```
( define ( add-2y y ) ( lambda ( x ) ( + x ( * 2 y ) ) ) )
```

The expression that we evaluated is:

```
( ( add-2y 3 ) 5 )
```

Then (add-2y 3) is a procedure. This procedure takes a parameter x, and the definition of this procedure is:

```
( lambda ( x ) ( + x ( * 2 y ) ) )
```

where y is 3. When x is 5, 5 + 2 * 3 = 11.

A formal parameter of a lambda expression may represent a procedure. Then the corresponding argument is also a procedure.

```
> ( ( lambda ( op ) ( op 2 3 ) ) + )
5
> ( ( lambda ( op ) ( op 2 3 ) ) * )
6
```

In this example, we used a procedure defined by:

```
( lambda ( op ) ( op 2 3 ) )
```

This procedure accepts one parameter, represented by *op*, and the procedure is to evaluate the expression (op 2 3). When this parameter takes the value +, it adds 2 and 3 to have value 5. When the operator is *, the result is 6.

Rewrite this program in two steps:

```
> ( define op-2-3 ( lambda ( op ) ( op 2 3 ) ) )
> ( op-2-3 + )
```

```
5
> ( op-2-3 * )
6
> ( op-2-3 sum-of-squares)
13
```

Using the implicit definition, we may write:

```
> ( define ( op-2-3 op ) ( op 2 3 ) )
```

Now it is clearer that (op-2-3 +) is the same as (+ 2 3).

In the following example, the procedure parameter takes the value of a lambda expression:

```
> ( ( lambda ( f  x ) ( f  x ) )
          ( lambda ( x ) ( * 3 x ) ) 4
  )
12
```

The operator is defined by (lambda (f x) (f x)). It takes two parameters: a procedure f and a value x. The argument corresponding to f is provided by another lambda expression (lambda (x) (* 3 x)). Although the formal parameter is also denoted by x, this formal parameter is local to this procedure, which is not related to the formal parameter x in the first procedure. The second parameter x of the first procedure takes the value 4. Therefore, the expression is the same as:

```
(  ( lambda ( x ) ( * 3 x ) ) 4 )
```

which has the value 12.

At the beginning of the body of a procedure definition, you may have nested definitions. A definition inside another definition defines a local variable or procedure, which is valid only in the outer definition:

```
> ( define ( add-x y ) ( define x 3 ) ( + x y ) )
> ( add-x 5 )
8
```

In explicit lambda form, this definition is:

```
> ( define add-x
    ( lambda ( y ) ( define x 3 ) ( + x y ) )
  )
```

The following example has local procedures:

```
( define     ( func p )
         ( define q ( - p 1 ) )
         ( define  ( square  x ) ( *    x    x ) )
         ( define ( sum-of-squares   x   y )
                       ( + ( square x )
                           ( square y )
                       )
         )
         ( sum-of-squares   ( + p q )
                            ( -  p q )
         )
)

> ( func 4 )
50
```

Environment

The identifiers of the formal parameters of a procedure are local to this procedure. In Scheme, the matching pairs of identifiers and their values in a part of the program is called the *environment*. Identifiers and procedures defined at the program level (i.e., not local to any other procedure) are in the *global environment*. Formal parameters of a procedure are in a part of the environment local to this procedure. In this way, different parts of an environment are nested the same way as nested procedures. If a name is defined in a procedure is in the part of the environment local to this procedure, it is *bound*; otherwise, it is *free*. If a name is free in a part of an environment, it takes the value defined in a nesting part of the environment. This is called *lexical* (or *static*) *scoping*.

In the previous example, name *p* is bound in the part of the environment local to procedure *func*, and it is free in the definition of *q* and in the definition of procedure sum-of-squares. Name *x* is local to procedure *square* and procedure *sum-of-squares*. Therefore, the name *x* in procedure *square* and in procedure *sum-of-squares* are different variables.

Let expressions

The formal parameters in a lambda expression are local to the procedure that the lambda expression defines.

The following procedure calculates $x^3 + x^2$ using a lambda expression:

```
> ( define  ( demo x )
         ( ( lambda ( a b )  ( + a b ) )
           ( * x x x ) ( * x x )
```

```
          )
  )
```

The lambda expressions define two local identifiers $a = x^3$ and $b = x^2$. This form can be re-written using a *let form*:

```
> ( define  ( demo x )
            ( let ( ( a ( * x x x ) )
                      ( b ( * x x ) )
                  )
                  ( + a b )
            )
  )
```

In general, a *let form* has the following format:

```
( let   ( ( < variable-1 >    < expression-1 > )
          ( < variable-2 >    < expression-2 > )
          …
          ( < variable-n >    < expression-n > )
        )
       ( < body > )
)
```

where < body > is an expression with arguments: variable-1, variable-2 …, variable-*n*.

This form is equivalent to:

```
( ( lambda ( < variable-1 > < variable-2 > … < variable-n > )
     ( < body > )
  )
  < expression-1 > < expression-2 > … < expression-n >
)
```

We may also use *let* to introduce a local name of a procedure.

Consider the expression:

```
> ( ( lambda ( x ) ( + x 1 ) ) 2 )
```

This expression defines a procedure by a lambda expression (lambda (x) (+ x 1)). Using the *let* form, we may write:

```
> ( let  ( ( add-1  ( lambda ( x ) ( + x 1 ) ) ) )
     ( add-1 2 )
  )
```

This is the same as:

```
> ( ( lambda ( add-1 )  ( add-1 2 ) )
    ( lambda ( x ) ( + x 1 ) )
  )
```

The first component of this expression defines a procedure by a lambda expression:

```
( lambda ( add-1 )  ( add-1 2 ) )
```

This expression takes a parameter that is also a procedure. This procedure is provided by the second component of the expression:

```
( lambda ( x ) ( + x 1 ) )
```

The use of *let* is just for convenience and is not necessary. We may always use the corresponding lambda expressions to replace a let expression. A form that is created just for convenience but is not necessary is called *syntactic sugar* in Scheme.

The *let* form may also be used to bind a value to a local variable to simplify an expression. For instance, the expression:

```
( + ( * 4 4 ) ( * 4 4 ) )
```

may be written as:

```
( let ( ( a (* 4 4) ) ) ( + a a ) )
```

so that (* 4 4) is evaluated only once.

This form is equivalent to:

```
( ( lambda ( a ) ( + a a ) ) ( * 4 4 ) )
```

The use of *let* is **similar** to the use of *define*. The previous example may also be written as the following:

```
> ( define  ( demo x )
            ( define  a  ( * x x x ) )
            ( define  b  ( * x x ) )
            ( + a b )
  )
```

However, they are **NOT** the same. In the previous example,

((define a (* 4 4)) (+ a a))

is not allowed.

Always remember that let form is just syntactic sugar for a lambda form. The following expression would give an error:

(let ((x 2) (y (+ x 1))) (* x y)).

Rewrite this in the lambda form, we have:

((lambda (x y) (* x y)) 2 (+ x 1))

where the last x is actually not in the scope of the binding of x.

If this let form is in an environment where x is bound to 4, the result is not 6 as you may expect. It will return 10 since x in the argument will take the value 4.

Another way to explain is that the binding specified by a *let* form is valid only in the body of the let form. In the expression:

(let ((x 2) (y (+ x 1))) (* x y)).

The variable x is bound to value 2 is valid only in the body (* x y), but not in the second part (y (+ x 1)).

§ 2. EVALUATION OF AN EXPRESSION - THE SUBSTITUTION MODEL

Scheme evaluates a compound expression by using the following rules:

First, all components of the expression are evaluated (**in an un-specified order**). Then apply the procedure, which is the first component of expression, to the arguments.

Examples. Suppose we want to evaluate the following expression

```
( / ( -  ( * 5 3 4 ) ( - 7 2 ) )
    ( +  5
         ( *  ( + 1 2)
              ( - 5 3 )
         )
     )
)
```

The evaluation of the expression goes in the following sequence:

```
( / ( -  60  5 )  ( + 5 ( *  3  2 ) )
( /  55  11 )
5
```

If we have a compound procedure, the body of the procedure is evaluated with each formal parameter replaced by corresponding arguments. This is the so-called *substitution model*.

Suppose we want to evaluate (func 4) in the previous example:

```
( define     ( func p )
          ( define q ( - p 1 ) )
          ( define  ( square  x ) ( *   x    x ) )
          ( define ( sum-of-squares   x   y )
                        ( + ( square x )
                             ( square y )
                        )
          )
          ( sum-of-squares   ( + p q )
                             ( -  p q )
          )
)
```

The process is as the following sequence:

```
( func 4 )
( ( sum-of-squares ( + p q ) ( - p q ) ) 4 )
( sum-of-squares ( + 4 3 ) ( - 4 3 )
( sum-of-squares  7 1 )
( + ( square 7 ) ( square 1 )
( +  ( * 7 7 ) ( * 1 1 ) )
( + 49 1 )
50
```

§ 3. CONDITIONAL EXPRESSION

An expression may have different values depending on different conditions. This is achieved using a *conditional form*. The general form of a conditional form is:

```
( cond  ( condition-1  form-1 )
        ( condition-2  form-2 )
        …
        ( condition-n  form-n )
        ( else  form-0 )
)
```

Each <condition, form> pair in a conditional form is called a *clause*. The first part of a clause is a *predicate*. A predicate is a logical expression that has a value true or false. We may use any expression as a predicate. Scheme treats #f as false, and **anything else as true**. The second part of a clause can be any form. This conditional form finds the first clause with a true condition, and evaluates the form in this clause. If none of the conditions is true, then this form evaluates the last form in the else part. The else part is optional. If the else part is missing and none of the condition is true, this form does nothing.

To compare numbers, Scheme uses the common predicates =, >, >=, <, <=.

For instance, the absolute value of a number x can be defined as follows:

```
( define     ( absolute x )
        ( cond  ( ( > x 0 ) x)
             ( ( = x 0 ) 0)
             ( ( < x 0 ) ( - x ) )
        )
)
```

Here we also used a unary operator - to negate x. (Hence, you must leave a blank between - and x). (In Scheme, there is a built-in procedure called abs that finds the absolute value of integers, but it may not work for real numbers in some implementations.)

Predicates can be combined using operators *and*, *or*, and *not*.

```
( define ( test  z )
     ( let ( ( x 1 ) ( y 3 ) )
        ( cond  ( ( and  ( >= z  x ) ( <= z  y ) )  'inside )
```

```
                ( ( or ( < z x ) ( > z y ) )  'outside )
        )
    )
)
```

```
> ( test 0 )
outside
> ( test 2 )
inside
```

Using the else part, the definition of the procedure *absolute* may also be written as follows:

```
( define    ( absolute x )
        ( cond  ( ( < x 0 ) ( - x ) )
                ( else x )
        )
)
```

If there are only two cases in a conditional form, it can be written as an *if* form. An if form has the general format:

(if (*condition*) form-1 form-2)

where *condition* has a value true or false. If the value of *condition* is true, form-1 is evaluated; otherwise, form-2 is evaluated. The procedure *absolute* can be defined using an if form as follows:

```
( define    ( absolute x )
        ( if ( < x 0 )
            ( - x )
            x
        )
)
```

§ 4. OUR FIRST SCHEME PROGRAM - FIND THE SQUARE ROOT OF A REAL NUMBER *X* USING NEWTON'S METHOD

Newton's method of finding the square root of a real number x is iterative: Start with an approximation r_0. Iteratively, let $r_{i+1} = (r_i + x / r_i) / 2$, until r_i^2 and x are close enough.

To write this program, we need to define a procedure that finds the average of a value r and the quotient x / r.

```
( define ( improve  r  x )
        ( / ( + r ( / x r ) ) 2 )
)
```

We also need a procedure that checks to see whether two consecutive approximations are close enough:

```
( define ( good-enough? p q precision )
        ( < ( absolute ( -  p q ) )  precision )
)
```

Procedure good-enough? is a user defined predicate. According to the convention in Scheme, the name of a user defined predicate ends with a '?'. Procedure *absolute* is as defined before to find the absolute value of a real number.

Now we define the procedure *sqrt-iterate*:

```
( define ( sqrt-iterate  new-r  old-r  x  precision )
        ( if ( good-enough?  new-r  old-r  precision)
            new-r
            ( sqrt-iterate ( improve new-r  x )
                new-r
                x
                precision
            )
        )
)
```

The final procedure *sqrt* can be defined with an initial approximation, and let x to be the old value:

```
( define ( sqrt x  initial  precision)
```

```
    ( sqrt-iterate ( improve initial ) initial  x  precision)
)
```

Since the procedures improve, good-enough?, and sqrt-iterate are used only in the procedure sqrt, we may make these procedures local to procedure sqrt.

```
( define ( sqrt x initial   precision)
        ( define ( improve r )
            ( / ( + r  ( / x r )  ) 2 )
        )
        ( define ( absolute x )
            ( if ( < x  0 )  ( - x ) x  )
        )
        ( define ( good-enough? p q )
            ( < ( absolute  ( -  p q ) ) precision )
        )
        ( define ( sqrt-iterate new-r old-r )
            ( if ( good-enough? new-r old-r ) new-r
                    ( sqrt-iterate ( improve new-r ) new-r )
            )
        )
        ( sqrt-iterate ( improve initial ) initial)
)
```

Since *x* and *precision* are bound in procedure *sqrt*, we do not have to pass it to other procedures.

§ 5. RECURSION AND ITERATION

In the previous example of finding the square root of a real number x, the procedure *sqrt-iterate* calls itself. A procedure that calls it self is a *recursive procedure*.

Recursion is a problem-solving method that solves a complex instance of a problem by reducing it eventually to a simple instance that can be solved directly.

The general process that recursion solves an instance of a problem is the following:

Suppose we want to solve an instance I_0 of a problem. Reduce this problem to a simpler instance I_1. If I_1 is still too complex to be solved directly, then reduce it again to an even simpler instance I_2. Continue in this way, until we reach an instance I_n that is simple enough to be solved directly. Then find the solution to I_n. Use the solution to I_n, we can build the solution to the previous instance I_{n-1}. From the solution to I_{n-1}, we can build the solution to I_{n-2}. Continue in this way, we can find the solution to the instance I_0, which is the solution that we want.

To develop a recursive algorithm, we need only the following:

- the base case: a case (or cases) that is simple enough to be solved directly
- the reduction step: a way to reduce a complex instance to a simpler instance (or simpler instances), and this procedure will eventually reach one of the base cases
- the solution build-up step: a way to construct the solution to a complex instance from the solution to a simpler instance (or solutions to simpler instances)

The solution build-up steps in the process of a recursive algorithm are *deferred operations*, since they are executed when the process comes back from the base case.

As an example, we look at a recursive algorithm that finds the sum of squares of the first n natural numbers: $S(n) = 1^2 + 2^2 + \ldots + n^2$.

The base case: $n = 0$. Then the sum is 0.

The reduction step: Ignore the last term n^2. We then have the sum of the square of the first $n - 1$ natural numbers.

The solution build-up step: If we have found the sum of the squares of the first $n - 1$ natural numbers, add n^2 to this sum. Then we have the sum of squares of the first n natural numbers.

Let $n = 4$. The process of finding S is as follows:

$S(4)$	
$S(3) + 4^2$	; $+ 4^2$ is the deferred operation
$S(2) + 3^2 + 4^2$	; $+ 3^2 + 4^2$ is the deferred operation
$S(1) + 2^2 + 3^2 + 4^2$	; $+ 2^2 + 3^2 + 4^2$ is the deferred operation
$S(0) + 1^2 + 2^2 + 3^2 + 4^2$	; $+ 1^2 + 2^2 + 3^2 + 4^2$ is the deferred operation
$0 + 1^2 + 2^2 + 3^2 + 4^2$	; $S(0) = 0$
$1 + 2^2 + 3^2 + 4^2$	; $S(1) = 1$
$5 + 3^2 + 4^2$	; $S(2) = 5$
$14 + 4^2$	; $S(3) = 14$
30	; $S(4) = 30$

The Scheme implementation of this algorithm is the following:

```
( define ( sum-of-squares n )
        ( define ( square n ) ( * n n ) )
        ( if ( = n 0 )
            0
            ( + ( sum-of-squares (- n 1) ) ( square n ) )
        )
)
```

In some cases, a recursive algorithm may be designed so that the solution build-up step can be omitted. In these cases, there are no deferred operations. When an instance is reduced to a simpler instance, the values of a fixed number of state parameters are maintained. Once the base case is reached, the values of state parameters are used to find the solution to the original instance immediately.

In the previous example, we may add up the last terms in each step to have a state parameter, called *temp-sum*, say, initialized as 0, and pass it every time a smaller instance is considered.

$S(4)$: temp-sum = 0.
$S(3)$: temp-sum = $0 + 4^2 = 16$.
$S(2)$: temp-sum = $16 + 3^2 = 25$
$S(1)$: temp-sum = $25 + 2^2 = 29$
$S(0)$: temp-sum = $29 + 1^2 = 30$.

The implementation of this algorithm uses a helper function *sos-iterate*:

```
( define ( sum-of-squares-iter n )
    ( define ( square n ) ( * n n ) )
    ( define ( sos-iterate temp-sum current-n )
        ( if ( = current-n 0 )
            temp-sum
```

```
        ( sos-iterate ( + temp-sum ( square current-n ) ) ( - current-n 1 ) )
      )
    )
    ( sos-iterate 0 n )
)
```

This kind of recursion is called the *tail-recursion*.

A process keeps the values of a set of state parameters and update the state parameters according a certain rule for a number of times is called an *iterative process*. We see that a tail-recursive algorithm generates an iterative process.

In Scheme, there are no loop structures (such as a '*for loop*', or a '*while loop*' in other languages). Scheme uses only recursive procedures to implement recursive or iterative algorithms.

The procedures *sqrt-iterate*, *sum-of-squares*, and *sos-iterate* are all recursive procedures. However, procedure *sum-of-squares* generates a **recursive process**, while *sqrt-iterate* and *sos-iterate* generate **iterative processes**.

Now let's look at some other examples of using recursive procedures in Scheme:

(i) A recursive algorithm of finding the power of a number

Suppose that we want to find the n-th power x^n of a number x, where n is a positive integer. If we multiply x by itself for n times, we need to do $n - 1$ multiplications. The following recursive algorithm computes x^n using $O(\log n)$ multiplications:

If $n = 1$, $x^n = x$. Otherwise, if n is even, let $m = n / 2$. We have $x^n = (x^m)^2$. If n is odd, $x^n = x^{n-1} \times x$.

The Scheme procedure is as follows:

```
( define ( power  x  n)
    (cond ( ( =  n 1 ) x )
          ( ( even? n  )                    ; even? is a built-in predicate
            ( square  ( power  x  ( /  n  2 ) ) )
          )
          ( else ( * ( power x ( -  n  1 ) ) x ) )
    )
)
```

This procedure generates a recursive process.

(ii) Finding the greatest common divisor of two positive integers by the Euclidean algorithm (Gcd is actually a built-in procedure)

The Euclidean algorithm finds the greatest common divisor (gcd) of two integers m and n by the following recursive method:

If $n = 0$, gcd $(m, n) = m$. Otherwise, let r be the remainder when divide m by n. Then gcd (m, n) = gcd (n, r).

```
( define ( gcd m n )
    ( if ( = n 0 ) m
        ( gcd n ( remainder m n ) )
    )
)
```

This procedure generates an iterative process.

There is a problem using the let form in a recursive procedure. The following procedure definition would result an "unbound variable: fact" error:

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

To see why fact is unbound, we rewrite this in the lambda form:

```
( ( lambda      ( fact )   ( fact  3 ) )
    ( lambda  ( n )
        ( if ( = n 0 )  1
            ( *  n   ( fact   ( -  n 1 ) ) )
        )
    )
)
```

The first lambda form defines fact, and the second lambda form is supposed to substitute for the formal parameter fact in the first lambda form. However, the identifier fact in the second lambda form is not bound.

To facilitate the use of let in recursive functions, Scheme provide another alternative for let. This is called "letrec". If we use letrec instead of let in the first definition, the definition of fact will be "carried over" to successive calls. Using letrec, the procedure factorial can be written as:

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

§ 6. CHARACTER AND STRING MANIPULATION

Scheme provides the following predicates to test the properties of characters:

```
char?           // test whether the argument is a character
char=?          // test whether two characters are equal
char-ci=?       // test whether two characters are the same
                // disregarding the case
char->integer   // convert a character to its ASCII code
integer->char   // converts the ASCII code of a character to the character
```

```
> ( char? #\x )
#t
> ( char? 5 )
#f
> ( define x #\a )
> ( char=? x #\a )
#t
> ( char=? x #\A )
#f
> ( char-ci=? x #\A )
#t
> ( char->integer #\x )
120
> ( char->integer x )   ; x is defined as #\a
97
> ( integer->char 120 )
#\x
> ( integer->char 97)
#\a
> ( integer->char 32 )
#\space
> ( integer->char 10 )
#\newline
```

If the argument of integer->char is not in the range of the ASCII code (0-255), there will be an error.

A string is a sequence of characters, denoted between a pair of double quotes.

```
> ( define str "a-string" )
> str
"a-string"
```

To test whether a given value is a string, use predicate "string?".

```
> ( define x "string" )
> ( string? x )
#t
> ( string? "a string" )
#t
> ( string? 123 )
#f
```

The characters in a string can be individually referred to by their *position*s. The positions count starting from 0. To find the character at a given position, use the built-in procedure *string-ref*. If the variable *str* represents the string "a string",

```
> ( string-ref str 3 )
#\t
> ( string-ref str 2)
#\space
> ( string-ref str 0 )
#\a
```

Strings are compared alphabetically by predicates:

```
string=?
string>?
string<?
string>=?
string<=?
```

```
> ( string=? str "a-string" )
#t
> (string=? str "A-string" )
#f
> ( string>? "abcd" "acde" )
#f
> ( string<=? "ab" "abc" )
#t
```

There are other built-in procedures used to manipulate strings:

```
string->number     // convert a string that represents a number to the number
number->string     // convert a number to a string
string->symbol     // convert a sting to a symbol
symbol->string     // convert a symbol to a string
string->list       // convert a string to a list
```

list->string // convert a list to a string
string-append // returns the concatenation of two or more strings

```
> (string->number "1.23")
1.23
> ( + ( string->number "23" ) ( string->number "51" ) )
74
> ( string->symbol "string" )
string
> ( symbol->string 'string )     ; string must be quoted
"string"
> ( string-append "a string " "b string " "c string")
"a string b string c string"
```

procedure string->list and list->string is to be covered later.

§ 7. SIMPLE INPUT / OUTPUT

Scheme provides a procedure *read* to read a value (a number, a string, a Boolean value, or any Scheme value) from the keyboard. Procedure read does not take any arguments, and it returns the input value. Note that the entered value has to be in Scheme constant format. If you enter:

```
> string
```

Then *string* will be recognized as a symbol. If you want to enter a string, you must include the double quotes:

```
> "string"
```

To enter a character *p*, you should do:

```
> #\p
```

You may also enter a list:

```
( define x ( read ) )
> (1 2)
> (car x)
1
> (cdr x)
(2)
```

(Note that, if you input an expression such as (+ 2 3), Scheme recognizes it as a list).

Scheme also provides procedures *write*, *display* and *newline* for output. Procedures *write* and *display* take one argument, which should be a value, and displays this value on the screen. The difference between *write* and *display* is that *write* outputs a string with double quote, and a character with #\, while *display* prints strings without double quotes and prints characters without #\. *Display* is used mainly for printing strings. Procedure *newline* moves the cursor to the beginning of the next line on the screen.

The following is an example of using the input/output procedures:

```
( define read-int
    ( lambda ( )
        ( display "Enter an integer or #f to stop." )
        ( newline )
```

```
        ( read-helper ( read ) )
    )
)
( define read-helper
    ( lambda ( x )
        ( if ( or ( Boolean? x ) ( number? x ) )
            x
            ( begin
                ( display "Invalid input. Try again." )
                ( newline )
                ( read-int )
            )
        )
    )
)
```

There is something new in these procedures.

(i) In the definition of the procedure read-helper, we have a form starting with *begin* with a sequence of other forms. The value of such a form is the value of the last form in the sequence. The forms after *begin* are evaluated in the specified order of this sequence.

(ii) In the body of procedure read-int, we have three forms in the body of the lambda expression. Actually, the lambda procedure is defined by the last form in this sequence. In this case, the sequence of forms in the lambda expression are evaluated according to the specified order, as if they are included in a begin form.

(iii) These two procedures are mutual recursive, since each of them calls the other.

Note that, since components in an expression are not necessarily evaluated in the order of the sequence, the following expression does not necessarily have the value as expected:

```
( - ( read ) ( read ) )
```

If the user enters a 5 and then a 2, the evaluator may give 3 or −3. To make it precise, we should have

```
( let  ( ( x   ( read ) ) )
    ( let  ( ( y  ( read ) ) )
         ( - x  y )
    )
)
```

Such a nested let form is used frequently so that Scheme provides a simplified version using a new keyword let*. The previous example can be written using let* as

```
( let*  ( ( x ( read ) )
          ( y ( read ) )
        )
        ( - x y )
)
```

CHAPTER 3. LISTS IN SCHEME

§ 1. PAIRS

We may need to combine a number of data items together and give it a single name. This becomes an aggregated data item. To create an aggregated data item, we need an operator to create it from the component data items. This operator is called a *constructor*. We also need operators to retrieve the components from an aggregated data item. These operators are called *selectors* (*accessors*). Using the constructor and the selectors, we may hide the implementation of an aggregated data type to achieve *data abstraction*. We use constructors and selectors to define a number of procedures to manipulate the aggregated data items. The user code needs only to know how to use these procedures to manipulate the aggregated data items but the detailed implementation is hidden from the user code.

Scheme uses *pair*s and *list*s to build aggregated data items.

A *pair* consists of two components, and it is referred to by a single identifier.

To construct a pair, use the built-in operator *cons* (meaning *construction*). The following definition defines a pair with two components x and y:

```
> ( define z ( cons 'x 'y ) )
> z
(x . y)
```

The last line is the way that Scheme writes a pair.

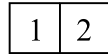
Operator *cons* is the constructor of the aggregated data type *pairs*.

We have two selectors that return the first and the second components, respectively, of a pair: Operator *car* returns the first component and operator *cdr* returns the second component.

```
> ( car z )
x
> ( cdr z )
y
```

A pair can be illustrated by a *box diagram*.

A pair constructed by (cons 1 2) can be illustrated by:



Note that, if we have:

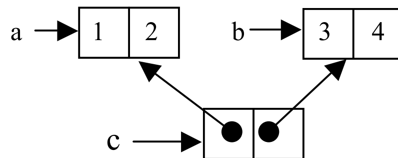
(define x 1)
(define y 2)
(define z (cons x y))

Then (car z) is a copy of x, and (cdr z) is a copy of y.

The name of a pair is a "pointer". If we have:

(define a (cons 1 2))
(define b (cons 3 4))
(define c (cons a b))

then (car c) **is** a, and (cdr c) **is** b. This can be illustrated by the following diagram:



Scheme provides a predicate (pair?) to check if a variable is a pair.

```
> ( define x ( cons 'a 'b ) )
> ( pair? x )
#t
> ( pair? 'a )
#f
> ( pair? 2 )
#f
```

§ 2. LISTS

The components of a pair may themselves pairs. We may use a nested pair structure to represent a sequence. Suppose we want to represent a sequence 1, 2, 3, 4, 5. We may build a nested pair structure as follows:

```
> ( define x ( cons 1 (cons 2 ( cons 3 ( cons 4 5) ) ) ) )
x
(1 2 3 4 . 5)
```

Such a nested pair structure is called a *list*. A special list called the *empty list*, denoted by ( ) or *null*, is defined to be a list without any member. If the last member of a list is an empty list, the list is a *proper list*. Otherwise, it is an *improper list* or *dotted list*. In this terminology, a pair constructed by operator cons with two data objects is an improper (or dotted) list with two members. A proper list is denoted by listing all members between parentheses. For instance, (1 2 3) is a proper list, which is constructed by

```
> ( define a-list ( cons 1 ( cons 2 ( cons 3 ( ) ) ) ) )
> a-list
(1 2 3)
```

An improper list is denoted the same way except that the last member is separated by a *dot*.

```
> ( define a ( cons 1 ( cons 2 ( cons 3  4 ) ) ) )
> a
(1 2 3 . 4)
```

This is why a pair (cons x y) is denoted by (x **.** y).

The construction of a proper list using nested cons can be simplified by operator *list*:

```
> ( define x ( list 1 2 3 4 5 ) )
> x
(1 2 3 4 5)
```

Because proper lists are used much more frequently than improper list, we usually say a *list* to mean a *proper list*.

The operator *list* is special because it may take any number of arguments. The other operators we learned so far can take only a fixed number of arguments.

```
( list )                ; returns an empty list
```

```
( list 1 )          ; returns a list with a single member 1
( list 'a 'b )      ; returns a list with two members a and b
( list 'a 'b 'c 'd) ; returns '(a b c d)
```

Scheme provides a way to define operators with any number of arguments.

The formal parameter list of such an operator is denoted by

```
. x         ; if the number of parameters is zero or more
x . y       ; if the number of parameters is one or more
x y . z     ; if the number of parameters is two or more
…
```

For instance, we may define an operator as follows:

```
( define ( op1 . x ) <body> )
```

Then op1 may accept zero or more parameters. In the *body* of the definition, x is used as a list of the parameters.

If an operator is defined as:

```
( define ( op2 x . y ) <body> )
```

Then op2 may accept one or more parameters. In the *body* of the definition, x is used to represent the first parameter, and y is used as a list of the other parameters.

In this way, operator *list* can be defined as follows:

```
( define ( list ) ( ) )
( define ( list x . y ) ( cons x y ) )
```

Since a list is actually a pair, we may use operator car and cdr to retrieve the components. If x is a list (1 2 3 4 5), then

```
> ( car x )
1

> ( cdr x )
(2 3 4 5)
```

List members may also be symbols. We may use the quote to define a list with symbol members. '(a b c) is the same as (list 'a 'b 'c).

```
> ( car '( a b c ) )
a
```

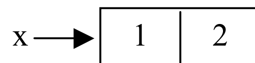
```
> ( cdr '( a b c ) )
(b c)
```

The members of a list may themselves lists. For instance, we may define:

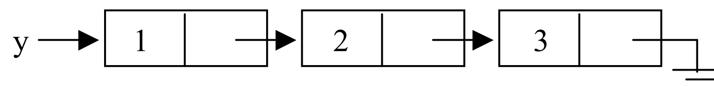
```
> ( define x ( list ( list 1 2 ) ( list 3 ) ( list ) 4 ) ) )
> x
((1 2) (3) ( ) 4)
```

Lists can be represented by box-diagrams as in the following examples:

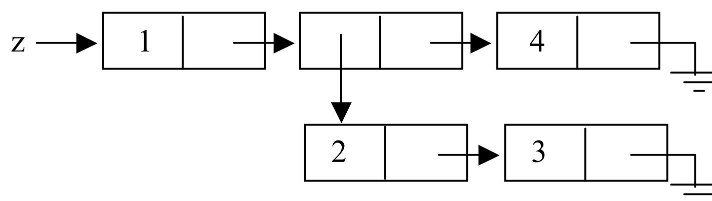
```
( define x ( cons 1 2 ) )
```




```
( define     y    ( list  1   2   3 ) )
```




```
( define     z    ( list 1  ( list 2 3 ) 4 ) )
```



As a pair, the name of a list is also a pointer to the physical list. This can be made clear if we use predicate eqv? to test the equality of lists.

```
> ( eqv?  ( list 1 2 )  ( list 1 2 ) )        ; we constructed two lists with the same value
                                              ; but they are physically different lists
#f
> ( define a ( list 1 2 ) )
> ( define  b a )                             ; b and a are two pointers to the same list
> ( eqv?  a b )
#t
```

The same as pairs, when we use procedure *cons* to construct a list with a list member, the result list has the member list as a part. Suppose we have the following definitions of four lists:

```
( define  x  ( list  1  2 ) )
( define  y  ( list  3  4 ) )
( define  z  ( cons  x  y ) )
( define  w ( cons  y  x ) )
```

Lists z and w share the same physical lists x and y.

```
> ( eqv?  ( car z )  ( cdr w ) )
#t
> ( eqv?  ( car w )  ( cdr  z ) )
#t
```

To test whether the value of two lists are the same (i.e., they have the same members), we use the predicate equal?

```
> ( equal?  ( list 1 2 )  ( list 1 2 ) )
#t
```

When the name of a list is passed to a function, the formal parameter takes the value of the argument, so that they point to the same physical list.

```
( define x ( list 1 2 3 ) )
( define ( check a )  a  )
( define y ( check x ) )
> ( eqv? y  x  )
#t
```

In this example, line one defines a list called x. In the second line, a function (check a) is defined. This function returns the parameter a. In this third line, list x is passed to function (check a) as an argument. Then y is defined as what is returned from function (check a), which is a. At the prompt, x and y are compared by predicate eqv?. The result is true.

We may combine procedures car and cdr to access the members of a list. For instance, if x is a list, then

(car (cdr x)) is the second member of the list,

(car (cdr (cdr x))) is the third member of x.

If the members of a list are themselves lists, then we may have something like:

(car (cdr (car (cdr x)))).

Scheme provides a simplified way to write these combined procedures. The name of such a combined procedure starts with a letter *c* and ends with a letter *r*. Every procedure car is represented by a letter *a*, and every procedure cdr is represented by a letter *d*. Therefore,

(caddr x) is the same as (car (cdr (cdr x)));

(cdaddr x) is the same as (cdr (car (cdr (cdr x)))).

Operators *display* and *write* can be used to output an entire list:

```
> ( define x ( list 1 2 3 ) )
> ( display x )
(1 2 3)
> ( write x )
(1 2 3)
```

A list of characters can be converted to a string, and vise versa, using the predicates list->string and string->list.

```
> ( string->list "string")
(#\s #\t #\r #\i #\n #\g)
> ( list->string (list #\a #\b #\space #\c) )
"ab c"
```

§ 3. MANIPULATING LISTS

The procedures used to manipulate lists are recursive in nature. Scheme provides a predicate to test whether a list is the empty list. This predicate is *null*?. Using this predicate, we can define a procedure that returns the length of a list.

```
( define ( length x )
    ( if ( null?  x )
        0
        ( + 1  ( length ( cdr x ) ) )
    )
)
```

Or, in an iterative procedure,

```
( define ( length x )
    ( define ( length-iter a len )
        ( if ( null? a )
            len
            ( length-iter ( cdr a ) ( + 1 len ) )
        )
    )
    ( length-iter x 0 )
)
```

The following procedure returns a member of the list. If n = 0, it returns the first member of the list; otherwise, it returns the (n + 1) th member of the list:

```
( define  ( nth  n x )
    ( if ( =  n 0 )
        ( car x )
        ( nth ( - n 1 ) ( cdr x )  )
    )
)
```

This process is iterative. We say that this procedure is "cdring down" the list.

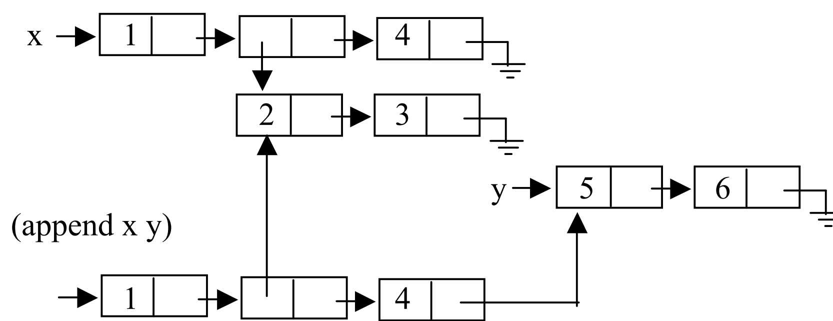
The following procedure "cons up" an answer list while cdring down another list:

```
( define ( append x y )
    ( if ( null? x )
        y
        ( cons  ( car x )
```

```
                ( append  ( ( cdr  x )  y ) )
        )
    )
)
```

Append is a built-in procedure in Scheme. Note that, in the list returned from procedure append, the second part is physically the list y, but the first part is only a copy of the list x, because, in the form (cons (car x) (append ((cdr x) y))), (car x) is a copy of the first member of x, while y is the "actual" list y. However, if a member of x is itself a list, then this list is shared by the returned list and x.

The following diagram shows two lists x and y, and the result of (append x y):



Assume the members of a list are comparable, such as numbers, characters, or strings. A common task to manipulate a list is to sort it, i.e., rearrange the members so that they are in an increasing order or a decreasing order.

There is a great number of sorting algorithms. Here, we implement a sorting algorithm known as the *Insertion Sort*.

First, we need a procedure to insert a new member into a sorted list. Assume we want to sort the members in a decreasing order, and the members of the list are numbers.

```
( define ( insert x sorted-list )
    ( cond  ( ( null? sorted-list )  ( list x ) )
            ( ( > x ( car sorted-list ) ) ( cons x sorted-list ) )
            ( else ( cons ( car sorted-list ) ( insert x ( cdr sorted-list ) ) ) )
    )
)
```

The insertion sort algorithm is implemented as follows:

```
( define ( insertion-sort a-list )
    ( if      ( or ( null? a-list ) ( null? ( cdr a-list ) ) )
```

```
            a-list
            ( insert ( car a-list ) ( insertion-sort ( cdr a-list ) ) )
    )
)
```

Three typical operations on lists are *mapping*, *filtering*, and *accumulating*.

A mapping operation constructs a list whose members are *maps* of the members of the original list. The *map* of a member of a list is the result of an operation on the member. For instance, the map procedure is the procedure *square*. Then a map operation on a list constructs another list whose members are squares of the members of the original list.

A map procedure can be defined as follows:

```
( define ( map  a-list a-procedure )
    ( if ( null?  a-list )  ( )
        ( cons  ( a-procedure   ( car a-list ) )
                ( map   ( cdr a-list ) a-procedure )
        )
    )
)
```

This procedure accepts another procedure *a-procedure* as a parameter. For instance, it could be the procedure square:

```
> ( define  ( square x ) ( * x x ) )
> ( map ( list 1 2 3 4 ) square )
(1 4 9 16)
```

A filtering operation constructs a new list that contains only the members of the original list that satisfy a condition. The condition is given by a predicate. Filtering procedures can be defined as follows:

```
( define ( filter predicate a-list )
    ( cond  ( ( null? a-list )  ( ) )
            ( ( not  ( predicate ( car a-list ) ) )
                    ( filter  predicate ( cdr a-list ) )
            )
            ( else
                ( cons ( car a-list )
                        ( filter  predicate  ( cdr a-list ) )
                )
            )
    )
)
```

For example, we want to find all odd members in a given list. The predicate odd? is a built-in predicate, which may be defined as:

```
( define ( odd? x )
     ( = ( remainder x 2 ) 1 )
)
```

Then we have:

```
> ( filter odd? ( list 1 2 4 7 10 15 ) )
(1 7 15)
```

An accumulating procedure finds a result obtained by accumulating the contributions of the members in a list. The way to accumulate is specified by a procedure. This procedure accepts two parameters: One is the contribution of the current member, and the other is the accumulated result from the previous members.

An accumulating procedure can be defined as follows:

```
( define ( accumulate a-procedure initial-value a-list )
     ( if  ( null? a-list ) initial-value
          ( a-procedure
               ( car a-list )
               ( accumulate a-procedure initial-value ( cdr a-list ) )
          )
     )
)
```

For instance, we have:

```
> ( accumulate + 0 ( list 1 2 3 4 ) )
10
> ( accumulate * 1 (list 1 2 3 4 ) )
24
```

We may combine these operations together. For instance, the following procedure constructs a list of the squares of odd members in a list:

```
( define ( square-odds a-list )
     ( map ( filter odd? a-list ) square )
)
```

The following procedure finds the sum of the odd members of a list:

```
( define ( sum-odds a-list )
     ( accumulate + 0 ( filter odd? a-list ) )
```

```
)
```

The following procedure finds the sum of the squares of a given list:

```
( define ( sum-squares a-list )
    ( accumulate + 0 ( map a-list  square )
)
```

The following procedure finds the sum of the squares of the odd members of a list:

```
( define ( sum-odd-squares a-list )
    ( accumulate + 0 ( map ( filter odd? a-list ) square ) )
)
```

§ 4. USING LISTS TO IMPLEMENT ABSTRACT DATA TYPES

The list is the main tool used in Scheme to implement abstract data types. The basic operations for manipulating lists are cons, car and cdr. As an abstract data type, we want to hide the implementation from the user code. In other words, the use of cons, car, and cdr should not be seen by the user of the data type. Here we use a number of *abstraction barrier* to separate different levels of abstraction: The first level is to use basic list operations to define procedures that construct a data object (the *constructor*), or selector the members of a data object (*selectors*). The second level is to use the constructors and selectors to define procedures that manipulate data objects. Then, finally, these procedures are used in the user code.

(1) Rational Numbers as Pairs

Rational numbers are defined as pairs of integers. The constructor is:

```
( define ( make-rational n d )
         ( cons  n d )
)
```

The accessors are:

```
( define ( numerator r ) ( car r ) )
( define ( denominator ) ( cdr  r ) )
```

Using this definition, we can implement the operations on rational numbers:

```
( define ( rat=?a    b )
         ( = ( * ( numerator a ) ( denominator b ) )
             ( * ( denominator a ) ( numerator b ) )
         )
)
( define ( rat+  a    b )
         ( make-rational  ( + ( *  ( numerator a )   ( denominator b ) )
                              ( * ( numerator   b )  ( denominator a ) )
                          )
                          ( * ( denominator a ) ( denominator b ) )
        )
)
( define     ( rat- a b )
             ( make-rational ( - ( *  ( numerator a ) ( denominator b ) )
                                 ( *  ( numerator b )  ( denominator a ) )
                             )
```

```
                              ( * ( denominator a ) ( denominator b ) )
              )
)
( define ( rat* a b )
          ( make-rational  ( * ( numerator a ) ( numerator b ) )
                           ( * ( denominator a )  ( denominator b ) )
          )
)
( define ( rat/ a b )
          ( make-rational  ( *  ( numerator a ) ( denominator b ) )
                           ( *  ( denominator a )  ( numerator b ) )
          )
)
( define ( print-rational x )
     ( let ( ( num  ( / ( numerator x )  ( gcd ( numerator x ) ( denominator x ) ) ) ) )
             ( den   ( / ( denominator x ) ( gcd ( numerator x) ( denominator x ) ) ) ) )
           )
           ( cond  ( ( = den  1 ) ( display     num ))
                   ( ( = den -1 ) ( display (- num )))
                   ( ( < den 0 )
                      ( begin
                           ( display ( - num ) )
                           ( display    "/")
                           ( display ( - den ) )
                      )
                   )
                   ( else
                      ( begin
                           ( display num )
                           ( display "/")
                           ( display den )
                      )
                   )
           )
     )
)
```

Note that, when the user uses these procedures to manipulate rational numbers, the implemention of rational numbers as pairs is hidden from the user. For instance, we may reduce the rational numbers at the definition stage as:

```
( define ( make-rational n d )
     ( let ( g( gcd n d ) )
     ; gcd is the procedure to find the greatest common divisor
     ( cons  ( / n  g ) ( / d g ) )
)
```

Then, the print-rational procedure may defined simply as

```
( define ( print-rational x )
    ( let ( ( num ( numerator x ) )
            ( den ( denominator x ) )
          )
        ( cond  ( ( = den  1 ) ( display num ))
                ( ( = den -1 ) ( display (- num )))
                ( ( < den 0 )
                    ( begin
                        ( display ( - num ) )
                        ( display    "/")
                        ( display ( - den ) )
                    )
                )
                ( else
                    ( begin
                        ( display num )
                        ( display "/")
                        ( display den )
                    )
                )
        )
    )
)
```

The following program can be used to test the implementation of rational numbers:

```
( define ( rational-demo )
  ( let ( ( x ( make-rational 3 -4 ) ) ( y ( make-rational 8  9 ) ) )
    ( begin
        ( display "Define a rational number x = " )
        ( print-rational x )
        ( newline )
        ( display "Define a rational number y = " )
        ( print-rational y )
        ( newline )
        ( display "Compare x and y: " )
        ( if ( rat=? x y )
            ( display "x and y are equal" )
            ( display "x and y are not equal" )
        )
        ( newline )
        ( display "The sum of x and y is " )
        ( print-rational ( rat+ x y ) )
```

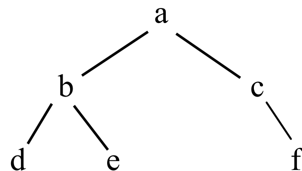
```
        ( newline )
        ( display "The difference of x and y is " )
        ( print-rational ( rat- x y ) )
        ( newline )
        ( display "The product of x and y is " )
        ( print-rational ( rat* x y ) )
        ( newline )
        ( display "The quotient of x and y is " )
        ( print-rational ( rat/ x y ) )
        ( newline )
    )
  )
)
```

The output of this program is:

```
Define a rational number x = -3/4
Define a rational number y = 8/9
Compare x and y: x and y are not equal
The sum of x and y is 5/36
The difference of x and y is -59/36
The product of x and y is -2/3
The quotient of x and y is -27/32
```

(2) Binary Search Trees as Lists

First, we introduce a way to denote binary trees: An empty binary tree is denoted by ( ). A binary tree is denoted recursively by (root; left-subtree; right subtree). In this way, a binary tree with a single node r, i.e., the root, will be denoted by (r; ( ); ( )). Just to simplify the notation, we will simply use (r;;) to denote such a binary tree. Then the binary tree represented by the following diagram will be denoted as:
(a; (b; (d;;); (e;;)); (c;;(f;;))).



This notation suggests naturally a way to represent at tree by a list. If the tree is empty, it is represented by the empty list. If the tree is not empty, then it is represented by a list with three members. The first member is the root, the second member is the left subtree and the third member is the right subtree.

We use the following constructor to construct a binary tree:

```
( define ( initialize-tree ) ( ) )       ; initialize a tree as an empty tree

( define ( make-tree root left right )
    ( list root left right )
)                                         ; construct a non-empty tree
```

The selectors are the following:

```
( define ( root t ) ( car t ) )
( define ( left-subtree  t ) ( cadr  t ) )
( define ( right-subtree t ) ( caddr  t ) )
```

A predicate tests whether a tree is empty:

```
( define ( empty-tree? t ) ( null? t ) )
```

These procedures establish the first abstraction barrier. Using these procedures, we can implement the operations to manipulate binary search trees.

The procedure to add a new member to a binary search tree is defined as follows (assume all keys in the binary search tree are distinct):

```
( define  ( insert  x  t )
  ( cond ( ( empty-tree? t ) ( make-tree x ( initialize-tree ) ( initialize-tree ) ) )
      ( ( > x ( root t ) )
        ( make-tree ( root t ) ( left-subtree t ) ( insert x ( right-subtree t ) ) )
      )
      ( ( < x ( root t ) )
        ( make-tree  ( root t ) ( insert x ( left-subtree t ) ) ( right-subtree t ) )
      )
  )
)
```

According to this definition, when a new member is added to a tree, we have a new tree with the new member added.

The procedure to search for a given key in a binary search tree is defined as follows:

```
( define ( in-tree? x t )
    ( cond  ( ( null? t ) #f )
            ( ( = x  ( root t ) )  #t )
            ( ( > x  ( root t ) ) ( in-tree? x ( right-subtree t ) ) )
            ( ( < x  ( root t ) ) ( in-tree? x ( left-subtree t ) ) )
    )
)
```

To print a tree, we have the following:

```
( define ( print-tree t )
  ( if (not ( empty-tree? t ) )
             ( begin
                 ( display "(")
                 ( display ( root t ) )
                 ( display ";")
                 ( print-tree ( left-subtree t ) )
                 ( display ";")
                 ( print-tree ( right-subtree t ) )
                 ( display ")")
             )
          )       ; empty tree is not printed
  )
)
```

The following is a test function:

```
( define ( bst-demo )
  ( begin
    ( define t0 ( initialize-tree ) )
    ( define t1 ( insert 5 t0 ) )
    ( define t2 ( insert 3 t1 ) )
    ( define t3 ( insert 7 t2 ) )
    ( define t4 ( insert 2 t3 ) )
    ( define t5 ( insert 6 t4 ) )
    ( print-tree t5 )
    ( newline )
    ( if ( in-tree? 6 t5 ) ( display "6 is in this tree" )
        ( display "6 is not in this tree" )
    )
    ( newline )
    ( if ( in-tree? 4 t5 ) ( display "4 is in this tree" )
        ( display "4 is not in this tree" )
    )
    ( newline )
  )
)
```

The output of this program is:

```
(5;(3;(2;;););(7;(6;;);))
6 is in this tree
4 is not in this tree
```

(3) Symbolic Algebra

Consider all polynomials with a single variable. A polynomial with a single variable will be represented by a list. The first component of the list is the variable. The other components of the list represent the terms in the polynomial. Each term is represented by a list with two components representing the degree of the term and the coefficient. These components are arranged in a decreasing order of the degrees. For instance, a polynomial

$$P(x) = 3x^5 + x^2 - 2x + 5$$

is represented by a list ('x (5 3) (2 1) (1 –2) (0 5)).

We will implement the addition and multiplication of polynomials. Since we consider only polynomials with a single variable, two polynomials can be added or multiplied only if they have the same variable.

We start with terms:

Constructor:
```
( define ( make-term degree coefficient )
    ( list degree coefficient )
)
```

Selectors:

```
( define ( degree term ) ( car term ) )
( define ( coefficient term ) ( cadr term ) )
```

Predicates:

```
( define ( zero-term? term ) ( = ( coefficient term ) 0 ) )
( define ( higher-degree? t1 t2 ) ( > ( degree t1 ) ( degree t2 ) ) )
( define ( same-degree? t1 t2 ) ( =  ( degree t1 ) ( degree t2 ) ) )
```

The next step is to define term lists:

Constructor:

```
( define ( initialize-termlist ) ( ) )
( define ( make-termlist x . y ) ( insertion-sort ( cons x y ) ) )
( define ( attach-at-beginning t p )
    ( if ( zero-term? t ) p
        ( cons t p )
    )
)    ; attach a new term t at the beginning of a term list p
```

Procedure make-termlist may accept one or more terms as parameters. Procedure insertion-sort is the same as defined before, but we need to compare terms by their degrees:

```
( define ( insert x sorted-list )
    ( cond  ( ( null? sorted-list )  ( list x ) )
            ( ( higher-degree? x  ( car sorted-list ) ) ( cons x sorted-list ) )
            ( else ( cons ( car sorted-list ) ( insert x ( cdr sorted-list ) ) ) )
    )
)
( define ( insertion-sort a-list )
    ( if    ( or ( null? a-list ) ( null? ( cdr a-list ) ) )
            a-list
            ( insert ( car a-list ) ( insertion-sort ( cdr a-list ) ) )
    )
)
```

Selectors:

```
( define ( first-term termlist ) ( car termlist ) )
( define ( rest-terms termlist ) ( cdr termlist ) )
```

A predicate:

```
( define ( empty-termlist? termlist ) ( null? termlist ) )
```

Now we can define polynomials:

Constructor:

```
( define ( make-polynomial variable term-list )
    ( cons  variable  term-list )
)
```

Selectors:

```
( define ( variable  poly )   ( car poly ) )
( define ( termlist  poly )   ( cdr poly ) )
```

A predicate:

```
( define ( same-var? p1 p2 ) ( eqv? ( variable  p1 ) ( variable p2 ) ) )
```

These procedures give the first abstraction barrier of the data type polynomial. Then we can write the procedures to add or multiply two polynomials:

```
( define ( polynomial+ p1 p2 )
 ( if ( same-var? p1 p2 )
   ( make-polynomial ( variable p1 ) ( termlist+termlist ( termlist  p1) ( termlist  p2 ) ) )
   #f
 )
)

( define ( polynomial* p1 p2 )
 ( if ( same-var? p1 p2 )
     ( make-polynomial ( variable p1 ) ( termlist*termlist ( termlist  p1) ( termlist  p2 ) ) )
     #f
 )
)
```

Now we need only to implement procedures *termlist+termlist* and *termlist*termlist*. To define termlist+termlist, first, we reduce it to add one term to a term list.

```
( define (termlist+termlist p1 p2 )
  ( if ( empty-termlist? p1 )
    p2
    (termlist+termlist ( rest-terms p1 ) ( term+termlist ( first-term p1 ) p2 ) )
; or (term+termlist ( first-term p1 ) ( termlist+termlist ( rest-term p1 ) p2 ) )
  )
)

( define ( term+termlist t p )
 ( cond ( ( empty-termlist? p )  ( make-termlist t ) )
        ( ( higher-degree? t ( first-term p ) ) ( attach-at-beginning t p ) )
        ( ( same-degree? t ( first-term p ) )
          ( attach-at-beginning ( term+term t ( first-term p )) ( rest-terms p ) )
        )
        ( else
          ( attach-at-beginning ( first-term p ) ( term+termlist t ( rest-terms p ) ) )
        )
 )
)

( define ( term+term t1 t2 )
 ( if ( same-degree? t1 t2 )
     ( make-term ( degree t1 ) ( + ( coefficient t1 ) ( coefficient t2 ) ) )
 )
)

( define  ( termlist*termlist  terms1  terms2 )
 ( if ( empty-termlist? terms1 )
```

```
        ( initialize-termlist )
        (termlist+termlist  (term*termlist ( first-term terms1 ) terms2 )
                   (termlist*termlist ( rest-terms terms1) terms2 )
        )
  )
)

( define ( term*termlist  t p )
    ( if  ( empty-termlist? p )
         ( initialize-termlist )
         ( let  ( ( first   ( first-term  p ) ) )
         ( term+termlist ( term*term t ( first-term p ) )
                         ( term*termlist t ( rest-terms p ) )
           )
       )
    )
)

( define ( term*term t1 t2)
    ( make-term ( + ( degree t1 ) ( degree t2 ) )
                ( * ( coefficient t1 ) ( coefficient t2 ) )
)
```

The following is a test function:

```
(define (test)
  (define p0 (make-polynomial 'x (initialize-termlist)))     ; zero polynomial
  (define p1 (make-polynomial 'x (list(list 0 -3))))         ; constant
  (define p2 (make-polynomial 'x (list (list 1 1))))         ; variable x
  (define tlist3 (make-termlist (list 5 1) (list 4 2) (list 2 -1) (list 0 3)))
  (define p3 (make-polynomial 'x tlist3))
  (define tlist4 (make-termlist (list 4 2) (list 3 -2) (list 2 11) (list 1 -3)))
  (define p4 (make-polynomial 'x tlist4))
  (define tlist5 (make-termlist (list 2 1)))
  (define p5 (make-polynomial 'x tlist5))
  (define tlist6 (make-termlist (list 2 1) (list 0 -1)))
  (define p6 (make-polynomial 'x tlist6))
  (define tlist7 (make-termlist (list 1 1) (list 0 -1)))
  (define p7 (make-polynomial 'x tlist7))
  (define tlist8 (make-termlist (list 2 1) (list 1 1) (list 0 1)))
  (define p8 (make-polynomial 'x tlist8))
  (define p9 (polynomial+ p0 p4))
  (define p10 (polynomial+ p1 p4))
  (define p11 (polynomial+ p3 (polynomial* p2 p4)))
  (define p12 (polynomial* p0 p3))
  (define p13 (polynomial* p1 p3))
```

```
 (define p14 (polynomial* p3 p4))
 (define p15 (scalar*polynomial 3 p0))
 (define p16 (scalar*polynomial 3 p1))
 (define p17 (scalar*polynomial 0 p4))
 (define p18 (scalar*polynomial 2 p4))

 (begin
   (display p1)
   (newline)
   (display p2)
   (newline)
   (display p3)
   (newline)
   (display p4)
   (newline)
   (display p5)
   (newline)
   (display p6)
   (newline)
   (display p7)
   (newline)
   (display p8)
   (newline)
   (display p9)
   (newline)
   (display p10)
   (newline)
   (display p11)
   (newline)
   (display p12)
   (newline)
   (display p13)
   (newline)
   (display p14)
   (newline)
   (display p15)
   (newline)
   (display p16)
   (newline)
   (display p17)
   (newline)
   (display p18)
  )
)
```

Here is the output:

```
> (test)
(x (0 -3))
(x (1 1))
(x (5 1) (4 2) (2 -1) (0 3))
(x (4 2) (3 -2) (2 11) (1 -3))
(x (2 1))
(x (2 1) (0 -1))
(x (1 1) (0 -1))
(x (2 1) (1 1) (0 1))
(x (4 2) (3 -2) (2 11) (1 -3))
(x (4 2) (3 -2) (2 11) (1 -3) (0 -3))
(x (5 3) (3 11) (2 -4) (0 3))
(x)
(x (5 -3) (4 -6) (2 3) (0 -9))
(x (9 2) (8 2) (7 7) (6 17) (5 -4) (4 -5) (3 -3) (2 33) (1 -9))
(x)
(x (0 -9))
(x)
(x (4 4) (3 -4) (2 22) (1 -6))
```

CHAPTER 4. MUTABLE DATA, OBJECTS, AND STREAMS

§ 1. ASSIGNMENT

So far, an identifier is just a name that is bound to a value. This name-value relationship is *permanent*. However, if we think a name is a state of an object, then the value of the name, i.e., the state of the object may *change*. We need to associate a new value with an existing name. This is achieved by assignment. The process of assignment uses a special form with the keyword set!. Note that set! is **NOT** a procedure. This means that the assignment form does not return a value. Assignment does not belong to the functional paradigm of the language, since, in a function, the value of variables will never change. There is no mathematical interpretation of assigning x + 2 to x.

```
> ( define  x   2 )
> ( set!  x   3 )
> x
3
```

With assignment, the same expression may be evaluated to be different values at different times.

```
> ( define ( add-to x )
    ( lambda ( y )  ( set! x ( + x y ) )  x )
  )
> ( define accum ( add-to  20 ) )
> ( accum  5 )
25
> ( accum 10 )
35
```

In many cases, using define has the same effect as to use set!. But **DO NOT** use define if you want to mutate an **EXISTING** variable. Define is only used to bind a new identifier to a data value or a procedure. Scheme allows you to redefine an identifier just to provide a way for you to correct errors.

We may also use assignment to lists. Scheme provides two forms to change the members of a list: set-car! and set-cdr!. Set-car! assigns a new value to the car of the list, and set-cdr! assigns a new list as the cdr of the list.

```
> ( define x 1 )                    ; define a variable x
> ( define y 6 )                    ; define another variable y
> ( define z ( list 2 3 ) )         ; define a list y
> ( define w ( list 4 5 ) )         ; define a list z
> ( define vec ( cons x z ) )       ; cons x with z to have a new list
> vec
(1 2 3)
> ( set-car!  vec  y )
> vec
(6 2 3)
> ( set-cdr! vec w )
> vec
(6 4 5)
```

Recall that the name of a list is just a pointer. The change of a physical list through one name will affect the other name. However, two names of primitive type variables refer to different data objects.

Question 1. Suppose we have the following:

```
> ( define x 1 )
> ( define y 2 )
> ( define z x )
> ( set! x y )              ; now x is 2
> ( set! y 3 )
```

What is the values of x, y, and z?

Answer. The form (set! x y) assigns the **value** of y to x. Variables x and y are still two independent variables. Any change to one variable does not have effect on the other variable. Then x is still 2. Variable z takes the original value 1 of x. When x is changed, z is still 1.

Question 2. If we have the following definitions:

```
> ( define x 1 )
> ( define y ( list 2 3 ) )
> ( define z ( cons x y ) )
> ( set! x 4 )
> ( set-car! y 5 )
```

what is the value of z?

Answer. Since (cons x y) take the value of x and the list y, the change of x does not affect z, but the change of y will change z. The list z becomes (1 5 3).

On the other hand, if x is a list, y is a number, and z is defined by (cons x y), then the change to x and the change to (car z) are related, and the change to y and the change to (cdr z) are not related.

```
> ( define x  ( list 1 2 ) )
> ( define y  3 )
> ( define z ( cons x y ) )
> z
((1 2) . 3)
> ( set  y  4 )
> (se-car!  x  5 )
> z
((5 2) . 3)
```

Also remember that, when a list is passed to a procedure, or returned from a procedure, the name of the list is just a pointer. Therefore, we have the following result:

```
> ( define x 1 )
> ( define atom ( lambda (a) a ) )
> ( define y ( atom x ) )
> y
1
> (set! x 2)
> y
1

> ( define z ( list 1 2 ) )
> ( define w ( atom z ) )
> w
(1 2)
> (set-car! z 3)
> w
(3 2)
> (set-car! ( cdr z ) 4 )
> w
(3 4)
```

Another thing to be watched out for is that, as with using the procedure append or cons to build lists with list members, when we use set-car! or set-cdr!, we may produce a result that consists of a complex structure of shared sublists.

§ 2. USING ASSIGNMENTS TO WRITE EXPLICIT ITERATIVE PROCEDURES

We have been using tail recursion to implement iterative procedures. After we introduced assignment, we may make iterative procedures explicit. A program using assignment extensively is said to have an *imperative style*, because this is the idea used in imperative languages. In Scheme, being, essentially, a language of the functional paradigm, this style is ***not*** encouraged.

In a functional style, i.e., use recursive procedure calls, a procedure that calculates the factorial of an integer n can be written as follows:

```
( define     ( factorial  n )
 (define  ( factorial-iter  result   count )
    ( if ( >  count  n )
         result
         ( factorial-iter ( * result count  ) ( + count  1 ) )
    )
 )
 (factorial-iter 1 1)
)
```

In the imperative style, this procedure may be written as follows:

```
( define  (factorial  n )
    ( let  ( ( result 1)  ( count 1 ) )
         ( define   ( iterate )                ; need parentheses to indicate a procedure call
             ( if  ( > count   n )
                  result
                  ( begin                      ; the order of the following forms is important
                      ( set!  result  ( * result  count ) )
                      ( set!  count  ( + count  1 ) )
                      ( iterate )
                  )
             )
         )
         ( iterate )
    )
)
```

§ 3. THE ENVIRONMENT MODEL

An expression with assignment can no longer be evaluated using the substitution model. For instance:

```
> ( define ( add-to  x )  ( lambda ( y )  ( set!   x    ( + x y ) ) x ) )
> ( define ( accum )  ( add-to  20 ) )
> ( accum  5 )
25
```

If we use the substitution model, we would have the following sequence:

```
( accum 5 )
( ( add-to  20 ) 5 )                                ; substitute the definition for accum
( ( lambda ( y ) ( set!  x  ( + x y ) ) x ) 20 ) 5 )   ; substitute the definition for ( add-to 20)
( ( lambda ( y ) ( set! 20 ( + 20 y ) ) 20 ) 5 )       ; substitute 20 for x
( ( set!  20  ( + 20  5 ) ) 20 )                    ; substitute 5 for y
( ( set!  20  25) 20 )
```

You cannot set 22 to 25!

The reason for this is that the substitution model is based on the functional paradigm of the language. But the set! form does not fit in this paradigm.

We have to use a new model of evaluation of expressions. This is the *environment model*, which is used in many programming languages, including Scheme. An *environment* is a list of binding of variables to their definitions associated with a procedure or an expression.

An environment consists of a list of *frames*. A frame is a table of variables and their values. At the end of the frame list is the *global environment*, in which the primitive identifiers and user-defined global definitions (variables or procedures) are bound to their values (data or operations). The global environment consists of two frames: The frame of primitive identifiers and the frame of user defined global identifiers.

A new frame is created when an expression is evaluated. When an expression is evaluated, a new frame with local definitions and parameter bindings is attached to the beginning of "*the environment associated with the procedure*" of this expression. The body of the procedure is evaluated in this environment. If a variable used in the evaluation of the body of the procedure is defined locally in the new frame of the expression, it is a *bound variable*, the value listed in the new frame will be used. If a variable is not found in the new frame of the expression, it is an *unbound*, or *free*, variable. Then, the value of this variable is searched for down through the frames in its

environment to find the first match. It a match cannot be found in the environment, an *unbound variable* error occurs.

The environment associated with every procedure is determined by its definition.

A procedure may be defined in two ways: It may be defined by a lambda expression such as:

```
( define square ( lambda ( x ) ( * x x ) ) )
```

(This is the same as:

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

which is a simplified version of the first definition.)

The definition of the procedure square is (lambda (x) (* x x)).

In this case, the environment associated with this procedure is the environment in which it is defined.

The second way to define a procedure is to define it by a procedure returned from another application expression, such as:

```
( define ( add-y x ) ( lambda ( y ) ( + x y ) ) )
( define add ( add-y 3 ) )
```

Procedure add is defined by the procedure returned from another application expression (add-y 3).

To find the definition of add, we must evaluate the expression (add-y 3). When this expression is evaluated, a new frame binding x to 3 is attached to the environment of the environment where add is defined. Then, this new environment is associated with the procedure add.

According to this frame-searching model, if an identifier is defined in more than one frame in the environment of a procedure, the "lowest" definition will be used.

The way of determining the binding of an identifier in a procedure is called *lexical scoping*, or *static scoping*. Almost all programming languages use this static scoping mechanism to determine the binding of identifiers.

Examples.

(i) Procedures without assignment:

```
( define x 3 )
( define ( square x ) ( * x x ) )
( define ( plus y ) ( + x y )
( define ( square-plus  x y )  ( - ( square x ) ( plus y ) )
> ( square-plus  4 5 )
```

In the global user frame, we have an identifier *x* bound to a value 3, an identifier *square* bound to a procedure

```
( lambda  ( x )  ( *  x  x ) ),
```

an identifier *plus* bound to procedure:

```
( lambda ( y ) ( + x y ) )
```

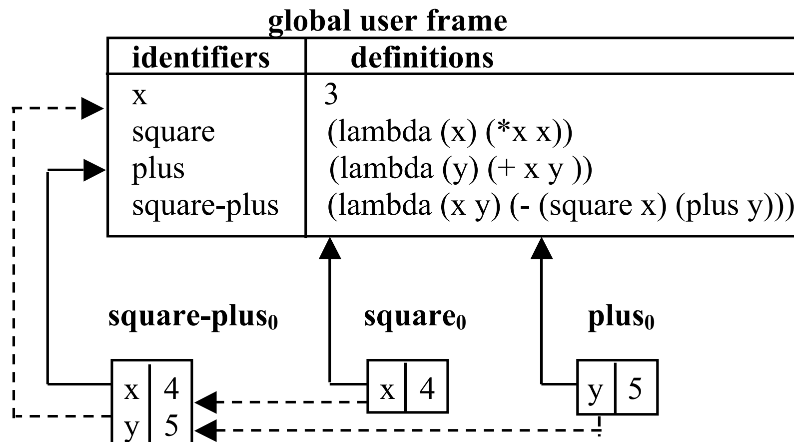
and an identifier *square-plus* bound to the procedure:

```
( lambda  ( x y )  ( -  ( square  x )  ( plus  y ) ).
```

To evaluate the expression (square-plus 4 5), a frame of the procedure square-plus is created with local bindings of x to 4 and y to 5. This procedure calls procedure square in the expression (square x) and passes the value x = 4 to this procedure. A frame associated with the evaluation of procedure square is created. In this frame, since x is passed to procedure square, x is bound to 4. This expression evaluates to 16. Then procedure square-plus calls procedure plus in the expression (plus y) and passes the value y = 5 to this procedure. A frame for the procedure plus is created. In this frame, the identifier y is bound to 5. There is also an identifier x used in procedure plus. However, this identifier is not bound in this frame. Since x is defined in the global environment, Scheme follows the static link of this frame to the global user frame, and find a binding x = 3 there. (Note that, x also has a binding to the value 4 in the frame of procedure square. If dynamic scoping were used, procedure would take x = 4 in its evaluation). Therefore, expression (plus 4) is evaluated to 5 + 3 = 8. Returning to procedure square-plus, expression (square-plus 4 5) is evaluated to 16 – 8 = 8.

In the following diagram a *static link* is to the environment associated with a procedure, and *a dynamic link* is to indicate where the procedure is called.

This situation can be illustrated by the following diagram:



In this diagram, we use solid arrows to denote the static link and the dashed line to denote the dynamic link. The subscript 0 in the names of procedures is used to distinguish different calls to the same procedure.

(ii) Nested procedures

Suppose we have the following definitions:

```
( define x 3 )
( define y 2 )
( define ( minus y ) ( - y x ) )
( define ( plus-minus  x y )
        ( define plus  ( lambda ( x ) ( + x y ) ) )
        ( * ( minus x ) ( plus y ) )
)
```

Evaluate the following expression:

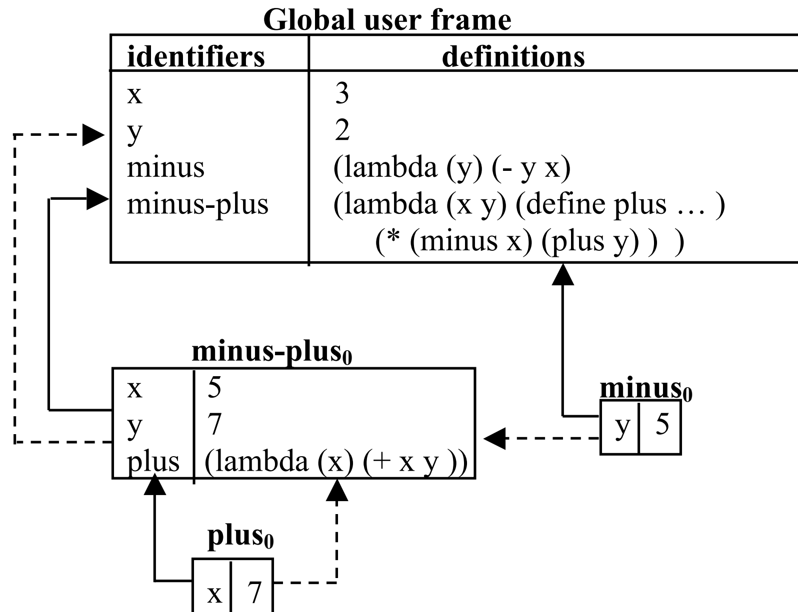
```
> ( plus-minus 5 7 )
28
```

According to the environment model, this expression is evaluated as follows:

Procedure plus-minus calls procedure minus and passes x = 5 to this procedure. The formal parameter y in procedure minus takes the value of the argument 5. It uses the value of x in the global environment, so that x = 3. The value returned from procedure minus is 5 – 3 = 2. Then, procedure minus-plus calls procedure plus, and passes the value y, i.e., 7, to procedure plus, which is defined inside procedure minus-plus. The formal parameter x of procedure plus takes the value of the argument, i.e., 7. Variable y is a free variable in procedure plus. According to the principle of static scoping, the

value y is found in the frame of procedure minus-plus, i.e., 7. Therefore, expression (+ x y) in procedure plus has a value 7 + 7 = 14. Finally, the expression (minus-plus 5 7) yields a value 2 * 14 = 28.

The diagram of this environment model is as follows:



(iii) Procedures return a procedure without mutable data.

```
( define x 2 )
( define ( f w ) ( lambda ( y ) ( + x y w ) ) )
( define ( g w )
   ( define x 10 )
   ( define h ( f 3 ) )
   ( h 5 )
)

(g 7)
```

(iv) Procedures with mutable data.

The following the program is that one that we used to show that the substitution model does not work.

```
> ( define   ( add-to x )
         ( lambda ( y )  ( set! x ( +  x y ) )   x )
  )
> ( define accum  ( add-to  20 ) )
```

```
> ( accum 5 )
25
> (accum 10 )
35
```

Expend the definition in the lambda form. We have

```
> ( define add-to
    ( lambda ( x )
        ( lambda ( y )  ( set! x  ( +  x  y ) )  x )
        )
     )
  )
```

At the beginning, we have the global user frame with the bindings of two identifiers *add-to* and *accum*. Procedure *add-to*, it is bound to a procedure:

```
( lambda ( x )  ( lambda ( y ) ( set!  x  ( + x y ) ) x ) )
```

This procedure accepts a parameter *x*, and returns an *anonymous* procedure:

```
( lambda ( y )  ( set!  x  ( +  x  y ) )  x )
```

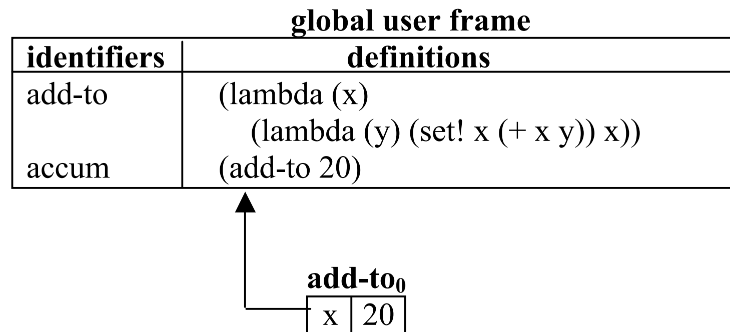
This anonymous procedure accepts a parameter *y*, sets *x* to *x* + *y*, and returns *x*.

Procedure *accum* is defined as (add-to 20). To find the definition of accum, the expression (add-to 20) is evaluated. This creates a new frame that binds a variable x to a value 20. This frame is attached to the global environment. According to the definition of add-to, the procedure returned from (add-to 20) is:

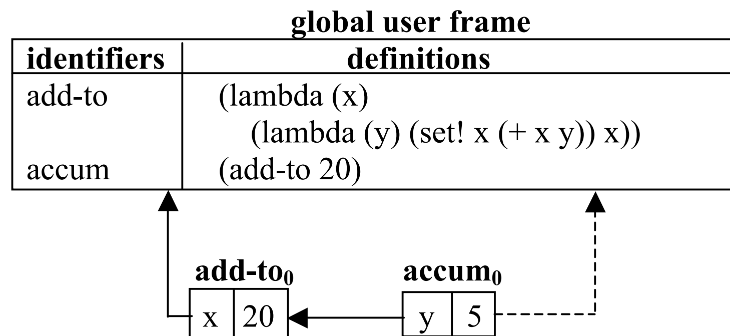
```
(set! x (+ x y) x),
```

and this procedure is to be evaluated in the new environment. We say that this new frame is *captured* by the procedure accum.

This situation can be illustrated by the following diagram:



When expression (accum 5) is called, a frame constructed with a binding of y to 5. Since accum is *defined* as add-to 20. This static link of this frame goes to the frame add-to$_0$, as illustrated by the following diagram:



Since the anonymous procedure does not have a binding of variable x, it takes the value of x from the nesting frame of add-to$_0$. The evaluation of accum$_0$ changes the value of x to 25, and returns the new value x, i.e., 25, along the dynamic link to the caller. After that, frame accum$_0$ is removed. The value of x in add-to$_0$ is 25.

The second time when (accum 10) is called, a new frame accum$_1$ is created and attached to add-to$_0$ again. It will take the new value of x, 25, and return 35.

(v) Recursive procedures

A recursive procedure calls itself. The following example shows how the environment model works to evaluate a recursive procedure expression.

```
( define  ( square x ) ( * x x ) )
( define ( sum-of-squares  n )
    ( if ( = n 0 ) 0
```

```
        ( + ( sum-of-squares ( - n  1 ) )  ( square n ) )
    )
)
> ( sum-of-squares 3 )
```

In the global environment, we have the definition of the procedure sum-of-squares. When (sum-of-squares 3) is evaluated, it calls itself with argument n = 2, and it calls a procedure square with x = 3. In turn, the second copy of sum-of-squares calls itself with argument n = 1, and calls procedure square with x = 2. Then, the third copy of sum-of-squares calls itself with n = 0, and it calls procedure square with x = 1. Finally, (sum-of-squares 0) return 0.

Back to the third copy of sum-of-squares, it returns 0 + 1 = 1. The second copy returns 1 + 4 = 5. Finally, the first copy of procedure sum-of-squares returns 5 + 9 = 14.

The diagram of this evaluation is left as an exercise.

§ 4. USING ASSIGNMENT WITH DATA STRUCTURE TO IMPLEMENT OBJECTS

Objects are data structures that have a number of *state variables*, which will change from time to time. To implement objects, we need to combine list structures and assignment. From the programmer's point of view, an object is just a list with mutable members.

The following is an example to create a class of objects called customer:

Constructors:

```
( define  ( make-name  first  last )  ( list  first  last ) )
( define  ( make-address number street city )  ( list number  street  city ) )
( define  ( make-customer  id  name  address  telephone )
          ( list  id name  address  telephone )
)
```

Selectors:

```
( define  ( get-id  customer)  ( car customer ) )
( define  ( get-name  customer )  ( cadr customer ) )
( define  ( get-address customer )  ( caddr customer ) )
( define  ( get-telephone  customer )  ( cadddr customer ) )
```

Mutators:

```
( define  ( change-name  customer new-name )
          ( set-car!  ( cdr customer )  new-name )
)
( define  ( change-address  customer new-address )
          ( set-car!  ( cddr customer )  new-address )
)
 ( define  ( change-telephone  customer new-telephone )
          ( set-car!  ( cdddr customer )  new-telephone )
)
```

Now we may define a customer such as:

```
( define john ( make-customer
                1234
                ( make-name  "John" "Smith" )
                ( make-address  12 "Baseline Avenue" "Ottawa" )
                888-6666
```

```
)

> john
(1234 ("John" "Smith") (12 "Baseline Avenue" "Ottawa") 888-6666)
> ( get-name john )
("John" "Smith")
> ( change-telephone  john  123-4567)
> john
(1234 ("John" "Smith") (12 "Baseline Avenue" "Ottawa") 123-4567)
```

§ 5. STREAMS

Assignment is used *to record the change of a variable over time*. If we record the values of the variable at different times, we have a sequence of values. If we use this sequence of values to represent the variable, there is no need for mutable variables or assignment.

The problem of using this approach is that we may need a very long, even infinitely long, list to store all observed values of a variable. Fortunately, in many cases, we do not need *all* of the values this variable takes. We may stop at a point when the value that we need is observed. Therefore, we need to construct only a part of the list. This approach is achieved in Scheme using a data structure called the *stream*.

From the user's point of view, a stream is just a list. But it is implemented slightly differently from a list: The members of a stream are not all evaluated at the time when the stream is created.

Another case where stream is useful is to write a generic algorithm that processes the members of a sequence of data items until a certain point. For instance, we want to add up all the squares of prime numbers that are less than 1000. The steps can be formulated as follows:

- enumerate all integers < 1000
- filter out primes
- find the square of each prime
- accumulate the sum

In short, we have an "enumerate - filter - map - accumulate" flow.

This problem can be solved by constructing a list of all positive integers less than 1000, and using the procedures mapping, filtering and accumulating that we defined before. However, what if the problem is changed to find the sum of squares of prime numbers until the sum is less than or equal to 1,000,000? In this case, we do not know in advance how many integers we would have to put in the list. We should construct the list step by step when the accumulation is carried out, and stop building the list when the sum exceeds 1,000,000.

We need an empty stream to start with. The empty stream is just an empty list.

```
( define empty-stream ( ) )
```

Use the following predicate to check whether a stream is empty:

```
( define empty-stream? null? )
```

Now let's see how to define and use a stream.

Suppose we have a procedure that generates the members of a list. Usually, this procedure is defined recursively with the following format:

```
( define list-generator  ( cons  first-term  ( list-generator  other-terms) ) )
```

If we want to use a stream instead of a list, we must delay the generation of the second part, which contains the terms in the cdr of the list. This is achieved by a special form with the keyword *delay*:

```
( define stream-generator  ( cons  first-term  ( delay  ( stream-generator  other terms ) ) ) )
```

Delay is not a procedure. (delay x) is a special form. If delay were a procedure, (delay x) would force x to be evaluated, which is exactly that we want to avoid. The notes use stream-cons.

Once a stream is built, we can use procedure car to access the first member:

```
( define ( head  strm )  ( car  strm ) )  // The notes use stream-car
```

To access the cdr of the stream, we have to build it. This is achieved by a procedure *force*.

```
( define ( tail strm ) ( force  ( cdr  strm ) ) )    // The notes use stream-cdr
```

Expression (force (cdr strm)) evaluates (cdr strm), which calls the stream-generator to generate the first term of cdr strm and put the rest part again in a delay form. For instance, the following defines a stream of integers starting from m:

```
( define  ( interval-stream  m  )  ( cons  m   ( delay  ( interval  ( + m 1 ) ) ) ) ) )
```

Then we have

```
> ( define x ( interval   20 ) )
> ( head x )
20
> ( head  ( tail x ) )
21
```

Now we can define procedures to process streams, such as mapping, filtering, and accumulating.

The following program finds the sum of the squares of positive multiples of 7 until the sum exceeds a given limit.

```
; filter-map-accumulate
( define  ( filter-map-accumulate  strm  sum  limit map-function )
    ( if ( >  sum  limit )  sum
        ( filter-map-accumulate
            ( tail  strm )
            ( +  ( square  ( head  strm ) )   sum )
            limit
            map-function
        )
    )
)
; stream facilities
( define  ( head  strm )  ( car  strm ) )
( define  ( tail  strm )  ( force  ( cdr  strm ) ) )
; stream generator
( define  ( multiples-of-7  n )
    ( cons  ( * n  7 )
            ( delay ( multiples-of-7  ( + n 1 ) ) )
    )
)
; application
( define  ( square  x )  ( * x x ) )
( define test-stream ( multiples-of-7 0 ) )
( define  ( test limit )  ( filter-map-accumulate  test-stream 0 limit square ) )

>  ( test 10000 )
13965
> (test 1000000)
1006460
```

Note that, using streams, we may deal with a list with infinitely many members. (Of cause, we can look at only a finite number of members). As in the previous example, the list of multiples of 7 is infinite. We can generate as many members as we need for the problem.

CHAPTER 5. A METACIRCULAR EVALUATOR

§ 1. METACIRCULAR EVALUATOR

We have been using some evaluators of Scheme (e.g., DrScheme) to create and run Scheme programs. An evaluator of a language is actually itself a program.

In this chapter, we are going to write an evaluator (interpreter) for (a simplified version of) Scheme in Scheme. An evaluator of a language written in this language itself is called a *metacircular evaluator*. The reasons for writing such an evaluator is to have better understanding of the behavior of the language, and to make it possible to change some specifications of the language. Also, this is a practice to write interpreters for other languages, in particular, new languages developed for specific purposes or enhancements of existing languages, in Scheme.

This simplified version of evaluator will support the following forms

```
self-evaluating                ; numbers
( quote <text-of-quotation> )
( set! <var> <value> )
( define <var> <value> )
( define ( <var> <parameter-1>... <parameter-n> ) <body> )
( lambda (<parameter-1> ... <parameter-n> ) <body> )
( cond  ( <predicate-1> <expression-1> )
        ( <predicate-2> < expression-2> )
        …
        ( <predicate-n> < expression-n> )
)
( procedure <argument-1> … <argument-n> )
```

§ 2. REPRESENTING THE ENVIRONMENT

An expression is evaluated in an environment. An environment in Scheme is represented by a list of *frames*. Every frame is a list of *bindings*. A binding is a pair of a *variable* and a *value*. Therefore, An environment is defined by the following definitions:

```
;; bindings - represented by a pair of a variable and a value

; constructor

( define  ( make-binding  var  value )  ( cons  var  value ) )

; accessors

( define  ( binding-var  a-binding )  ( car  a-binding ) )
( define  ( binding-value  a-binding )  ( cdr  a-binding ))

; modifier

( define  ( set-binding-value!  a-binding  new-value )
          ( set-cdr!  a-binding  new-value )
)

;; frames - represented by a list of bindings

; constructors

; initialize a frame as an empty list
( define ( init-frame ) ( ) )
; construct a frame from a list of variables and a list of values
( define ( make-frame variables values )
  ( cond
     ; if the variable list and the value list are empty,
      ; return empty list - base case
     ( ( and (null? variables)  (null? values)  ) ( init-frame ) )
     ; if one list is empty but the other is not, an error
     ( ( null? variables )    "Too many values supplied" )
     ( ( null? values )   "Too few values supplied" )
     ; bind the first value to the first variable, and recursion
     ( else ( cons ( make-binding ( car variables ) ( car values ) )
                   ( make-frame ( cdr variables ) ( cdr values ) )
            )
     )
 )
)

; add a new binding to a frame
(define ( adjoin-binding new-binding frame )
```

```
        ( cons new-binding frame )
)

; accessors
( define ( first-binding frame ) ( car frame ) )
( define ( rest-bindings frame ) ( cdr frame ) )

; predicate
( define ( empty-frame? frame ) ( null? frame ) )

; searching a frame to find a binding of a given variable.
( define ( search var frame )
    ( cond
        ( ( empty-frame? frame ) #f )
        ; frame is empty means not found, return #f
        ( ( eqv? var ( binding-var ( first-binding frame ) ) )
           ;found at the beginning of frame
           ( first-binding frame )              ; return first binding
        )
        ( else ( search var ( rest-bindings frame ) ) )
        ; otherwise, continue look at the rest bindings
        ; of the frame recursively
    )
)

;; environment - represented by a list of frames

; constructor

; initialize an environment
( define ( init-env ) ( ) )
; adding a new frame to an environment, given the new frame
( define ( add-frame new-frame env ) ( cons new-frame env ) )
; adding a new frame to an environment, given a list
; of variables and a list of values
( define ( extend-env var-list value-list env )
     ( add-frame ( make-frame var-list value-list ) env )
)

; accessors

( define ( first-frame env ) ( car env ) )
( define ( rest-frames env ) ( cdr env ) )

; modifier - change the first frame to a new frame

( define ( set-first-frame! env new-frame ) ( set-car! env new-frame )
)

; predicate

( define ( empty-env? env ) ( null?  env ) )

; searching an environment to find the binding of a variable
( define ( binding-in-env var env )
    ( if ( empty-env? env )         ; not found
         #f
```

```
        ( let ( ( binding ( search var ( first-frame env ) ) ) )
            ( if binding                ; found
                 binding
                ( binding-in-env var ( rest-frames env ) )
                ; continue
            )
        )
    )
)

; set a new value to a variable in an environment
( define ( set-variable-value! var new-value env )
     ( let ( ( binding ( binding-in-env var env ) ) )
            ; try to find the variable in env
           ( if binding                            ; found
                ( set-binding-value! binding new-value )
                "Unbound Variable for set"
           )
     )
)
; define or redefine a variable in the first frame of an environment
( define ( define-var! var value env )
     ( let ( ( binding ( search var ( first-frame env ) ) ) )
           ( if binding                   ; var is in the first frame
                ( set-binding-value! binding value )  ; re-define
                ; else, add the new binding to the first frame to
                ; have a new frame, and change the first frame to
                ; the new frame
                ( set-first-frame! env
                   ( adjoin-binding ( make-binding var value )
                                    ( first-frame env ) )
                )
           )
     )
)
```

§ 3. EVALUATE EXPRESSIONS

To evaluate an expression is to evaluate the sub-expressions and then apply the value of the *operator sub-expression* to the values of the *operand sub-expressions*.

The procedure "eval" is a dispatcher. According to different kinds of forms, it calls corresponding procedures to find their values:

```
; a procedure to evaluate an expression
( define ( eval  exp  env)
    ; evaluate expression exp in environment env
    ( cond
            ; if exp is self-evaluating, return exp
            ( ( self-evaluating?  exp )  exp )
            ; if exp is a quoted expression, find the text of exp
            ( ( quoted? exp ) ( text-of-quotation  exp  ) )
            ; if exp is a variable, call a procedure to look up
            ; the environment to find it value
            ( ( variable? exp) ( lookup-variable-value  exp  env ) )
            ; if exp is a definition, call a procedure to bind
            ; the definition to the variable in the environment
            ( ( definition? exp ) ( eval-definition  exp  env ) )
            ; if exp is an assignment, change the binding of
            ; the variable in the environment
            ( ( assignment? exp ) ( eval-assignment  exp  env ) )
            ; if exp is a lambda expression, create a procedure
            ( ( lambda? exp ) ( make-procedure  exp  env ) )
            ; if exp is a conditional expression,
            ; look up the clauses in the environment
            ( ( conditional? exp ) ( eval-cond ( clauses exp ) env ) )
            ; if exp is an application expression, find the operands
            ; and the operator, then call eval recursively
            ( ( application? exp)
              ( apply ( eval ( operator exp ) env )
                      ( list-of-values ( operands exp ) env )
              )
            )
            ( else ( error "Illegal Expression" exp ) )
     )
)
```

Now we look at the different categories of form:

(i) The only self-evaluating expressions are numbers.

```
; a predicate to check if an expression is self-evaluating
( define ( self-evaluating? exp ) ( number? exp ) )
```

where number? is a built-in predicate.

(ii) A quote expression starts with the key word "quote":

```
; a predicate to check if an expression is a quote expression
( define  ( quoted?  exp )
   ( if  ( atom?  exp )  #f          ; expression is not a list
         ( eqv?  ( car exp )  'quote )
   )
)
```

Here, predicate atom? is defined as

```
; define predicate atom?
( define ( atom? x ) ( not ( pair? x ) ) )
```

where pair? is a pre-defined predicate. it returns true if x is pair (or a list).

To find the text of a quoted expression, look at the cadr of the expression:

```
; a procedure to find the text of a quote expression
( define  ( text-of-quotation  exp )  ( cadr exp ) )
```

(iii) To test whether an expression is a variable, use the built-in predicate symbol?:

```
; a predicate to check if an expression is a variable
( define  ( variable?  exp )  ( symbol?  exp ) )
```

If the expression is a variable, use the procedure binding-in-env to return the binding:

```
; a procedure to find the value of a variable in an environment
( define ( lookup-variable-value var env )
   ( let ( ( binding ( binding-in-env var env )  ) )
         ( if  binding                    ; found
              ( binding-value binding )
              "Unbound variable for lookup"
         )
   )
)
```

(iv) A definition starts the keyword "define":

```
; a predicate to check if an expression is a definition
( define ( definition?   exp )
     ( if ( atom?  exp )  #f              ; expression is not a list
         ( eqv? ( car  exp ) 'define )
     )
)
```

A definition form has two formats: The first is (define <*variable*> <*definition*>) and the second is (define (<*variable*> <*parameter-list*>) <*procedure-definition*>). The second format is used to replace a definition of a procedure of the first format using a lambda form.

In the first case, the variable is the second member of the expression, and the body of the definition is the third member of the expression. In the second case, the variable is the first member of the second member of the expression. The body of the definition is a lambda expression: (lambda (<*parameter-list*>) <*procedure-definition*>). The parameter list is found as the cdr of the second member of the expression. The procedure definition is the third member of the expression.

```
; a procedure to find the variable (i.e., identifier)
; that a definition defines
( define ( definition-var  exp )
    ( if ( variable? ( cadr exp ) )
    ; if the second member of the expression is a variable
          ( cadr exp )  ; the variable is the second member
          ( caadr exp ) ; otherwise, the variable is the first
                        ; member of the expression
    )
)
```

```
; a procedure to find the body of the definition
( define ( definition-value exp )
    ( if ( variable? ( cadr exp ) )    ; a definition of a variable
         ( caddr  exp )       ; the body is the second member of exp
         ( cons 'lambda       ; construct a lambda expression
                ( cons ( cdadr exp ); parameter list is cdr of cadr
                       ( cddr exp ) ; body is the third member as a
list
                )
         )
    )
)
```

The purpose of procedure *eval-definition* is to add (or change) the definition of an identifier to the environment. To do this, use procedure definition-var to find the variable in the expression, then use procedure definition-value to find the body of the definition and **call procedure eval to evaluate the body of the definition.** Finally, call procedure define-var! to add (or change) the definition in the environment. Note that, when eval(uating) the body of the definition, a new frame may attached to the environment. But, the definition is added to the old environment.

```
; a procedure to add the definition of an identifier to the environment
( define ( eval-definition  exp  env )
     ( let ( ( var ( definition-var exp ) ) )
      ( define-var! var ( eval ( definition-value exp ) env ) env )
      var                                       ; returns var
     )
)
```

(v) An assignment form starts with the keyword set!.

```
; a predicate to check if an expression is an assignment
( define ( assignment? exp )
```

```
    ( if ( atom? exp )  #f          ; expression is not a list
         ( eqv? ( car exp ) 'set! )
    )
)
```

We need to find the variable and the new-value of the variable given by the assignment:

```
; procedures to fine the variable and the new value of the assignment
( define  ( assign-var exp ) ( cadr exp ) )
( define  ( assign-value exp ) ( caddr exp ) )
```

Procedure eval-assignment is used to set the new value to the variable. The new value may be represented by an expression. Call procedure eval to find the new value:

```
; a procedure to assign the new value to the variable
( define ( eval-assignment exp env )
      ( let ( ( new-value ( eval ( assign-value exp ) env ) ) )
      ; find the new value
            ( set-variable-value! ( assign-var  exp ) new-value env )
            ; assign
            new-value                           ; return new value
      )
)
```

(vi) A lambda expression is specified by the keyword lambda:

```
; a predicate to check if an expression is a lambda expression
( define ( lambda? exp )
     ( if ( atom?   exp )     #f
           ( eqv?  ( car  exp )  'lambda )
     )
)
```

The make-procedure procedure is used to create a procedure, as a list. The first member of the list is a symbol *procedure* to identify that this is a procedure, the second component is the expression representing a procedure (which consists of three parts: the symbol *'lambda*, the parameter list, and the body of the procedure), and the third part is the environment associated with the lambda expression.

```
; a procedure that makes a procedure, it returns a list
; containing the body and the environment of the procedure
( define ( make-procedure  exp  env )
      ( list 'procedure exp env )
)
```

To access different components of a procedure, we have the following procedures:

```
; the body of the procedure is the third part,
; of the second member of the procedure list.
; Note that operator cddr returns a list, which may contain
; more than one expression
( define ( get-proc-body proc ) ( cddr ( cadr  proc ) ) )
```

```
; the parameter list is the second member of the second member
; of the procedure list
( define ( get-parameters  proc ) ( cadr ( cadr  proc ) ) )
; the environment is the third part of the procedure list
( define ( get-proc-env proc ) ( caddr  proc ) )
```

(vii) Conditional expressions are specified by the keyword cond:

```
; a predicate to check whether an expression is a conditional
expression
( define ( conditional? exp )
     ( if ( atom? exp ) #f
          ( eqv? ( car  exp ) 'cond )
     )
)
```

There are a number of clauses in a conditional expression. To retrieve these clauses, we use the following procedures:

```
; an accessor to access clauses in a conditional expression
( define ( clauses exp ) ( cdr exp ) )
; accessors to individual clauses in the clause list
( define ( first-clause clause-list ) ( car clause-list ) )
( define ( rest-clauses clause-list ) ( cdr clause-list ) )
; a predicate to mark the end of a clause list
( define ( no-clauses?  clause-list ) ( null?  clause-list ) )
```

Each clause has two parts: the condition, and the value.

```
; accessors to find the predicate and the action to be taken if the
predicate is true
( define ( predicate a-clause ) ( car a-clause ) )
( define ( action a-clause ) ( cdr a-clause ) )
```

There is a special clause that does not have a condition - the else clause:

```
; a predicate to recognize the else clause
( define ( else-clause? clause ) ( eqv?  'else ( predicate clause ) ) )
```

To evaluate a conditional expression, we have to look at each of the conditions:

```
; evaluate the conditional expression
( define ( eval-cond  clause-list   env )
   ; clause-list is obtained from procedure (slauses exp)
   ( cond ( ( no-clauses?  clause-list ) #f )
          ; the first clause is an else-clause or the
          ; predicate of the first clause is true
          ( ( or ( else-clause? ( first-clause  clause-list ) )
                 ( eval ( predicate ( first-clause clause-list ) ) env )
            )
            ( eval-sequence ( action ( first-clause clause-list ) ) env
)
```

```
           ; procedure eval-sequence evaluates a sequence of
expressions
         )
         ; else, continue
         ( else ( eval-cond ( rest-clauses clause-list )  env ) )
      )
)
```

(viii) An application is a procedure call, which starts with a procedure followed by a list of arguments.

```
; a predicate to check if an expression is an application
( define ( application?  exp ) ( not ( atom?  exp ) ) )

; procedures to find the operator and the operands
( define ( operator exp ) ( car exp ) )
( define ( operands exp ) ( cdr exp ) )
```

Each of the operands may itself be a compound expression. We have to evaluate the operands in an environment to get a list of arguments to be used in evaluating the expression:

```
; a procedure that converts a list of expressions to a list of values,
; and return it
( define ( list-of-values operands env )
      ( if ( null? operands ) ( )
           ( cons ( eval ( car operands ) env )
                  ( list-of-values ( cdr operands )  env )
           )
      )
)
```

The procedure "apply" treats two different cases according to that the procedure is a primitive procedure or a compound procedure. If it is a primitive procedure, directly evaluate it with the given set of arguments by a procedure *apply-primitive-proc*. If it is a compound procedure, use a procedure *eval-sequence* to evaluate it. This procedure *eval-sequence* takes two parameters: the body of the procedure returned from a procedure *get-proc-body*, and an environment obtained by extending the base environment of the procedure by adding a new frame with the bindings of parameters to arguments. This environment is constructed by the procedure *extend-env*, which accepts the parameters of the procedure (returned from a procedure *get-parameters*), the arguments, and the base environment of the procedure obtained from the procedure *get-proc-env*.

```
; a procedure to apply a procedure to its arguments
( define ( apply proc args )
   ; proc is obtained by ( eval ( operator exp ) env )
   ; args is obtained by ( list-of-values ( operand exp ) env )
   ( cond
      ( ( primitive-proc?  proc )   ; the procedure is primitive
            ( apply-primitive-proc proc args ) )
      ( ( compound-proc? proc )     ; the procedure is compound
        ( eval-sequence ( get-proc-body  proc )
```

```
                        ; add a new frame in the environment
                        ; of the procedure and return the new
                        ; environment
                        ( extend-env ( get-parameters  proc )
                                     args
                                     ( get-proc-env proc)
                                     ; Note the environment is the
                                     ; environment of the procedure,
                             but
                                     ; not the environment in which
                             this
                                     ; expression is evaluated
                        )
        )
      )
      ( else ( error  "Illegal Application" ) )       ; neither
   )
)
```

Procedure eval-sequence is defined as follows:

```
; evaluate a sequence of expressions
( define ( eval-sequence body env )
   ( if ( last-expression?   body )
         ( eval   ( first-expression   body )   env )
         ( begin
            ( eval ( first-expression   body  )   env )
            ( eval-sequence ( rest-expressions body ) env )
         )
   )
)

; accessors to access individual expressions in a sequence
( define  ( last-expression?  body )  ( null?  ( cdr  body ) ) )
( define  ( first-expression  body )  ( car  body ) )
( define  ( rest-expressions  body )  ( cdr  body ) )
```

§ 4. RUNNING THE EVALUATOR

First, we need to set up a global primitive environment. This global primitive environment has a frame that binds the names of primitive procedures used in this evaluator to the real primitive procedures defined in Scheme.

```
; defining primitive procedures
( define   primitive-procedure-names
     '( car  cdr  cons  eqv? null? + - * / > < = and or not )
      ; more primitives could be added here
)
( define   primitive-procedure-objects
     '( ( primitive car )
        ( primitive cdr )
        ( primitive cons )
        ( primitive eqv? )
        ( primitive null? )
        ( primitive + )
        ( primitive - )
        ( primitive * )
        ( primitive / )
        ( primitive > )
        ( primitive < )
        ( primitive = )
        ( primitive and )
        ( primitive or )
        ( primitive not )
        ; more primitives could be added here
      )
)
```

According to this definition, the name of a primitive procedure is recognized by the following predicate:

```
; predicates to check if a procedure is a primitive procedure
; or a compound procedure
( define ( primitive-proc? proc ) ( eqv? 'primitive ( car proc ) ) )
( define ( compound-proc? proc ) ( not ( primitive-proc? proc ) ) )
```

The primitive procedures can be applied by the following procedures:

```
; procedures to apply primitive procedures
( define ( apply-primitive-proc  proc  args )
; args is a list of arguments
    ( let ( ( id ( primitive-id proc ) ) )
          ( cond
            ( ( eqv? id 'car ) ( car  ( car  args ) ) )
            ; In this case, args is a list with a single list.
            ; (car args) returns the list, and car (car args)
            ;returns car of the list.
```

```
            ( ( eqv? id 'cdr ) ( cdr  ( car args ) ) )
            ( ( eqv? id 'cons ) ( cons ( car args ) ( cadr args ) ) )
            ( ( eqv? id 'null? ) ( null? ( car args ) ) )
            ( ( eqv? id '+ ) ( add args ) )
            ( ( eqv? id '- ) ( - ( car args ) ( acdr args ) ) )
            ( ( eqv? id '* ) ( multiply args ) )
            ( ( eqv? id '/ ) ( / ( car args ) ( cadr args ) ) )
            ( ( eqv? id '> ) ( > ( car args ) ( cadr args ) ) )
            ( ( eqv? id '< ) ( < ( car args ) ( cadr args ) ) )
            ( ( eqv? id '= ) ( = ( car args ) ( cadr args ) ) )
            ( ( eqv? id 'and ) ( and ( car args ) ( cadr args ) ) )
            ( ( eqv? id 'or ) ( or ( car args ) ( cadr args ) ) )
            ( ( eqv? id 'not ) ( not ( car args ) ) )
            ;  more primitives should be added here
          )
    )
)
( define ( add args )
   ( if ( null? args )
        0
        ( + ( car args ) ( add ( cdr args ) ) )
   )
)
(define ( multiply args )
  ( if ( null? args )
       1
       ( * ( car args ) ( multiply ( cdr args ) ) )
  )
)
```

The name of the primitive procedure is retrieved by the procedure:

```
; find the name of a primitive procedure
( define ( primitive-id proc ) ( cadr proc ) )
```

The global primitive environment is constructed by the procedure extend-env:

```
; construct the global environment
( define ( set-global-environment )
   ( let ( ( initial-env ( extend-env primitive-procedure-names
                                      primitive-procedure-objects
                                      ( )
                         )
           )
         )
         ( define-var! '#f #f initial-env )          ; define false
         ( define-var! '#t ( not #f ) initial-env )  ; define true
         initial-env                ; return initial environment
   )
)
```

Finally, we need to define a user interface:

```
; user interface
( define  ( run )
```

```
      ( newline )              ; outputs a new-line
      ( display "==> " )
      ; a cursor different from the original evaluator
      ( let ( ( x ( read ) ) )
         ( if ( and ( symbol? x )
                    (string=? ( symbol->string x ) "quit")
              )
              "Bye"
              ( begin
                ( print ( eval x global-env ) )
                ( run )
              )
         )
      )
)

; define the global environment
(define global-env ( set-global-environment ) )
```

To run this evaluator, type at the original evaluator cursor

> (run)

Then you will see the cursor of the new evaluator:

==>

Enter an expression at the new cursor, it will respond as you designed.

Enter *quit* at the cursor will quit from your evaluator.

CHAPTER 6. INTRODUCTION TO PROLOG

§ 1. INTRODUCTION

Prolog program = database + question.

Prolog usually works interactively with the programmer (the *user*) through an interpreter. The user establishes a *database* (also called a *knowledge base*) consisting of *clauses*, which are either *facts* or *rules*. When the program is run, the Prolog interpreter prompts for a *question* (also called a *query*). If the user inputs a question after the prompt ('?-'), Prolog gives an answer: 'yes' or 'no', and, if the answer is 'yes', it identifies the variables in the question.

Prolog compilers are also available to create executable program files, but this will not be discussed in this course. We will use SWI-Prolog, an interpreter.

An Example.

The database:

```
parent(tom, bob).                 % Tom is a parent of Bob
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).

male(tom).                        % Tom is male
male(bob).
male(pat).
female(liz).                      % Liz is female
female(ann).

father(X, Y) :-                   % X is the father of Y, if
        parent(X, Y),             % X is a parent of Y, and
        male(X).                  % X is male

predecessor(X, Y) :-              % X is a predecessor of Y, if
        parent(X, Y).             % X is a parent of Z
predecessor(X, Z) :-              % or
        parent(X, Y),             % X is a parent of Y, and
        predecessor(Y, Z).        % Y is a predecessor of Z
```

<u>Questions</u>:

```
?- parent(bob, ann).

yes

?- parent(pat, bob).

no

?-  parent(bob, ann), male(bob).

yes

?- parent(bob, ann), female(bob).

no

?- female(X).

X = liz->;
X = ann->;
no

?- parent(tom, X).

X = bob->;
X = liz->;
no

?- parent(X, Y), female(Y).

X = tom
Y = liz->;
X = bob
Y = ann->;
no

?- parent(X, _).

X = tom->;
X = tom->;
X = bob->;
X = bob->;
no
```

```
?- parent(_, X), parent(X, _).

X = bob->;
X = bob->;
no

?- father(Who, bob).

Who = tom->;
no

?- father(bob, Who), male(Who).

Who = pat->;
no

?- predecessor(bob, pat).

yes

?-  predecessor(tom, Who).

Who = bob->;
Who = liz->;
Who = ann->;
Who = pat->;
no

?-  predecessor(Who, ann).

Who = bob->;
Who = tom->;
no
```

The symbol -> is used to prompt if the user needs alternative answers. If the user types a semicolon after ->, Prolog generates the next alternative answer. But the interface may vary from interpreter to interpreter. SWI-Prolog does not print the -> prompt; it simply waits for a user to input a semi-colon.

§ 2. BASIC CONCEPTS AND SYNTAX RULES

(i) A database consists of *facts* and *rules*. Facts and rules are *clauses*. Every clause ends with a period.

(ii) A *fact* states a known fact, which consists of a *predicate* and a number of *arguments* included in parentheses. No space is between the predicate and the left parenthesis, but you may have space between parentheses. The number of arguments in a fact is called the *arity* of the predicate. **Predicates with the same name but different arities are regarded as different predicates**. To specify the arity of a predicate, a predicate may be denoted with its parity as *predicate / arity*. For instance, the predicate *parent* can be denoted as parent/2, and the predicate *male* can be denoted as male/1. We may define another predicate parent/3, which is different from parent/2. An argument of a fact is a data object, called a *term*, which may be a *constant*, a *variable* or a *structure*.

(iii) A constant can be an *atom* or a number.

An atom represents a specific data object or a relation between data objects. A constant atom must start with a lowercase letter, such as john, or include in single quotes, such as 'John'. It may consist of letters, digits, and underscores. An atom representing a relation consists of symbols, such as "?-" and ":-".

Most Prolog interpreters support both integer or real numbers, but some Prolog interpreters may only recognize integers. SWI-Prolog recognizes both.

(iv) Variables are strings of characters, digits, and underscores, starting with an uppercase letter, or an underscore. (Variables starting with underscores are not recommended). The scope of a variable is the clause. In other words, two variables of the same name in different clauses are not related to each other.

An underscore alone is used as an *anonymous variable*. Anonymous variables in the same clause may have different values. Anonymous variables in the question will not be identified in the output.

When a variable is given a value (a constant), it is *instantiated.* Any occurrence of the same variable in the same clause is also instantiated to the given value.

(v) A structure is a compound data object with a name, called a *functor*, and a number of *components*. Structures are to be introduced later.

(vi) A *rule* states a fact with conditions. A rule consists of the *head*, which is the conclusion fact, and the *body*, i.e., a list of facts which are conditions, separated by ':-', the *'neck symbol'*. More than one rule with the same head are understood as being

connected by *or*. Conditions in the body separated by comma(s) are understood as connected by *and*. Note that the order of a sequence of rules with the same name and the order of the conditions in the body of a rule is significant.

(vii) A *question* is a conjectured rule (with an anonymous head) to be verified. The question is given after the *Prolog prompt* "?-". If there is more than one condition in the question, separated by comma(s), then it is in a *conjunctive form*.

If a program is intended only for a single question, then we may put this question as the body of a rule without arguments. Then we can use the head of this rule to activate the program. For instance, if a program is always used to check if a conditions X and Y are both true, we may add a rule:

```
go :- X, Y.
```

in the database. Then, each time we can just type 'go' at the Prolog prompt to get an answer yes or no.

(viii) The symbol % is used to start a comment, which is effective until the end of the line, or use /* ... */ pair to mark what ever in between as a comment. A comment is ignored by the interpreter.

§ 3. SOME BUILT-IN FACILITIES

(1) Arithmetic

Prolog uses +, –, *, /, and 'mod' (the remainder) to construct arithmetic expressions. But, in Prolog, **an expression is not evaluated**. To evaluate an expression, Prolog provides a built-in operator '*is*'. Predicate 'is' is an infix operator that accepts two operands: A variable and an expression. This operator is used as in the following example:

```
?- X is 3 + 4 / 2.
X = 5 ->
```

The goal with operator 'is' will fail when it is re-satisfied during backtracking (to be explained later).

(2) Relational Operators

Prolog uses operators = and \= to compare two names or two numbers. These operators are used as in the following examples:

```
?- john = john.
yes
?- X = john.
X = john ->;
no
?- X is 3, X = Y.
X = 3
Y = 3 ->;
no                          % because condition X = 3 is re-satisfied
?- X = 3, Y = 4, X = Y.
no
?- X = 3, Y = 4, X \= Y.
yes
```

In the first example of this chapter, if we add a predicate 'sibling' defined as follows:

```
sibling(X, Y) :- parent(A, X), parent(A, Y).
```

The question

```
?- sibling(X, Y).
```

will give answers:

```
X = bob
Y = bob ->;

X = bob
Y = liz ->;

X = liz
Y = liz ->;

...
```

To avoid having a person as its own sibling, we should write the predicate 'sibling' as:

```
sibling(X, Y) :- parent(A, X), parent(A, Y), X \= Y.
```

Prolog also uses >, <, >=, and =< (not as other languages that use <=) to compare numbers. The condition X \= Y means that *X should be different from Y*.

A goal with a relational operator will fail when it is re-satisfied.

(3) Simple I/O

Prolog uses built-in predicates 'read' and 'write' to do simple input and output of terms as in the following examples:

```
?- write(john).

john
yes

?- write(32).

32
yes

?- read(X), write(X).

john.          % This is the input from the user ended with a period.
john           % The program writes the input
X = john ->;   % The program identifies the variable.

no
```

A goal with predicate 'read' or 'write' will fail if it is re-satisfied. For instance,

```
?- read(X), write(X), X > 4.
5.            % the input
5             % the output
X = 5 ->;     % succeeded

no            % backtrack and fail
```

(4) Declarative and procedural meanings of Prolog programs

Prolog programs have both *declarative* and *procedural* meanings.

On one hand, a Prolog program gives a description of the known facts and the relationships between facts (by rules). The program *describes* the problem.

On the other hand, the order of conjectured facts in the question, the order of facts and rules, and the order of conditions in the body of a rule, determine the way Prolog searches the database, and therefore specifies the processing procedure.

§ 4. THE GOALS

A *goal* is a conjectured fact that is to be *satisfied*. A goal is satisfied if:

(1) it *matches* a fact in the database, or

(2) it matches the head of a rule, and the conditions in the body of the rule, taking as goals themselves (called the *subgoals* of the original (*parent*) goal), are satisfied.

When Prolog is trying to satisfy a goal, this goal is *called*. If a goal is satisfied, it *succeeds*; otherwise, it *fails*.

At the beginning of the execution of a program, the goal is the question (regarded as a rule with an anonymous head). It succeeds if all the conjectured facts in the question (taken as subgoals) are satisfied; otherwise, it fails. If this goal fails, the program fails, and Prolog answers 'no'. If this goal succeeds, the program ends successfully, Prolog answers 'yes' and outputs the values of all (non-anonymous) variables in the question.

According to this structure, the goals and subgoals can be represented by a tree.

Remarks:

(i) The goal-subgoal relation is determined dynamically during the execution of the program. A goal is reduced to its subgoals only if it matches the head of a rule, and then the conditions in body of the rule become subgoals.

(ii) The order of the subgoals is determined by the order of conditions in the body of the rule. *This order is significant*.

A goal *matches* a fact in the database if:

(i) they have the same predicate, and

(ii) the arguments in the goal matches the arguments in the fact in the same order.

Two arguments *match* each other if:

(i) both arguments are the same constant, (an instantiated variable is regarded as a constant), or

(ii) at least one argument is an uninstantiated variable.

In the second case, if a variable matches a constant, the variable is *instantiated*, i.e., it is replaced by the matching constant; if a variable matches another variable, these two variables are *unified*, i.e., they are regarded identical.

Once a variable in a clause is instantiated or unified, it is instantiated or unified in the same way in other parts of the clause.

§ 5. SCANNING AND BACKTRACKING

When a goal is called, Prolog searches the database **from the top down**. If it matches the head of a rule, then the conditions in the rule are called one by one as subgoals **from left to right**. This procedure is called *scanning*. If a subgoal *succeeds*, i.e., a match is found, then:

(i) if the subgoal is the right-most subgoal of the parent goal, then the parent goal succeeds;

(ii) if the subgoal is not the right-most subgoal of the parent goal, the next subgoal is called.

If a subgoal *fails*, i.e., no match was found, then:

(i) if the subgoal is the left-most subgoal of a parent goal, then the parent goal fails;

(ii) if the subgoal is not the left-most subgoal of the parent goal, then the previous subgoal on its left is *resatisfied*. This is called *backtracking*.

When a goal is to be resatisfied, any variable instantiated during the last match are *uninstantiated*, and Prolog searches the database again starting from the clause where the last match was found. If the clause was a fact, this match is abandoned, and the search continues from the next clause. If the clause was a rule, then the conditions in the body of the rule are to be re-satisfied **from right to left**.

§ 6. EXAMPLES

(1) The program:

```
is_integer(0).
is_integer(X) :- is_integer(Y), X is Y + 1.

?- is_integer(X).
```

The output:

```
X = 0 ->
X = 1 ->
X = 2 ->
...
```

Explanation:

The first call to the goal is_integer(X) matches the first clause, which generates the answer X = 0.

When it is resatisfied, the second clause is called. It, in turn, calls is_integer(Y) in its body. This instantiates Y to 0 by the first clause. Then X = Y + 1 = 1. The second answer results.

When the second clause is re-satisfied again, Y is uninstantiated, and this goal is resatisfied. Since the first clause has been used, this goal is_integer(Y) calls the second clause. In this case, the variable X in the second goal is unified to Y, and the Y in the body becomes a new variable, say Y1. When the first condition in the body, namely is_integer(Y1), is called, by the first clause, Y1 is instantiated to be 0. Then Y (which is unified with X) is 1, and X becomes 2.

We may represent this processing procedure in a layered table:

```
                goal :   is_integer(X).
Layer   0       is_integer(0)                    called
        0       is_integer(0)                    succeeds

output X = 0.

                goal :   is_integer(X)            resatisfied
Layer   0       is_integer(X)
```

```
                :- is_integer(Y), X is Y + 1.        called
        1               is_integer(Y)                called
        1               is_integer(0)                called
        1               is_integer(0)                succeeds
                        result : Y = 0
        1               X is Y + 1                   called
                        result : X = 1
        0       is_integer(X)
                :- is_integer(Y), X is Y + 1         succeeds
```

output X = 1.

```
                goal : is_integer(X)                 resatisfied
Layer   0       is_integer(X)
                :- is_integer(Y), X is Y + 1         resatisfied
        1               X is uninstantiated in X = Y + 1
        1               is_integer(Y)                resatisfied
        1               is_integer(Y)
                        :- is_integer(Y1), Y is Y1 + 1 called
        2                       is_integer(Y1)       called
        2                       is_integer(0)        called
        2                       is_integer(0)        succeeds
        2                       result : Y1 = 0
        2                       Y = Y1 + 1           called
                                result : Y = 1
        1               is_integer(Y)
                        :- is_integer(Y1), Y is Y1 + 1 succeeds
        1               X = Y + 1                    called
                        result : X = 2
        0       is_integer(X)
                :- is_integer(Y), X is Y + 1         succeeds
```

output X = 2.

(2) Going through a maze.

```
go(a, b).               % One may go from a to b.
go(a, c).
go(c, d).
go(c, e).
reach(X, Y) :- go(X, Y).
reach(X, Y) :- go(X, Z), reach(Z, Y).
```

?- reach(a, e).

```
Layer   0       reach(a, e) :- go(a, e)                        called
```

1	go(a, e)	called
1	go(a, e)	failed
0	reach(a, e) :- go(a, Z), reach(Z, e).	called
1	go(a, Z)	called
1	go(a, b)	matched
1	go(a, b)	succeeds
	result : Z = b.	
1	reach(b, e)	called
1	reach(b, e) :- go(b, e)	called
1	go(b, e)	called
1	go(b, e)	failed
1	reach(b, e) :- go(b, U), reach(U, e)	called
2	go(b, U)	called
	go(b, U)	failed
1	reach(b, e) :- go(b, U), reach(U, e)	failed
1	go(a, Z)	resatisfied
1	go(a, c)	matched
1	go(a, c)	succeeds
	result : Z = c	
1	reach(c, e)	called
1	reach(c, e) :- go(c, e)	called
2	go(c, e)	called
2	go(c, e)	matched
1	reach(c, e) :- go(c, e)	succeeds
0	reach(a, e) :- go(a, Z), reach(Z, e)	succeeds

output yes.

(3) Suppose we have defined the facts:

```
parent(tom, bob).          % Tom is a parent of Bob
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
```

Four potentially different ways to define the predicate *ancestor*/2 are:

(i)
```
ancestor_1(X, Z) :- parent(X, Z).
ancestor_1(X, Z) :- parent(X, Y), ancestor_1(Y, Z).
```

(ii)
```
ancestor_2(X, Z) :- parent(X, Y), ancestor_2(Y, Z).
ancestor_2(X, Z) :- parent(X, Z).
```

(iii)
```
ancestor_3(X, Z) :- parent(X, Z).
ancestor_3(X, Z) :- ancestor_3(X, Y), parent(Y, Z).
```

(iv)
```
ancestor_4(X, Z) :- ancestor_4(X, Y), parent(Y, Z).
```

```
ancestor_4(X, Z) :- parent(X, Z).
```

The first three works well with the question:

```
?- ancestor_i(tom, pat).                    % where i is 1, 2, 3
```

But the last version does not work. The first three answers this question with different search paths.

For predicate *ancestor*_1, the first clause fails. The second clause instantiates X to tom, and Z to pat. The first subgoal instantiates Y to bob, and the second subgoal tries to satisfy ancestor_1(bob, pat). Going back to the first rule, Prolog tries to satisfy parent(bob, pat). This subgoal matches the last fact. The goal succeeds.

For predicate *ancestor*_2, the first clause instantiates X to tom, and Z to pat.

Reduce to subgoals: Prolog tries to satisfy parent(tom, Y), and ancestor_2(Y, pat). Searching the database, the first subgoal instantiates Y to bob. Now the problem is to satisfy ancestor_2(bob, pat).

The first clause of ancestor_2(bob, pat) will not work, because the first subgoal finds Y is ann, and Y is pat. Neither of them will satisfy the second subgoal ancestor_2(Y, pat).

Then, move on to the second clause. The second clause is satisfied, because the subgoal parent(bob, pat) satisfies.

For *ancestor*_3, the first clause instantiates X to tom, and Z to pat. Reduce to the subgoal, Prolog tries to satisfy parent(tom, pat),which fails. Now the second clause is called. The head instantiates X to tom, and Z to pat.

Reduce to subgoals: Prolog tries to satisfy ancestor_3(tom, Y) and parent(Y, pat).

Use the first clause of predicate ancestor_3, Y is instantiated to bob. Then second subgoal parent(bob, pat) is satisfied.

For predicate *ancestor*_4, the first clause instantiates X to tom, and Z to pat. Then, the first subgoal tries to find a Y to satisfy ancestor_4(tom, Y). This matches the first clause again, and the first subgoal is again ancestor_4(tom, Y). This results an infinite loop.

Now look at another question:

```
?- ancestor_i(X, Y).
```

This question tries to find all pairs of X and Y that satisfy the predicate ancestor_*i*(X, Y). For this question, the first two definitions work well, but the last two do not work,

because both of them has the recursive call as the first predicate, which will cause an infinite loop.

In most cases, **a recursive predicate should not be the first subgoal of a rule**.

(4) The following are four different ways to define integers:

(i)
```
integer(0).
integer(X) :- Y is X - 1, integer(Y).
```

It works for a question like

```
?- integer(3).
```

For question

```
?- integer(X).  (We want to generate all integers.)
```

It does not work because, when X is not instantiated, Y is X - 1 cannot be evaluated.

(ii)
```
integer(0).
integer(X) :- integer(Y), X is Y + 1.
```

It works for both questions. If one keeps asking for alternatives with the second question, the program generates 0, 1, 2, ... without an end.

(iii)
```
integer(0).
integer(X) :- X is Y + 1, integer(Y).
```

It does not work for either question because Y is not instantiated, and X is Y + 1 does not make sense.

(iv)
```
integer(0).
integer(X) :-  integer(Y), Y is X - 1.
```

This definition does not work for either question.

CHAPTER 7. STRUCTURES AND LISTS

§ 1. INTRODUCTION TO STRUCTURES

A *structure* is an ordered collection of data objects, called the *arguments* of the structure, with a name, called the *functor* of the structure. An argument of a structure may also be a structure. Hence, in general, a structure can be represented by a tree.

A structure is referred to by a single variable to achieve data abstraction.

Two structures match if they have the same functor, and the corresponding arguments in these two structures match.

Examples

(1) A structure

```
family(
    father('John', 48),
    mother('Alan', 45),
    children(
        first('Bob', 22),
        second('Mary', 18)
            )
        ).
```

This structure matches the following goals:

(i) family(X, Y, Z).

(ii) family(father(Father, 48), _, _).

(iii) family(father(Father,_), mother(Mother,_), _children(_, Second)).

Another family may be represented by a structure as follows:

```
family(
    father('Jack', 30),
    mother('Liz', 29),
    children(
```

```
                first('Lou', 8),
                second('Rob', 5),
                third('Dave', 2)
                    )
                ).
```

Then the goal:

```
        family(_, _, children(_,_, _))
```

matches the second family but not the first family.

(2) Another structure

The layout of this structure shows its tree representation.

```
        employee(
                Id,
                name(
                        Last,
                        First
                    ),
                address(
                        Number,
                        Street,
                        City,
                        ),
                salary,
                date_of_birth(
                        Day,
                        Month,
                        Year
                            )
                  ).
```

We may use questions to retrieve information from structures.

Example. Suppose a database contains the following structures:

```
        employee(20163, names('Smith', 'John'), address(35, 'King_street', 'Ottawa'),
                        30000, date_of_birth(30, 9, 1953)).
        employee(20164, names('Palmer', 'David'), address(26, 'Queen_street', 'Ottawa'),
                        35000, date_of_birth(12, 1, 1962)).
        employee(20165, names('Ray', 'Anna'), address(1234, 'Open_street', 'Nepean'),
                        40000, date_of_birth(1, 1, 1945)).
        employee(20166, names('Black', 'Edward'), address(62, 'Small_drive', 'Hull'),
```

```
                  58000, date_of_birth(31, 12, 1938)).
employee(20167, names('Norman', 'Robert'), address(25, 'Parkway', 'Ottawa'),
                  45000, date_of_birth(15, 10, 1929)).

% Retrieving the information about employee number 20165.
?- employee(20165, Names, Address, Salary, Date_of_birth).

Name = names('Ray', 'Anna')
Address = address(1234, 'Open_street', 'Nepean')
Salary = 40000
Date_of_birth = date_of_birth(1, 1, 1945) ->;

no

%  Who lives in Ottawa?
?- employee(_, Names, address(_,_,'Ottawa') ,_, _).

Names = names('Smith', 'John') ->;
Names = names('Palmer', 'David') ->;
Names = names('Norman', 'Robert') ->;

no
```

More clauses may be added to the database to simplify the questions. In this way, the user of the database does not have to know the details of the structure. This separates the implementation and the interface of the structures. For instance, add the following clauses to the database:

```
salary (Last_name, First_name, Salary) :-
        employee(_, names(Last_name, First_name), _, Salary, _).

address(ID, Number, Street, City) :-
        employee(ID, _, address(Number, Street, City), _, _).
```

Then, to find the salary of John Smith:

```
?-  salary('Smith', 'John', Salary).
Salary = 30000 ->
```

To find the address of an employee with ID number 20166:

```
?- address(20166, Number, Street, City).
Number = 62
Street = 'Small_drive'
City = 'Hull' ->
```

§ 2. BINARY TREES AS STRUCTURES

(1) Definition and notation.

An empty tree is represented by the atom *nil*, and a non-empty binary tree is represented by a structure

```
t(Left, Root, Right),
```

where Left and Right are also binary trees, the *left* and *right subtree*, and Root is a data object of any kind.

A *binary search tree* is a binary tree with the property that every node is greater than any node in its left subtree, and is less than any node in its right subtree.

(2) Binary search trees.

The following algorithm searches a binary search tree to check if a given node is in a binary search tree:

```
in(X, t(_,X,_)).                % If the root matches the key, then stop.
in(X, t(Left, Root, Right)) :-
        gr(Root, X),
        in(X, Left).    % If the root is greater than the key,
                        % then search the left subtree.
in(X, t(Left, Root, Right)) :-
        gr(X, Root),
        in(X, Right).   % If the root is less than the key,
                        % then search the right subtree.
```

In this program, another predicate *gr* is defined, according to the nature of the key, to compare the nodes: gr(X, Y) succeeds if X is greater, in some sense, than Y, and it fails otherwise.

(3) Insertion and deletion.

To add a new node to a binary search tree, it becomes the root if the tree is empty, or its key is compared with the root. If it is greater than the root, it is added to the right subtree; if it is less than the root, it is added to the left subtree. Hence, we have the following predicate:

```
addleaf(nil, X, t(nil, X, nil)).          % The tree is empty.
addleaf(t(Left, Root, Right), X, t(Left1, Root, Right)) :-
```

```
        gr(Root, X),
        addleaf(Left, X, Left1)).          %  Add to the left subtree.
addleaf(t(Left, Root, Right), X, t(Left, Root, Right1)) :-
        gr(X, Root),
        addleaf(Right, X, Right1).      %Add to the right subtree.
```

The predicate gr is defined to compare the nodes. In this definition, we assume that all the keys of nodes are distinct. The definition can be easily modified so that, if a key already exists in the tree, an error message will be produced, or no action is taken, or we may allow duplication, then the new node may be inserted into either the left or the right subtree.

To delete a node from a binary search tree follows the same procedure as in other languages. We have to consider a number of cases. If the node to be deleted has two children, we have to find a node to fill in the place of the deleted node. This node can be the smallest node in its right subtree.

Here is the definition of the predicate deletenode:

```
deletenode(t(nil, X, Right), X, Right).
        %  If the left subtree is empty, and the root is to be deleted,
        %  then return the right subtree.
deletenode(t(Left, X, nil), X, Left).

deletenode(t(Left, X, Right), X, t(Left, Y, Right1)) :-
        delmin(Right, Y, Right1).
        %  The root X is replaced by the minimum Y of the right subtree,
        %  and Y is deleted from the right subtree.

deletenode(t(Left, Root, Right), X, t(Left1, Root, Right)) :-
        gr(Root, X),
        deletenode(Left, X, Left1).
        %  If X is less than the root, delete it from the left subtree.
deletenode(t(Left, Root, Right), X, t(Left, Root, Right1)) :-
        gr(X, Root),
        deletenode(Right, X, Right1).
        %  If X is greater than the root, delete it from the right subtree.

delmin(t(nil, Y, Right), Y, Right).
        %  If the left subtree is nil, the root is the minimum.
        %  Reurn it , and the right subtree.
delmin(t(Left, Root, Right), Y, t(Left1, Root, Right)) :-
        delmin(Left, Y, Left1).
        %  If the left subtree is not nil, the minimum is returned
        %  and deleted from the left subtree.
```

§ 3. LISTS

A *list* is a special kind of structure with the period (.) as the functor, and two arguments, called the *head* and the *tail*. The head is a data object of any kind, and the tail is itself a list. There is a special list which has no arguments, neither head nor tail, called the *empty list* and denoted by [ ].

Examples

(i) .(a, [ ]), head = a, tail = [ ].

(ii) .(a, .(b, [ ])), head = a, tail = .(b, [ ]).

(iii) .(a, .(b, .(c, .(d, [ ])))), head = a, tail = .(b, .(c, .(d. [ ]))).

(iv) .(.(a, .(b, [ ])), .(c, .(d, [ ]))), head = .(.(a, .(b, [ ])), tail = .(c, .(d, [ ])).

Alternative notations:

(a) The [Head | Tail] notation. Use a pair of square brackets and a vertical bar to separate the head and the tail.

(i) .(a, [ ]) = [a | [ ]],

(ii) .(a, .(b, [ ])) = [a | [b | [ ]]],

(iii) .(a, .(b, .(c, .(d, [ ])))) = [a | [b | [c | [d | [ ]]]]],

(iv) .(.(a, .(b, [ ])), .(c, .(d, [ ]))) = [[a | [b | [ ]]] | [c | [d | [ ]]]].

(b) The [*, *, …, *] notation. In the [Head | Tail] notation, we may remove a vertical bar and the following pair of square brackets, and replace them by a comma.

(i) .(a, [ ]) = [a | [ ]] = [a],

(ii) .(a, .(b, [ ])) = [a | [b | [ ]]] = [a, b | [ ]] = [a, b],

(iii) .(a, .(b, .(c, .(d, [ ])))) = [a | [b | [c | [d | [ ]]]]] = [a, b | [c | [d | [ ]]]]
= [a, b, c | [d | [ ]]] = [a, b, c, d | [ ]] = [a, b, c, d]
= [a | [b, c, d]] = [a, b | [c, d]],

(iv) .(.(a, .(b, [ ])), .(c, .(d, [ ]))) = [[a | [b | [ ]] | [c | [d | [ ]]] = [[a, b], [c, d]].

Note that (iii) and (iv), in both (a) and (b), are different, since the head of (iii) is the term a, while the head of (iv) is a structure [a, b].

§ 4. LIST PROCESSING

(1) The Length of a List

The built-in predicate *length*(L, N) accepts a list L and returns N to be the length of L. The length of a list is the number of arguments. For instance, the length of [a, b, c, d] is 4, and the length of [[a, b], c, d] is 3. This predicate can be implemented as follows (although you don't have to do it because it is 'built-in').

```
length([ ], 0).
length([X | Y], N) :- length (Y, M), N is M + 1.
```

(2) Membership

The membership relation determines if an object X is in a list L. The predicate *member* accepts X and L as parameters. It succeeds if X is a member of L, and fails otherwise.

Definition:

```
member(X, [X | _]).                      %  X is the head of L
member(X, [_ | T]) :- member(X, T). %  X is a member of the tail of L
```

Example

```
student([ann, bill, cam]).
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
admissible(X) :- student(L), member(X, L).

?-admissible(bill).
yes.

?- admissible(X).
X = ann ->;
X = bill ->;
X = cam ->;

no
```

(3) Concatenation

This predicate concatenates two lists L1 and L2, and the resulting list is represented by L3.

```
conc([ ], L, L).
conc([H | T1], L2, [H | T3]) :- conc(T1, L2, T3).
```

Using concatenation

(i) Concatenate two known lists: L1, L2 are known, return L by conc(L1, L2, L).

(ii) Decompose a list into two parts: L is known, return L1, L2 by conc(L1, L2, L). If a goal conc(L1,L2,[1,2,3]) is resatisfied again and again, it will return all possible decompositions of [1,2,3] into two sublists.

(iii) Find a specified leading or ending section of a list. L1 (or L2) and L is known, return L2 (or L1) by conc(L1, L2, L).

(iv) Find a sublist of a given list. For instance, the following predicate finds a sublist in a list:

```
sublist(S, L) :- conc(L1, L2, L), conc(S, L3, L2).
```

We may use this goal to check whether a given list S is a sublist of L or generate all sublists of a given list. If the goal sublist(S, [1,2,3,4]) is resatisfied again and again, it gives all sections of the given list in an order [ ], [1], [1,2], [1,2,3], [1,2,3,4], [ ], [2], [2,3], [2,3,4], [ ], [3], [3,4], [ ], [4], [ ]. If you do not want these empty lists, add a subgoal S \= [ ] at the end.

(v) Find the last member of a list.

```
last(L, M) :- conc(_, [M], L).
% Since M is a term, you must write [M] to make a list.
```

Delete the last member of a list L.

```
nolast(L, R) :- conc(R, [M], L).
```

(vi) Implementing membership by *conc*:

```
member(X, L) :- conc(L1, [X | Tail], L).
```

(4) Adding and Deleting Items in a List

(i) Deletion.

```
del([X | Tail], Tail].
del([Y | Tail], [Y | Tail1]) :- del(Tail, Tail1).
```

If the question '?- del([a,b,c], L).' is asked, the program deletes a, b, and c in turn.

Another version of deletion using conc:

```
del1(X, L, R) :- conc(L1, [H | T], L), conc(L1, T, R).
```

(ii) Adding at the beginning.

```
add(X, L, [X | L]).
```

(iii) Insertion.

```
insert(X, L, NewL) :- conc(L1, L2, L), conc(L1, [X|L2], NewL).
```

Another version of insert can be written by the predicate 'del':

```
insert1(X, L, NewL) :- del1(X, NewL, L).
```

If '?- insert(a, [b,c], L).' is asked, it returns [a, b, c], [b, a, c] and [b, c, a] in turn.

(5) Mapping

Let f(X, Y) be a predicate that processes an item X and returns a modified item Y. Then, the elements of a list L can be processed by this predicate using the following clauses:

```
process([ ], [ ]).
process([X | T1], [Y | T2]) :- f(X, Y), process(T1, T2).
```

Examples

(i) Number lists

```
sqr(X, Y) :- Y is X * X.
squared([ ], [ ]).
squared([X | T1], [Y | T2]) :- sqr(X, Y), squared(T1, T2).

?- square([1, 2, 3, 4, 5], Z).

Z = [1, 4, 9, 16, 25] ->;

no

summation([ ], 0).
summation([H | T], X) :- summation(T, Y), X is H + Y.
```

```
?- summation([1, 2, 3, 4], X).

X = 10 ->;

no
```

(ii) Other lists.

```
f(you, i).
f(are, 'am not').
f(X, X).
change([ ], [ ]).
change([X | T1], [Y | T2]) :- f(X, Y), change(T1, T2).

?- change([you, are, a, computer], Z).

Z = [i, 'am not', a, computer] ->;

no
```

(6) Reversing a list

```
reverse([ ], [ ]).
reverse([X | T], Y) :- reverse(T, Z), conc(Z, [X], Y).

?- reverse([a, b, c], L).

L = [c, b, a] ->;

no
```

(7) Permutation generation

```
permutation([ ], [ ]).
permutation([H | T], X) :- permutation(T, Y), insert(H, Y, X).

?- permutation([a, b, c], X).

X = [a, b, c] ->;
X = [b, a, c] ->;
X = [b, c, a] ->;
X = [a, c, b] ->;
X = [c, a, b] ->;
X = [c, b, a] ->;

no
```

(8) Accumulation

When a list is processed sequentially, an accumulator can be set to reflect the accumulation of the process. The final value of the accumulator is assigned to an output variable.

Examples.

(i) The length of a list.

```
length(List, Length) :- accum(List, 0, Length).
accum([ ], Accumulator, Accumulator).
accum([H|T], Accumulator, Length) :- New_accumulator is Accumulator + 1,
                    accum(T, New_accumulator, Length).
```

The last argument Length is the output variable, which is assigned the value of the final value of the Accumulator when the List variable is reduced to an empty list.

(ii) Another version of the summation example.

To find the sum of the numbers in a list of integers, the accumulator can be used.

```
accusum([ ], S, S).          % All terms are added, output the accumulator.
accusum([H | T], A, S) :- A1 is A + H, accusum(T, A1, S).
sum(L, S) :- accusum(L, 0, S).
```

(iii) A list may have components that are themselves lists. The following program constructs a list that decomposes all list components into single components. For instance, if the input list is [[a, b], d, [a, [b, c]], then the output list will be [a, b, d, a, b, c].

We are going to use a predicate is_list/1 that checks to see if a term is a list or an atom:

```
is_list([ ]).
is_list([_|_]).
```

The program is as follows:

```
simple_list(L, S) :- accum(L, [ ], S).
accum([ ], Accumulator, Accumulator).
accum([H|T], Accumulator, S) :- is_list(H), simple_list(H, H1),
                    conc(Accumulator, H1, New_accumulator),
                    accum(T, New_accumulator, S).
accum([H|T], Accumulator, S) :- conc(Accumulator, [H], New_accumulator),
                    accum(T, New_accumulator, S).
```

(9) Difference Lists

A list L can be represented by a pair of lists (L1, L2) such that the concatenation of L and L2 is L1. In this representation, L is denoted by L1 - L2. For instance, a list [a, b, c] can be represented by such as [a, b, c] - [ ], or [a, b, c, b, a] - [b, a], or [a, b, c, d] - [d].

Using difference lists makes the concatenation more efficient. Using difference lists, the concatenation predicate can be defined simply as:

```
conc1(A - B, B - C, A - C).
```

This version uses one clause instead of two clauses, and, more importantly, it works more efficiently. If we want to concatenate [a, b] and [c, d, e], the goal is called by:

```
conc1([a, b | T1] - [T1], [c, d, e | T2] - T2, Result).
```

Then A is instantiated to [a, b | T1], B is instantiated to T1, which is in turn instantiated to [c, d, e | T2], and C is instantiated to T2. Since A is now [a, b, c, d, e | T2], we have the result immediately as [a, b, c, d, e | T2] - T2, i.e., [a, b, c, d, e].

If we want to concatenate a number of lists, say L1, L2, L3, L4, then the clause is simply:

```
multiconc(L1, L2, L3, L4, Result) :-
        conc1([L1 | T1] - T1, [L2 | T2] - T2, M1]),
        conc1([M1 | T3] - T3, [L3 | T4] - T4 , M2),
        conc1([M2 | T5] - T5, [L4 | T6] - T6, Result).
```

In this case, M1 is the concatenation of L1 and L2, M2 is the concatenation of M1 and L3, and Result is the concatenation of M2 and L4.

§ 5. STRING MANIPULATION

Character strings are represented by lists of the ASCII code of the string characters. A string constant is denoted between double quotes, such as "string", "a string". A string is recognized by Prolog as a list of the ASCII codes of the characters in this string:

```
"abc" = [97, 98, 99];

"This is a string."
= [84, 104, 105, 115, 32, 105, 115, 32, 97, 32, 115, 116, 114, 105, 110, 103, 46].
```

The following predicates can be used to compare two strings according to the alphabetical order:

```
strcmp( [ ], [ ], 0 ).
strcmp( S1, [ ], 1 ).
strcmp( [ ], S2, -1).
strcmp( [H1 | T1], [H2 | T2], 1 ) :- H1 > H2.
strcmp( [H1 | T1], [H2 | T2], -1 ) :- H1 < H2.
strcmp( [H | T1], [H | T2], N ) :- strcmp( T1, T2, N ).
```

This predicate instantiates N to be 1 if the first string is alphabetically greater than the second; -1 if the first string is alphabetically less than the second; 0 if they are identical.

Using the predicate conc/3 for general lists, we can concatenate two strings. Using the predicate sublist/2, we can check whether a string is a substring of the other.

```
conc([ ], L, L).
conc([H | T1], L2, [H | T3]) :- conc(T1, L2, T3).
sublist(S, L) :- conc(L1, L2, L), conc(S, L3, L2).

strcat(S1, S2, S) :- conc(S1, S2, S).
substr(S, T) :- sublist(S, T).          % this predicate succeeds if S is a substring of T
```

Prolog provides a predicate to output the ASCII code of a character as the character. The predicate is put/1. The argument of put/1 is an integer between 1 and 127. The predicate converts the integer to a character and output to the current output stream.

To output a string, we may use the following predicate:

```
print_string( [ ] ).
print_string( [H | T] ) :- put(H), print_string(T).
```

CHAPTER 8. CONTROLLING BACKTRACKING

§ 1. PREDICATE CUT

(1) What is 'cut'?

Predicate *cut*, written as !, is a built-in predicate with arity zero used as a condition in the body of a rule or a subgoal in the question. When it is to be satisfied from the left, it always succeeds; if it is resatisfied from the right, then

(i) the predicate cut fails,
(ii) the entire rule fails, and
(iii) other rules with the same predicate also fail.

(2) Using the cut

The following are main situations where cut is needed:

(i) Suppose a predicate is defined by a number of clauses. When one rule is satisfied, the other rules must fail, or other rules may produce false or useless solutions. Then, add a cut to the end of the clause to prevent unnecessary backtracking.

(ii) Suppose a rule has a number of conditions in the body. When a part of the conditions are satisfied with some variables instantiated, if we do not want the values of these variables to be changed, i.e., uninstantiated, in case of a backtracking, then add a cut after these conditions.

If a cut is used to avoid unnecessary backtracking, i.e., if this cut is removed, the predicate will work the same way, this cut is called a *green cut*. If removing the cut, the program will give a wrong answer, or will not work properly, then this cut is called a *red cut*.

(3) Examples

(i) The following program defines a function Y = f(X) such that Y = 0 if $X \le 0$, and Y = 1 if X > 0.

```
f(X, 0) :- X =< 0, !.
f(X, 1) :- X > 0.
```

Without the cut, when a backtracking occurs after the first rule succeeds, the other rule will be tried and is bounded to fail. (green cut.)

(ii) The same function can also be defined as follows:

```
f(X, 0) :- X =< 0, !.
f(X, 1).        %  When this rule is tried, the first rule must have failed, so X > 0.
```

Without the cut, when backtracking occurs after the first rule succeeds, the other rule will give a false solution. (red cut.) Try a question f(0, A) with and without the cut, and keep asking for alternative solutions.

(iii) Computing maximum

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

(iv) Membership test.

```
in(X, [X | T]) :- !.
in(X, [_ | T]) :- in(X, T).
```

If L contains a number of copies of X, this predicate gives once. If it should be resatisfied due to backtracking, it would give the same result, which is bound to fail as with the previous try.

(v) The following program is intended to find a nonnegative integer K such that:

$$2^K \leq N < 2^{K+1}$$

for a given number N:

```
power(0, 1).                %  2^0 = 1.
power(K, X) :- J is K - 1, power(J, Y), X is 2 * Y.
                            %  Calculate 2^K recursively.
least(K, N) :- int(K), J is K + 1, power(J, X), X > N.
                            %  Try all integers until 2^(K+1) > N.
int(0).
int(K) :- int(J), K is J + 1.
                            %  Generates all nonnegative integers.
```

There's a problem with this program:

Suppose a question '?- least(K, 3).' is asked. Prolog finds K = 0, J = 1, X = 2, but the last subgoal fails. Then power(1, X) is resatisfied. From right to left, it leads to resatisfying power(0, Y). It goes to the second goal and we have trouble. Actually, when K is

specified, the predicate power(K,N) generates only one result. It should not be resatisfied.

Therefore, a cut is added to the end of the each rule of the predicate power:

```
power(0, 1) :- !.                % 2^0 = 1.
power(K, X) :- J is K - 1, power(J, Y), X is 2 * Y, !.
                          % Calculate 2^K recursively.
least(K, N) :- int(K), J is K + 1, power(J, X), X > N.
                          % Try all integers until 2^(K+1) > N.
int(0).
int(K) :- int(J), K is J + 1.
                          % Generates all integers.
```

The same question is tried again. This time, as before, K = 0, J = 1, X = 2, are instantiated, and the last subgoal X > 3 fails. Then backtracking reaches the first subgoal int(K). The subgoal int(K) is resatisfied and gives K = 1. Next, J = 2, power(2, X) yields X = 4, and the last subgoal succeeds. Prolog outputs K = 1. What if an alternative solution is asked? (It is possible, if this goal is a subgoal in another clause).

Then backtracking reaches int(K) again and it generates a solution K = 2. If more solutions are asked for, the program would find K = 3, 4, ... This is not what we expected.

This means the predicate least should not be resatisfied. In the last version of the program, another cut is added to the end of the clause defining predicate *least*:

```
power(0, 1).              % 2^0 = 1.
power(K, X) :- J is K - 1, power(J, Y), X is 2 * Y, !.
                          % Calculate 2^K recursively.
least(K, N) :- int(K), J is K + 1, power(J, X), X > N, !.
                          % Try all integers until 2^(K+1) > N.
int(0).
int(K) :- int(J), K is J + 1.
                          % Generates all integers.
```

(vi) The following predicate is intended to find all sublists of a list starting with the first occurrence of a given member.

```
sublist(A, [A | T], [A | T1]) :-
        conc(T1, _, T).
        % If L starts with the given member A, then the result
        % is A followed by a prefix of the tail.
sublist(A, [X | T], L) :-
        sublist(A, T, L).       % If the head of L is not A, then delete it.
```

The problem with this definition is that, when the first clause fails, the second rule applies and generates all sublists after the second or the other A's. If a question ?- sublist(a, [b, a, c, a, b], S) is asked, then the program will generate S = [a], S = [a, c], S = [a, c, a], S = [a, c, a, b], S = [a] and S = [a, b]. But the last two answers are not what we wanted. To avoid this problem a cut is added in the first clause as the following:

```
sublist(A, [A | T], [A | T1]) :- !, conc(T1, _, T).
        %  If L starts with the given member A, then the result
        %  is A  followed by a prefix of the tail.
sublist(A, [X | T], L) :-
        sublist(A, T, L).        %  If the head of L is not A, then delete it.
```

Suppose the program is activated by a question:

```
?- sublist(a, [b, a, c, a], S).
```

Then it executes as like this:

First, the second rule matches the question, and it is reduced to the subgoal:

```
sublist(a, [a, c, a], S).
```

The first clause applies. The first subgoal '!' succeeds, and the second subgoal gives T1 = [ ]. Then the first clause gives the first answer S = [a]. During backtracking, the second subgoal of the first clause is resatisfied, and generates T1 = [c]. Then we have the second answer S = [a, c]. Backtrack again, we have the third answer S = [a, c, a]. Backtrack again, the second subgoal of the first clause fails. Now the cut is encountered. It fails, and the second clause will not be tried. This ends the execution of the program.

(vii) The following predicate tests if an integer is a prime number.

```
upto(M,N,M) :- M =< N.
upto(M,N,A) :- M < N, M1 is M + 1, upto(M1,N,A).
prime(X) :- upto(2,X,M), Z is X mod M, Z = 0, !, M = X.
```

§ 2. THE CUT/FAIL COMBINATION AND PREDICATE NOT

The predicate *fail* is a built-in predicate with arity zero. It always fails and causes backtracking. We may use fail to generate all alternative solutions to goal.

```
?- ancestor(pat, X), write(X), fail.
```

This question will write all ancestors of pat.

Predicate fail is usually used with the cut, as in the following example:

```
A :- B, C, !, fail.
```

This means that, if B and C are satisfied, then the goal A fails without trying any other alternative clause for the same predicate.

Examples

(i) The following predicate defines when two variables are different:

```
different(X, X) :- !, fail.
different(X, Y).
```

(ii) The following predicate defines the factorial of an integer:

```
factorial(X, Y) :- X < 0, !, fail.
factorial(0, 1) :- !.
factorial(X, Y) :- Z is X - 1, factorial(Z, W), Y is X*W, !.
```

Predicate *not* takes a fact A as the argument, written as not(A), or not A. The clause not(A) is true if and only if A is false.

Predicate 'not' can be used to replace cut or cut/fail combination. The second example can be written as follows:

```
factorial(0, 1) :- !.
factorial(X, Y) :- not(X =< 0), Z = X - 1, factorial(Z, W), Y is X*W, !.
```

Predicate 'not' can be used in the following cases:

(i) If a clause is defined in form:

```
A :- B, !, fail.
```

```
A :- D.
```

Then it can be rewritten as

```
A :- not(B), D.
```

(ii) If a clause is defined in the form

```
A :- B, !, C.
A :- D.
```

In the definition, we mean that the second clause is used only when B fails. Then the clauses can be written as:

```
A :- B, C.
A :- not(B), D.
```

Use 'not' instead of cut or cut/fail combination increases the readability of the program. But, in the second case above, the program is less efficient because, when the condition B fails, the program has to check it twice.

Note that the predicate 'not' can check whether the argument fact is false, but it cannot find something that does not satisfy the fact.

Example

```
f(a).
f(b).
g(a).
h(X) :- not g(X).
```

If the question is:

```
?- f(X), h(X).
```

then the answer is X = b. But if the question is:

```
?- h(X), f(X).
```

then the answer will be: no. When h(X) is called, the subgoal is 'not g(X)'. The program finds a match of g(X), i.e., X = a. Therefore, the subgoal fails, and the answer is *no.*

§ 3. PREDICATE REPEAT

The predicate *repeat* acts like a bouncing wall. It always succeeds during scanning or backtracking. When it is used as a subgoal of a rule, it succeeds when it is invoked from left, and when a backtrack reaches it from right, it also succeeds and forces the subgoals on its right to be satisfied again. It is used mainly to create a loop (similar to a 'repeat-until loop' as in Pascal, or do-while loop in C/C++) in the following format:

a_goal :- repeat, *do_something*, *condition*.

When a_goal is invoked, *repeat* succeeds. Then it does *something*. If the condition fails, backtrack to repeat, and it succeeds again. Do_something is satisfied again, and the condition is checked again until the condition succeeds. There is an important difference between the use of repeat and the 'repeat-until loop' in, say, Pascal: All variables instantiated during the previous pass are uninstantiated, so they cannot be carried over to the next pass. Therefore, there must be some input from outside to terminate the loop.

Example. The following program outputs the square of a sequence of positive integers until the user enters a zero.

```
square :-
        repeat,
        write('Please enter a positive integer or zero to stop.'), nl,
        read(X), nl,
        Y is X*X,
        write('The square of '), write(X), write(' is '), write(Y), nl,
        Y =:= 0.
```

It is also possible to write the program so that it does not output the square of the ending zero.

```
square :-
        repeat,
        write('Please enter a positive integer or zero to stop.'), nl,
        read(X), nl,
        (
                X =:= 0
                ;
                Y is X*X,
                write('The square of '), write(X), write(' is '), write(Y), nl,
                fail
        ).
```

This program uses a semicolon to create a disjunctive goal. If the goals on the left-hand side of the semicolon succeed, the right-hand side is not executed. If the left-hand side fails, then the right-hand side goals are invoked. When both sides fail, backtrack to the previous goal. During backtracking, the side that succeeded is resatisfied.

Without using disjunction, this program can also be written as the following:

```
square :-
        repeat,
        write('Please enter a positive integer or zero to stop.'), nl,
        read(X), nl,
        process(X).
process(0).
process(X) :-
        Y is X*X,
        write('The square of '), write(X), write(' is '), write(Y), nl,
        fail.
```

CHAPTER 9. INPUT/OUTPUT, BUILT-IN PREDICATES AND OPERATORS

§ 1. Reading and Writing Terms

To read a term from the current input stream, use the predicate read(*Name*). If Name is an uninstantiated variable, then it is instantiated to the input data object. If Name is an instantiated variable or a term, then it is compared with the input. The goal succeeds if they are identical, or fails otherwise. If the end of the file is reached, Name is instantiated to *end_of_file*. This predicate cannot be resatisfied. The input, an atom or a number has to be followed by a period to mark the end.

To write a term to the current output stream, use the predicate write(Name). If Name is a string starting with a capital letter or contains characters other than lowercase letters or underscore (used between lowercase letters), then it has to be quoted.

Format the output

The following predicates can be used to format the output:

(i) nl. It outputs a new line.

(ii) tab(X). It outputs X spaces.

The following program writes the members of a list, which are lists of integers, each in a separate line:

```
writelist([ ]).
writelist([H|T]) :-
        doline(H),
        nl,
        writelist(T).
doline([ ]).
doline([H|T]) :-
        write(H),
        tab(1),
        doline(T).
```

The following is an example of the execution of the program:

```
?- writelist([[1, 2, 3], [4, 5], [6, ,7, 8, 9]).

1 2 3
4 5
6 7 8 9

yes
```

Building a user interface.

(1) The predicate write can be used to prompt the user to enter an input, explain the meaning of an output, and, if necessary, trace the execution of the program.

(2) As mentioned before, if a fixed question is always used in a program, it can be included in the database, and use a single word to activate the program.

For instance, the following program finds the squares of input integers until the user input a sentinel *stop*:

```
square :-
        write('Please enter an integer or type 'stop' to stop: '),
        read(X),
        nl,
        process(X).

process(stop) :- !.
process(X) :-
        Y is X*X,
        write('The square of '),
        write(X),
        write( ' is '),
        write(Y),
        square.
```

Then the program is activated simply by a question *square*.

(Note: This program also serves as an example of indirect recursion used to imitate a loop.)

§ 2. COMMUNICATING WITH FILES

Prolog reads input from and writes output to *files*. The default file is the standard input and output devices (i.e., the keyboard and the screen), called *user*.

To change the I/O stream, use the following predicates:

(i) see(*file_name*).

This predicate changes the input stream to a file called *file_name*. After this predicate, Prolog reads in input from this file. If the file has not yet been opened (by a previous call of the predicate see), then Prolog opens it, and read from the beginning of the file. If it has been opened before, Prolog reads from where Prolog left off. During backtracking, this predicate is skipped over without undoing.

(ii) seeing(X).

This predicate identifies the current input file. Prolog returns X = *current_input_file.*

(iii) seen.

This 0-arity predicates closes the input file, and returns to the standard input stream. If you want to return to the standard input stream without closing the current input file, use see(user).

(iv) tell(*file_name*).

This predicates changes the output stream to a file called *file_name* to store the output. If the file does not exist, Prolog creates one. If this file already exists, Prolog opens it for output. After it is opened, Prolog writes into it from the beginning, the previous content will be overwritten. Prolog continues write into it until it is closed. During backtracking, this predicate is skipped over without undoing.

(v) telling(X).

This predicate identifies the current output file. Prolog returns X = *current_output_file.*

(vi) told.

This 0-arity predicate closes the output file, and returns to the standard output stream. If you want to return to the standard output before closing the current output file, use tell(user).

The file name used in these predicates can be full path-names such as 'a:\prolog\outfile'.

Example. The following sequence of goals writes X to fileA, Y to the screen, and Z to fileA again and overwrite exiting contents in fileA:

```
..., tell(fileA), write(X), told, write(Y), tell(fileA), write(Z), ...
```

The following sequence of goals writes X to fileA, Y to the screen, and Z to fileA again following X:

```
..., tell(fileA), write(X), tell(user), write(Y), tell(fileA), write(Z), ...
```

These are standard file manipulation predicates defined by the language. But there are also other input/output predicates provided by various implementations of the language.

Example. Suppose we have a file, called *wagelist,* of structures, each has the format

```
employee(Name, HourlyWage, HoursWorked).
```

(It is important to have a period after each structure in the file.) We want to create a file, called *payroll*, listing the names of the employees and the pay amount. The program is as follows:

```
calculate :-
        see(wagelist),
        tell(payroll),
        processfile,
        seen,
        told,
        write('The process is complete.').
processfile :-
        read(Employee),
        process(Employee).
process(end_of_file).
process(employee(Name, HourlyWage, HoursWorked)) :-
        Pay is HourlyWage * HoursWorked,
        write(Name),
        tab(5),
        write(Pay),
        nl,
        processfile.
```

§ 3. CHARACTERS AND STRINGS I / O

There are input/output predicates that take characters as their ASCII codes:

get0(X): It reads a character from the input stream, and X is instantiated to the ASCII code (as an integer) of the input character.

get(X): It skips over all non-printable characters (including blanks), and reads the first printable character from the input stream, and X is instantiated to the ASCII code (as an integer) of the input character.

These two predicates read the input character immediately, unlike predicate *read*, which needs a period to mark the end of a term, and the input is buffered until you press ENTER.

Prolog uses predicate put/1 to output a character as we talked about before.

These predicates cannot be resatisfied. Every character from the input stream is read by get0 or get only once.

For instance, the following program leaves exactly one blank space between two words in the input sentence:

```
squeeze :-
        get0(C),        % Read the first character.
        put(C),         % Write it.
        dorest(C).      % Depending on what C is, process the rest.

dorest(46) :- !.        % If C is a period, stop.
dorest(32) :- !,        % If C is a blank, then
        get(C),         % read the next printable character,
        put(C),         % write it,
        dorest(C).      % and continue.
dorest(Letter) :-       % If C is a letter, then
        squeeze.        % go back to squeeze.
```

(This program has to read input from a file. Otherwise, the input and output will be mixed up on the screen. If you want to input from the keyboard, another predicate, provided by some Prolog interpreters has to be used to eliminate the echo of the input, so that the input will not be shown on the screen.)

The following program creates a nice table of the name and the telephone number of the employees.

```
maketable :-
    tell(user),
    write('Enter the name of the employee between double quotes, '),
    write('or enter a pair of double quotes to stop.'),
    nl,
    read(Name),
    process(Name).

process([ ]) :- told, !.
process(Name) :-
    length(Name, L),
    tell('c:\csi2165\direct.ari'),
    writestring(Name),
    W is 25 - L,
    tab(W),
    tell(user),
    write('Enter the telephone number of '),
    writestring(Name),
    write('between double quotes.'),
    nl,
    read(Num),
    tell('c:\csi2165\direct.ari'),
    writestring(Num),
    nl,
    maketable.
writestring([ ]).
writestring([H|T]) :-
    put(H),
    writestring(T).
```

§ 4. MORE BUILT-IN PREDICATES

This section introduces more built-in predicates that manipulate terms and the database.

(1) Testing the Type of Terms

var(X) : It succeeds if X is an uninstantiated variable.

nonvar(X) : It succeeds if X is not a variable, or it is an instantiated variable.

atom(X) : It succeeds if X currently stands for an atom.

integer(X) : It succeeds if X currently stands for an integer.

float(X) : It succeeds if X currently stands for a real number.

atomic(X) : It succeeds if X currently stands for an integer or an atom.

(2) Manipulating Database

listing : It displays the clauses in the database.

consult(File) : It adds the given file to the end of the current database. If no database was consulted during the same session, the database will contain the clauses in File.

reconsult(File) : The same as *consult* but if a predicate in File is the same as a predicate currently existing in the database, then the new definition substitutes the original one.

retract(C) : It deletes the clause C from the database.

assert(C) (the same as assertz(C)) : It adds the clause C to the database after all other clauses with the same predicate.

asserta(C) : It adds the clause C to the database before all other clauses with the same predicate.

(The last two predicates add a new clause C in the database during the execution of a program, but the file that contains the program is not changed. In other words, when next time the program is activated, clause C is not in the program before the assert clause is reached.)

(3) Manipulating Terms

name(Atom, List) : It converts an atom to a string, as a list of the ASCII code of the characters in the string, or vise versa. For instance, the goal name(atom, X) causes X to be instantiated to "atom" (i.e., [97, 116, 111, 109]), and name(X, "abc") causes X to be instantiated as abc.

=.. : This is an infix predicate that converts a structure to a list with the functor as the head and the argument list as the tail. For instance, the goal names(first, last) =.. L causes L to be instantiated to [names, first, last], and the goal S =.. [family, Parents, Children] causes S to be instantiated to family(Parents, Children).

This predicate can be used to define a 'variable' predicate. Suppose we define:

S =.. [F, A, B, C].

Then, when F is instantiated to *family*, S is family(A, B, C), and when F is instantiated to *triangle*, S becomes triangle(A, B, C).

functor(Structure, Functor, Arity) : This predicate returns the functor, as an atom, and the arity, i.e., the number of arguments, of a structure Structure. Or, it defines a structure Structure with Functor as the functor with arity Arity.

arg(N, Structure, A) : This predicate returns, or defines, the Nth argument A of the structure Structure.

A clause in the database of a program is, actually, a structure. The predicate of a fact is the functor, and the arguments of the fact are arguments of the structure. A rule is an infix structure with ':-' as the principal functor, the head as the first argument and the body as the second argument. In the body, subgoals are connected by functor ',', which is of the type xfy. For instance, a rule 'A :- B, C, D.' can be written as:

':-'(A, ','(','(B, C), D)).

According to this point of view, combining predicates *functor* and *arg*, we may create a clause during the execution of the program and assert it into the database, so that the program may change during its execution, or, in other words, the program can 'get experience' or 'learn' from its execution. For instance, if, at some point in the program, we want to add an additional clause:

A(B, C).

into the database, where A, B, and C are instantiated during the execution of the program, then the following goals serve the purpose:

functor(Clause, A, 2), arg(1, Clause, B), arg(2, Clause, C), assert(Clause).

This goal can later be called by a predicate *call*(Clause).

bagof(X, Goal, L) : This predicate returns a list L consisting of all X's that satisfy the goal Goal. It fails if L is an empty list. If there are other variables, besides X in Goal, then L has multiple values according to the values of the other variables. If we want to ignore the difference between the values of another variable Y, say, put Y^ before Goal. Then, all X's corresponding to different values of Y will be bagged into a single list.

For instance, if the database is the following:

```
parent(joe, bob).
parent(ann, bob).
parent(joe, cindy).
parent(john, dave).
parent(mary, dave).

?- bagof(X, parent(X, Y), L).

X = _0086                (X is given by an address without a value.)
Y = bob
L = [joe, ann];

X = _0086
Y = cindy
L = [joe];

X = _0086
Y = dave
L = [john, mary];

?- bagof(X, Y^parent(X, Y), L).

X = _0086
Y = _ 0088
L = [joe, ann, joe, john, mary];
```

setof(X, Goal, L) : This predicates works the same as bagof, except that the terms in L are ordered, numerically or alphabetically, and duplicated terms are discarded. For instance, for the previous database, the answer to the question '?- setof(X, Y^parent(X, Y), L).' will be:

```
X = _0086
Y = _0088
L = [ann, joe, john, mary];
```

findall(X, Goal, L) : This predicate works the same as bagof, except that, if there are other variables other than X in Goal, L contains all X's that satisfy Goal without classifying according to the values of the other variables. For instance, the goal:

```
?- findall(X, parent(X, Y), L).
```

is the same as:

```
?- bagof(X, Y^parent(X, Y), L).
```

Another difference between *bagof* and *findall* is that, if the result list is empty, *findall* succeeds with the empty list as the result while *bagof* fails in this case.

(4) Examples

(i) Generate sentences (The use of *name*).

The following program transforms an input sentence into a list of atoms, which are words in the sentence. The input sentence is assumed to contain only letters and blanks, and end with a period.

```
getsentence(Wordlist) :-        % Read from the input file and output Wordlist.
        get0(Char),             % Read the first character.
        getrest(Char, Wordlist).
                                % Process the character and generate Wordlist.

getrest(46, [ ]) :- !.          % If the character is a period, generate [ ].
getrest(32, Wordlist) :-        % If the character is a blank, ignore it.
        !,
        getsentence(Wordlist).
getrest(Letter, [Word|Wordlist]) :-             % If the character is a letter,
        getletters(Letter, Letters, Nextchar),
                        % read the following letters into a list
                        % until the next character is a period or a blank.
        name(Word, Letters),            % Transform the list into an atom.
        getrest(Nextchar, Wordlist).    % Process the next character.

getletters(46, [ ], 46) :- !.           % If the character is a period, generate [ ],
                                        % and return a period.
getletters(32, [ ], 32) :- !.           % If the character is a blank, generate [ ],
                                        % and return a blank.
getletters(Letter, [Letter|Letters], Nextchar) :-
                                % If the character is a letter,
                                % then put it as the head of the list.
        get0(Char),                     % Read the next character Char, and
```

```
        getletters(Char, Letters, Nextchar).
                        % generate the tail of the list by Char,
                        % and return Nextchar, which is either 46 or 32.
```

(ii) The following program defines new relations and adds them to the database:

```
parent(a, c).
parent(b, c).
parent(a, d).
parent(b, d).

define_new(X, Z) :-
        write('What is their relation?'), nl,
        read(Relation),
        functor(C, Relation, 2),
        arg(1, C, X),
        arg(2, C, Z),
        assertz(C).
```

Then we may have the following question sequence:

```
?- parent(b, c).

yes

?- brother(c, d).

no.

?- define_new(c, d).
What is their relation?
brother.

yes

?- brother(c, d).

yes.
```

§ 5. OPERATORS

(1) Precedence and Associativity of Operators

The *precedence level* of an operator is given by an integer between 1 and 1200. When different operators are connected together without parentheses, the operator with the **lowest** precedence is grouped first with its arguments. The operator with the **highest** precedence level in an expression is called the *principal* operator of the expression.

The associativity of an operator specifies the order of grouping when two or more operators with the same precedence are connected together without parentheses. The associativity of an operator is given by a *type specifier* as in the following table:

Type	specifier	associativity
infix	xfx	nonassociative
	xfy	right to left
	yfx	left to right
prefix	fx	non-associative
	fy	right to left
postfix	xf	non-associative
	yf	left to right

If an operator is non-associative, then the operators with the same precedence cannot appear in the same expression. If an operator has a right to left associativity, then the rightmost operator is grouped first with its arguments. If an operator has a left to right associativity, then the leftmost operator is grouped first with its arguments.

The precedence and associativity of the most common operators are as in the following table:

operator	type specifier	precedence
:-	xfx	1200
:-, ?-	fx	1200
;	xfy	1100
,	xfy	1000
not	fy	900
=	xfy	700
is, \=, ==, \==	xfx	700
or	yfx	675
and	yfx	670
=\=, =:=	yfx	650
<, =<, >, >=	yfx	600
+, -	yfx	500

*, /, mod, //	yfx	400

(2) Defining Operators

Using user-defined operators enables the user to write clauses and queries in a convenient format (infix, prefix, or postfix) making the program more readable. These 'operators' are actually predicates. No operations are associated with user-defined operators.

Operators are defined by *directives* with the following format:

```
:- op(precedence, type specifier, name of the operator).
```

The name of the operator is an atom. For instance,

```
:- op(600, xfx, has).
```

defines an operator called *has*, which is infix and non-associative with precedence level 600. Then a clause:

```
'John' has a_son.
```

means the same as:

```
has('John', a_son).
```

Logical operators: A logical formula contains a number of atoms and variables connected together by logical operators: ~ (negation), v (disjunction), & (conjunction), <==> (equivalence). These operators can be declared as follows:

```
:- op(500, fy, ~).
:- op(600, xfy, v).
:- op(700, xfy, &).
:- op(800, xfx, <==>).
```

Therefore, a logic expression of:

```
~(~AvB)<==>A&~B
```

is interpreted by Prolog as:

```
<==>(~(v(~(A), B)), &(A, ~(B))).
```

(3) More Arithmetic and Logical Operators

(i) Equality predicates '=', '==', '=:=', '\='. '\==', '=\='

The infix predicate '=' unifies two arguments if both arguments are uninstantiated variables, or instantiates a variable to a value if one argument is a un-instantiated variable and the other is an instantiated variable or a constant term. If both sides are constants or instantiated variables, then this goal succeeds if and only if both arguments are identical.

For instance, the clause X = Y unifies X and Y if both are not yet instantiated. After this predicate, Y and Y are regarded the same. If X is un-instantiated and Y is instantiated, then this predicate assigns the value of Y to X, so that X is instantiated with the value of Y. The clause X = 3 instantiates the un-instantiated variable to 3. This works similar to the assignment statement in procedural languages but there is an important difference: If X has been instantiated to a value, say 1, then the clause X = 3 will not change the value of X, but simply fails.

Predicate '==' is used to compare two arguments. If they are the same, it succeeds; otherwise, it fails. Therefore, if X and Y are un-instantiated variables, X == Y fails. If one side is instantiated, and the other side is not instantiated, it fails as well.

Predicate '=:=' is used to compare two numbers on its two sides. If they have the same value, it succeeds; otherwise, it fails. If one or both sides are expressions, the expressions are evaluated.

Predicate '\=' succeeds, if the two argument cannot be unified. Predicate '\==' succeeds, if its two sides are not the same. Predicate '=\=' succeeds if its two sides have different values.

(ii) Predicates comparing atoms

@>, @>=, @<, and @=<, are predicates used to compare atoms by their lexicographic order.

For instance, jack @< john succeeds, and 2 + 3 @< 1 + 5 fails.

(iii) More arithmetic operators

Some common mathematical functions are also defined as built-in predicates in almost all interpreters or compilers of Prolog language.

> abs(X) evaluates the absolute value of X.
> sin(X), cos(X), tan(X) are trigonometric functions as usual.
> ln(X) gives the natural logarithm of X, and exp(X) give the power e^X.
> sqrt(X) gives the square root of X.

(vi) Examples

(a) Calculate the factorial.

The factorial of a non-negative integer N can be calculated by the following predicate:

```
factorial(0, 1).
factorial(N, M) :- L is N - 1, factorial(L, K), M is K * N.
```

(b) Solving quadratic equations.

A quadratic equation $ax^2 + bx + c = 0$ is specified by the coefficients a, b, c, where $a \neq 0$. The following predicate solves quadratic equations:

```
roots(A, B, C, X1, X2) :-
        A =\= 0,
        D is B*B - 4*A*C,
        D >= 0,
        E is sqrt(D),
        X1 is (-B+E)/(2*A),
        X2 is (-B-E)/(2*A).
```

(c) Find the greatest common divisor of two natural numbers.

The greatest common divisor of two positive integers M and N can be obtained by the Euclidean algorithm. The following is an implementation of this algorithm in Prolog:

```
gcd(X, 0, X).           % gcd(X, 0) = X.
gcd(X, Y, Z) :-
        W is X mod Y,
        gcd(Y, W, Z).
```

CHAPTER 10. PROBLEM SOLVING BY PROLOG

§ 1. SORTING ALGORITHMS

(1) Bubble Sort:

```
bubblesort(List, Sorted) :-
        swap(List, List1), !,
        bubblesort(List1, Sorted).
bubblesort(Sorted, Sorted).
% The predicate is used to find the first pair of consecutive members
% that are in a wrong order, and interchange them.
% If there is no such pair, this goal fails.
swap([X, Y | Rest], [Y, X | Rest]) :-
        X > Y.
swap([Z | Rest], [Z | Rest1]) :-
        swap(Rest, Rest1).
```

Note that this bubble sort program works a little differently from the algorithm used in other languages.

(2) Insertion Sort:

```
insertion([ ], [ ]).
insertion([X | Tail], Sorted) :-
        insertion(Tail, Sortedtail),        % Sort the tail.
        insert(X, Sortedtail, Sorted).  % Insert the head into the sorted tail.

insert(X, [H | T], [H | T1]) :-
        X > H, !,                           % If X > H, insert X into the tail of the tail.
        insert(X, T, T1).
insert(X, T, [X | T]).                      % If X < H, X is the first member.
```

(3) QuickSort:

```
quick([ ], [ ]).
quick([X | T], Sorted) :-
        split(X, T, Pre, Post),
        quick(Pre, Sortedpre),
        quick(Post, Sortedpost),
```

```
        conc(Sortedpre, [X | Sortedpost], Sorted).

split(_, [ ], [ ], [ ]).                    % [ ] is split into [ ] and [ ].
split(X, [Y | T], [Y | Pretail], Post) :-
        X > Y, !,
        split(X, T, Pretail, Post).         % If X > Y, leave it in Pre.
split(X, [Y | T], Pre, [Y | Posttail]) :-
        split(X, T, Pre, Posttail).         % If X =< Y, put Y in Post.

conc([ ], L, L).
conc([X | L1], L2, [X | L]) :-
        conc(L1, L2, L).
```

(4) Merge Sort:

```
partition([ ], [ ], [ ]).
partition([H|T], [H|T1], L) :- partition(T, L, T1).
merge([ ], L, L).
merge(L, [ ], L).
merge([H1|T1], [H2|T2], [H2|T3]) :- H1 > H2, !,
                merge([H1|T1], T2, T3).
merge([H1|T1], [H2|T2], [H1|T3]) :- merge(T1, [H2|T2], T3).
merge_sort([ ], [ ]) :- !.
merge_sort([H], [H]) :- !.
merge_sort(L, Sorted) :- partition(L, L1, L2), merge_sort(L1, Sorted1),
                merge_sort(L2, Sorted2), merge(Sorted1, Sorted2, Sorted).
```

§ 2. THE HANOI TOWER PROBLEM

The problem: There are three pegs numbered 1, 2, 3. N disks with different diameters are on Peg 1 from top down in an increasing order with respect to their diameters. The problem asks a user to move all disks to Peg 3 according to the following rules:

(i) Each time only one disk at the top of a peg can be moved to the top of another peg.

(ii) A larger disk cannot be put on top of a smaller disk.

The solution uses a recursive predicate, called hanoi(N, X, Y, Z), say, where N is the number of disks, X, Y and Z are numbers of pegs, and the disks are to be moved from Peg X to Peg Z using Peg Y as the intermediate peg.

If N = 0, no action is taken. This leads to a clause hanoi(0, X, Y, Z).

When N > 0, we first move the top N - 1 disks from Peg X to Peg Y, then move the last disk from Peg X to Peg Z, and, finally, move the N - 1 disks on Peg Y to Peg Z. This leads to the second clause:

```
hanoi(N, X, Y, Z) :-
        M is N -1, hanoi(M, X, Z, Y), move(X, Z), hanoi(M, Y, X, Z).
```

Finally, the predicate move(X, Y) outputs the move, which depends on the format of the output you want. For instance, it can be simply implemented as:

```
move (X, Y) :- write(X), write(' -> '), write(Y), nl.
```

Therefore, the program is:

```
hanoi(0, X, Y, Z).
hanoi(N, X, Y, Z) :-
        M is N - 1, hanoi(M, X, Z, Y), move(X, Z), hanoi(M, Y, X, Z).
move(X, Y) :- write(X), write(' -> '), write(Y), nl.
```

When it is activated by the question, say,

```
?- hanoi(4, 1, 2, 3).
```

it outputs the moves four disks from Peg 1 to Peg 3.

§ 3. THE EIGHT-QUEENS PROBLEM

The Eight Queens Problem asks a user to put eight queens on a chessboard (8 by 8) so that no queen can attack another queen. A solution is specified by a list [X1/Y1, X2/Y2, ..., X8/Y8] that gives the 'coordinates' of the queens. Since the queens have to be in different columns, (so no two queens can attack each other vertically), we may simply assume X1 = 1, X2 = 2, ..., X8 = 8. Then a solution is a list [1/Y1, ..., 8/Y8] such that

(i) Y1, Y2, ..., Y8 are all between 1 and 8 inclusive.

(ii) Y1, Y2, ..., Y8 are all distinct. (So no two queens can attack each other horizontally.)

(iii) Two queens at XI/YI and XJ/YJ cannot attack each other diagonally. In other words, $|XI - XJ| \neq |YI - YJ|$.

The program constructs a solution by adding queens one-by-one on to the board. In other words, Prolog constructs a solution with eight queens from a solution with no queen, one queen, two queens, ..., seven queens. The problem works in the following 'stupid' way: There is a trivial solution with no queen at all. Then Prolog finds a solution with one queen. Then a solution with two queens. Suppose a solution with five queens, say, has been constructed. Then Prolog tries to put the sixth queen. If it is impossible, Prolog removes the fifth queen in the current the solution and tries to relocate the fifth queen to find another solution with five queens. If all possible locations of the fifth queen cannot be augmented to a solution of six queens, then the fourth queen is relocated. Backtracking in this way (possibly the first queen has to relocated) until a solution with six queens is found. Then, Prolog tries to add the seventh queen by the same method.

The program is the following:

```
solution([ ]).                  %  The solution with no queen at all.
solution([X/Y | Others]) :-
        %  A new queen located at X/Y is added to the solution
        %  with the Others solution(Others),
        %  The Others is already a solution.
        solution(Others),
        member(Y, [1, 2, 3, 4, 5, 6, 7, 8],
                                %  Y is between 1 and 8.
        noattack(X/Y, Others).
                    %  The new queen cannot attack any queen in Others.
                    %  The noattack relation is defined recursively:
noattack(X/Y, [ ]).             %  If there is no Others, then no attack.
noattack(X/Y, [X1/Y1 | Rest]) :-
```

```
                    % X/Y cannot attack the Others if X/Y cannot attack
                    % the head of the Others and X/Y cannot attack the
                    % Rest of the Others.
        % The next three conditions specifies X/Y cannot attack X1/Y1:
        Y =\= Y1,           % X/Y cannot attack X1/Y1 horizontally.
        X - X1 =\= Y - Y1,
        X - X1 =\= Y1 - Y,  % X/Y cannot attack X1/Y1 diagonally.
                            % And X/Y cannot attack the Rest:
        noattack(X/Y, Rest).

    member(X, [X | _]).
    member(X, [_ | T]) :- member(X, T).
```

The format of the solution can be specified in the question: (In the program in many textbooks, the format of the solution is specified by a predicate in the database. Then this predicate has to included in the question.)

```
?- solution([1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7, 8/Y8]).
```

Use the following simple case to illustrate the execution of the program: Similar to eight queen problem, we may ask for the solution of a four queen problem, i.e., put four queens on a 4 by 4 chessboard so that no two queens may attack each other. The program can be obtained from the above by making obvious changes.

When the program is executed, one queen (Queen #4) is put at the square 4/1. To put the second queen (Queen #3), Prolog tries 3/1, 3/2, but both fail. It finds a possible place 3/3. Now the third queen (Queen #2) is introduced. 2/1 is attacked by the first queen, 2/2 is attacked by the second queen. 2/3 is even attacked by both queens. 2/4 is attacked by the second queen. Therefore, the second queen is relocated. The only possible place is 3/4. Now the third queen can be put at 2/2. The fourth queen (Queen #1) cannot be located without being attacked. The third queen is relocated, but Prolog finds no other place available. So the second queen is relocated. The second queen does not have another available place either. Then the first queen is relocated at 4/2. The second queen is relocated at 3/4. The third queen is relocated at 2/1. Finally, the fourth queen can be located at 1/3. An answer is found!

In the case of eight queens, Prolog finds the following solutions one by one:

```
4, 2, 7, 3, 6, 8, 5, 1.    5, 2, 4, 7, 3, 8, 6, 1.    3, 5, 2, 8, 6, 4, 7, 1.
3, 6, 4, 2, 8, 5, 7, 1.    5, 7, 1, 3, 8, 6, 4, 2.    4, 6, 8, 3, 1, 7, 5, 2.
... .
```

There are 56 solutions. But some of them are 'essentially' equivalent, in the sense that one can be obtained from the other by rotating the board or taking a mirror image with respect to a certain axis.

§ 4. A SHORTEST PATH PROBLEM

```
go(a, b, 100).
go(a, c, 150).
go(b, d, 200).
go(b, e, 180).
go(c, d, 130).
go(c, e, 100).
go(d, f, 80).
go(e, f, 170).

shortest(X, Y, P) :- go(X, Y, P).
shortest(X, Y, P) :- go(X, Y, PY), go(X, Z, PZ), shortest(Y, AY, QY),
        shortest(Z, AZ, QZ), PY + QY < PZ + QZ, !, P is PY + QY.
shortest(X, Z, P) :- go(X, Y, PY), go(X, Z, PZ), shortest(Y, AY, QY),
        shortest(Z, AZ, QZ), P is PZ + QZ.

find(X, [X, f], P) :- go(X, f, P), !.
find(X, [X|T], P) :- shortest(X, Y, P), find(Y, T, _).

reach(f, [f], 0) :- !.
reach(X, [X|T], P) :- go(X, Y, P1), reach(Y, T, P2), P is P1 + P2.
```

§ 4. REPRESENTING GRAMMAR RULES BY PROLOG

(1) Languages and Grammars

A *language* is a set of *sentences*, constructed from a set of *words*, according to a given set of *grammar rules*. (In the theory of formal languages, sentences are usually called 'words', and words are characters.)

The grammar rules specify how a sentence is constructed from words. In Prolog, grammar rules are presented using the DCG (Definite Clause Grammar) notation, which consists a set of *terminals* (that is, the words in the language), a set of *nonterminals*, and a set of *production rules*. Nonterminals represent a part of a sentence that can be replaced by a sequence of terminals and/or nonterminals. Among nonterminals, there is one, called the *starting symbol*, is at the top level representing an entire sentence. A production rule specifies how a nonterminal can be constructed from a sequence of terminals and/or nonterminals (a *phrase*). A terminal is recognized by Prolog as a term, and a sequence of terminals and/or nonterminals (such as a phrase or a sentence) is recognized by Prolog as a list.

The following example shows how a grammar is presented in DCG notation, and how the sentences are constructed by the grammar:

The language PALINDROM with alphabet {a, b} consists of all sequences of a and b that is symmetric (i.e., the sequence is identical to its reverse). Now the words of the language are a and b, which are terminals. We need only one nonterminal denoted by s, which is also the starting symbol. And the production rules are:

```
s --> [a].
s --> [b].
s --> [a, a].
s --> [b, b].
s --> [a], s, [a].
s --> [b], s, [b].
```

The rules mean that s can be replaced by [a], or [b], or [a, a], or [b, b], or a sequence [a], s, [a], or a sequence [b], s, [b]. All rules about the same nonterminal can be put together using the vertical bar to represent 'or'. Hence, the grammar can also be written in one production rule:

```
s --> [a] | [b] | [a, a] | [b, b] | [a], s, [a] | [b], s, [b].
```

Note that, in DCG, terminals are included in square brackets, words on the right-hand side of the production rule are separated by commas, and each rule is terminated by a period.

The main tasks in processing a language are:

(i) generate sentences in the language; and

(ii) check whether a given sequence of words is a legitimate sentence in the language, and, if it is, how the sentence is generated by the grammar (this procedure is called *parsing*).

For instance, the sequence [a, b, a, a, b, a] is a sentence in PALINDROM because it can be generated by the following steps:

(i) Starting from s, replace it by [a], s, [a];

(ii) replace s again by [b], s, [b], to have [a, b], s, [b, a];

(iii) and finally, replace s by [a, a].

The following is another example of a grammar that specifies fixed-point real numbers with or without decimal point, with or without a sign:

```
fixed --> num.
fixed --> sign, num.
num --> digits.
num -->  digits, dot, digits.
digits --> digit, digits.
digits --> digit.
sign --> ['+'] | ['-'].
dot --> ['.'].
digit --> [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9].
```

According to this grammar, 12, 1.20, +12.3, -02.4, and 00.00 are legitimate fixed-point real numbers, and 1.2.3, 12., +-1.2 are not legitimate fixed-point real numbers.

(2) Representing Grammar Rules by Clauses

Lists used to represent sentences in a language are in the form of a difference list. Recall that a list L is denoted by L1 - L2 if L1 is the concatenation of L and L2.

Grammar rules in DCG are transformed into ordinary clauses by Prolog and stored in the database. When a production rule is transformed, the nonterminal n on the left-hand side

becomes the predicate that takes a pair of lists L and M as arguments. This means n can be replaced by L - M.

If the right-hand side of the rule is a list L0 of terminals, then L - M = L0. In this case, the clause is simply a fact:

```
n([L0 | T], T).
```

If the right-hand side of the rule is a sequence of nonterminals n1, n2, ..., nk, then the clause is transformed into a rule:

```
n(L, M) :-
        n1(L, L2),
        n2(L2, L3),
        ...,
        nk(Lk, M).
```

This means that L - L2 corresponds to n1, L2 - L3 corresponds to n2, ..., and, finally, Lk - M corresponds to nk. Hence, L - M is the concatenation of L - L2, L2 - L3, ..., Lk - M.

If the right-hand side of the rule is a sequence mixed with terminals and nonterminals, then the terminals are included in the arguments of the nonterminals on the right-hand side. For instance, a rule:

```
n --> [t1], n1, [t2], n2, n3.
```

is transformed as:

```
n([t1 | L], M) :-
        n1(L, [t2 | L2]),
        n2(L2, L3),
        n3(L3, M).
```

This means that, n is replaced by [t1|L - M], where t2 is in L, and what before t2 in L corresponds to n1, after t2, L2 - L3 corresponds to n2, and L3 - M corresponds to n3. Similarly, a rule:

```
n --> n1, [t1, t2], n2, [t3].
```

is transformed into:

```
n(L, M) :-
        n1(L, [t1, t2 | L2]),
        n2(L2, [t3 | M).
```

Therefore, the grammar of PALINDROM is transformed as the following:

```
s([a | L], L).
s([b | L], L).
s([a, a | L], L).
s([b, b | L], L).
s([a | L], M) :-
        s(L, [a | M]).
s([b | L], M) :-
        s(L, [b | M]).
```

By this representation of lists, if we want to check if the string abba is a sentence in PALINDROM, the question shall be:

```
?- s([a, b, b, a], [ ]).
```

The predicate of the question is the starting symbol of the grammar, and it takes two arguments to represent a list in the difference list representation. Similarly, if we want to check if -12.3 is a fixed-point real, the question is:

```
?- fixed(['-', 1, 2, '.', 3], [ ]).
```

To have a better interface, more predicates can be added into the program to read a real number in the ordinary form, and convert it to a list, then call the goal *fixed* with an empty list as the second argument. The following is the modified version of the program fixed:

```
fixed --> num.
fixed --> sign, num.
num --> digits.
num -->  digits, dot, digits.
digits --> digit, digits.
digits --> digit.
sign --> ['+'] | ['-'].
dot --> ['.'].
digit --> [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9].
check :-
        write('Please enter a fixed-point real number between single quotes.'),
        read(R),
        name(R, L),
        convert(L, L1),
        answer(L1).
convert([ ], [ ]).
convert([H |T], [X |T1]) :-
        name(X, [H]),
        convert(T, T1).
answer(L1) :-
```

```
        fixed(L1, [ ]), !,
        write('Yes, this is a fixed-point real.').
answer(L1) :-
        write('No, this is not a fixed-point real.').
```

(3) Adding Parameters into Nonterminals

In the grammars introduced in previous paragraphs, a nonterminal can be replaced by any sequence specified by the product rules without considering how the other in which the nonterminals are replaced. But, in some cases, as in natural languages, which sequence can be used to replace a nonterminal depends on how the other nonterminals are replaced. In other words, some kinds of agreement, such as the agreement of *number*, *person,* and *time* in natural languages, have to be maintained. Then additional parameters may be introduced into nonterminals to mark the difference.

The following is an example to show how to use parameters in DCG to recognized natural language sentences:

```
sentence --> noun_phrase(Number, sub), verb_phrase(Number).
noun_phrase(Number, Type) --> determiner, noun(Number, Type).
verb_phrase(Number) --> verb(Number), noun_phrase(Number1, ob).

determiner --> [the].
noun(singular, sub) --> [student] | [employee].
noun(plural, sub) --> [students] | [employees].
verb(singular) --> [learns].
verb(plural) --> [learn].
noun(singular, ob) --> [formula].
noun(plural, ob) --. [formulas].
```

This program will recognize sentences like 'the students learn the formulas', 'the employee learns the formulas', and 'the employees learn the formula', but not 'the employees learns the formula', or 'the formula learns the student'.

When DCG is used to recognize a natural language sentence, the sentence has to be written as a list in the difference list format. To have a better interface, the predicate *getsentence* introduced in Chapter 9 can be used to transform a sentence into a list of words.

When a production rule with parameters is transformed into an ordinary Prolog clause, the parameters are included in the parameter list of the predicate, by convention, before the difference list parameter.

(4) Showing Parsing Trees

The procedure of parsing a sentence into words according to the production rules can be represented by a tree, called the *parsing tree* of the sentence. By including another parameter into the production rules, we can identify this parsing tree.

The following example shows how it works:

```
sentence(sentence(NP, VP)) -->
        noun_phrase(Number, sub, NP),
        verb_phrase(Number, VP).
noun_phrase(Number, Type, noun_phrase(Det, Noun)) -->
        determiner(Det),
        noun(Number, Type, Noun).
verb_phrase(Number, verb_phrase(Verb, NP)) -->
        verb(Number, Verb),
        noun_phrase(_, ob, NP).

determiner(determiner(the)) --> [the].
noun(singular, sub, noun(student)) --> [student].
noun(singular, sub, noun(employee)) --> [employee].
noun(plural, sub, noun(students)) --> [students].
noun(plural, sub, noun(employees)) --> [employees].
verb(singular, verb(learns)) --> [learns].
verb(prural, verb(learn)) --> [learn].
noun(singular, ob, noun(formula)) --> [formula].
noun(plural, ob, noun(formulas)) --> [formulas].
```

When a question:

```
?- sentence(T, [the, student, learns, the, formulas], [ ]).
```

is asked, Prolog will output the parsing tree of the sentence (as a structure):

```
T = sentence(noun_phrase(determiner(the), noun(student)),
verb_phrase(verb(learns), noun_phrase(determiner(the)), noun(formulas))).
```