Chapter 1

The basic evaluator

This chapter describes a simple language, whose constructs should be regarded as simplified versions of the constructs of PASCAL, written in a syntax designed for ease of parsing. We then present the interpreter for that language, giving the code, in PASCAL, in Appendix B. The chapter ends with an outline of what this book is about, and why.

1.1 The language

Our interpreter is interactive. The user will enter two kinds of inputs: function definitions, such as:

```
(define double (x) (+ x x))
```

and expressions, such as:

```
(double 5).
```

Function definitions are simply "remembered" by the interpreter, and expressions are evaluated. "Evaluating an expression" in this language corresponds to "running a program" in most other languages.

The subsections of this section present the language's syntax, its semantics, and examples. It is to be hoped that the reader will soon find the syntax, if not elegant, at least not a major hindrance.

1.1.1 Syntax

The syntax of our language is like that of LISP, and can be very simply defined[1]:

```
input       ⟶  expression | fundef
fundef      ⟶  ( define function arglist expression )
arglist     ⟶  ( variable* )
expression  ⟶  value
            |   variable
```

[1]Appendix A gives an explanation of this notation.

```
                |   ( if expression expression expression )
                |   ( while expression expression )
                |   ( set variable expression )
                |   ( begin expression+ )
                |   ( optr expression* )
optr       -->  function | value-op
value      -->  integer
value-op   -->  + | - | * | / | = | < | > | print
function   -->  name
variable   -->  name
integer    -->  sequence of digits, possibly preceded by minus sign
name       -->  any sequence of characters not an integer, and not containing
                '(', ')', ';', or space.
```

A function cannot be one of the "keywords" `define`, `if`, `while`, `begin`, or `set`, or any of the value-op's. Aside from this, names can use any characters on the keyboard. Comments are introduced by the character '`;`', and continue to the end of the line; this is why '`;`' cannot occur within a name. A session is terminated by entering "`quit`"; thus, it is highly inadvisable to use this name for a variable.

Expressions are fully parenthesized. Our purpose is to simplify the syntax by eliminating such syntactic recognition problems as operator precedence. Thus, the PASCAL assignment

$$i := 2{*}j + i - k/3;$$

becomes

```
(set i (- (+ (* 2 j) i) (/ k 3))).
```

The advantage is that the latter is quite trivial to parse, where the former is not. Our form may be unattractive, but then most PASCAL programmers will agree that assignments of even this much complexity occur rarely.

1.1.2 Semantics

The meanings of expression's are presented informally here (and more formally in section 1.2.8). Note first that integers are the only values; when used in conditionals (`if` or `while`), zero represents false and one (or any other non-zero value) represents true.

(`if` e_1 e_2 e_3) — If e_1 evaluates to zero, then evaluate e_3, otherwise evaluate e_2.

(`while` e_1 e_2) — Evaluate e_1; if it evaluates to zero, return zero; otherwise, evaluate e_2 and then re-evaluate e_1; continue this until e_1 evaluates to zero. (A `while` expression always returns the same value, but that really doesn't matter since it is being evaluated only for its side-effects.)

(`set x e`) — Evaluate `e`, assign its value to variable `x`, and return its value.

(`begin` e_1 ... e_n) — Evaluate each of e_1 ... e_n, in that order, and return the value of e_n.

(f e_1 ... e_n) — Evaluate each of e_1 ... e_n, and apply function f to those values. f may be a value-op or a user-defined function; if the latter, its definition is found and the expression defining its body is evaluated with the variables of its arglist associated with the values of e_1 ... e_n.

if, while, set, and begin are called *control operations.*

All the value-op's take two arguments, except print, which takes one. The arithmetic operators +, -, *, and / do the obvious. The comparison operators do the indicated comparison and return either zero (for false) or one (for true). print evaluates its argument, prints its value, and returns its value.

As in PASCAL, there are global variables and formal parameters. When a variable reference occurs in an expression at the top level (as opposed to a function definition), it is necessarily global. If it occurs within a function definition, then if the function has a formal parameter of that name, the variable is that parameter, otherwise it is a global variable. This corresponds to the *static scope* of PASCAL, in the (relatively uninteresting) case where there are no nested procedure or function declarations[2]. There are no local variables *per se*, only formal parameters.

1.1.3 Examples

As indicated earlier, the user enters function definitions and expressions interactively. Function definitions are stored; expressions are evaluated and their values are printed. Our first examples involve no function definitions. "->" is the interpreter's prompt; an expression following a prompt is the user's input, and everything else is the interpreter's response:

```
-> 3
3

-> (+ 4 7)
11

-> (set x 4)
4

-> (+ x x)
8

-> (print x)
4
4
```

Notice that (print x) first prints the value of x, then returns the value; the interpreter always prints the value of an expression, so in this case 4 is printed twice.

```
-> (set y 5)
5

-> (begin (print x) (print y) (* x y))
```

[2]For more on static scope, see section 4.7 in the SCHEME chapter, and also Chapter 9.

```
4
5
20

-> (if (> y 0) 5 10)
5
```

The body of a `while` expression (that is, its second argument) will almost always be a `begin` expression.

```
-> (while (> y 0)
>      (begin (set x (+ x x)) (set y (- y 1))))
0

-> x
128
```

Note that the interpreter allows expressions to be entered on more than one line; if an incomplete expression is entered, the interpreter gives a different prompt and waits for more input.

User-defined functions work very much as in PASCAL: the arguments to the function are evaluated first, then the function body is evaluated with its *formal parameters* (i.e. variables occurring in the function's arglist) "bound" to the *actual parameters* (i.e. evaluated arguments of the application). When a function is entered, the interpreter responds by echoing the name of the function.

```
-> (define +1 (x) (+ x 1))
+1
-> (+1 4)
5

-> (define double (x) (+ x x))
double
-> (double 4)
8
```

It is important to realize that the variable "x" occurring as the function's formal parameter is distinct from the "x" used earlier as a global variable. This is again similar to PASCAL, where a global variable may have the same name as a formal parameter and the two occurrences represent distinct variables. A reference to `x` *within* the function refers to the formal parameter; a reference outside the function refers to the global variable. Unlike PASCAL, our language uses call-by-value parameter-passing exclusively (no **var** parameters); this means that a global variable can never have its value altered by an assignment to a formal parameter within a function.

```
-> x
128

-> (define setx (x y) (begin (set x (+ x y)) x))
setx
-> (setx x 1)
```

```
129

-> x
128
```

Most function definitions use either iteration or recursion. Here is a PASCAL version of the gcd function:

```
function gcd (m, n: integer): integer;
var r: integer;
begin
      r := m mod n;
      while r <> 0 do begin
          m := n;
          n := r;
          r := m mod n
          end;
      gcd := n
end;
```

and here it is in our language[3]:

```
-> (define not (boolval) (if boolval 0 1))

-> (define <> (x y) (not (= x y)))

-> (define mod (m n) (- m (* n (/ m n))))

-> (define gcd (m n)
         (begin
            (set r (mod m n))
            (while (<> r 0)
               (begin
                  (set m n)
                  (set n r)
                  (set r (mod m n))))
            n))
-> (gcd 6 15)
3
```

A recursive version in PASCAL is:

[3]From now on, and throughout the rest of the book, when presenting an interpreter session, both the echoing of function names and the secondary prompts (">") will be elided.

```
function gcd (m, n: integer): integer;
begin
    if n=0
    then gcd := m
    else gcd := gcd(n, m mod n)
end;
```

which would be rendered in our language as:

```
-> (define gcd (m n)
        (if (= n 0) m (gcd n (mod m n))))
```

The reader should work some of the problems in the "Learning about the language" section of the exercises before going on to the next section.

1.2 The structure of the interpreter

The code for our interpreter, written in Pascal, appears in Appendix B. This section is essentially the documentation of that program. The listing is divided into eight sections, and this documentation is structured likewise. The most important function is *eval*, in section *EVALUATION*, which evaluates an expression; it is explained last, and in great detail.

1.2.1 *DECLARATIONS*

This section contains the declarations of the records *EXPREC*, *EXPLISTREC*, *FUNDEFREC*, and *ENVREC*, and their associated pointer types *EXP*, *EXPLIST*, *FUNDEF*, and *ENV* (we will often use these shorter names when referring to the corresponding records):

```
EXP = ↑EXPREC;
EXPLIST = ↑EXPLISTREC;
VALUELIST = ↑VALUELISTREC;
NAMELIST = ↑NAMELISTREC;
FUNDEF = ↑FUNDEFREC;
```

```
NUMBER = integer;
EXPTYPE = (VALEXP,VAREXP,APEXP);
EXPREC = record
          case etype: EXPTYPE of
               VALEXP: (num: NUMBER);
               VAREXP: (varble: NAME);
               APEXP: (optr: NAME; args: EXPLIST)
      end;

EXPLISTREC = record
          head: EXP;
          tail: EXPLIST;
      end;

FUNDEFREC = record
          funname: NAME;
          formals: NAMELIST;
          body: EXP;
          nextfundef: FUNDEF
      end;

NAMELISTREC = record
          head: NAME;
          tail: NAMELIST;
      end;
```

As given in the syntactic description of the language, an expression can be either a *value* (integer), a *variable* (formal parameter or global variable), or an *application* (function call). These correspond to *EXP*'s of types *VALEXP*, *VAREXP*, and *APEXP*, respectively. (*NAME* is declared as an integer subrange, as explained under *NAME MANAGEMENT*.) A user-defined function is stored internally as a record containing the name of the function (*funname*), its formal parameters (*formals*), and its body (*body*); the *nextfundef* field in *FUNDEFREC* is used to chain together all the function definitions the user has entered. The *INPUT* section contains functions that translate the user's input into its internal representation as an *EXP* or *FUNDEF*.

Types containing the word "*LIST*" all have the same form: a *head* field and a *tail* field, defining a linked-list of whatever type.

Environments (*ENV*'s) are used to hold the values of variables, whether global variables or formal parameters. Note that the value of every variable is an integer.

```
ENV = ↑ENVREC;
ENVREC = record
            vars: NAMELIST;
            values: VALUELIST
         end;

VALUELISTREC = record
            head: NUMBER;
            tail: VALUELIST;
         end;
```

Operations on *ENV* records are given in section *ENVIRONMENTS.*

The values of scalar type *BUILTINOP* correspond to the built-in operations of the language. (All end with the characters "*OP*" to avoid collisions with PASCAL keywords.)

```
BUILTINOP = (IFOP,WHILEOP,SETOP,BEGINOP,PLUSOP,MINUSOP,
             TIMESOP,DIVOP,EQOP,LTOP,GTOP,PRINTOP);
VALUEOP = PLUSOP .. PRINTOP;
CONTROLOP = IFOP .. BEGINOP;
```

1.2.2 *DATA STRUCTURE OP'S*

This section gives operations for constructing expressions, environments, and various kinds of lists, and for computing the lengths of lists.

1.2.3 *NAME MANAGEMENT*

All function names and variables are translated to integers by the input routines. This is accomplished by calling *install* (from *parseName* in *INPUT*), which installs new names and recognizes previously-installed ones. From then on, all references to that name use that integer (which is why the type *NAME* is an integer subrange). This is just an optimization to avoid having to store multiple copies of each name, and to save some time in looking up variables.

initNames is called at the beginning of each session; it places the built-in names into the array *printNames*, and initializes the global variables *numBuiltins* and *numNames* to contain the highest occupied index in *printNames* (i.e. the number of built-in operations).

primOp translates *NAME*'s corresponding to built-in operations (more precisely, those in the range 1 to *numBuiltins*) to the corresponding element of *BUILTINOP.*

1.2.4 *INPUT*

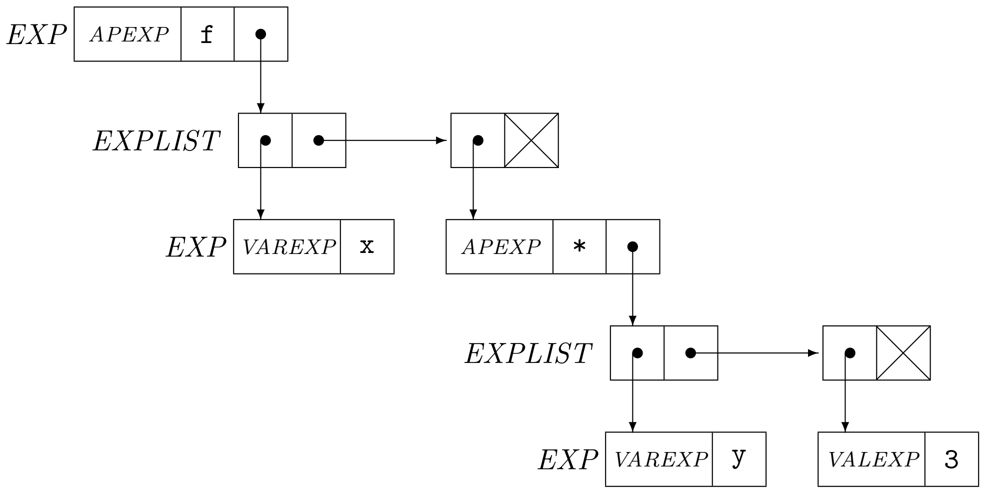
There are three important routines in this section — *reader*, *parseExp*, and *parseDef* — plus various supporting functions.

The procedure *reader* reads characters from the terminal and places them into the array *userinput.* It reads a single line, unless that line contains an unmatched left parenthesis, in which case it reads until all parentheses are balanced. It places a sentinel value (namely, a semi-colon) after the last input character in *userinput*, and sets the variable *pos* to the index of the first non-blank character in *userinput.*

At the top level (*READ-EVAL-PRINT LOOP*), *reader* is called, and then the input is examined to determine if it is an expression or a function definition, calling either *parseExp* or *parseDef* accordingly.

parseExp and *parseDef* use a "recursive-descent" approach to parsing. Each syntactic category has a function which recognizes inputs in that category and translates them to internal form. Thus, *parseDef* is for function definitions, *parseExp* for expressions, *parseEL* for expression lists, *parseNL* for name lists, *parseVal* for integers, and *parseName* for names. These functions are mutually recursive.

For example, if the user enters the expression `(f x (* y 3))`, *reader* will place these characters in *userinput*. The read-eval-print loop will recognize that this is an expression (because it doesn't start with the word "`define`"), so will call *parseExp*, which will construct, and return a pointer to, the following structure:



Whenever a *parse*··· function is called, *pos* — the current location in the input string *userinput* — points to the first non-blank character of the corresponding syntactic item; when it is finished, it has produced a structure of its particular kind, and has updated *pos* to point to the first non-blank character after the end of that item. (In the case of parenthesized lists, *pos* starts out pointing to the first non-blank character after the opening parenthesis, and ends up at the first non-blank after the closing parenthesis.) These details must be borne in mind when adding new syntax to the language, that is, when modifying these parsing routines.

1.2.5 *ENVIRONMENTS*

The values of variables must be stored in symbol tables. The structure of symbol tables is defined by the *ENVREC* record. Each table is a pair, containing a *NAMELIST* and a *VALUELIST*. The *NAMELIST* contains the variables' names and the *VALUELIST* the corresponding values; of course, these two lists must be of the same length.

There are two types of variables, global variables and formal parameters of functions. Global variables are contained in the environment *globalEnv*, which is initialized before the read-eval-print loop and updated in *eval* (see case *SETOP*). The environments giving bindings for formal parameters are different for each function call, so these are passed as parameters to *eval*; they are constructed in *applyUserFun* in *EVALUATION*.

1.2.6 *NUMBERS*

The only kinds of values in this language are integers. However, this should not be expected to hold true in the future. On the contrary, the principal changes in the languages of Part 2 will be in replacing integers by other values, such as matrices. The operations defined here are: *prValue* (called from the top level and from *applyValueOp*); *isTrueVal* (called from *eval*); and *applyValueOp* (called from *eval*). The latter gets a *VALUEOP* and a *VALUELIST* as its arguments, and, after checking that the right number of arguments has been provided, applies the *VALUEOP* and returns the result.

There is another value-dependent function, *parseVal*, in *INPUT*, which will change when there are values other than integers.

1.2.7 *READ-EVAL-PRINT LOOP*

The body of the program is a *read-eval-print loop.* It *reads* an expression or function definition from the terminal; if an expression, it *evaluates* it by calling the function *eval*, and *prints* the value.

When a function definition is entered, it is stored in an unordered linked-list of *FUNDEFREC*'s, whose beginning is given by the global variable *fundefs*. Each *FUNDEFREC* contains a function's name, formal parameter list, and body.

1.2.8 *EVALUATION*

eval is the heart of the interpreter. Indeed, it is not too much to say that all the rest is just bookkeeping for *eval*. Accordingly, we will study it in some detail, quoting extensively from the code, which appears in Appendix B.

An expression is evaluated in a context in which the global variables have certain values, and, if the expression occurs within a function definition, the formal parameters likewise have values. Evaluation of an expression leads eventually to its value, and may also change the values of global variables or formal parameters. To explain the computation process, we need a definition:

Definition 1.1 An *environment* is an association of values (that is, integers) with variables. For environment ρ and variable $\mathtt{x}$, $\rho(\mathtt{x})$ denotes the value associated with $\mathtt{x}$ in ρ; we will say $\mathtt{x}$ is *bound* to $\rho(\mathtt{x})$ in ρ, or that $\rho(\mathtt{x})$ is the *binding* of $\mathtt{x}$ in ρ. $\{\mathtt{x}_1 \mapsto n_1, \ldots, \mathtt{x}_m \mapsto n_m\}$ denotes the environment that associates value n_i with variable $\mathtt{x}_i$, for $1 \leq i \leq m$.

So, to restate the preceding paragraph, an expression is evaluated with respect to an environment giving bindings for global variables — the *global* environment — and an environment giving bindings for the formal parameters of the function in which the expression occurs — the *local* environment. Note that `set` can be used to assign a new value to either a global variable or formal parameter; thus, an expression evaluation can result in a change to either environment. The global environment is given by the variable *globalEnv*, which is initialized in *READ-EVAL-PRINT LOOP*. The local environment passed to *eval* from *READ-EVAL-PRINT LOOP* is empty, but new local environments are created during the evaluation process, as we will see; the formal parameter *rho* of *eval* contains the local environment for the evaluation of *e*:

function *eval* (*e*: *EXP*; *rho*: *ENV*): *NUMBER*;

The action of *eval* depends upon the type of the expression *e* (integer constant, variable, or application) and, if it is an application, the type of its operator (user-defined, value-op, or control). Thus, the body of *eval* is a **case** statement:

```
begin (* eval *)
    with e↑ do
        case etype of
            VALEXP:
                eval := num;
            VAREXP:
                if isBound(varble, rho)
                then eval := fetch(varble, rho)
                else if isBound(varble, globalEnv)
                    then eval := fetch(varble, globalEnv)
                    else ... undefined variable error ...;
            APEXP:
                if optr > numBuiltins
                then eval := applyUserFun(optr, evalList(args))
                else begin
                    op := primOp(optr);
                    if op in [IFOP .. BEGINOP]
                    then eval := applyCtrlOp(op, args)
                    else eval := applyValueOp(op, evalList(args))
                end
        end (* case and with *)
end; (* eval *)
```

Thus, if *e* is an integer constant, the integer is returned; if it is a variable, it is looked up first in the local, and then in the global, environment; and if it is an application, one of the functions *applyUserFun*, *applyValueOp*, or *applyCtrlOp* is called. To elaborate further on the *APEXP* case:

- *applyUserFun* is called to apply a user-defined function. Before it is called, all the arguments are recursively evaluated by *evalList*, yielding a list of values, say $n_1, \ldots, n_m$ (and also possibly modifying the local and global environments). Then, if the function was defined as:

  (define f (x$_1$ ... x$_m$) e),

  e is evaluated in environment $\{\mathtt{x}_1 \mapsto n_1, \ldots, \mathtt{x}_m \mapsto n_m\}$:

```
function applyUserFun (nm: NAME; actuals: VALUELIST ): NUMBER;
var
    f: FUNDEF;
    rho: ENV;
begin
    f := fetchFun(nm);
    . . . check if f defined . . .
    with f↑ do begin
        . . . check number of arguments . . .
        rho := mkEnv (formals, actuals);
        applyUserFun := eval(body, rho)
        end
end; (* applyUserFun *)
```

- *applyValueOp* is called to apply a *VALUEOP*. Before it is called, all its arguments are (recursively) evaluated by calling *evalList*. *applyValueOp* was described under *NUMBERS*.
- *applyCtrlOp* applies *CONTROLOP*'s, which correspond to control structures in Pascal. Unlike user-defined functions and *VALUEOP*'s, the arguments to *CONTROLOP*'s are not evaluated. For instance, consider the expression `(set x e)`; its arguments are `x` and `e`, but clearly it would be wrong to evaluate `x`. The body of *applyCtrlOp* is a **case** statement:

```
function applyCtrlOp (op: CONTROLOP;
                      args: EXPLIST): NUMBER;
var n: NUMBER;
begin
    with args↑ do
        case op of
            IFOP: · · ·
            WHILEOP: · · ·
            SETOP: · · ·
            BEGINOP: · · ·
        end (* case and with *)
end; (* applyCtrlOp *)
```

The processing of these operators is straight-forward:

***IFOP*:** evaluate argument one; if it evaluates to a true (non-zero) value, then evaluate, and return the value of, argument two; otherwise, evaluate, and return the value of, argument three:

```
IFOP:
    if isTrueVal(eval(head, rho))
    then applyCtrlOp := eval(tail↑.head, rho)
    else applyCtrlOp := eval(tail↑.tail↑.head, rho);
```

WHILEOP: repeatedly evaluate argument one and argument two in turn, until argument one evaluates to false (zero):

```
WHILEOP:
   begin
      n := eval(head, rho);
      while isTrueVal(n)
      do begin
            n := eval(tail↑.head, rho);
            n := eval(head, rho)
         end;
      applyCtrlOp := n
   end;
```

SETOP: evaluate argument two and assign it to argument one, after determining whether the latter is a global variable or formal parameter:

```
SETOP:
   begin
      n := eval(tail↑.head, rho);
      if isBound(head↑.varble, rho)
      then assign(head↑.varble, n, rho)
      else if isBound(head↑.varble, globalEnv)
         then assign(head↑.varble, n, globalEnv)
         else bindVar(head↑.varble, n, globalEnv);
      applyCtrlOp := n
   end;
```

BEGINOP: evaluate all arguments and return the last value. (*BEGINOP* could have been treated as a *VALUEOP*, since it evaluates all its arguments, but it fits better with the control structures.)

```
BEGINOP:
   begin
      while args↑.tail <> nil do
      begin
         n := eval(args↑.head, rho);
         args := args↑.tail
      end;
      applyCtrlOp := eval(args↑.head, rho)
   end
```

1.3 Where we go from here

PASCAL is a representative of the class of *imperative programming languages*. This class includes such languages as FORTRAN, COBOL, ALGOL-60, and C, among many others. Indeed, these languages — often called "conventional," usually in a sense denoting mild exasperation — account for the vast majority of programming done in the real world.

Since the reader knows PASCAL well, he knows about imperative programming languages; or at least he knows about *imperative programming*, the prevailing programming style associated with these languages. Some of the characteristics of this style and of these languages are:

- "Word-at-a-time" processing. A computation consists of many individual movements and computations of small items of data.
- Programming by side-effect. The computation proceeds by continually changing the "state" of the machine — the values contained in the various machine words — by assignment.
- Iteration is the predominant control structure. Whether by explicit control structures (**while**, **repeat**) or **goto**, iteration is the primary method of control. Procedures, and particularly recursion, take a back seat.
- The language structures, both data and control, are fairly close to the underlying machine architecture. **goto**'s are unconditional jumps (**if**'s and **while**'s are bundled conditional and unconditional jumps), arrays are contiguous blocks of memory, pointers are pointers, assignment is data movement, and so on. For example, debuggers for imperative languages are often simple adaptations of machine-level debuggers.

Of course, knowing no alternatives, the reader may not realize that he has been programming in a particular style — he's just been programming! It is precisely the purpose of this book to acquaint the reader with the principal alternatives.

Our presentation of them is structured thusly:

Part 2. The languages LISP and APL differ from the language of this chapter in a conceptually small way: instead of manipulating (that is, assigning, passing as arguments, returning as results) integers, they manipulate larger values — lists and arrays, respectively. It transpires that this conceptually small change has a deep effect on one's programming habits. One result is that in LISP, recursion is the predominant control structure; where recursion in PASCAL is rather exotic, in LISP it is iteration that is rare.

Part 3. The *functional languages* are characterized by their treating functions as "first-class" (i.e. assignable, passable, returnable) values. A highly distinctive programming style emerges from this feature. The best-known functional language is SCHEME, a dialect of LISP. The language SASL incorporates *lazy evaluation*, which even more strongly supports the functional programming style. (λ-calculus, a mathematical system that inspired these languages, is covered in the SASL chapter.)

Part 4. The *object-oriented languages* share the feature that they allow users to add to the set of data types. The first one presented is CLU, which adds this capability in a very clean and simple way. SMALLTALK represents a more radical departure based on the same idea; here, the ability of a data type to *inherit* properties from a previously-defined type allows for effective code re-use. (Also briefly covered in these chapters are ADA and C++.)

Part 5. The concept of *logic programming*, whereby the principles of mathematical logic are adapted to programming, is represented by Prolog. Also discussed in this Part is the subject of *mechanical theorem-proving*, to which Prolog traces its origins.

Part 6. As far as these languages may take us from the world of imperative programming, eventually programs must run on computers. This Part discusses compilation in general, and an implementation issue especially important to non-imperative languages, that of *memory management.* (Note that the interpreters provided for the languages are little more than existence proofs, and bear little relation to how the languages are actually implemented.)

For each of the principal languages covered — Lisp, APL, Scheme, Sasl, Clu, Smalltalk, and Prolog — we present a simplified subset, giving its syntax (formally, as in this chapter) and semantics (informally), and many examples. In addition, we present an interpreter, adapted from the one presented in this chapter, which the reader can use to run programs in the language. That interpreter is documented in the chapter and listed in an Appendix.

The presentation of these languages, and of the other topics covered along the way, has ultimately one goal: to foster an appreciation of the range of programming languages and styles that are possible.

1.4 Exercises

1.4.1 Learning about the language

The first six exercises define some number-theoretic functions for you to program. Try to use recursion; it will be good practice for Chapter 2:

1. `(sigma` m n`)` $= m + (m+1) + \cdots + n$.

2. `(exp` m n`)` $= m^n$ $(m, n \geq 0)$. `(log` m n`)` = the least integer l such that $m^{l+1} > n$ $(m > 1, n > 0)$.

3. `(choose` n k`)` is the number of ways of selecting k items from a collection of n items, without repetitions, n and k non-negative integers. This quantity is called a *binomial coefficient*, and is notated $\binom{n}{k}$. It can be defined as $\frac{n!}{k!(n-k)!}$, but the following identities are more helpful computationally: $\binom{n}{0} = 1$ $(n \geq 0)$, $\binom{n}{n} = 1$ $(n \geq 0)$, and $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ $(n, k > 0)$.

4. `(fib` m`)` is the mth Fibonacci number. The Fibonacci numbers are defined by the identities: `(fib` 0`)` $= 0$, `(fib` 1`)` $= 1$, and for $m > 1$, `(fib` m`)` $=$ `(fib` $m-1$`)+(fib` $m-2$`)`.

5. `(prime` n`)` = true (1) if n is prime, false (0) otherwise. `(nthprime` n`)` = the nth prime number. `(sumprimes` n`)` = the sum of the first n primes. `(relprime` m n`)` = true if m and n are relatively prime (have no common divisors except 1), false otherwise.

6. `(binary` m`)` = the number whose decimal representation is the binary representation of m. E.g. `(binary 12)` $= 1100$, since $1100_2 = 12_{10}$.

7. The scope rule of our language is identical to that of Pascal, when we consider a Pascal without nested function definitions. Consider this Pascal program: