Comparison of New (Enhanced) vs Old (Original) Chapter 1 Interpreter

When running the original interpreter, users entered either a function definition or an expression using a Lisp-style syntax.

I made the following enhancements.

1. Replace the Lisp-style syntax with a Pascal-like syntax.
2. Add new commands (load & sload) that read user input from a file.

To implement these, I modified *parseDef* and added *parseExpr, processCmd* and their supporting functions to handle the following grammar rules.

input −→ quit | cmdline | fundef | expr
fundef −→ fun name (arglist) := expr nuf
expr −→ ifex | whileex | seqex | exp1
cmdline −→) command
command −→ load filename | sload filename

Below is a side by side comparison of the old vs. new grammar (Table 1) and the old vs. new syntax (Table 2) followed by an overview of the program changes (Table 3).

Table 1 Old vs. New Grammar

Old Grammar	New Grammar
input −→ expression | fundef	userinput −→ input $ input −→ quit | cmdline | fundef | expr cmdline −→) command note: the right parenthesis must be first char of input line command −→ load filename | sload filename filename −→ any valid filename for the operating system
fundef −→ (define function arglist expression) arglist −→ (variable*)	fundef −→ fun name (arglist) := expr nuf arglist −→ null | name [, name]*
expression −→ value | variable | (if expression expression expression) | (while expression expression) | (set variable expression) | (begin expression^{+}) | (optr expression*)	expr −→ ifex | whileex | seqex | exp1 ifex −→ if expr then expr else expr fi whileex −→ while expr do expr od seqex −→ seq explist qes explist −→ expr [; expr]* exp1 −→ exp2 [:= exp1]* exp2 −→ [prtop] exp3 exp3 −→ exp4 [relop exp4]* exp4 −→ exp5 [addop exp5]* exp5 −→ [addop] exp6 [mulop exp6]* exp6 −→ name | integer | funcall | (expr)
optr −→ function | value-op value −→ integer value-op −→ + | - | * | / | = | < | > | print function −→ name variable −→ name	funcall −→ name (actparmlist) actparmlist −→ null | expr [, expr]* prtop −→ print relop −→ = | < | > addop −→ + | - mulop −→ * | /
integer −→ sequence of digits, possibly preceded by minus sign[1] name −→ any sequence of characters not an integer and not containing a delimiter. delimiter −→ ’(’, ’)’, ’;’ or space	integer −→ sequence of digits name −→ any sequence of characters not an integer and not containing a delimiter. delimiter −→ '(', ')', ';', '+', '-', '*', '/', ':', '=', '<', '>', ',', '$', '!' or space note: [and] enclose optional parts of a production. [ ] indicates the option may occur once

The semicolon begins a comment. [1]Unary minus (and plus) are now implemented by the optional addop in the production for exp5 in the new grammar.	[ ]* indicates the option may occur more than once The exclamation point begins a comment.

Old vs. New Syntax

In the original interpreter, if the input begins with an open parenthesis then a corresponding close parenthesis will mark the completion of input and the function readParens is called from readInput to read until the parentheses are balanced.

In the enhanced interpreter, a dollar sign symbol marks the completion of user input and the function readDollar is called from readInput to read until a $ symbol is entered. If the user forgets to enter it then a continuation prompt is displayed until the $ is entered.

Table 2 Old vs. New Syntax

Chap1 Orig	Chap1 Enhanced
~$ ./chap1_orig -> 3 3 -> (+ 4 7) 11 Assignment is done via the set function. -> (set x 4) 4 -> (+ x x) 8 -> (print x) 4 4 -> (set y 5) 5 -> (begin (print x) (print y) (* x y)) 4 5 20 -> (if (> y 0) 5 10) 5 -> (while (> y 0) > (begin (set x (+ x x)) (set y (- y 1)))) 0 -> x 128 -> (define +1 (x) (+ x 1)) +1 -> (+1 4) 5 -> (define double (x) (+ x x)) double -> (double 4) 8 -> x 128 -> (define setx (x y) (begin (set x (+ x y)) x)) setx -> (setx x 1) 129	~$ ./chap1 -> 3$ 3 -> 4+7$ 11 Assignment is done via the := operator. -> x:=4$ 4 -> x+x$ 8 -> print x$ 4 4 -> y:=5$ 5 -> seq print x; print y; x*y qes$ 4 5 20 -> if y>0 then 5 else 10 fi$ 5 -> while y>0 do > seq x:=x+x; y:=y-1 qes > od$ 0 -> x$ 128 -> fun #1 (x) := x + 1 nuf$ #1 -> #1(4)$ 5 -> fun double(x):=x+x nuf$ double -> double(4)$ 8 -> x$ 128 -> fun setx(x,y):= seq x:=x+y; x qes nuf$ setx -> setx(x,1)$ 129

```
-> x
128

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

<> can be a function name since <, > are not delimiters.

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




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

-> (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




-> (gcd 6 15)
3

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

-> (gcd 6 15)
3
```

Lisp syntax forces correct operator precedence and associativity.

```
-> (+ (* 5 3) 7)
22

-> (+ 5 (* 3 7))
26

-> (- (- 14 7) 3)
4

-> (/ (/ 48 12) 2)
2
```

Keywords may be redefined but new definition is ignored since builtins occur before user-defined definitions in symbol table.

```
-> (define set (x) (+ x 5))
set
```

Above redefinition is ignored and builtin definition is applied.

```
-> (set y 20)
20
-> y
20
```

Keywords can be reused as variable names.

```
-> (set set 20)
20
-> (if (> set 15) 10 30)
10
```

```
-> x$
128

-> fun not(boolval):= if boolval then 0 else 1 fi nuf$
not
```

<> cannot be a function name since <, > are now delimiters.

```
-> fun <> (x, y):= not(x=y) nuf$
mutate: found <  where nameid or funid is expected.
```

is not a delimiter and can be used in a name.

```
-> fun ## (x,y):= not(x=y) nuf$
##

-> fun mod(m,n):=m-n*(m/n)nuf$
mod

-> fun gcd(m,n):=
>    seq
>      r:=mod(m,n);
>      while ##(r,0) do
>        seq
>          m:=n;
>          n:=r;
>          r:=mod(m,n)
>        qes
>      od;
>      n
>    qes
>  nuf$
gcd

-> gcd(6,15)$
3

-> fun gcd(m,n):=
>    if n=0 then m else gcd(n,mod(m,n)) fi nuf$
gcd

-> gcd(6,15)$
3
```

Normal operator precedence and associativity are implemented.

```
-> 5*3+7$
22

-> 5+3*7$
26

-> 14-7-3$
4

-> 48/12/2$
2
```

Keywords cannot be redefined.

```
-> fun if (x) := x+5 nuf$
mutate: found if  where nameid or funid is expected.
```

Keywords cannot be reused as variable names.

```
-> if := 20$
Error parsing expr.  Found := where one of the
following is expected: "if", "while", "seq", "print",
nameid, funid, number, or "("
```

Names are any sequence of chars not an integer and not containing a delimiter.

```
-> (set ~12#ab 25)
25
-> ~12#ab
25
-> (set x (+ (- 15 ~12#ab) 7))
-3
```

A string of digits is a valid name. This seems to work even though the grammar says that a name is not an integer.

```
-> (define 222 (x) (+ x 222))
222
-> (222 3)
225
-> 222
222
```

Inserting a delimiter in a name confuses the parser and causes erroneous results. Below, the name ″a(b″ causes unbalanced parentheses which evokes the continuation prompt to input a final right parenthesis.

```
-> (set a(b 25)
> )
Undefined function: b
```

Function name may be reused as a variable name.

```
-> (define inc10 (x) (+ x 10))
inc10
-> (set inc10 50)
50
-> (inc10 inc10)
60
```

Multiple assignment

```
-> (set i (set j (set k 25)))
25
-> i
25
-> j
25
-> k
25
```

Names are any sequence of chars not an integer and not containing a delimiter.

```
-> ~12#ab:=25$
25
-> ~12#ab$
25
-> x:=15-~12#ab+7$
-3
```

A string of digits is not a valid name.

```
-> fun 222 (x):= x+22 nuf$
mutate: found 222  where nameid or funid is expected.
-> fun 2#2 (x):=x+22 nuf$
2#2
-> 2#2(33)$
55
-> a:=55-2#2(33)$
0
```

Inserting a delimiter in a name confuses the parser and causes erroneous results.

```
-> a(b:=25$
Undefined variable: a
```

Function name may not be reused as a variable name.

```
-> fun inc10 (x) := x+10 nuf$
inc10
-> inc10 := 50$
Error in match. Found :=  where ( is expected.
```

Multiple assignment. **:=** is the only right associative operator.

```
-> i:=j:=k:=25$
25
-> i$
25
-> j$
25
-> k$
25
```

New load from file commands are illustrated below.
They must be entered on a single input line. The $ end of input marker may be omitted since it is appended by the program.

load echoes the text read from the file.
sload (silent load) does not echo the text read.

```
-> )load mod.txt

Current Directory is :
/home/dawsond/pascal/proglang/chap1
 Loading file : mod.txt

 fun mod(m,n):=
 m-n*(m/n)
 nuf$
mod

-> )sload not.txt

Current Directory is :
/home/dawsond/pascal/proglang/chap1
 Loading file : not.txt

not
-> )load gcd_only.txt$
```

```
Current Directory is :
/home/dawsond/pascal/proglang/chap1
 Loading file : gcd_only.txt

fun gcd(m,n):=
 seq
  r:=mod(m,n);
  while not(r=0) do
   seq
    m:=n;
    n:=r;
    r:=mod(m,n)
   qes
  od;
  n
 qes
nuf$
gcd

gcd(51,34)$
17

gcd(225,300)$
75

mod(10,7)$
3

mod(100,50)$
0

not(3)$
0

not(0)$
1
```

Program Changes

The original source code is divided into the following 8 sections: DECLARATIONS, DATA STRUCTURE OPS, NAME MANAGEMENT, INPUT, ENVIRONMENTS, NUMBERS, EVALUATION, MAIN.

In order to implement the new grammar presented in Table 1 above, the DECLARATIONS, NAME MANAGEMENT, INPUT and MAIN sections were modified and a section entitled NEW PARSING ROUTINES was added. The DATA STRUCTURE OPS, ENVIRONMENTS, NUMBERS, EVALUATION sections were not changed.

The overall design of the enhanced interpreter is the same as the original with the same builtin, control and value operations. The only thing that has changed is the input syntax and the added feature of reading user input from a file.

Table 3 on the next page below lists the complete source code for the enhanced interpreter. The code is organized so that the main program body is at the bottom of the file and so that lower-level functions & procedures are declared before the higher-level ones that call them. This avoids the need for forward declarations in most cases.

For example, consider the following grammar rules for expressions and their associated function prototypes below. The rules are presented in Table 1 in a top-down, recursive-descent order but the corresponding functions are defined in a reverse, bottom-up order in the NEW PARSING ROUTINES section.

Top-Down Order in Grammar	Bottom-Up Order in Source Code
expr −→ ifex | whileex | seqex | **exp1** **exp1** −→ exp2 [:= exp1]* exp2 −→ [prtop] exp3 exp3 −→ exp4 [relop exp4]* exp4 −→ exp5 [addop exp5]* exp5 −→ [addop] exp6 [mulop exp6]* exp6 −→ name | integer | funcall | (**expr**)	function parseExp6:EXP; function parseExp5:EXP; function parseExp4:EXP; function parseExp3:EXP; function parseExp2:EXP; function parseExp1:EXP; function parseExpr;

So if you would like a top-down view of the code, it will be best to read it from the bottom up after reviewing the DECLARATIONS section.

Table 3 Description

Table 3 below lists the source code and describes the changes that were made.
The color-coding in the columns have the following meanings.

Column 2 below lists the complete source code for the Enhanced version.
Color-coding:

- Black = original source code
- Red = new or changed source code
- Bold Blue = Extra comments about the changes that were not included with the source code.
  - These comments are prefixed with my initials so they can easily be reviewed by searching for ″DD:″.

Column 1 below lists original code that was changed in or omitted from Column 2.
Color-coding:

- Black = original code that is changed on the right
- Purple = original code that is omitted on the right

Table 3 Program Changes

COMMENTCHAR = ';';	(*** * DECLARATIONS * ***) PROGRAM chapter1 (input, output); Uses sysutils; label 99; CONST NAMELENG = 20; (* Maximum length of a name *) MAXNAMES = 100; (* Maximum number of different names *) MAXINPUT = 500; (* Maximum length of an input *) CMDLENG = 8; (* Maximum length of a command name *) NUMCMDS = 2; (* Number of commands currently defined *) ARGLENG = 40; (* Maximum length of a command argument *) PROMPT = '-> '; (* Initial prompt *) PROMPT2 = '> '; (* continuation prompt *) **DD: The char for starting a comment was changed from a semicolon to an exclamation point.** COMMENTCHAR = '!'; TABCODE = 9; LINEFEED = 10; CR = 13; DOLLAR = '$'; (* marks end of the expr or fundef input by the user *) TYPE NAMESIZE = 0..NAMELENG; NAMESTRING = packed array [1..NAMELENG] of char; NUMBER = integer; NAME = 1 .. MAXNAMES; (* a NAME is an index in printNames *) CMDSIZE = 0..CMDLENG; CMDSTRING=packed array [1..CMDLENG] of char; CMD = 1..NUMCMDS; (* a CMD is an index in printCmds *) ARGSIZE = 0..ARGLENG; ARGSTRING = packed array [1..ARGLENG] of char;

```
 BUILTINOP = (IFOP,WHILEOP,ASSIGNOP,SEQOP,ADDOP,SUBOP,MULOP,DIVOP,EQOP,
LTOP,GTOP,PRINTOP);
 VALUEOP = ADDOP .. PRINTOP;
 CONTROLOP = IFOP .. SEQOP;
```

DD: Below are symbolic names for each token. They are stored in the toktable array at the same index as the corresponding token in the printNames array by the initNames & install functions.

```
 TOKEN = (nameidsy,numsy,funidsy,ifsy,thensy,elsesy,fisy,whilesy,dosy,
 odsy,seqsy,qessy,funsy,nufsy,assignsy,rparsy,
 lparsy,semsy,comsy,addsy,subsy,mulsy,divsy,eqsy,lssy,
 gtsy,printsy,quitsy,dollarsy);


  EXP = ^EXPREC;
  EXPLIST = ^EXPLISTREC;
  ENV = ^ENVREC;
  VALUELIST = ^VALUELISTREC;
  NAMELIST = ^NAMELISTREC;
  FUNDEF = ^FUNDEFREC;

  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;

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

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

  ENVREC = record
              vars: NAMELIST;
              values: VALUELIST
```

```
            end;

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

var
   fundefs: FUNDEF;

   numval:NUMBER;

   globalEnv: ENV;

   currentExp: EXP;

   punctop: set of char;    (* set of punctuation & operator chars *)

   userinput: array [1..MAXINPUT] of char;
   inputleng, pos: 0..MAXINPUT;

   printNames: array [NAME] of NAMESTRING;
   numNames, numBuiltins, tokindex, mulsy_index: NAME;

   tokstring:NAMESTRING;  (* string & length for display in error msgs *)
   tokleng:NAMESIZE;

   infilename:ARGSTRING;
   infile:text;    (* file variable for input source file *)

   printCmds:array [CMD] of CMDSTRING;
   load,sload:CMD;

   toksy: TOKEN;   (* current token returned from getToken or install *)
   toktable: array [NAME] of TOKEN; (* holds symbolic name of each token in
                                       the printnames array. Corresponding
                                       toktable & printNames entries have the
                                       same index. See initNames & install. *)
   addops,mulops,relops:set of token;  (* sets of operators *)

   quittingtime,
   dollarflag,       (* true = $ was input. $ marks end of current expr or
                        fundef being input *)
   echo,             (* true = echo characters during a load command *)
   readfile:boolean; (* true if an input file is being loaded *)
```

```
(*****************************************************************
 *                      DATA STRUCTURE OP'S                      *
 *****************************************************************)

(* mkVALEXP - return an EXP of TYPE VALEXP with num n            *)
function mkVALEXP (n: NUMBER): EXP;
var e: EXP;
begin
   new(e);
   e^.etype := VALEXP;
   e^.num := n;
   mkVALEXP := e
end; (* mkVALEXP *)

(* mkVAREXP - return an EXP of TYPE VAREXP with varble nm        *)
function mkVAREXP (nm: NAME): EXP;
var e: EXP;
begin
   new(e);
   e^.etype := VAREXP;
   e^.varble := nm;
   mkVAREXP := e
end; (* mkVAREXP *)

(* mkAPEXP - return EXP of TYPE APEXP w/ optr op and args el     *)
function mkAPEXP (op: NAME; el: EXPLIST): EXP;
var e: EXP;
begin
   new(e);
   e^.etype := APEXP;
   e^.optr := op;
   e^.args := el;
   mkAPEXP := e
end; (* mkAPEXP *)

(* mkExplist - return an EXPLIST with head e and tail el         *)
function mkExplist (e: EXP; el: EXPLIST): EXPLIST;
var newel: EXPLIST;
begin
   new(newel);
   newel^.head := e;
   newel^.tail := el;
   mkExplist := newel
end; (* mkExplist *)

(* mkNamelist - return a NAMELIST with head n and tail nl        *)
function mkNamelist (nm: NAME; nl: NAMELIST): NAMELIST;
var newnl: NAMELIST;
```

```
begin
   new(newnl);
   newnl^.head := nm;
   newnl^.tail := nl;
   mkNamelist := newnl
end; (* mkNamelist *)

(* mkValuelist - return an VALUELIST with head n and tail vl     *)
function mkValuelist (n: NUMBER; vl: VALUELIST): VALUELIST;
var newvl: VALUELIST;
begin
   new(newvl);
   newvl^.head := n;
   newvl^.tail := vl;
   mkValuelist := newvl
end; (* mkValuelist *)

(* mkEnv - return an ENV with vars nl and values vl              *)
function mkEnv (nl: NAMELIST; vl: VALUELIST): ENV;
var rho: ENV;
begin
   new(rho);
   rho^.vars := nl;
   rho^.values := vl;
   mkEnv := rho
end; (* mkEnv *)

(* lengthVL - return length of VALUELIST vl                      *)
function lengthVL (vl: VALUELIST): integer;
var i: integer;
begin
   i := 0;
   while vl <> nil do begin
      i := i+1;
      vl := vl^.tail
      end;
   lengthVL := i
end; (* lengthVL *)

(* lengthNL - return length of NAMELIST nl                       *)
function lengthNL (nl: NAMELIST): integer;
var i: integer;
begin
   i := 0;
   while nl <> nil do begin
      i := i+1;
      nl := nl^.tail
      end;
```

```
   lengthNL := i
end; (* lengthNL *)

(*************************************************************
 *                     NAME MANAGEMENT                       *
 *************************************************************)

(* fetchDef - get function definition of fname from fundefs      *)
function fetchDef (fname: NAME): FUNDEF;
var
   f: FUNDEF;
   found: Boolean;
begin
   found := false;
   f := fundefs;
   while (f <> nil) and not found do
      if f^.funname = fname
      then found := true
      else f := f^.nextfundef;
   fetchDef := f
end; (* fetchDef *)

(* newDef - add new function fname w/ parameters nl, body e   *)
procedure newDef (fname: NAME; nl: NAMELIST; e: EXP);
var f: FUNDEF;
begin
   f := fetchDef(fname);
   if f = nil (* fname not yet defined as a function *)
   then begin
           new(f);
           f^.nextfundef := fundefs; (* place new FUNDEFREC *)
           fundefs := f                      (* on fundefs list *)
        end;
   f^.funname := fname;
   f^.formals := nl;
   f^.body := e
end; (* newDef *)

(* initNames - place all pre-defined names into printNames
               and corresponding token symbols in toktable. *)

procedure initNames;
var i: integer;
begin
   fundefs := nil;
   i := 1;
   printNames[i] := 'if                 '; toktable[i] := ifsy;     i := i+1;
   printNames[i] := 'while              '; toktable[i] := whilesy;  i := i+1;
```

```
   printNames[i] := ':=                  '; toktable[i] := assignsy; i := i+1;
   printNames[i] := 'seq                 '; toktable[i] := seqsy;    i := i+1;
   printNames[i] := '+                   '; toktable[i] := addsy;    i := i+1;
   printNames[i] := '-                   '; toktable[i] := subsy;    i := i+1;

(* To handle negative numbers (unary minus), we build an expr with the multiply
operator and operand -1. Below we save the multiply symbol index for this
purpose. This avoids having to do a lookup to obtain the index. *)

   mulsy_index:=i;
   printNames[i] := '*                   '; toktable[i] := mulsy;   i := i+1;
   printNames[i] := '/                   '; toktable[i] := divsy;   i := i+1;
   printNames[i] := '=                   '; toktable[i] := eqsy;    i := i+1;
   printNames[i] := '<                   '; toktable[i] := lssy;    i := i+1;
   printNames[i] := '>                   '; toktable[i] := gtsy;    i := i+1;
   printNames[i] := 'print               '; toktable[i] := printsy; i := i+1;
   printNames[i] := 'quit                '; toktable[i] := quitsy;  i := i+1;
   printNames[i] := 'then                '; toktable[i] := thensy;  i := i+1;
   printNames[i] := 'else                '; toktable[i] := elsesy;  i := i+1;
   printNames[i] := 'fi                  '; toktable[i] := fisy;    i := i+1;
   printNames[i] := 'do                  '; toktable[i] := dosy;    i := i+1;
   printNames[i] := 'od                  '; toktable[i] := odsy;    i := i+1;
   printNames[i] := 'qes                 '; toktable[i] := qessy;   i := i+1;
   printNames[i] := 'fun                 '; toktable[i] := funsy;   i := i+1;
   printNames[i] := 'nuf                 '; toktable[i] := nufsy;   i := i+1;
   printNames[i] := '(                   '; toktable[i] := lparsy;  i := i+1;
   printNames[i] := ')                   '; toktable[i] := rparsy;  i := i+1;
   printNames[i] := ';                   '; toktable[i] := semsy;   i := i+1;
   printNames[i] := ',                   '; toktable[i] := comsy;   i := i+1;
   printNames[i] := '$                   '; toktable[i] := dollarsy;
   numNames := i;
   numBuiltins := i
end; (* initNames *)

(* install - insert new name into printNames                          *)
function install (nm: NAMESTRING): NAME;
var
   i: integer;
   found: Boolean;
begin
   i := 1; found := false;
   while (i <= numNames) and not found
   do if nm = printNames[i]
      then found := true
      else i := i+1;
   if not found
   then begin
           if i > MAXNAMES
```

```
          then begin
                  writeln('No more room for names');
                  goto 99
               end;
          numNames := i;
          printNames[i] := nm;
          toktable[i] := nameidsy
       end;
   toksy := toktable[i];   (* return token symbol in global var *)
   install := i
end; (* install *)

(* initCmds -  place all pre-defined commands into printCmds  *)
procedure initCmds;
var i: integer;
begin
   i := 1;
   printCmds[i] := 'sload   '; sload  := i; i := i+1;
   printCmds[i] := 'load    '; load   := i; i := i+1;
(* printCmds[i] := 'xxxxxx  '; xxxxxx := i; i := i+1; *)
end;  (* of initCmds *)

(* initParse - initialization of variables *)
procedure initParse;

begin
  initCmds;
  readfile:=false;
  echo := false;
  addops := [addsy,subsy];
  mulops := [mulsy,divsy];
  relops := [lssy,eqsy,gtsy];
  punctop := ['(', ')', '+', '-', '*', '/', ':', '=', '<', '>', ';', ',', '$',
COMMENTCHAR];
end;

(* prName - print name nm                                          *)
procedure prName (nm: NAME);
var i: integer;
begin
   i := 1;
   while i <= NAMELENG
   do if printNames[nm][i] <> ' '
      then begin
              write(printNames[nm][i]);
              i := i+1
           end
      else i := NAMELENG+1  (* exit while loop *)
```

```
isDelim := c in ['(', ')', ' ', COMMENTCHAR]
```

```
end; (* prName *)

(* primOp - translate NAME optr to corresponding BUILTINOP       *)
function primOp (optr: NAME): BUILTINOP;
var
   op: BUILTINOP;
   i: integer;
begin
   op := IFOP; (* N.B. IFOP is first value in BUILTINOPS *)
   for i := 1 to optr-1 do op := succ(op);
   primOp := op
end; (* primOp *)

(*****************************************************************
 *                          INPUT                                *
 *****************************************************************)
```

DD: The functions parseCmd, parseName and isNumber each call the isDelim function below. parseCmd and parseName read chars until a delimiter is encountered in order to obtain the command/variable/function name. isNumber reads digits until a nondigit is encountered and requires that the nondigit be a delimiter.

```
(* isDelim - check if c is a delimiter                           *)
function isDelim (c:char): Boolean;
begin
  isDelim := (c = ' ') or (c in punctop)
end;

(* skipblanks - return next non-blank position in userinput      *)
function skipblanks (p: integer): integer;
begin
   while userinput[p] = ' ' do p := p+1;
   skipblanks := p
end; (* skipblanks *)

(* reader - read char's into userinput; be sure input not blank  *)
procedure reader;

(* readInput - read char's into userinput                        *)
   procedure readInput;

   var c: char;
```

DD: New code was added to the nextchar function below to deal with reading input chars from a file instead of the terminal.

```
(* nextchar - read next char - filter and comments
```

```
 * DD: Also filter CR/LF which were returned in input stream under WSL/Cygwin.
 *)
     procedure nextchar (var c: char);
     begin
        if readfile then
          begin
            read(infile,c);  (* read file *)
            if eof(infile) then
              begin
                readfile:=false;
                echo:=false;
(*
   The next line below assigns a '$' to c to mark the end of the input.
   Returning the space that is read at eof is not acceptable because the
   program will proceed by displaying the continuation prompt (PROMPT2).
   If the user wants to enter another command (e.g. a 2nd load command)
   then he cannot do so in response to PROMPT2. Commands are only checked for
   and processed in reponse to the main prompt (PROMPT).
   To avoid this, we return $ in order to force the program
   to process the current userinput and then display the main prompt.
*)
                c := DOLLAR;
                CLOSE(infile)
              end;
            if echo then write(c);
          end
        else
          read(c);  (* read standard input *)

(* Replace tab and eoln chars with space, skip comments *)
        if (c = chr(TABCODE)) or (c = chr(LINEFEED)) or (c = chr(CR))
        then c := ' '
        else if c = COMMENTCHAR then
             begin
               if readfile then
                 while not eoln(infile) do
                   begin
                     read(infile,c);
                     if echo then write(c)  (*echo comment *)
                   end
               else
                 while not eoln do read(c);
               c := ' '   (* replace eoln char *)
             end
     end; (* nextchar *)

DD: The readDollar function below replaces readParens.
readInput used to call readParens to read chars until a closing right
```

parenthesis was entered which marked the end of the current expr or fundef. Now readDollar is used to read until a dollar sign is entered which marks the completion of the current input.

```
(* readDollar - read char's, ignoring newlines, till '$' is read  *)
(*              '$' marks end of the fundef or expr that is being input *)
     procedure readDollar;
     var
        c: char;
     begin
        c := ' ';
        repeat
           if not readfile and eoln then write(PROMPT2);
           nextchar(c);
           pos := pos+1;
           if pos = MAXINPUT
           then begin
                   writeln('User input too long');
                   goto 99
                end;
           userinput[pos] := c
        until c = dollar;
        dollarflag := true;
     end; (* readDollar *)
```

DD: The next four functions, readCmd, parseCmd, parseCmdArg and processCmd handle reading, parsing and executing the new load and sload commands. If readInput detects a right parenthesis as first character of the input line then it calls processCmd (which calls the others) to open the file and set readfile to true. While readfile is true, the nextchar function will read input chars from the opened file instead of the terminal.

```
(* readCmd - read command line into the userinput buffer for processing. *)
procedure readCmd;
var
 c:char;
begin
  c := ' ';
  while not eoln do  (* commands are assumed to be entered on one line *)
    begin
      pos:=pos+1;
      nextchar(c);
      userinput[pos]:=c;
    end;

  inputleng:=pos;
  if userinput[inputleng] = DOLLAR then
     inputleng := inputleng - 1;  (* exclude $ from command line, if any *)
```

```
(*
  Next read removes the LF (under WSL) or CR (under Cygwin)
  that follows the $ in the input stream so it is not
  accepted as input once the main prompt is displayed
*)
  read(c)
end; (* of readCmd *)

(* parseCmd - return Cmd starting at userinput[pos] *)
function parseCmd: CMD;
var
   nm: CMDSTRING; (* array to accumulate characters *)
   leng: CMDSIZE; (* length of CMD *)
   i:integer;
   found: Boolean;

begin
   nm[1] := #0;
   leng := 0;
   while (pos < inputleng) and not isDelim(userinput[pos])
   do begin
         if leng = CMDLENG
         then begin
                 writeln('Command Name too long, begins: ', nm);
                 goto 99
              end;
         leng := leng+1;
         nm[leng] := userinput[pos];
         pos := pos+1
      end;

   if leng = 0
   then begin
          writeln('Error: expected Command name, instead read: ',
                   userinput[pos]);
          goto 99
       end;
   for leng := leng+1 to CMDLENG do nm[leng] := ' ';

   i := 1; found := false;
   while (i <= NUMCMDS) and not found
   do if nm = printCmds[i]
      then found := true
      else i := i+1;

   if not found then
     begin
       writeln('Unrecognized Command Name  begins: ',nm);
```

```
      goto 99
    end;

  pos := skipblanks(pos); (* skip blanks after command name *)
  parseCmd := i;
end; (* parseCmd *)

(* parseCmdArg - return the character string argument starting at
userinput[pos]*)
(*  This function is currently used to parse the filename argument from the
    load & sload commands *)
function parseCmdArg: ARGSTRING;
var
  nm: ARGSTRING; (* array to accumulate characters *)
  leng: ARGSIZE; (* length of name *)
begin
  nm[1] := #0;
  leng := 0;
  while (pos <= inputleng) and not (userinput[pos] = ' ')
  do begin
       if leng = ARGLENG
       then begin
              writeln('Argument name too long, begins: ', nm);
              goto 99
            end;
       leng := leng+1;
       nm[leng] := userinput[pos];
       pos := pos+1
    end;
  if leng = 0
  then begin
         writeln('Error: expected argument name, instead read: ',
                 userinput[pos]);
         goto 99
       end;
  for leng := leng+1 to ARGLENG do nm[leng] := ' ';
  parseCmdArg := nm
end; (* parseCmdArg *)

(* processCmd - input, parse, and execute the command *)
procedure processCmd;
var
  i,j: integer;
  cmdnm:CMD;  (* cmdnm is an index to printCmds *)
begin
  readCmd;
  pos:=skipblanks(1);     (* get pos of ")" which begins each command *)
  pos:=skipblanks(pos+1); (* get pos of 1st letter of command name *)
```

DD: Old logic below: If input begins with a "(" then call readParens to read chars until the parentheses are balanced which marks the end of input.
New logic to the right of old logic below: call

```
   cmdnm:=parseCmd;
   if (cmdnm = sload) or (cmdnm = load) then
     begin
       infilename:=parseCmdArg;   (* parse filename argument *)
       i := 1;
       while (infilename[i] <> ' ') do
         i := i + 1;
       for j := i to ARGLENG do infilename[j] := #0;   (*Null padding fixes
File Not Found on WSL*)
       writeln;
       writeln('Current Directory is : ',GetCurrentDir);
       writeln(' Loading file : ',infilename);
       writeln;
       Assign(infile,infilename);
       RESET(infile);
       readfile:=true;  (* tell nextchar function to read from file *)
       if cmdnm = load then
         echo:=true;
     end;
end;  (* of processCmd *)

   begin (* readInput *)
      c := ' ';
      dollarflag := false;
      if not readfile then write(PROMPT);
      pos := 0;
      repeat
         pos := pos+1;
         if pos = MAXINPUT
         then begin
                 writeln('User input too long');
                 goto 99
              end;
         nextchar(c);
         userinput[pos] := c;

         if (pos=1) and (c=')') then   (* if it's a command, then execute it*)
           begin
             processCmd;
             if not readfile then write(PROMPT);
             pos:=0
           end
         else    (* otherwise read expr or fundef terminated by dollar sign *)
             if userinput[pos] = dollar then
               dollarflag:=true
             else
               if readfile then
                 begin
```

readDollar to read until a "$" is entered which marks the completion of the input.

```
        if userinput[pos] = '(' then readParens
      until eoln;
```

```
                  if eoln(infile) then readDollar
                end
              else
                if eoln then readDollar
      until dollarflag;

      pos:=pos-1;  (* exclude $ from user input *)
      inputleng := pos;
      if readfile and echo then writeln (* echo blank line between inputs *)
   end; (* readInput *)

begin (* reader *)
    repeat
       readInput;
       pos := skipblanks(1);
    until pos <= inputleng (* ignore blank lines *)
end; (* reader *)

(* parseName - return (installed) NAME starting at userinput[pos]*)
function parseName: NAME;
var
   nm: NAMESTRING; (* array to accumulate characters *)
   leng: NAMESIZE; (* length of name *)
begin
   nm[1] := #0;
   leng := 0;
   while (pos <= inputleng) and not isDelim(userinput[pos])
   do begin
         if leng = NAMELENG
         then begin
                 writeln('Name too long, begins: ', nm);
                 goto 99
              end;
         leng := leng+1;
         nm[leng] := userinput[pos];
         tokstring[leng]:=userinput[pos];
         pos := pos+1
      end;
   tokleng:=leng;
   if leng = 0
   then begin
          writeln('Error: expected name, instead read: ',
                  userinput[pos]);
          goto 99
        end;
   for leng := leng+1 to NAMELENG do nm[leng] := ' ';
   pos := skipblanks(pos); (* skip blanks after name *)
   parseName := install(nm)
```

DD: The purple text below is omitted from the new logic on the right since unary minus is now handled in the parseExp5 function.

```
  isNumber := isDigits(pos) or
     ((userinput[pos] = '-') and isDigits(pos+1))
```

```
var n, sign: integer;
begin
   n := 0; sign := 1;
   if userinput[pos] = '-'
   then begin
           sign := -1;
           pos := pos+1
        end;
```

```
end; (* parseName *)

(* isNumber - check if a number begins at pos                    *)
function isNumber (pos: integer): Boolean;

(* isDigits - check if sequence of digits begins at pos          *)
   function isDigits (pos: integer): Boolean;
   begin
      if not (userinput[pos] in ['0'..'9']) then
        isDigits := false
      else
        begin
          isDigits := true;
          while userinput[pos] in ['0'..'9'] do pos := pos + 1;
          if not isDelim(userinput[pos])
          then isDigits := false
        end
   end; (* isDigits *)

begin (* isNumber *)
   isNumber := isDigits(pos)
end; (* isNumber *)

(* parseVal - return number starting at userinput[pos]           *)
function parseVal: NUMBER;
var n: integer;
begin
   n := 0;
   tokleng:=0;




   while userinput[pos] in ['0'..'9'] do
      begin
         n := 10*n + (ord(userinput[pos]) - ord('0'));
         tokleng:=tokleng+1;
         tokstring[tokleng]:=userinput[pos];
         pos := pos+1
      end;
   pos := skipblanks(pos); (* skip blanks after number *)
   parseVal := n
end; (* parseVal *)

(*************************************************************************
                          NEW PARSING ROUTINES
 *************************************************************************)
procedure writeTokenName(t:token);
```

```
(* write the specific token name in printnames array that corresponds to
   token symbol t.  If t is generic (i.e. nameidsy,funidsy,numsy) then
   write that generic name *)
var
 i:NAME;
 generic:set of token;
 j:NAMESIZE;
begin
 generic := [nameidsy,numsy,funidsy];
 if t in generic then
(* output generic name *)
   begin
     case t of
       nameidsy:write('nameid');
         numsy:write('number');
       funidsy:write('funid');
       otherwise;
     end;
   end
 else
(* output specific name *)
  begin
    i:=1;
    while (toktable[i] <> t) and (i <= numBuiltins) do
      i:= i+1;
    if i <= numBuiltins then
    (* write the name of the token *)
      begin
        j:=1;
        while (printNames[i][j] <> ' ') and (j <= NAMELENG) do
          begin
            write(printNames[i][j]);
            j:=j+1
          end
      end
    else   (* name not found, write the symbolic name *)
      write(t)
  end
end;  (* of writeTokenName *)

procedure writeTokenString;
(* Write out chars of token string.
   During errors, this function is used to display invalid string
   found in the userinput *)
var
 i:integer;
begin
 for i:= 1 to tokleng do
```

```
   write(tokstring[i]);
 write(' ');
end;

procedure errmsg(errnum:integer);
(* displays error messages based on the given error number *)
begin
  writeln;
  CASE errnum of
    1:begin
       write('Error parsing arglist.  Found ');
       writeTokenString;
       writeln('where ")" or nameid is expected.');
      end;
    2:begin
       write('Error parsing function name.  Found ');
       writeTokenString;
       writeln('funid or nameid is expected.');
      end;
    3:begin
       write('Error parsing exp6.  Found ');
       writeTokenString;
       writeln('where nameid, funid, "(", or a number  is expected.');
      end;
    4:begin
       write('Error parsing expr.  Found ');
       writeTokenString;
       writeln('where one of the following is expected: ');
       writeln('"if", "while", "seq", "print", nameid, funid, number, or "("
');
      end;
    otherwise;
  end;
  writeln;
  goto 99
end;  (* of errmsg *)

procedure getToken;
(* Identify token that begins at userinput[pos], return its symbol in global
   variable toksy and leave pos pointing to first nonblank that follows. *)
var
   nm: NAMESTRING; (* array to accumulate characters *)
   leng: NAMESIZE; (* length of name *)

begin
  if isNumber(pos) then   (* parse a number *)
    begin
      numval := parseVal;
```

```
      toksy := numsy
    end

  else if (userinput[pos] = ':') and (userinput[pos+1] = '=') then
   (* parse an assignment *)
    begin
      leng := 2;
      nm[1] := ':';
      nm[2] := '=';

      tokleng := leng;
      tokstring[1] := ':';
      tokstring[2] := '=';

      pos := pos + 2;
      for leng := leng+1 to NAMELENG do nm[leng] := ' ';
      pos := skipblanks(pos);
      tokindex := install(nm);
      toksy := toktable[tokindex]
    end

  else if userinput[pos] in punctop then  (* parse single char punct or
operator *)
    begin
      leng := 1;
      nm[1] := userinput[pos];

      tokleng := leng;
      tokstring[1] := userinput[pos];

      pos := pos + 1;
      for leng := leng+1 to NAMELENG do nm[leng] := ' ';
      pos := skipblanks(pos);
      tokindex := install(nm);
      toksy := toktable[tokindex]
    end

  else    (* else parse a name *)
    tokindex := parseName
end; (* getToken *)

procedure mutate(newtype:token);
(* change nameidsy to funidsy or vice versa *)
begin
  if (toksy <> nameidsy) and (toksy <> funidsy) then
    begin
      write('mutate: found ');
      writeTokenString;
```

parseNL below is now renamed to parseParams on the right since it parses parameter list of a fundef.

```
(* return NAMELIST starting at userinput[pos] *)
function parseNL: NAMELIST;
var
   nm: NAME;
   nl: NAMELIST;
begin

   if userinput[pos] = ')'
   then begin
           pos := skipblanks(pos+1); (* skip ')' *)
           parseNL := nil
        end

   else begin
           nm := parseName;




           nl := parseNL;
```

```
      writeln(' where nameid or funid is expected.');
      goto 99
    end
  else
    toktable[tokindex] := newtype
end;   (* of mutate *)

(* match the expected token t and get next one.
   Explanation: If the expected token t matches the current one in toksy
   then call getToken to return the next token from userinput in toksy *)
procedure match(t:token);
begin
   if toksy = t then
     getToken
   else
     begin
       write('Error in match. Found ');
       writeTokenString;
       write(' where ');
       writeTokenName(t);
       writeln(' is expected.');
       goto 99
     end;
end;  (* of match *)

function parseExpr:EXP;forward;

(* parse parameters of a fundef *)
function parseParams:NAMELIST;
var
   nm:NAME;
   nl:NAMELIST;
begin
   CASE toksy of
     rparsy:  parseParams := nil;




     nameidsy: begin
                 nm:=tokindex;
                 match(nameidsy);
                 if toksy = comsy then
                   begin
                     match(comsy);
                     nl:=parseParams
                   end
```

```
          parseNL := mkNamelist(nm, nl)
        end
end; (* parseNL *)
```

```
  pos := skipblanks(pos+1); (* skip '( ..' *)
  pos := skipblanks(pos+6); (* skip 'define ..' *)
  fname := parseName;
```

```
  pos := skipblanks(pos+1); (* skip '( ..' *)
  nl := parseNL;

  e := parseExp;
  pos := skipblanks(pos+1); (* skip ') ..' *)
```

```
              else
                nl:=nil;
              parseParams := MkNamelist(nm,nl)
            end;
     otherwise;
         errmsg(1)
   end
end;  (* of parseParams *)

(* parseDef - parse function definition at userinput[pos]        *)
function parseDef:NAME;
var
   fname: NAME;        (* function name *)
   nl: NAMELIST;       (* formal parameters *)
   e: EXP;             (* body *)
begin
   match(funsy);
   mutate(funidsy);
   fname := tokindex;

   CASE toksy of
     nameidsy:match(nameidsy);
     funidsy:match(funidsy);
     otherwise;
       errmsg(2)
   end;
   match(lparsy);
   nl := parseParams;
   match(rparsy);
   match(assignsy);
   e := parseExpr;
   match(nufsy);
   newDef(fname, nl, e);
   parseDef := fname
end; (* parseDef *)

(* parse arguments of a function call *)
function parseArgs:EXPLIST;
var
  ex:EXP;
  eL:EXPLIST;
begin
  if toksy = rparsy then
    parseArgs := nil
  else
    begin
      ex:=parseExpr;
      if toksy = comsy then
```

```
        begin
          match(comsy);
          eL := parseArgs
        end
      else
        eL := nil;
      parseArgs := mkEXPLIST(ex,eL)
    end
end; (* of parseArgs *)

(* parse a function call *)
function parseCall:EXP;
var
  eL:EXPLIST;
  nm:NAME;
begin
  nm:=tokindex;
  match(funidsy);
  match(lparsy);
  eL := parseArgs;
  match(rparsy);
  parseCall := mkAPEXP(nm,eL)
end; (* parseCall *)

(* parse an expression list separated by semicolons *)
function parseEL:EXPLIST;
var
  ex:EXP;
  eL:EXPLIST;
begin
  ex:=parseExpr;
  if toksy = semsy then
    begin
      match(semsy);
      eL := parseEL
    end
  else
    eL := nil;
  parseEL := mkExplist(ex,eL)
end;  (* parseEL *)

(* parse an if expression *)
function parseIf:EXP;
var
  e1,e2,e3:EXP;
  eL:EXPLIST;
  nm:NAME;
begin
```

```
  nm := tokindex;
  match(ifsy);
  e1 := parseExpr;
  match(thensy);
  e2 := parseExpr;
  match(elsesy);
  e3 := parseExpr;
  match(fisy);
  eL := mkExplist(e3,nil);
  eL := mkExplist(e2,eL);
  eL := mkExplist(e1,eL);
  parseIf := mkAPEXP(nm,eL)
end; (* parseIf *)

(* parse a while expression *)
function parseWhile:EXP;
var
  e1,e2:EXP;
  eL:EXPLIST;
  nm:NAME;
begin
  nm := tokindex;
  match(whilesy);
  e1 := parseExpr;
  match(dosy);
  e2 := parseExpr;
  match(odsy);
  eL := mkExplist(e2,nil);
  eL := mkExplist(e1,eL);
  parseWhile := mkAPEXP(nm,eL)
end; (* parseWhile *)

(* parse a sequence expression *)
function parseSeq:EXP;
var
  eL:EXPLIST;
  nm:NAME;
begin
  nm := tokindex;
  match(seqsy);
  eL := parseEL;
  match(qessy);
  parseSeq := mkAPEXP(nm,eL)
end; (* parseSeq *)

(*
   The following functions (parseExp1 through parseExp6) implement the
   following grammar rules.
```

```
  exp1 −→ exp2 [ :=  exp1 ]*
  exp2 −→ [ prtop ] exp3
  exp3 −→ exp4 [ relop exp4 ]*
  exp4 −→ exp5 [ addop exp5 ]*
  exp5 −→ [ addop ] exp6 [ mulop exp6 ]*
  exp6 −→ name | integer | funcall | ( expr )

  The recursive structure of these rules yields the following list from
  lowest to highest precedence:

    :=
    prtop
    relop
    addop
    unary addop, mulop
    variable name, integer, function call, expression in parentheses

  Since the functions call each other recursively, they are implemented in
  reverse order below to avoid forward declarations.

*)

(* parse variable name, integer, function call, parenthesized expression *)
function parseExp6:EXP;
var
     ex:EXP;
  varnm:NAME;
    num:NUMBER;
begin
  case toksy of
    nameidsy:begin
               varnm:=tokindex;
               match(nameidsy);
               ex:=mkVAREXP(varnm)
             end;
     numsy:begin
               num:=numval;
               match(numsy);
               ex:=mkVALEXP(num)
             end;
    lparsy:begin
               match(lparsy);
               ex:=parseExpr;
               match(rparsy)
             end;
    funidsy: ex:=parseCall;
   otherwise;
```

```
      errmsg(3)
  end; (* case *)

  parseExp6 := ex     (* return ptr to an expression *)
end;  (* parseExp6 *)

(* parse unary addop, binary mulop *)
function parseExp5:EXP;
var
  nm:NAME;
  ex,e1,e2:EXP;
  eL:EXPLIST;
  addop_token:token;
  sign:NUMBER;
begin
  addop_token:=dollarsy; (* Initialize so its prior value is not reused. E.g.
for *)
                         (* -10-7$, after negating 10, 7 was incorrectly
negated. *)

  if toksy in addops then (* unary + or - *)
    begin
      addop_token:=toksy;
      match(toksy)
    end;

  e1:=parseExp6;

  if addop_token = subsy then
  (* for unary minus, make an expr to multiply e1 by -1 *)
    begin
      sign:=-1;
      ex:=mkVALEXP(sign);
      eL:=mkExplist(ex,nil);
      eL:=mkExplist(e1,eL);
      nm:=mulsy_index;
      e1:=mkAPEXP(nm,eL)
    end;

  while toksy in mulops do
    begin
      nm:=tokindex;
      match(toktable[nm]);
      e2:=parseExp6;
      eL:=mkExplist(e2,nil);
      eL:=mkExplist(e1,eL);
      e1:=mkAPEXP(nm,eL)
    end;
```

```
  parseExp5:=e1;
end;  (* parseExp5 *)

(* parse binary addop *)
function parseExp4:EXP;
var
  nm:NAME;
  e1,e2:EXP;
  eL:EXPLIST;
begin
  e1:=parseExp5;
  while toksy in addops do
    begin
      nm:=tokindex;
      match(toktable[nm]);
      e2:=parseExp5;
      eL:=mkExplist(e2,nil);
      eL:=mkExplist(e1,eL);
      e1:=mkAPEXP(nm,eL)
    end;
  parseExp4:=e1;
end;  (* parseExp4 *)

(* parse binary relop *)
function parseExp3:EXP;
var
  nm:NAME;
  e1,e2:EXP;
  eL:EXPLIST;
begin
  e1:=parseExp4;
  while toksy in relops do
    begin
      nm:=tokindex;
      match(toktable[nm]);
      e2:=parseExp4;
      eL:=mkExplist(e2,nil);
      eL:=mkExplist(e1,eL);
      e1:=mkAPEXP(nm,eL)
    end;
  parseExp3:=e1;
end;  (* parseExp3 *)

(* parse print op *)
function parseExp2:EXP;
var
  eL:EXPLIST;
  ex:EXP;
```

DD: The original parseExp & parseEL below are omitted from the enhanced interpreter since expression syntax has changed significantly. The original Lisp-style syntax was easy to parse so that parseEXP below could expect a value, a variable, or an application followed by an expression list. Expression parsing is completely redefined to handle the Pascal-style syntax beginning with the new parseExpr on the right on next page. parseEL now parses an expList as defined in the new grammar (expressions separated by semicolons).

```
  nm:NAME;
  printflag:boolean;
begin
  printflag:=false;
  if toksy = printsy then
    begin
      printflag:=true;
      nm:=tokindex;
      match(printsy)
    end;
  ex:=parseExp3;
  if printflag then
    begin
      eL:=mkExplist(ex,nil);
      parseExp2:=mkAPEXP(nm,eL)
    end
  else
    parseExp2:=ex;
end;  (* parseExp2 *)

(* parse assignment *)
function parseExp1:EXP;
var
  eL:EXPLIST;
  ex,e2:EXP;
  nm:NAME;
begin
  ex:=parseExp2;
  while toksy = assignsy do
  (* build an assignment expression *)
    begin
      nm:=tokindex;
      match(assignsy);
      if ex^.etype = VAREXP then (* l.h.s. must be a variable *)
        begin
          e2:=parseExp1;     (* process r.h.s.*)
          eL:=mkExplist(e2,nil);
          eL:=mkExplist(ex,eL);
          ex:=mkAPEXP(nm,eL)
        end
      else (* illegal l.h.s. *)
        begin
          writeln('parseExp1: left hand side of assignment must be a
variable');
          goto 99
        end;
    end;  (* of while *)
    parseExp1:=ex
```

```
function parseEL: EXPLIST; forward;

(* return EXP starting at userinput[pos] *)
function parseExp: EXP;
var
  nm: NAME;
  el: EXPLIST;
begin
  if userinput[pos] = '(' then
    begin   (* APEXP *)
      pos := skipblanks(pos+1); (* skip '( ..' *)
      nm := parseName;
      el := parseEL;
      parseExp := mkAPEXP(nm, el)
    end
  else if isNumber(pos) then
    parseExp := mkVALEXP(parseVal) (* VALEXP *)
  else
    parseExp := mkVAREXP(parseName) (* VAREXP *)
end; (* parseExp *)

(* return EXPLIST starting at userinput[pos] *)
function parseEL;
var
  e: EXP;
  el: EXPLIST;
begin
  if userinput[pos] = ')' then
    begin
      pos := skipblanks(pos+1); (* skip ') ..' *)
      parseEL := nil
    end
  else
    begin
      e := parseExp;
      el := parseEL;
      parseEL := mkExplist(e, el)
    end
end; (* parseEL *)
```

```
end;  (* parseExp1 *)

(* parse if, while, seq, exp1 *)
function parseExpr;
 var
  ex:EXP;
begin
  case toksy of
       ifsy: ex:=parseIf;
    whilesy: ex:=parseWhile;
      seqsy:  ex:=parseSeq;
      nameidsy,numsy,subsy,funidsy,printsy,lparsy: ex:=parseExp1;

    otherwise;
       errmsg(4)
  end; (* case *)
  parseExpr:=ex;
end; (* parseExpr *)
```

```
(*************************************************************
 *                      ENVIRONMENTS                         *
 *************************************************************)

(* emptyEnv - return an environment with no bindings          *)
function emptyEnv: ENV;
begin
   emptyEnv := mkEnv(nil, nil)
```

```
end; (* emptyEnv *)

(* bindVar - bind variable nm to value n in environment rho      *)
procedure bindVar (nm: NAME; n: NUMBER; rho: ENV);
begin
   rho^.vars := mkNamelist(nm, rho^.vars);
   rho^.values := mkValuelist(n, rho^.values)
end; (* bindVar *)

(* findVar - look up nm in rho                                   *)
function findVar (nm: NAME; rho: ENV): VALUELIST;
var
   nl: NAMELIST;
   vl: VALUELIST;
   found: Boolean;
begin
   found := false;
   nl := rho^.vars;
   vl := rho^.values;
   while (nl <> nil) and not found do
      if nl^.head = nm
      then found := true
      else begin
              nl := nl^.tail;
              vl := vl^.tail
           end;
   findVar := vl
end; (* findVar *)

(* assign - assign value n to variable nm in rho                 *)
procedure assign (nm: NAME; n: NUMBER; rho: ENV);
var varloc: VALUELIST;
begin
   varloc := findVar(nm, rho);
   varloc^.head := n
end; (* assign *)

(* fetch - return number bound to nm in rho                      *)
function fetch (nm: NAME; rho: ENV): NUMBER;
var vl: VALUELIST;
begin
   vl := findVar(nm, rho);
   fetch := vl^.head
end; (* fetch *)

(* isBound - check if nm is bound in rho                         *)
function isBound (nm: NAME; rho: ENV): Boolean;
begin
```

```
   isBound := findVar(nm, rho) <> nil
end; (* isBound *)

(*************************************************************
 *                      NUMBERS                              *
 *************************************************************)

(* prValue - print number n                                  *)
procedure prValue (n: NUMBER);
begin
   write(n:1)
end; (* prValue *)

(* isTrueVal - return true if n is a true (non-zero) value   *)
function isTrueVal (n: NUMBER): Boolean;
begin
   isTrueVal := n <> 0
end; (* isTrueVal *)

(* applyValueOp - apply VALUEOP op to arguments in VALUELIST vl  *)
function applyValueOp (op: VALUEOP; vl: VALUELIST): NUMBER;

var n, n1, n2: NUMBER;

(* arity - return number of arguments expected by op         *)
   function arity (op: VALUEOP): integer;
   begin
      if op in [ADDOP .. GTOP] then arity := 2 else arity := 1
   end; (* arity *)

begin (* applyValueOp *)
   if arity(op) <> lengthVL(vl)
   then begin
           write('Wrong number of arguments to ');
           prName(ord(op)+1);
           writeln;
           goto 99
        end;
   n1 := vl^.head; (* 1st actual *)
   if arity(op) = 2 then n2 := vl^.tail^.head; (* 2nd actual *)
   case op of
      ADDOP: n := n1+n2;
      SUBOP: n := n1-n2;
      MULOP: n := n1*n2;
      DIVOP: n := n1 div n2;
      EQOP: if n1 = n2 then n := 1 else n := 0;
      LTOP: if n1 < n2 then n := 1 else n := 0;
      GTOP: if n1 > n2 then n := 1 else n := 0;
```

```
      PRINTOP:
         begin prValue(n1); writeln; n := n1 end
   end; (* case *)
   applyValueOp := n
end; (* applyValueOp *)

(*************************************************************
 *                      EVALUATION                           *
 *************************************************************)

(* eval - return value of expression e in local environment rho  *)
function eval (e: EXP; rho: ENV): NUMBER;

var op: BUILTINOP;

(* evalList - evaluate each expression in el                    *)
   function evalList (el: EXPLIST): VALUELIST;
   var
      h: NUMBER;
      t: VALUELIST;
   begin
      if el = nil then evalList := nil
      else begin
              h := eval(el^.head, rho);
              t := evalList(el^.tail);
              evalList := mkValuelist(h, t)
           end
   end; (* evalList *)

(* applyUserFun - look up definition of nm and apply to actuals  *)
   function applyUserFun (nm: NAME; actuals: VALUELIST): NUMBER;
   var
      f: FUNDEF;
      rho: ENV;
   begin
      f := fetchDef(nm);
      if f = nil
      then begin
              write('Undefined function: ');
              prName(nm);
              writeln;
              goto 99
           end;
      with f^ do begin
         if lengthNL(formals) <> lengthVL(actuals)
         then begin
                 write('Wrong number of arguments to: ');
                 prName(nm);
```

```
                writeln;
                goto 99
             end;
          rho := mkEnv(formals, actuals);
          applyUserFun := eval(body, rho)
          end
   end; (* applyUserFun *)

(* applyCtrlOp - apply CONTROLOP op to args in rho                *)
   function applyCtrlOp (op: CONTROLOP;
                         args: EXPLIST): NUMBER;
   var n: NUMBER;
   begin
      with args^ do
         case op of
           IFOP:
              if isTrueVal(eval(head, rho))
              then applyCtrlOp := eval(tail^.head, rho)
              else applyCtrlOp := eval(tail^.tail^.head, rho);
           WHILEOP:
              begin
                 n := eval(head, rho);
                 while isTrueVal(n)
                 do begin
                       n := eval(tail^.head, rho);
                       n := eval(head, rho)
                    end;
                 applyCtrlOp := n
              end;
           ASSIGNOP:
              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;
           SEQOP:
              begin
                 while args^.tail <> nil do
                    begin
                       n := eval(args^.head, rho);
                       args := args^.tail
                    end;
                 applyCtrlOp := eval(args^.head, rho)
              end
```

DD: The matches function below is omitted from the enhanced interpreter. It checked if keywords "quit" or "define" were entered as shown in the old main program logic below. The new logic on the right below now calls getToken then checks if the "quit" or "fun" symbol was returned.

```
(* check if nm matches userinput[s.. s+leng] *)
function matches (s: integer; leng: NAMESIZE;
                  nm: NAMESTRING): Boolean;
var
   match: Boolean;
   i: integer;
begin
   match := true; i := 1;
   while match and (i <= leng) do begin
      if userinput[s] <> nm[i] then match := false;
      i := i+1;
```

```
         end (* case and with *)
   end; (* applyCtrlOp *)

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 begin
                       write('Undefined variable: ');
                       prName(varble);
                       writeln;
                       goto 99
                     end;
         APEXP:
           if optr > numBuiltins
           then eval := applyUserFun(optr, evalList(args))
           else begin
                  op := primOp(optr);
                  if op in [IFOP .. SEQOP]
                  then eval := applyCtrlOp(op, args)
                  else eval := applyValueOp(op,
                                  evalList(args))
                end
      end; (* case and with *)
end; (* eval *)
```

```
     s := s+1
     end;
   if not isDelim(userinput[s]) then match := false;
   matches := match
end; (* matches *)
```

```
   if matches(pos, 4, 'quit                ')

   else if (userinput[pos] = '(') and
matches(skipblanks(pos+1),6,'define              ')
```

```
             currentExp := parseExp;
```

```
(*************************************************************
 *                     READ-EVAL-PRINT LOOP                  *
 *************************************************************)

begin (* chapter1 main *)
   initParse;
   initNames;
   globalEnv := emptyEnv;
   quittingtime := false;

99:

   while not quittingtime do
     begin
      reader;
      getToken;  (* return the first token from userinput in toksy *)

      if toksy = quitsy then
        quittingtime := true
      else if toksy = funsy then
             begin
               prName(parseDef);
               writeln
             end
           else
             begin
               currentExp := parseExpr;
               prValue(eval(currentExp, emptyEnv));
               writeln
             end
     end (* while *)
end. (* chapter1 *)
```