

FComparison of C vs Pascal Version of Enhanced Chapter 1 Interpreter

This document describes the issues and changes involved in translating the Pascal version (chap1.pas) of the enhanced interpreter into C (chap1.c).

The grammar (section 1 below) and instruction syntax (section 2 below) did not change.

Section 3 below discusses some of the translation issues and presents a side by side comparison of the Pascal and C source code in part 8 of that section.

Section 1. Grammar for Enhanced Chap1 Interpreter

The grammar for chap1.c is still the same as it was for chap1.pas and is as follows.

```
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     → fun name ( arglist ) := expr nuf
arglist    → null | name [ , name ]*
```

```
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 )
```

```
funcall    → name ( actparmlist )
actparmlist → null | expr [ , expr ]*
prtop      → print
relop      → = | < | >
addop      → + | -
mulop      → * | /
```

```
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

[]* indicates the option may occur more than once

The exclamation point begins a comment.

Section 2. Syntax for Enhanced Chap1 Interpreter

The syntax for chap1.c is still the same as it was for chap1.pas and is illustrated in the following examples.

```
~$ ./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
```

```
-> 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.

```
-> ~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 incl0 (x) := x+10 nuf$  
incl0  
-> incl0 := 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.

```
-> )load mod.txt
```

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

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

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

```
-> )sload not.txt
```

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

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

```
Current Directory is : /home/dawson/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

Section 3. Program organization, Translation Considerations and Program Changes

Part 1. Program Organization

The main program is a Read-Eval-Print-Loop (REPL) which reads user input, parses it, evaluates it, prints its value then returns to the beginning of the loop. If an error occurs then the errmsg() function displays error information and calls longjmp() to return to the main program and restart the REPL.

The code is still organized as it was in the Pascal version with the main function at the bottom of the file and the other functions listed in the same order under the following sections: DECLARATIONS, DATA STRUCTURE OPS, NAME MANAGEMENT, INPUT, NEW PARSING ROUTINES, ENVIRONMENTS, NUMBERS, EVALUATION.

Lower-level functions are still, for the most part, defined before higher-level ones that call them which helps reduce the need for forward declarations. For example, consider the grammar rules for expressions and their associated function prototypes in the following table. The rules are presented in a top-down, recursive-descent order but the corresponding functions are in reverse, bottom-up order from the lowest-level function to the highest.

Top-Down Order of Grammar Rules.	Bottom-Up Order of Functions in Source Code
<div>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)</div>	<div>function parseExp6:EXP; function parseExp5:EXP; function parseExp4:EXP; function parseExp3:EXP; function parseExp2:EXP; function parseExp1:EXP; function parseExpr;</div>

So to obtain a top-down view of the program logic, it may be best to read the NEW PARSING ROUTINES and other sections from the bottom up, after reviewing the DECLARATIONS section.

Part 2. Translation of Structure Pointers using Typedef

The *REC names in the Pascal version are only used to declare the REC structure type and to obtain a pointer to it. For example, we have the following.

```
EXPLIST = ^EXPLISTREC;  
EXPLISTREC = record  
    head: EXP;  
    tail: EXPLIST  
END;  
  
FUNCTION mkExplist (e: EXP; el: EXPLIST): EXPLIST;  
    var newel: EXPLIST;  
BEGIN  
    new(newel);  
    newel^.head = e;  
    newel^.tail = el;  
    mkExplist = newel  
END; /* mkExplist */
```

EXPLISTREC is not used anywhere else except in the above red text.

EXPLIST is used throughout the program to declare variables, arguments, function return types.

There is a debate at <http://stackoverflow.com/questions/252780/why-should-we-typedef-a-struct-so-often-in-c> about whether or not typedef should be used in C. I decided to use it and rewrite the above Pascal code as shown below in the middle column. I like the cleaner look and that it made it easier to visually confirm that my C code is equivalent to the Pascal code. If I did not use typedef then the C code would be written as shown in the last column below.

Pascal Code	C Code with typedef	C code without typedef
<pre>EXPLIST = ^EXPLISTREC; EXPLISTREC = record head: EXP; tail: EXPLIST END; FUNCTION mkExplist (e: EXP; el: EXPLIST): EXPLIST; var newel: EXPLIST; BEGIN new(newel); newel^.head = e; newel^.tail = el; mkExplist = newel END; /* mkExplist */</pre>	<pre>struct EXPLISTREC { EXP head; struct EXPLISTREC *tail; }; typedef struct EXPLISTREC* EXPLIST; EXPLIST mkExplist (EXP e, EXPLIST el) { EXPLIST newel; newel = (EXPLIST) malloc(sizeof(struct EXPLISTREC)); newel->head = e; newel->tail = el; return newel; }; /* mkExplist */</pre>	<pre>struct EXPLISTREC *mkExplist (EXP e, struct EXPLISTREC *el) { struct EXPLISTREC *newel; newel = (struct EXPLISTREC *) malloc(sizeof(struct EXPLISTREC)); newel->head = e; newel->tail = el; return newel; }; /* mkExplist */</pre>

Part 3. Translation of Pascal Variant Record using C Union

Expressions may be of type number (integer constant), variable or application (of a function or operator). The Pascal variant record structure that was used to store expressions will now be represented with a C union construct as shown below.

Pascal Variant Record	C Union
<pre>EXPTYPE = (VALEXP,VAREXP,APEXP); EXPREC = record case etype: EXPTYPE of VALEXP: (num: NUMBER); VAREXP: (varble: NAME); APEXP: (optr: NAME; args: EXPLIST) end;</pre>	<pre>enum EXPTYPE {VALEXP,VAREXP,APEXP}; struct EXPREC { enum EXPTYPE etype; union { NUMBER num; NAME varble; struct { NAME optr; EXPLIST args; } ap; } u; };</pre>

Part 4. Translation of Pascal Sets

The following Pascal sets were defined.

```
addop, mulop, relop:set of token; // sets of operators
```

And they were initialized in `initParse` as follows.

```
addop = [addsy,subsy];  
mulop = [mulsy,divsy];  
relop = [lssy,eqsy,gtsy];
```

For the C conversion, I replaced tests for set membership like the following

```
if toksy in mulop then
```

with tests of the specific values as follows.

```
if (toksy == mulsy || toksy == divsy)
```

Part 5. Translation of Pascal `eoln()` and `eof()`

Functions `eoln()` and `eof()` were replaced with booleans named `eoln` and `eof` respectively. These booleans are set to true in the `nexchar()` function as `eoln` and `eof` conditions are encountered.

Part 6. Scattered `gotos` replaced single `longjmp()`

In the Pacal version, the Read-Eval-Print-Loop (REPL) was preceded by label 99. Each error message was followed by a “goto 99” statement. Then the REPL would restart.

In the C version, a call to `setjmp()` replaces label 99 prior to the REPL. All error messages are now inside an `errmsg()` function which calls `longjmp()` at the end to return to the `setjmp` location and restart the REPL.

Part 7. Space-padded strings replaced with null-terminated strings

The `printNames` array is used to store the names of all operators, variables and functions.

In the Pascal version, all such names are padded with trailing spaces to a length of 20 characters. In the C version, the trailing spaces are omitted and the names are null-terminated. So all the loops that compared and inserted trailing spaces in the Pascal version have been removed in C version.

Part 8. Side by Side Comparison of Pascal vs C source code

Pascal code is in column 1 and corresponding C code is in column 2 of the table below. The formatting differs from the actual source code in places due to adjusting spacing or moving lines so that corresponding source lines are side by side.

I added some clarifying comments in bold blue text below which are not contained in the actual source code.
They are prefixed by my initials and a colon. Search for “DD:” to read them.

<pre> (***** * * 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 *) TABCODE = 9; LINEFEED = 10; CR = 13; COMMENTCHAR = '!'; DOLLAR = '\$'; (*marks end of expr or fundef input by the user*) TYPE NAME_SIZE = 0..NAMELENG; NAMESTRING = packed array [1..NAMELENG] of char; NAME = 1 .. MAXNAMES; (* a NAME is an index in printNames *) CMD_SIZE = 0..CMDLENG; </pre>	<pre> /***** * * DECLARATIONS * *****/ #define _POSIX_C_SOURCE 200809L #include <stdio.h> #include <stddef.h> #include <string.h> #include <setjmp.h> #include <stdlib.h> #include <ctype.h> #include <errno.h> #include <limits.h> #include <unistd.h> DD: setjmp(JL99) replaces label 99: in main. errmsg() does a longjmp back to that point after an error message is displayed. jmp_buf JL99; #define NAMELENG 20 // Max length of a name #ifdef TESTSIZE #define MAXNAMES 29 // Max no. of names to test err_max_names #define MAXINPUT 50 // Max input length to test err_input_len #else #define MAXNAMES 100 // Max number of different names #define MAXINPUT 500 // Max length of an input #endif #define MAXCMDLENG 8 // Max length of a command name #define MAXCMDS 3 // Max number of different commands #define ARGLENG 40 // Maximum length of a command argument #define MAXDIGITS 19 // Max digits in LONG_MAX in limits.h // for __WORDSIZE == 64 #define PROMPT "-> " // Initial prompt #define PROMPT2 "> " // continuation prompt #define TAB 9 #define LF 10 #define CR 13 #define COMMENTCHAR '!' #define DOLLAR '\$' // marks END of expr or fundef input by user #define SPACE ' ' #define RPAREN ')' // marks beginning of a Cmd typedef short int NAME_SIZE; typedef char *NAMESTRING; typedef short int NAME; typedef short int CMD_SIZE; </pre>
---	--

```
CMDSTRING=packed array [1..CMDLENG] of char;  
CMD = 1..NUMCMDS;      (* a CMD is an index in printCmds *)
```

```
NUMBER = integer;  
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;
```

```
typedef char *CMDSTRING;  
typedef short int CMD;
```

```
typedef long NUMBER;  
typedef short int ARGSize;
```

```
typedef enum {false, true} BOOLEAN;
```

```
typedef enum {IFOP,WHILEOP,ASSIGNOP,SEQOP,ADDOP,SUBOP,MULOP,DIVOP,  
EQOP,LTOP,GTOP,PRINTOP} BUILTINOP;  
DD: VALUEOP range replaced with firstValueOp, lastValueOp indexes.  
CONTROLOP range replaced with firstControlOp, lastControlOp indexes.
```

```
typedef enum {nameidsy=1,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} TOKEN;
```

DD: Error messages in the Pacal version were displayed at the point of occurrence and followed by a goto label 99 in the main function. Those are now replaced with a call to errmsg() which is passed one of the error codes below. errmsg() prints the associated message then calls longjmp(JL99, errnum) to return to main to read the next input.

```
typedef enum {err_arglist=1, err_function, err_exp6, err_expr, err_cwd,  
err_open, err_max_names, err_input_len, err_cmd_len, err_no_cmd,  
err_bad_cmd, err_arg_len, err_no_name, err_name_len,  
err_no_name2, err_digits, err_no_name3, err_mismatch,  
err_not_var, err_num_args, err_undef_func, err_num_args2,  
err_undef_var, err_undef_op, err_nested_load  
} ERROR_NUM; // error codes passed to errmsg()
```

```
typedef struct EXPREC* EXP;  
typedef struct EXPLISTREC* EXPLIST;  
typedef struct ENVREC* ENV;  
typedef struct VALUELISTREC* VALUELIST;  
typedef struct NAMELISTREC* NAMELIST;  
typedef struct FUNDEFREC* FUNDEF;
```

```
enum EXPTYPE {VALEXP,VAREXP,APEXP};  
struct EXPREC {  
    enum EXPTYPE etype;  
    union {  
        NUMBER num;  
        NAME varble;  
        struct {  
            NAME optr;  
            EXPLIST args;  
        } ap;  
    } u;
```

```

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;
    printCmds: array [CMD] of CMDSTRING;

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

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

    load, sload: CMD;

    numNames, numBuiltin, tokindex, mulsy_index: NAME;

```

```

};

struct EXPLISTREC {
    EXP head;
    EXPLIST tail;
};

struct VALUelistREC {
    NUMBER head;
    VALUelist tail;
};

struct NAMElistREC {
    NAME head;
    NAMElist tail;
};

struct ENVREC {
    NAMElist vars;
    VALUelist values;
};

struct FUNDEFREC {
    NAME    funname;
    NAMElist formals;
    EXP    body;
    FUNDEF    nextfundef;
};

FUNDEF fundefs;
NUMBER numval;
ENV globalEnv;
EXP currentExp;

char punctop[] = "()+-*/:=<>,$!"; // punctuation and operator chars

char userInput[MAXINPUT];
short int j, inputleng, pos;

NAMESTRING printNames[MAXNAMES]; //built-in & user-defined names
CMDSTRING printCmds[MAXCMDS];    //built-in command names

char tokstring[NAMLENG+1]; // token string for display in error messages
NAMESIZE tokleng;

char infilename[ARGLENG];
FILE *infp;    // input source file pointer

CMD load,        // load file, echo characters
    sload,       // silently load file, no echo
    user;        // print user defined names

NAME numNames, numCmds, numBuiltin, tokindex, mulsy_index,
    // initNames() saves index of first/last ControlOp & ValueOp
    // in following variables for checking which range Op is in.

```

<pre> toksy: TOKEN; (* current token returned from getToken or install *) toktable: array [NAME] of TOKEN; (*holds symbolic name of each token in printNames array. Corresponding toktable & printNames entries have same index. *) 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; </pre>	<pre> firstValueOp, lastValueOp, firstControlOp, lastControlOp; TOKEN toksy; // symbolic name of token TOKEN toktable[MAXNAMES]; // symbolic name of each token in printNames. // Corresponding toktable & printNames elements // have same index DD: The Pascal sets on the left were replaced with direct comparison to the set values as indicated in Section 3 part 4 above. BOOLEAN quittingtime, // true = exit program dollarflag, // true = \$ was entered which marks end of input echo, // true = echo chars during a load command readfile, // true = an input file is being loaded eof, // eof and eoln are used to mimic the boolean eoln; // Pascal functions eof() and eoln() void errmsg(ERROR_NUM, char[], int); // forward declaration //----- // DATA STRUCTURE OPERATIONS //----- // mkVALEXP - return an EXP of type VALEXP with num n EXP mkVALEXP (NUMBER n) { EXP e; e = malloc(sizeof(struct EXPREC)); e->etype = VALEXP; e->u.num = n; return e; } // mkVALEXP // mkVAREXP - return an EXP of type VAREXP with varble nm EXP mkVAREXP (NAME nm) { EXP e; e = malloc(sizeof(struct EXPREC)); e->etype = VAREXP; e->u.varble = nm; return e; } // mkVAREXP // mkAPEXP - return EXP of type APEXP with optr op and args eL EXP mkAPEXP (NAME op, EXPLIST eL) { EXP e; e = malloc(sizeof(struct EXPREC)); e->etype = APEXP; e->u.ap.optr = op; </pre>
--	---

```

    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 *)

```

```

    e->u.ap.args = eL;
    return e;
} // mkAPEXP

// mkExplist - return an EXPLIST with head e and tail eL
EXPLIST mkExplist (EXP e, EXPLIST eL)
{
    EXPLIST newel;

    newel = malloc(sizeof(struct EXPLISTREC));
    newel->head = e;
    newel->tail = eL;
    return newel;
} // mkExplist

// mkNamelist - return a NAMELIST with head n and tail nl
NAMELIST mkNamelist (NAME nm, NAMELIST nl)
{
    NAMELIST newnl;

    newnl = malloc(sizeof(struct NAMELISTREC));
    newnl->head = nm;
    newnl->tail = nl;
    return newnl;
} // mkNamelist

// mkValuelist - return an VALUelist with head n and tail vl
VALUelist mkValuelist (NUMBER n, VALUelist vl)
{
    VALUelist newvl;

    newvl = malloc(sizeof(struct VALUelistREC));
    newvl->head = n;
    newvl->tail = vl;
    return newvl;
}

// mkEnv - return an ENV with vars nl and values vl
ENV mkEnv (NAMELIST nl, VALUelist vl)
{
    ENV rho;

    rho = malloc(sizeof(struct ENVREC));
    rho->vars = nl;
    rho->values = vl;
    return rho;
} // mkEnv

// prEnv - print vars & values in an ENV - for debugging
void prEnv(ENV env)
{
    NAMELIST nl;
    VALUelist vl;

    int i;

```

```
(* 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 *)
```

*)

*)

```
i=0;
nl = env->vars;
vl = env->values;

while (nl != 0)
{
  i = i + 1;
  printf("%d. %s = %ld", i, printNames[nl->head], vl->head);
  nl = nl->tail;
  vl = vl->tail;
}
} // prEnv
```

```
// lengthVL - return length of VALUelist vl
int lengthVL (VALUelist vl)
{
  int i;

  i = 0;
  while (vl != 0)
  {
    i = i+1;
    vl = vl->tail;
  }
  return i;
} // lengthVL
```

```
// lengthNL - return length of NAMElist nl
int lengthNL (NAMElist nl)
{
  int i;

  i = 0;
  while (nl != 0)
  {
    i = i+1;
    nl = nl->tail;
  }
  return i;
} // lengthNL
```

```
void prExplist(EXPLIST); //forward declaration
```

DD: Below are more print functions for debugging

```
// print an EXP
void prExp(EXP e)
{
  switch (e->etype)
  {
    case VALEXP:
      printf("etype = VALEXP\n");
      printf("  num = %ld\n", e->u.num);
      break;
```



```

(*****
*                               *
*               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);

```

```

case VAREXP:

```

```

    printf(" etype = VAREXP\n");

```

```

    printf("varble = %s\n", printNames[e->u.varble]);

```

```

    break;

```

```

case APEXP:

```

```

    printf(" etype = APEXP\n");

```

```

    printf(" optr = %s\n", printNames[e->u.ap.optr]);

```

```

    prExplist(e->u.ap.args);

```

```

    break;

```

```

default:

```

```

    printf("Invalid etype = %d", e->etype);

```

```

    break;

```

```

}

```

```

}

```

```

// prExplist - print an Explist

```

```

void prExplist(EXPLIST eL)

```

```

{

```

```

    prExp(eL->head);

```

```

    if (eL->tail != 0)

```

```

        prExplist(eL->tail);

```

```

    return;

```

```

} // prExplist

```

```

//-----

```

```

// NAME MANAGEMENT

```

```

//-----

```

```

// fetchDef - get FUNCTION definition of fname from fundefs

```

```

FUNDEF fetchDef (NAME fname)

```

```

{

```

```

    FUNDEF f;

```

```

    BOOLEAN found;

```

```

    found = false;

```

```

    f = fundefs;

```

```

    while (f != 0 && !found)

```

```

        if (f->funname == fname)

```

```

            found = true;

```

```

        else

```

```

            f = f->nextfundef;

```

```

    return f;

```

```

} // fetchDef

```

```

// newDef - add new FUNCTION fname with parameters nl, body e

```

```

void newDef (NAME fname, NAMELIST nl, EXP e)

```

```

{

```

```

    FUNDEF f;

```

```

    f = fetchDef(fname);

```

```

    if (f == 0) // fname not yet defined as a FUNCTION

```

```

    {

```

```

        f = malloc(sizeof(struct FUNDEFREC));

```

```

        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 a lookup later 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] := fisys;    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;

```

```

        f->nextfundef = fundefs; // place new FUNDEFREC
        fundefs = f;             // at front of fundefs list
    }
    f->funname = fname;
    f->formals = nl;
    f->body = e;
} // newDef

```

DD: Null-terminated strings are used in initNames below instead of the space-padded strings used in the Pascal version. So all the loops that deal with space-padding are removed in the C version.

```

// initNames - place all pre-defined (built in) names into printNames
// and corresponding token symbols in toktable. user-defined names
// (functions, variables) are also kept here
void initNames()
{
    int i;

    fundefs = 0; //empty list of fundefs
    firstControlOp = i = 0;
    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;

    lastControlOp = i;
    printNames[i] = "seq";   toktable[i] = seqsy;   i = i+1;

    firstValueOp = i;
    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 APEXP with the
    // multiply operator and operand -1. Below we save the multiply symbol
    // index so we can insert it into the APEXP later.

    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;

    lastValueOp = i;
    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] = fisys;   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 *)

(* 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 *)

(* 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 *)
end; (* prName *)

(* install - insert new name into printNames *)
function install (nm: NAMESTRING): NAME;
var
i: integer;

found: Boolean;
begin
i := 1;
found := false;

```

```

printNames[i] = ";"; toktable[i] = semsy; i = i+1;
printNames[i] = ","; toktable[i] = comsy; i = i+1;
printNames[i] = "$"; toktable[i] = dollarsy; i = i+1;

numNames = numBuiltins = i; // no. of entries in 0 to numBuiltins-1
} // initNames

// initCmds - place all pre-defined commands into printCmds
void initCmds()
{
short int i;

i = 0;
printCmds[i] = "sload"; sload = i; i = i+1;
printCmds[i] = "load"; load = i; i = i+1;
printCmds[i] = "user"; user = i; i = i+1;
numCmds = i;
} //initCmds

// prName - print name nm
void prName(NAME nm)
{
printf("%s",printNames[nm]);
} // prName

DD: Following function implements command "user" which is for debugging
// prUserList - List user-defined names and token type id
void prUserList()
{
int i;

for(i = numBuiltins; i < numNames; i++)
{
printf("printNames[%d]= ", i);
prName(i);
printf(" toktable[%d]= %d\n", i, toktable[i]);
}
} // prUserList

// install - insert new name into printNames, set its type in toktable
// - save its type in toksy, return its index in printNames array
NAME install(char *nm)
{
NAME i;
int result;
BOOLEAN found;

i = 0;
found = false;

```


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 for the number to be valid. Digits followed by non-delimiter chars are treated as names of variables or functions.

```
(* 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 tabs and comments. 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.
This causes the main prompt to be displayed. Displaying the continuation
```

```
// isDelim - check if c is a delimiter
BOOLEAN isDelim (char c)
{
  if (c == SPACE || strchr(punctop, c) != NULL)
    return true;
  else
    return false;
} // isDelim

// skipblanks - return next non-blank pos in userInput
int skipblanks (int p)
{
  while (userinput[p] == SPACE)
    p = p + 1;
  return p;
} // skipblanks
```

DD: Global boolean variables eoln and eof are added to the C version to mimic the Pascal eoln and eof functions. The nextchar function below sets their values appropriately.

```
// nextchar - read next char - filter tabs,LF, comments
// Also filter CR/LF which were returned in input stream under WSL/Cygwin
char nextchar()
{
  int c;

  eoln = false;

  if (readfile)
  {
    if ((c = getc(infp)) == EOF) // read from file
    {
      readfile = false;
      echo = false;
      eoln = true;
      eof = true;
```

prompt at EOF is undesirable because it would prohibit the user from entering another load command since such commands are only checked for in response to 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);

```

```

        c = DOLLAR; // exit read loop in readDollar and readInput
        fclose(infp);

```

```

    }
    else if (echo)
        printf("%c", c); // echo char
    }
else
    c = getc(stdin); // read from stdin

```

```

    if (c == LF)
        eoln = true;

```

// replace tab and eoln chars with space, skip comments

```

    if (c == TAB || c == LF || c == CR)
        c = SPACE;
    else if (c == COMMENTCHAR) // skip comment chars
    {
        if (readfile)
        {
            while ((c = getc(infp)) != LF)
                if (echo) printf("%c", c); // echo comment chars
            if (echo) printf("%c", c); // echo LF
        }
        else
            while ((c = getc(stdin)) != LF)
                ;
        eoln = true;
        c = SPACE; // replace LF with space
    }
    return c;
} // nextchar

```

```

// readDollar - read char's, ignoring newlines, till '$' is read
// '$' marks END of the fundef or expr that is being input
void readDollar()
{
    char c = SPACE;
    char str[2]="";

    do
    {
        if (!readfile && eoln)
        {

```

```

    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 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 *)

```

```

        printf(PROMPT2); //continuation prompt
        fflush(stdout);
    }
    c = nextchar();
    pos = pos+1;
    if (pos == MAXINPUT)
    {
        str[0] = c;      //last char read
        errmsg(err_input_len, str, MAXINPUT);
    }
    userinput[pos] = c;
} while (c != DOLLAR);
dollarflag = true;
} // readDollar

```

/* The next four functions, readCmdLine, parseCmdName, parseCmdNameArg 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. */

```

// readCmdLine - read command line into userinput buffer.
//          When readInput detects RPAREN as first char of input,
//          it calls processCmd which calls readCmdLine to read
//          rest of the cmd line. On entry to this function,
//          pos==0 and userinput[0]==RPAREN.
void readCmdLine()
{
    char c = SPACE, *dollarPtr;

    while (!eoln)
    {
        c = nextchar(); // sets eoln true on LF & returns SPACE
        pos = pos + 1;
        userinput[pos] = c;
    }

    // use dollar position (if any) or LF position to determine input length

    if ((dollarPtr = strchr(userinput, DOLLAR)) != NULL)
        inputleng = dollarPtr - userinput;
    else
        inputleng = pos;
} // readCmdLine

```

```

// parseCmdName - return Cmd starting at userinput[pos]
CMD parseCmdName()
{
    char nm[MAXCMDLENG+1]; // for accumulating characters in Cmd

```

```

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;

```

```

CMDSIZE leng;
CMD i;
BOOLEAN found;
int result;

nm[0] = 0;
leng = 0;
result = 0;
DD: Command names contain only alpha chars
while (pos < inputleng && isalpha(userinput[pos]))
{
  if (leng == MAXCMDLENG)
  {
    nm[leng] = 0;
    errmsg(err_cmd_len, nm, MAXCMDLENG);
  }
  nm[leng] = userinput[pos];
  leng = leng+1;
  pos = pos+1;
}

if (leng == 0)
  errmsg(err_no_cmd, null_str, null_int);

nm[leng] = 0; //null terminate Cmd name

// Determine Cmd index in printCmds

i = 0;
found = false;
while (i < MAXCMDS && !found)
{
  result = strcmp(nm, printCmds[i]);
  if (!result)
    found = true;
  else
    i = i+1;
}

if (!found)
  errmsg(err_bad_cmd, nm, null_int);

pos = skipblanks(pos); /* skip blanks after command name */
return i;
} // parseCmdName

/* 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 */
void parseCmdArg(char nm[])

```



```

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 *)
  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*)

    Assign(infile,infilename);
    RESET(infile);

    writeln;
    writeln('Current Directory is : ',GetCurrentDir);

```

```

{
  ARGSize leng; // length of argumnet name

  nm[0] = 0;
  leng = 0;
  while (pos < inputleng && userinput[pos] > SPACE)
  {
    if (leng == ARGLENG)
    {
      nm[leng] = 0;
      errmsg(err_arg_len, nm, ARGLENG);
    }
    nm[leng] = userinput[pos];
    leng = leng+1;
    pos = pos+1;
  }

  if (leng == 0)
    errmsg(err_no_name, null_str, null_int);

  nm[leng] = 0; //null terminate string
} // parseCmdArg

// processCmd - input, parse, and execute the command
void processCmd()
{
  char cwd[PATH_MAX];
  CMD cmdnm; // cmdnm is an index to printCmds

  readCmdLine();
  pos = 1; // pos of 1st letter of command name

  cmdnm = parseCmdName(); // parse command name
  if (cmdnm == sload || cmdnm == load)
  {
    parseCmdArg(infilename); // parse filename argument

    // Load commands cannot be issued inside another file being loaded.
    if (!readfile) //if not already loading a file
    {
      if ((infp = fopen(infilename, "r")) == NULL)
        errmsg(err_open, null_str, null_int);

      if (getcwd(cwd, sizeof(cwd)) != NULL)
        printf("\nCurrent Directory is: %s\n", cwd);
      else
        errmsg(err_cwd, null_str, null_int);
    }
  }
}

```

```

writeln(' Loading file : ',infilename);
writeln;
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 command, execute it*)
        begin
            processCmd;
            if not readfile then write(PROMPT);

            pos:=0
        end
        else (* otherwise read expr or fundef terminated by dollar *)

            if userinput[pos] = dollar then
                dollarflag:=true
            else
                if readfile then
                    begin

```

```

printf(" Loading file : %s\n\n",infilename);

readfile = true;          // tell nextchar to read from file
eof = false;
if (cmdnm == load)
    echo = true;          // remains false for sload (silent load)
}
else //error on nested load cmds
    errmsg(err_nested_load, infilename, null_int);
}
if (cmdnm == user)
    prUserList(); // display user-defined entries
} // processCmd

// readInput - read char's into userinput array until dollar sign is input
void readInput()
{
    char c = SPACE;
    char str[2]="";

    dollarflag = false;
    if (!readfile)
    {
        printf(PROMPT); //display main prompt for new input
        fflush(stdout);
    }
    pos = -1;
    do
    {
        c = nextchar();
        pos = pos+1;
        if (pos == MAXINPUT)
        {
            str[0] = c; //last char read
            errmsg(err_input_len, str, MAXINPUT);
        }

        userinput[pos] = c;
    } // parse a command or fundef or expression
    if (pos == 0 && c == RPAREN) // If it's a cmd (e.g. load or sload)
    {
        processCmd(); //opens file, sets echo
        if (!readfile)
        {
            printf(PROMPT); // redisplay main prompt after a cmd
            fflush(stdout);
        }
        pos = -1; //restart userinput index
    }
    else //parse fundef or expression
    {
        if (userinput[pos] == DOLLAR)
            dollarflag = true;
        else

```

```

        if eoln(infile) then readDollar
        end
        else (* reading stdin *)
            if eoln then readDollar
        until dollarflag;

        pos:=pos-1; (* exclude $ from user input *)
        inputleng := pos;
        if readfile and echo then writeln (* echo LF 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;
end;

```

```

        if (eoln) readDollar();
    }

    } while (!dollarflag);

    inputleng = pos; // pos of '$' is length of input in 0 to pos-1

    //remove LF and any other chars that follow dollar sign from input buffer
    while (!eoln)
        c = nextchar(); // sets eoln true on LF & returns SPACE
} // readInput

// reader - read char's into userinput; be sure input is not blank
void reader()
{
    do
    {
        readInput(); // read input into userinput array
        pos = skipblanks(0); // advance to first non-blank
        if (userinput[pos] == DOLLAR) // if it is '$' then
            inputleng = 0; // it is a blank line
    } while (inputleng == 0); // ignore blank lines
} // reader

NAME install(char *); // forward declaration

// parseName - return (installed) NAME starting at userinput[pos]
NAME parseName()
{
    char nm[NAMELENG+1]; // array to accumulate characters
    NAMESIZE leng; // length of name

    nm[0] = tokstring[0] = 0;
    leng = 0;
    while (pos < inputleng && !isDelim(userinput[pos]))
    {
        if (leng == NAMELENG)
        {
            nm[leng] = 0;
            errmsg(err_name_len, nm, NAMELENG);
        }
        nm[leng] = tokstring[leng] = userinput[pos];
        leng = leng+1;
        pos = pos+1;
    }

    if (leng == 0)
        errmsg(err_no_name2, null_str, null_int);
}

```

```

for leng := leng+1 to NAMELENG do nm[leng] := ' ';

pos := skipblanks(pos); (* skip blanks after name *)
parseName := install(nm)
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
    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

```

```

nm[leng] = tokstring[leng] = 0; //null terminate
tokleng = leng;
pos = skipblanks(pos); /* skip blanks after name */
return install(nm);
} // parseName

// isNumber - check if a number begins at userinput[i]
// It must be a sequence of digits followed by a delimiter
// otherwise it will be treated as a name.
// E.g. 232+ is parsed as the number 232 followed by a plus sign.
// but 232abc is parsed as a name since the digits are not
// followed by a delimiter.
BOOLEAN isNumber(short int i)
{

    if (!isdigit(userinput[i]))
        return false;
    else
    {

        while (isdigit(userinput[i])) i = i + 1;
        if (isDelim(userinput[i]))
            return true;
        else
            return false;
    }

} // isNumber

// parseVal - return number starting at userinput[pos]
NUMBER parseVal()
{
    NUMBER n;
    char numString[MAXDIGITS+1]; //digits in value being parsed

    numString[0] = tokstring[0] = 0;
    n = 0;
    tokleng = 0;
    while (isdigit(userinput[pos]) && tokleng < MAXDIGITS)
    {
        n = 10*n + (userinput[pos] - '0');
        numString[tokleng] = tokstring[tokleng] = userinput[pos];
        tokleng = tokleng+1;
        pos = pos+1;
    }
    numString[tokleng] = tokstring[tokleng] = 0; // null terminate

    if (isdigit(userinput[pos]))
        errmsg(err_digits, null_str, MAXDIGITS); // Too many digits

pos = skipblanks(pos); //skip any blanks after number
return n;

```

<pre> end; (* parseVal *) (***** NEW PARSING ROUTINES *****) (* writeTokenName - 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 *) procedure writeTokenName(t:token); 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 end; (* of writeTokenName *) (* writeTokenString - Write out chars of token string. During errors, this function is used to display invalid string found in the userinput *) procedure writeTokenString; </pre>	<pre> } //parseVal //----- // NEW PARSING ROUTINES //----- // writeTokenName - Display the name corresponding to token symbol t. void writeTokenName(TOKEN t) { NAME i; if (t == nameidsy t == numsy t == funidsy) // display generic name for user-defined symbols switch (t) { case nameidsy: printf("nameid"); break; case numsy: printf("number"); break; case funidsy: printf("funid"); break; default: break; } else { // display builtin name i = 0; while (toktable[i] != t && i < numBuiltins) i = i+1; if (i < numBuiltins) // write the name of the token printf("%s", printNames[i]); else // name not found, display token id printf("name not found for token %d", t); } } // of writeTokenName // writeTokenStr - Display token string. During errors, this function // is used to display the invalid token found in userinput. void writeTokenStr() </pre>
--	--

```

var
  i:integer;
begin
  for i:= 1 to tokleng do
    write(tokstring[i]);
  write(' ');
end;

(* displays error messages based on the given error number *)
procedure errmsg(errnum:integer);
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 *)

```

```

{
  printf("%s ", tokstring);
} // writeTokenStr

```

DD: All error messages in the Pascal version were given an error code and are now displayed by calling the errmsg() function below.

```

// errmsg - display error message for given error number and jump to JL99
// in main function
void errmsg(ERROR_NUM errnum, char err_str[], int err_int)
{
  printf("*****");
  switch (errnum)
  {
    case err_arglist:
      printf("Error parsing arglist. Found ");
      writeTokenStr();
      printf("where \"")\" or nameid is expected.");
      break;
    case err_function:
      printf("Error parsing function name. Found ");
      writeTokenStr();
      printf("where funid or nameid is expected.");
      break;
    case err_exp6:
      printf("Error parsing exp6. Found ");
      writeTokenStr();
      printf("where nameid, funid, \"(\", or a number is expected.");
      break;
    case err_expr:
      printf("Error parsing expr. Found ");
      writeTokenStr();
      printf("where one of the following is expected:\n");
      printf("\"if\", \"while\", \"seq\", \"print\", nameid, funid,
number, or \"(\"");
      break;
    case err_cwd:
      perror("getcwd() error");
      break;
    case err_open:
      perror("fopen() error");
      break;
    case err_max_names:
      printf("No more room for names");
      break;
    case err_input_len:
      printf("Input exceeds %d chars. Last char read = %s", err_int,
err_str);
      printf("\nSkipping rest of input and quitting.");
      quittingtime = true;
      if (readfile) fclose(infp);
      break;
    case err_cmd_len:
      printf("Command Name exceeds %d chars, begins: %s", err_int,
err_str);

```

```

        break;
    case err_no_cmd:
        printf("Error: expected Command name, instead read: %c",
userinput[pos]);
        break;
    case err_bad_cmd:
        printf("Unrecognized Command Name, begins: %s", err_str);
        break;
    case err_arg_len:
        printf("Argument name exceeds %d chars, begins: %s", err_int,
err_str);
        break;
    case err_no_name:
        printf("Error: expected name, instead read: %c",
userinput[pos]);
        break;
    case err_name_len:
        printf("Name exceeds %d chars, begins: %s", err_int, err_str);
        break;
    case err_no_name2:
        printf("Error: expected name, instead read: %c",
userinput[pos]);
        break;
    case err_digits:
        printf("parseVal: Max digits allowed in 64 bit signed long
is %d", err_int);
        break;
    case err_no_name3:
        printf("mutate: found ");
        writeTokenStr();
        printf(" where nameid or funid is expected.");
        break;
    case err_mismatch:
        printf("Error in match. Found ");
        writeTokenStr();
        printf(" where ");
        writeTokenName(err_int);
        printf(" is expected.");
        break;
    case err_not_var:
        printf("parseExp1: left hand side of assignment must be a
variable");
        break;
    case err_num_args:
        printf("Wrong number of arguments to ");
        prName(err_int);
        break;
    case err_undef_func:
        printf("Undefined function: ");
        prName(err_int);
        break;
    case err_num_args2:
        printf("Wrong number of arguments to: ");
        prName(err_int);
        break;
    case err_undef_var:

```

```
(* Identify token that begins at userInput[pos], return its symbol in
global variable toksy and leave pos pointing to first nonblank that
follows. *)
```

```
procedure getToken;
```

```
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
```

```
            printf("Undefined variable: ");
```

```
            prName(err_int);
```

```
            break;
```

```
        case err_undef_op:
```

```
            printf("eval: invalid value for op = %d", err_int);
```

```
            break;
```

```
        case err_nested_load:
```

```
            printf("Load commands cannot occur inside a file being
loaded.\n");
```

```
            printf("*****");
```

```
            printf("Remove the load command for file %s", err_str);
```

```
            quittingtime = true;
```

```
            fclose(infp);
```

```
        default:
```

```
            break;
```

```
    }
```

```
    printf("\n\n");
```

```
    longjmp(JL99, errnum);
```

```
} // errmsg
```

```
// getToken - get next token in userInput, set toksy, tokstring, tokleng.
```

```
// For names and operators, also set tokindex. For numbers, tokindex does
```

```
// not apply since they are not stored in printNames.
```

```
void getToken()
```

```
{
```

```
    char nm[2]; // array to accumulate characters
```

```
    nm[0] = tokstring[0] = 0;
```

```
    if (isNumber(pos)) // parse a number
```

```
    {
```

```
        numval = parseVal(); // set tokstring, tokleng
```

```
        toksy = numsy;
```

```
    }
```

```
    else if (userinput[pos] == ':' && userinput[pos+1] == '=')
```

```
        // parse an assignment
```

```
    {
```

```
        tokleng = 2;
```

```
        nm[0] = tokstring[0] = ':';
```

```
        nm[1] = tokstring[1] = '=';
```

```
        nm[2] = tokstring[2] = 0;
```

```
        pos = pos + 2;
```

```
        pos = skipblanks(pos);
```

```
        tokindex = install(nm); // set toksy
```

```
    }
```

```
    else if (strchr(punctop, userInput[pos]) != NULL)
```

```
        // parse single char punctuation or operator
```

```
    {
```



```

    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 *)

(* change nameidsy to funidsy or vice versa *)

procedure mutate(newtype:token);
begin
    if (toksy <> nameidsy) and (toksy <> funidsy) then
        begin
            write('mutate: found ');
            writeTokenString;
            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;
    end; (* of match *)

    (* parse parameters of a fundef *)
    function parseParams:NAMELIST;
    var
        nm:NAME;
        nl:NAMELIST;

```

```

    tokleng = 1;
    nm[0] = tokstring[0] = userInput[pos];
    nm[1] = tokstring[1] = 0; // null terminate 1 char string

    pos = pos + 1;

    pos = skipblanks(pos);
    tokindex = install(nm); // set toksy
}

else // else assume it is a name
    tokindex = parseName(); // set toksy, tokstring, tokleng
} // getToken

// mutate - Change type of toksy in toktable to newtype.
// This function is currently used to redefine a variable name (nameidsy)
// as a function name (funidsy)
void mutate(TOKEN newtype)
{
    if (toksy != nameidsy && toksy != funidsy)
        errmsg(err_no_name3, null_str, null_int);

    else
        toktable[tokindex] = toksy = newtype;
} // of mutate

// match - If the expected token t matches the current one in toksy
// then call getToken() to get the next toksy
// else print mismatch error
void match(TOKEN t)
{
    if (toksy == t)
        getToken();
    else
        errmsg(err_mismatch, null_str, t);
} // match

// parse parameters of a function definition
NAMELIST parseParams()
{
    NAME nm;
    NAMELIST nl;

```

```

begin
  CASE toksy of

    rparsy: parseParams := nil;

    nameidsy: begin
      nm:=tokindex;
      match(nameidsy);
      if toksy = comsy then
        begin
          match(comsy);
          nl:=parseParams
        end
      else
        nl:=nil;
        parseParams := MkNamelist(nm,nl)
      end;
    otherwise;
      errmsg(1)
    end
  end; (* of parseParams *)

function parseExpr:EXP;forward;

(* 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);

```

```

switch (toksy)
{
  case rparsy:      // end of list, return null
    nl = 0;
    break;
  case nameidsy:
    nm = tokindex;
    match(nameidsy);
    if (toksy == comsy) // recursively parse remainder of list
    {
      match(comsy);
      nl = parseParams();
    }
    else
      nl = 0;
      nl = mkNamelist(nm, nl);
      break;
  default:
    errmsg(err_arglist, null_str, null_int);
    break;
}
return nl;
} // parseParams

EXP parseExpr(void); //forward declaration

// parseDef - parse function definition at userinput[pos]
// syntax: fun fun_name(arglist):=eL nuf
NAME parseDef()
{
  NAME fname;      // function name
  NAMELIST nl;     // formal parameters
  EXP e;           // body

  match(funsy);    // match "fun" and get next toksy

  switch (toksy) // match name
  {
    case nameidsy:
      mutate(funidsy); // set type to funidsy & do case funidsy
    case funidsy:
      fname = tokindex; // save name index for use below
      match(funidsy);
      break;
    default:
      errmsg(err_function, null_str, null_int);
      break;
  }
  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
    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

```

```

    e = parseExpr();
    match(nufsy);
    newDef(fname, nl, e);
    return fname;
} // parseDef

// parse arguments of a function call
EXPLIST parseArgs()
{
    EXP e;
    EXPLIST eL;

    if (toksy == rparsy)        // RPAREN marks end of arg list
        return 0;
    else
    {
        e = parseExpr();
        if (toksy == comsy)    // recursively parse rest of arg list
        {
            match(comsy);
            eL = parseArgs();
        }
        else
            eL = 0;
        return mkExplist(e,eL);
    }
} // parseArgs

// parse function call
// syntax: f(e1,e2, . . . , en)
EXP parseCall()
{
    EXPLIST eL;
    NAME nm;

    nm = tokindex;
    match(funidsy);
    match(lparsy);
    eL = parseArgs();
    match(rparsy);
    return mkAPEXP(nm,eL);
} /* parseCall */

// parseExplist - parse expression list
EXPLIST parseExplist()
{
    EXP e;
    EXPLIST eL;

    e = parseExpr();
    if (toksy == semsy)
    {
        match(semsy);
        eL = parseExplist();
    }
    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)

```

```

    eL = 0;
    return mkExplist(e,eL);
} // parseExplist

// parse if expression
// syntax: if e1 then e2 else e3 fi
EXP parseIf()
{
    EXP e1,e2,e3;
    EXPLIST eL;
    NAME nm;

    nm = tokindex;
    match(ifsy);
    e1 = parseExpr();
    match(thensy);
    e2 = parseExpr();
    match(elsesy);
    e3 = parseExpr();
    match(fisy);
    eL = mkExplist(e3,0);
    eL = mkExplist(e2,eL);
    eL = mkExplist(e1,eL);
    return mkAPEXP(nm,eL);
} // parseIf

// parse while expression
// syntax: while e1 do e2 od
EXP parseWhile()
{
    EXP e1,e2;
    EXPLIST eL;
    NAME nm;

    nm = tokindex;
    match(whilesy);
    e1 = parseExpr();
    match(dosy);
    e2 = parseExpr();
    match(odsy);
    eL = mkExplist(e2,0);
    eL = mkExplist(e1,eL);
    return mkAPEXP(nm,eL);
} // parseWhile

// parse sequence expression
EXP parseSeq()
{
    EXPLIST eL;
    NAME nm;

    nm = tokindex;
    match(seqsy);
    eL = parseExplist();
    match(qessy);
    return mkAPEXP(nm,eL);
}

```

<pre> 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 var 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) </pre>	<pre> } // 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 precedence from lowest to highest: := 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, parenthesized expr, function call EXP parseExp6() { EXP ex; NAME varnm; NUMBER num; switch (toksy) { case nameidsy: varnm = tokindex; match(nameidsy); ex = mkVAREXP(varnm); break; case numsy: num = numval; match(numsy); ex = mkVALEXP(num); break; case lparsy: match(lparsy); ex = parseExpr(); match(rparsy); break; case funidsy: ex = parseCall(); break; default: errmsg(err_exp6, null_str, null_int); </pre>
--	--

```

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 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

```

```

    break;
  }
  return ex;      // return pointer to expression
} // parseExp6

// parse unary addop, binary mulop
EXP parseExp5()
{
  NAME nm;
  EXP ex,e1,e2;
  EXPLIST eL;
  TOKEN addop_token = 0;
  NUMBER sign;

  if (toksy == addsy || toksy == subsy) // unary + or -
  {
    addop_token = toksy;
    match(toksy);
  }

  e1 = parseExp6();

  if (addop_token == subsy) // unary minus
  {
    // make an expr to multiply e1 by -1
    sign = -1;
    ex = mkVALEXP(sign);
    eL = mkExplist(ex,0);
    eL = mkExplist(e1,eL);
    nm = mulsy_index;
    e1 = mkAPEXP(nm,eL);
  }

  while (toksy == mulsy || toksy == divsy) // binary mulops
  {
    nm = tokindex;
    match(toktable[nm]);
    e2 = parseExp6();
    eL = mkExplist(e2,0);
    eL = mkExplist(e1,eL);
    e1 = mkAPEXP(nm,eL);
  }
  return e1;
} // parseExp5

// parse binary addop
EXP parseExp4()
{
  NAME nm;
  EXP e1,e2;
  EXPLIST eL;

```

```

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;
    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;

```

```

e1 = parseExp5();
while (toksy == addsy || toksy == subsy)
{
    nm = tokindex;
    match(toktable[nm]);
    e2 = parseExp5();
    eL = mkExplist(e2,0);
    eL = mkExplist(e1,eL);
    e1 = mkAPEXP(nm,eL);
}
return e1;
} // parseExp4

// parse binary relop
EXP parseExp3()
{
    NAME nm;
    EXP e1,e2;
    EXPLIST eL;

    e1 = parseExp4();
    while (toksy == lssy || toksy == eqsy || toksy == gtsy)
    {
        nm = tokindex;
        match(toktable[nm]);
        e2 = parseExp4();
        eL = mkExplist(e2,0);
        eL = mkExplist(e1,eL);
        e1 = mkAPEXP(nm,eL);
    }
    return e1;
} // parseExp3

// parse print op
EXP parseExp2()
{
    EXPLIST eL;
    EXP ex;
    NAME nm;
    BOOLEAN printflag;

    printflag = false;
    if (toksy == printsy)
    {
        printflag = true;
        nm = tokindex;
        match(printsy);
    }
    ex = parseExp3();
    if (printflag)
    {
        eL = mkExplist(ex,0);
        return mkAPEXP(nm,eL);
    }
    else
        return 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 must be a variable');
          goto 99
        end;
      end; (* of while *)
      parseExp1:=ex
    end; (* parseExp1 *)

  (* parse if, while, seq, expl *)
  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;

```

```

  } // parseExp2

  // parse assign op
  EXP parseExp1()
  {
    EXPLIST eL;
    EXP ex,e2;
    NAME nm;

    ex = parseExp2();
    while (toksy == assignsy)
    {
      // build an assignment expression
      nm = tokindex;
      match(assignsy);
      if (ex->etype == VAREXP) // lhs must be a variable
      {
        e2 = parseExp1(); // process rhs
        eL = mkExplist(e2,0);
        eL = mkExplist(ex,eL);
        ex = mkAPEXP(nm,eL);
      }
      else // illegal lhs

        errmsg(err_not_var, null_str, null_int);

    } // while
    return ex;
  } // parseExp1

  // parse if, while, seq, Expl
  EXP parseExpr()
  {
    EXP ex;

    switch (toksy)
    {
      case ifsy:
        ex = parseIf();
        break;
      case whilesy:
        ex = parseWhile();
        break;
      case seqsy:
        ex = parseSeq();
        break;
      case nameidsy:
      case numsy:
      case subsy:
      case funidsy:
      case printsy:
      case lparsy:
        ex = parseExp1();
        break;
      default:

```


<pre> 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 *) </pre>	<pre> errmsg(err_expr, null_str, null_int); break; } // case return ex; } /* parseExpr */ //----- // ENVIRONMENTS //----- // emptyEnv - return an environment with no bindings ENV emptyEnv() { return mkEnv(0, 0); } // emptyEnv // bindVar - bind variable nm to value n in environment rho void bindVar (NAME nm, NUMBER n, ENV rho) { rho->vars = mkNamelist(nm, rho->vars); rho->values = mkValuelist(n, rho->values); } // bindVar // findVar - look up nm in rho VALUelist findVar (NAME nm, ENV rho) { NAMELIST nl; VALUelist vl; BOOLEAN found; found = false; nl = rho->vars; vl = rho->values; while (nl != 0 && !found) { if (nl->head == nm) found = true; else { nl = nl->tail; vl = vl->tail; } } return vl; } // findVar // assign - assign value n to variable nm in rho void assign (NAME nm, NUMBER n, ENV rho) { VALUelist varloc; varloc = findVar(nm, rho); varloc->head = n; } // assign // fetch - return number bound to nm in rho </pre>
---	--


```

        goto 99
    end;
    n1 := v1^.head; (* 1st actual *)
    if arity(op) = 2
    then n2 := v1^.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 *)

```

```

n1 = v1->head;           // 1st actual
if (arity(op) == 2)
    n2 = v1->tail->head; // 2nd actual

switch (op)
{
    case ADDOP:
        n = n1 + n2;
        break;
    case SUBOP:
        n = n1 - n2;
        break;
    case MULOP:
        n = n1 * n2;
        break;
    case DIVOP:
        n = n1 / n2;
        break;
    case EQOP:
        if (n1 == n2)
            n = 1;
        else
            n = 0;
        break;
    case LTOP:
        if (n1 < n2)
            n = 1;
        else
            n = 0;
        break;
    case GTOP:
        if (n1 > n2)
            n = 1;
        else
            n = 0;
        break;
    case PRINTOP:
        prValue(n1);
        printf("\n");
        n = n1;
        break;
    default: // this case should never occur
        printf("applyValueOp: bad value for op = %d\n", op);
        break;
} // switch
return n;
} // applyValueOp

```

```

//-----
// EVALUATION
//-----

```

<pre> 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 CONTROL OP op to args in rho *) function applyCtrlOp (op: CONTROL OP; args: EXPLIST): NUMBER; var n: NUMBER; begin with args^ do case op of IFOP: </pre>	<pre> NUMBER eval (EXP, ENV); // forward declaration // evalList - evaluate each expression in eL VALUELIST evalList (EXPLIST eL, ENV rho) { NUMBER h; VALUELIST t; if (eL == 0) return 0; else { h = eval(eL->head, rho); t = evalList(eL->tail, rho); return mkValuelist(h, t); } } // evalList // applyUserFun - look up definition of nm and apply to actuals NUMBER applyUserFun (NAME nm, VALUELIST actuals) { FUNDEF f; ENV rho; f = fetchDef(nm); if (f == 0) errmsg(err_undef_func, null_str, nm); if (lengthNL(f->formals) != lengthVL(actuals)) errmsg(err_num_args2, null_str, nm); rho = mkEnv(f->formals, actuals); return eval(f->body, rho); } // applyUserFun // applyCtrlOp - apply CONTROL OP op to args in rho NUMBER applyCtrlOp (BUILTINOP op, EXPLIST args, ENV rho) { NUMBER n; switch (op) { case IFOP: </pre>
--	--

```

    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

    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)

```

```

        if (eval(args->head, rho))
            n = eval(args->tail->head, rho);
        else
            n = eval(args->tail->tail->head, rho);
        break;
    case WHILEOP:

        n = eval(args->head, rho);
        while (n)
        {
            n = eval(args->tail->head, rho);
            n = eval(args->head, rho);
        }
        break;

    case ASSIGNOP:

        n = eval(args->tail->head, rho);
        if (isBound(args->head->u.varble, rho))
            assign(args->head->u.varble, n, rho);
        else if (isBound(args->head->u.varble, globalEnv))
            assign(args->head->u.varble, n, globalEnv);
        else
            bindVar(args->head->u.varble, n, globalEnv);
        break;

    case SEQOP:

        while (args->tail != 0)
        {
            n = eval(args->head, rho);
            args = args->tail;
        }
        n = eval(args->head, rho); //value of last statement in seq
        break;
    default:
        printf("applyCtrlOp: bad value for op = %d\n", op);
        break;
    } // switch
    return n;
} // applyCtrlOp

// eval - return value of expression e in local environment rho
NUMBER eval (EXP e, ENV rho)
{
    BUILTINOP op;
    NUMBER n;

    switch (e->etype)
    {
        case VALEXP:
            n = e->u.num;
            break;
        case VAREXP:
            if (isBound(e->u.varble, rho))
                n = fetch(e->u.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 *)

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

begin (* chapter1 main *)

    initParse;
    initNames;

    globalEnv := emptyEnv;
    quittingtime := false;

99:

    while not quittingtime do
        begin
            reader;

```

```

        else if (isBound(e->u.varble, globalEnv))
            n = fetch(e->u.varble, globalEnv);
        else
            errmsg(err_undef_var, null_str, e->u.varble);
        break;

    case APEXP:
        op = e->u.ap.optr;
        if (op > numBuiltins-1)
            n = applyUserFun(op, evalList(e->u.ap.args, rho));
        else
            {
                if (op >= firstControlOp && op <= lastControlOp)
                    n = applyCtrlOp(op, e->u.ap.args, rho);
                else if (op >= firstValueOp && op <= lastValueOp)
                    n = applyValueOp(op, evalList(e->u.ap.args, rho));
                else
                    errmsg(err_undef_op, null_str, op);
            }
        break;
    default:
        printf("Invalid etype = %d", e->etype);
        break;
    } // switch
    return n;
} // eval

//-----
// READ-EVAL-PRINT LOOP (REPL)
//-----

int main (int argc, char **argv)
{
    short int error_no=0;

    initNames();
    initCmds();
    initParse();

    globalEnv = emptyEnv();
    quittingtime = false;

    // Set the jump location for longjmp.
    // After a parse error, errmsg() displays the error message and does
    // a longjmp to here. The program then continues with the REPL below.
    if (setjmp(JL99))
        ;

    // repeatedly read, parse and evaluate the input
    // until "quit$" is entered

    while (!quittingtime)
    {
        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 *)
```

```
getToken();

if (toksy == quitsy)
    quittingtime = true;
else if (toksy == funsy)
{
    prName(parseDef());
    printf("\n\n");
}
else
{
    currentExp = parseExpr();
    prValue(eval(currentExp, emptyEnv()));
    printf("\n\n");
}
} // while
exit (EXIT_SUCCESS);
}
```