EPC2

Documentation

Ronald Blaschke
Bowling Green State University
Computer Science

Bowling Green State University
Bowling Green, Ohio 43402

EPC2: Documentation
by Ronald Blaschke

Published $Date: 2000/03/30 20:50:39 $

Revision History

Revision $Revision: 1.3 $	$Date: 2000/03/30 20:50:39 $	Revised by: rb
Spelling corrections.		
Revision 1.2	2000/03/30 20:34:04	Revised by: rb
Added required project report.		
Revision 1.1	2000/03/27 01:18:58	Revised by: rb
First completed revision.		

Table of Contents

Foreword

This is the documentation for the project EPC2. Chapter 1, which is provided by Dr. Lancaster, describes the compiler to be built. It is based upon EPC, created by Jim Holmes [1]. Chapter 2 describes the design of the compiler, Chapter 3 is about testing and Chapter 4 provides the project report.

Appendix A includes the API and Appendix B the final source code.

The documentation is available online at
http://www.cosy.sbg.ac.at/~rblasch/study/cs609/epc2/ after the project's due date, that is April 12, 2000.

Chapter 1. Specification

The specifications are only provided for the printed version. If this is the printed version the specification will be inserted after this page.

Chapter 2. Design

This chapter describes the overall design. Since it is only a small compiler, only a very rough outline seems necessary.

Symbol Table

The symbol table holds three objects for each identifier. The name, the data type and the base address.

Declarations

The declarations form a block, just like the statements. There are single declarations and a block consists of a sequence of such declarations.

A single declaration consists of a list of identifiers and a data type. As soon as all information is available (that is list of identifiers and their data type) it is inserted in the symbol table.

The memory for a symbol is allocated when it is inserted into the symbol table. A memory block of the data type size is requested from a global memory management module.

The declarations are no further needed, because they put their information into the symbol table.

Data Items

Since we no longer have just simple variables, but also arrays, we need a generalization of identifiers. These are called data items and are basically pointers to elements.

Expressions

There are several ways to create arithmetic expressions including precedence. I decided for the ambiguous version with additional precedence rules. The grammar looks, at least for me, more natural this way.

Statements

The only interesting statement is the assignment statement, since multiple assignments are allowed.

An easy way seems to be a sequence of data items to which an expression is assigned. This fits good into SLICK, because I just need to emit all data items, then the expression, emit a assignment opcode for each data item and pop the expression off the stack.

All other statements are rather straightforward and not worth mentioning.

Chapter 3. Testing

Testing is done with the test suite provided in the subdirectory t/ of the project source code. Read its README for further details.

The following program was provided with the Specification (see Chapter 1).

```
program test1;
  var a, b: array [1..10] of integer;
      i, j: integer;

begin
  i := j := 2;        // Multiple assignments
  loop
    if i > 10 then exit end if;
    a[i] := j * 2 + i;
    b[i] := 11 - i;
    i := i + 1;
    j := j + 3
  end loop;
  b[5] := 0-2; ;     // Empty statement
  writeln (a[b[4]+1]);
  if i < j then
    if i = 0 then
      writeln (a[5])
    else
      writeln (a[3]);  // Empty statement
    end if
  end if;
  writeln (a[10])
end.
```

Running the program leads to the following result.

```
&gt;./epc test1.p | ./slick
slick interpreter version 1.2

48
```

13
62

Symbol Creation

A declaration is defined as follows (see the section called parser.y in Appendix B).

```
Declaration:
    IdentifierSeq COLONTK DataType
        {$$ = new DeclarationCls($1, $3, textline);}
```

The constructor of DeclarationCls then iterates over the IdenfifierSeq. For each identifier we do the following. First, look if it is already in the symbol table and output an error if so. If not, then allocate memory for the identifier with the memory management class MMCls. The required size is determined by the DataType, which is stored in a DataTypeCls. The symbol is finally inserted into the symbol table.

Multiple Assignments

Multiple assignments are defined as follows (see the section called parser.y in Appendix B).

```
AssignmentStmt:
    AssignmentSeq Expr
{$$ = new AssignmentStmtCls($1,$2,textline);}
    ;

AssignmentSeq:
    AssignmentSeq DataItem ASGTK
        {$$ = PAssignmentSeqCls($1)->append($2);}
    | DataItem ASGTK
        {$$ = new AssignmentSeqCls($1);}
    ;
```

An assignment consists of a sequence of data items (see Chapter 2) and an expression.

Code generation works as follows. First we put all data items onto the stack. Then we put the expression onto the stack. After that, we create as many assignment operations as there are data items. Finally, we pop of the expression from the stack. The code for this follows below (see also the section called emit.C in Appendix B).

```
int AssignmentStmtCls :: emit() {
  if (debugFlag) cout << "AssignmentStmtCls::emit()" << endl;
  StatementCls::emit();

  assign_seq->emit();
  expr->emit();

  for (int i=0; i<PAssignmentSeqCls(assign_seq)->getLength(); i++)
    cout << ":=" << endl;

  cout << "pop" << endl;

  return 0;
}
```

Array subscripts

Array elements are data items. The grammar looks as follows (see also the section called parser.y in Appendix B).

```
DataItem:
   Identifier
      {$$ = new DataItemCls($1);}
   | DataItem LBRACKTK Expr RBRACKTK
      {$$ = new DataItemCls($1, $3);}
   ;
```

Code generation works as follows. For data items that are simple identifiers we put the address onto the stack. If we have a subscript we assume that the address we have put onto the stack is the base address; therefore we put the expression onto the stack (by emitting it) and add it to the base address. The stack now contains the address of the specified array element.

Disambiguating Features

Scanner

For flex (see the section called scanner.l in Appendix B) we have the following situation.

```
":=" {ckout();
TOKEN(ASGTK, 0);
 return ASGTK;}

":"          {ckout();
            TOKEN(COLONTK, 0);
            return COLONTK;}

"="           {ckout();
            TOKEN(EQTK, 0);
             return EQTK;}
```

This is ambiguous, because := can be resolved as ASGTK or COLONTK EQTK. The “longest-match” rule applies, which resolves this to be an ASGTK.

Parser

The following precedence rules are used. Only the latter two are needed to disambiguate the grammar. The former two are for documentation only.

```
%nonassoc ASSIGNTK
%nonassoc LTTK LETK EQTK NETK GETK GTTK
%left PLUSTK MINUSTK
%left STARTK SLASHTK
```

These are needed because the expression grammar, which is shown below, is ambiguous.

```
Expr:
    Expr PLUSTK Expr
    | Expr MINUSTK Expr
    | Expr STARTK Expr
    | Expr SLASHTK Expr
    | Factor
    | LPARENTK Expr RPARENTK
    ;
```

Extensions

This version of epc2 is able to use multidimensional arrays. To do so just change this to the following in the Makefile.

```
#E_MULT_ARR= -DMULT_ARR
E_MULT_ARR=
```

```
E_MULT_ARR= -DMULT_ARR
#E_MULT_ARR=
```

Then you are able to write programs like this.

```
program marray;
  var a: array[1..2] of array[1..4] of integer;
begin
    a[2][3] := 25;
    writeln(a[2][3]);
end.
```

Running the program results in this.

```
&gt;./epc marray.p | ./slick
slick interpreter version 1.2

25
```

Error Checking

Array indices must start at 1. An error is raised if it is not.

The upper array bound may not be lower then the lower array bound.

Multiple declaration of variables raises an error.

Using an undeclared variable raises an error.

Minimal type checking is done. You cannot use array subscripts for simple types, eg the following raises an “Type and reference mismatch” error.

```
program wrong;
  var a: integer;
begin
    a[2] := 25;
end.
```

> Note: Using too few subscripts does not lead to an error. Instead the first element of the array is referenced (just as in the bash-shell).

Limitations

None.

Known Bugs

None.

Implementation Strategie

I followed Dr. Lancasters recommendation for the implementation.

First, the new tokens were added. This was done by adding their internal representation to the parser (the section called parser.y in Appendix B) and their patterns to the scanner (the section called scanner.l in Appendix B). Also, they were added to the scanner test program (the section called mainScan.C in Appendix B).

Then, the emitter (the section called emit.C in Appendix B) was modified to emit SLICK opcode for the existing language.

After that, I modified the grammar (the section called parser.y in Appendix B) to recognize the new language. The production actions were skipped for the moment.

Finally, I wrote the implementation of the production actions and the new classes. The order was: Declarations, array subscripts, expressions and statements.

Project Efforts

The following table summarizes the hours needed to create the compiler. In my opinion, these numbers should be considered as "faster than average."

Task	Time (h)
Design	1
Implementation (Declarations)	4
Implementation (Block)	3
Documentation	12
Tests	4
Total	24

Appendix A. API

The API documentation is generated with an external tool. In the printed version, it will follow this page. Otherwise, it is available online at http://www.cosy.sbg.ac.at/~rblasch/study/cs609/epc2/.

Appendix B. Source Code

These are the source code files for the project. Modifications and additions are emphasized.

scanner.l

```
%{
/*
 * scanner.l
 */

#include "scanparse.h"

// If we are just scanning and not parsing, we don't want to refer
// to parser classes.
#if defined SCANONLY
    #define TOKEN(ttype, ttext)
    #define YYSTYPE int
#else
    #define TOKEN(type,txt) lex_tok = new LexTokCls(yylineno, type, txt)
    #define YYSTYPE PPTreeNodeCls
    #include "p_tree.h"
    PLexTokCls lex_tok;
#endif

#include &lt;string.h&gt;
#include &lt;iostream.h&gt;

#include "ctrl.h"
#include "parser.tab.h"

char *textline = new char[257];
int yylineno = 1;

int ck_reserved_wd();
void ckout();
```

```
%}

digit [0-9]
digits {digit}+
letter [A-Za-z]
letter_or_digit  ({letter}|{digit})
ident {letter}{letter_or_digit}*
whitespace [ \t]
cr [\n]
other .

%x COMMENT

%%

"//"            BEGIN(COMMENT);
&lt;COMMENT&gt;[^\n]*\n        { textline[0] = '\0'; BEGIN(INITIAL); }

{whitespace} {ckout();}

{cr} {ckout();}


";" {ckout();
TOKEN(SCTK, 0);
return SCTK;}

"(" {ckout();
TOKEN(LPARENTK, 0);
 return LPARENTK;}

")" {ckout();
TOKEN(RPARENTK, 0);
 return RPARENTK;}

","             {ckout();
```

```
                TOKEN(COMMATK, 0);
                return COMMATK;}

":"              {ckout();
                TOKEN(COLONTK, 0);
                return COLONTK;}

"["              {ckout();
                TOKEN(LBRACKTK, 0);
                 return LBRACKTK;}

"]"              {ckout();
                TOKEN(RBRACKTK, 0);
                 return RBRACKTK;}

":=" {ckout();
TOKEN(ASGTK, 0);
 return ASGTK;}

".."             {ckout();
                TOKEN(RANGETK, 0);
                 return RANGETK;}

"." {ckout();
TOKEN(DOTTK, 0);
 return DOTTK;}

"+"              {ckout();
                TOKEN(PLUSTK, 0);
                 return PLUSTK;}

"-"              {ckout();
                TOKEN(MINUSTK, 0);
                 return MINUSTK;}

"*"              {ckout();
                TOKEN(STARTK, 0);
```

```
              return STARTK;}

"/"           {ckout();
             TOKEN(SLASHTK, 0);
              return SLASHTK;}

"&lt;"           {ckout();
             TOKEN(LTTK, 0);
              return LTTK;}

"&lt;="           {ckout();
             TOKEN(LETK, 0);
              return LETK;}

"="           {ckout();
             TOKEN(EQTK, 0);
              return EQTK;}

"&lt;&gt;"           {ckout();
             TOKEN(NETK, 0);
              return NETK;}

"&gt;="           {ckout();
             TOKEN(GETK, 0);
              return GETK;}

"&gt;"           {ckout();
             TOKEN(GTTK, 0);
              return GTTK;}

{digits} {ckout();
TOKEN(NUMLITERALTK, yytext);
return NUMLITERALTK;}

{ident} {ckout();
int actual_tk = ck_reserved_wd();
TOKEN( actual_tk, yytext);
```

```
return actual_tk;}

{other} {ckout();
return yytext[0];}

%%

void ckout() {
   textline = strcat(textline,yytext);
   if (yytext[0] == '\n') {
       #if !defined SCANONLY
       if (OptionCls::option_info() % 2) {  //List option is a 1
      cout &lt;&lt; "[";
   cout.width(5);
   cout &lt;&lt;  yylineno -1 &lt;&lt; "]     " &lt;&lt; textline ;
       }
       #endif
       textline[0] = '\0';
   }
}

struct rwtable_str {
    char *rw_name; /* lexeme */
    int  rw_yylex; /* token  */
};

rwtable_str rwtable[] = {
    "", IDENTIFIERTK,
    "array",             ARRAYTK,
    "begin", BEGINTK,
    "else",              ELSETK,
    "end", ENDTK,
    "exit",              EXITTK,
    "if",                IFTK,
    "integer",           INTEGERTK,
    "loop",              LOOPTK,
    "of",                OFTK,
```

```
    "program", PROGRAMTK,
    "then",          THENTK,
    "var",           VARTK,
    "writeln", WRITETK
};

#define LEN(x)  (sizeof(x)/sizeof((x)[0]))
#define ENDTABLE(v) (v - 1 + LEN(v))

int ck_reserved_wd() {
rwtable_str *low = rwtable;
rwtable_str *high = ENDTABLE(rwtable);
rwtable_str *mid;
int comp;
char temp[80];

strcpy(temp,yytext);

while (low &lt;= high)
{ mid = low + (high-low)/2;

if ((comp=strcmp(mid-&gt;rw_name, temp)) == 0)
return mid-&gt;rw_yylex;
else if (comp &lt; 0)
low = mid+1;
else
high = mid-1;
}
return rwtable-&gt;rw_yylex;  /* ie. token: IDENTIFIER! */
}

#if !defined __alpha
// For some reason, alpha doesn't like us to define yywrap!
int yywrap() {
   return 1;
}
#endif
```

parser.y

```
%{
/*
 * parser.y
 */

#include &lt;iostream.h&gt;
#include &lt;string.h&gt;

#define YYSTYPE PPTreeNodeCls

#include "p_tree.h"
#include "scanparse.h"

int yylex();
void yyerror(char*);

extern char* textline; //Defined in scanner.l
%}

%token PROGRAMTK
%token BEGINTK
%token ENDTK
%token SCTK
%token ASGTK
%token DOTTK
%token IDENTIFIERTK
%token NUMLITERALTK
%token WRITETK
%token LPARENTK
%token RPARENTK

%token VARTK
%token ARRAYTK
```

```
%token RANGETK
%token OFTK
%token LOOPTK
%token IFTK
%token THENTK
%token ELSETK
%token EXITTK

%token COMMATK
%token COLONTK
%token LBRACKTK
%token RBRACKTK

/* arithmetic operators */
%token PLUSTK
%token MINUSTK
%token STARTK
%token SLASHTK

/* relational operators */
%token LTTK
%token LETK
%token EQTK
%token NETK
%token GETK
%token GTTK

%token INTEGERTK


%nonassoc ASSIGNTK
%nonassoc LTTK LETK EQTK NETK GETK GTTK
%left PLUSTK MINUSTK
%left STARTK SLASHTK


%start Program
```

```
%%

Program:
    PROGRAMTK Identifier SCTK
    DeclarationBlock Block
{PProgramCls pgm = new ProgramCls($2,$5);}
    ;

DeclarationBlock:
    VARTK DeclarationSeq
        {$$ = $2;}
    | /* empty */
        {$$ = new DeclarationSeqCls();}
    ;

DeclarationSeq:
    DeclarationSeq SCTK Declaration
        {$$ = PDeclarationSeqCls($1)-&gt;append($3);}
    | Declaration
        {$$ = new DeclarationSeqCls($1);}
    ;

Declaration:
    IdentifierSeq COLONTK DataType
        {$$ = new DeclarationCls($1, $3, textline);}
    | /* empty */
        {$$ = new DeclarationCls();}
    ;

IdentifierSeq:
    IdentifierSeq COMMATK Identifier
        {$$ = PIdentSeqCls($1)-&gt;append($3);}
    | Identifier
        {$$ = new IdentSeqCls($1);}
    ;

DataType:
```

```
    BasicType
    | CompoundType
    ;

BasicType:
    INTEGERTK
        {$$ = new IntTypeCls();}
    ;

CompoundType:
    ARRAYTK LBRACKTK Number RANGETK Num-
ber RBRACKTK OFTK DataType
        {$$ = new ArrayTypeCls($3, $5, $8);}
    ;

Block:
    BEGINTK
StatementSeq
    ENDTK DOTTK
{$$ = new BlockCls($2);}
    ;
StatementSeq:
    Statement
{$$ = new StatementSeqCls($1);}
    | StatementSeq SCTK Statement
{$$ = PStatementSeqCls($1) -&gt; append($3);}
    ;
Statement:
    /* empty */
{$$ = new EmptyStmtCls;}
    | AssignmentStmt
    | WriteStmt

    | LOOPTK StatementSeq ENDTK LOOPTK
        {$$ = new LoopStmtCls($2);}
    | EXITTK
        {$$ = new ExitStmtCls();}
```

```
    | IFTK Condition THENTK StatementSeq ElseBlock ENDTK IFTK
        {$$ = new IfStmtCls($2, $4, $5);}
    ;

ElseBlock:
    /* empty */
        {$$ = new ElseStmtCls();}
    | ELSETK StatementSeq
        {$$ = new ElseStmtCls($2);}
    ;

Condition:
    | Expr RelOp Expr
        {$$ = new ConditionCls($1, $2, $3);}
    ;

RelOp:
    LTTK
        {$$ = new RelOpCls(RelOpCls::ltOp);}
    | LETK
        {$$ = new RelOpCls(RelOpCls::leOp);}
    | EQTK
        {$$ = new RelOpCls(RelOpCls::eqOp);}
    | NETK
        {$$ = new RelOpCls(RelOpCls::neOp);}
    | GETK
        {$$ = new RelOpCls(RelOpCls::geOp);}
    | GTTK
        {$$ = new RelOpCls(RelOpCls::gtOp);}
    ;

AssignmentStmt:
    AssignmentSeq Expr
{$$ = new AssignmentStmtCls($1,$2,textline);}
    ;
```

```
AssignmentSeq:
    AssignmentSeq DataItem ASGTK
        {$$ = PAssignmentSeqCls($1)-&gt;append($2);}
    | DataItem ASGTK
        {$$ = new AssignmentSeqCls($1);}
    ;

WriteStmt:
    WRITETK LPARENTK Expr RPARENTK
{$$ = new WriteStmtCls($3,textline);}
    ;

Expr:
    Expr PLUSTK Expr
        {$$ = new ArithmCls($1, $3, ArithmCls::PlusOp);}
    | Expr MINUSTK Expr
        {$$ = new ArithmCls($1, $3, ArithmCls::MinusOp);}
    | Expr STARTK Expr
        {$$ = new ArithmCls($1, $3, ArithmCls::MultOp);}
    | Expr SLASHTK Expr
        {$$ = new ArithmCls($1, $3, ArithmCls::DivOp);}
    | Factor
    | LPARENTK Expr RPARENTK
        {$$ = $2;}
    ;

Factor:
    Number
{$$ = new NumFactorCls($1);}
    | DataItem
{$$ = new VarAccessFactorCls($1);}
    ;

DataItem:
    Identifier
        {$$ = new DataItemCls($1);}
/* | Identifier LBRAKTK Expr RBRACKTK */
```

```
    | DataItem LBRACKTK Expr RBRACKTK
        {$$ = new DataItemCls($1, $3);}
    ;

Identifier:
    IDENTIFIERTK
{$$ = new IdentCls();}
    ;
Number:
    NUMLITERALTK
{$$ = new NumLiteralCls();}
    ;
```

main.C

```
/*
 * main.C  – For Example Compiler
 */

#include &lt;iostream.h&gt;

#include "ctrl.h"

int main(int argc, char *argv[])
{
    PControllerCls ctl = new ControllerCls(argc,argv);
}
```

mainScan.C

```
// mainScan.C
//
// This will test the scanner.
//
// Example:  scan sample.p
```

```
#include &lt;iostream.h&gt;
#include &lt;stdio.h&gt;
#include &lt;string.h&gt;
#include "parser.tab.h"

int yylex();

int main(int argc, char *argv[])
{
  int token = 0;
  int count = 0;
  char *tokenString[100];
  char *badToken = "Invalid token";
  char fileName[100];

  // Initialize character version of tokens
  for (int i = 0; i &lt; 100; ++i)
    tokenString[i] = badToken;
  tokenString[PROGRAMTK   -255] = "program";
  tokenString[BEGINTK     -255] = "begin";
  tokenString[ENDTK       -255] = "end";
  tokenString[SCTK        -255] = ";";
  tokenString[ASGTK       -255] = ":=";
  tokenString[DOTTK       -255] = ".";
  tokenString[IDENTIFIERTK-255] = "&lt;identifier&gt;";
  tokenString[NUMLITERALTK-255] = "&lt;literal&gt;";
  tokenString[WRITETK     -255] = "writeln";
  tokenString[LPARENTK    -255] = "(";
  tokenString[RPARENTK    -255] = ")";
  tokenString[VARTK       -255] = "var";
  tokenString[ARRAYTK     -255] = "array";
  tokenString[RANGETK     -255] = "..";
  tokenString[OFTK        -255] = "of";
  tokenString[LOOPTK      -255] = "loop";
  tokenString[IFTK        -255] = "if";
  tokenString[THENTK      -255] = "then";
  tokenString[ELSETK      -255] = "else";
```

```
tokenString[EXITTK -255] = "exit";
tokenString[COMMATK -255] = ",";
tokenString[COLONTK -255] = ":";
tokenString[LBRACKTK -255] = "[";
tokenString[RBRACKTK -255] = "]";
tokenString[PLUSTK -255] = "+";
tokenString[MINUSTK -255] = "-";
tokenString[STARTK -255] = "*";
tokenString[SLASHTK -255] = "/";
tokenString[LTTK -255] = "&lt;";
tokenString[LETK -255] = "&lt;=";
tokenString[EQTK -255] = "=";
tokenString[NETK -255] = "&lt;&gt;";
tokenString[GETK -255] = "&gt;=";
tokenString[GTTK -255] = "&gt;";
tokenString[INTEGERTK -255] = "integer";

// Get the name of the source file
if (argc &lt; 2) {
  cout &lt;&lt; "Source program: ";
  cin &gt;&gt; fileName;
}
else
  strncpy (fileName, argv[1], 80);

// Do nothing if no filename provided
if (strlen(fileName) == 0)
  return 0;

// Redirect cin to the source file
if (freopen(fileName, "r", stdin)) {
  ios::sync_with_stdio();
}
else {
  cerr &lt;&lt; "*** Unable to open file " &lt;&lt; fileName &lt;&lt; endl;
  return 1;
}
```

```
  // Call yylex repeatedly until a dot token is found
  while ( /*(count++ &lt; 100) &&*/  (token != DOTTK)) {
    token = yylex();
    cout &lt;&lt; tokenString[token-255] &lt;&lt; endl;
  }
  return 0;
}
```

ctrl.h

```
/*
 * ctrl.h Controller definition module
 */

typedef class OptionCls *POptionCls;
class OptionCls {
  public:
static int option_info();
friend class ControllerCls;
  private:
OptionCls();   //Called only by ControllerCls
static int list;
static int emit;
};
/* end-OptionCls */

class SymtabCls;  // 'forward'
class PTreeCls;   //     references

typedef class ControllerCls *PControllerCls;
class ControllerCls {
  public:
ControllerCls(int argc, char** argv);
void print();
  private:
SymtabCls  *std_table;
```

```
PTreeCls *parse_tree;
int open_file(char*);
};
/* end-ControllerCls */

typedef class MMCls *PMMCls;
class MMCls {
 public:
  static int allocate(int size);

 private:
  static int mem_pos;
};
```

ctrl.C

```
/* header */
/*
 * ctrl.C  ControllerCls Implementation Module
 */

#include &lt;iostream.h&gt;
#include &lt;stdio.h&gt;
#include &lt;string.h&gt;

#include "scanparse.h"
#include "p_tree.h"
#include "symtab.h"

#include "ctrl.h"
/* end-header */

// global declaration
bool debugFlag;

// static data
int OptionCls::list = 0;
```

```
int OptionCls::emit = 1;

OptionCls :: OptionCls() {
    if (debugFlag) cout &lt;&lt; "OptionCls() " &lt;&lt; endl;
}

int OptionCls :: option_info() {
    return (list );
}
/* end-OptionCls */
int  ControllerCls :: open_file(char* source_file) {
    if (debugFlag) cout &lt;&lt; "Controller-
Cls::open_file()" &lt;&lt; source_file &lt;&lt; endl;
    int length = strlen(source_file);
    //Check for  .p  extension
    if ((length &gt; 1) && ((source_file[length -2] == '.') &&
              (source_file[length -1] == 'p')  )) {
if (!freopen(source_file, "r", stdin)) {
    cout &lt;&lt; " Cannot open file – Sorry " &lt;&lt; endl;
    //should be done by an ErrorCls object
    return 0;
} else {
    return 1;
}
    } else if (length == 0) {
        cout &lt;&lt; " No file specified" &lt;&lt; endl;
        return 0;
    } else {
        cout &lt;&lt; " File must have a  .p  extension" &lt;&lt; endl;
        //should be done by an ErrorCls object
return 0;
    }
}
/* end-ControllerCls-open */

ControllerCls :: ControllerCls(int argc, char** argv) {
    if (debugFlag) cout &lt;&lt; "ControllerCls() " &lt;&lt; endl;
```

```
    this -&gt; std_table = new SymtabCls;
    PScopeCls scp = new ScopeCls;
    scp -&gt; vista = this -&gt; std_table;

    char *source_file = new char[80];
    this -&gt; parse_tree = 0;
    if (argc &lt;= 1) {
cout &lt;&lt; " Usage:  epc [-elds] &lt;filename&gt;.p" &lt;&lt; endl;
return;
    } else {
for (int i = 1; i &lt; argc; i++) {
    if (*argv[i] == '-') {
while (*++argv[i]) {
    switch(*argv[i]) {
    case 'l':
OptionCls :: list = 1;
continue;

    case 'e':
OptionCls :: emit = 2;
continue;

    case 'd':
                    debugFlag = true;
                    continue;

    default:
cerr &lt;&lt; "Unknown option " &lt;&lt;
*argv[i] &lt;&lt; endl;
    }
}
    } else {
    source_file = argv[i];
    }
}
```

```
    }
    if (open_file(source_file)) {
ios::sync_with_stdio();
PScanParseCls sp = new ScanParseCls;
this -&gt; parse_tree = sp -&gt; parse_tree;

if (OptionCls::emit) {
    parse_tree -&gt; emit();
} else {
            cout &lt;&lt; "Unable to execute: interpreter not implemented." &lt;&lt; endl;
}
    }
}
/* end-ControllerCls-Cont */

int MMCls::mem_pos = 0;

int MMCls::allocate(int size) {
  int mem;

  if (mem_pos + size &lt; 200) {
    mem = mem_pos;
    mem_pos +=size;
  }
  else {
    cout &lt;&lt; "error: memory overflow!" &lt;&lt; endl;
    exit(1);
  }

  return mem;
}
```

scanparse.h

```
/*
 * scanparse.h
 */
```

```
typedef class LexTokCls *PLexTokCls;
class LexTokCls {
  public:
LexTokCls(int LineNo, int Token, char *Lexeme);
char* get_lexeme() {return lexeme;}
  private:
int line_no;
char   *lexeme;
int token;
};

class ControllerCls;
class PTreeCls;

typedef class ScanParseCls *PScanParseCls;
class ScanParseCls {
  public:
ScanParseCls();
void     print();
friend class ControllerCls;
  private:
PTreeCls    *parse_tree;
};
```

scanparse.C

```
/*
 * scanparse.C
 */

#include &lt;stream.h&gt;
#include &lt;string.h&gt;
#include "p_tree.h"
#include "scanparse.h"

int yyparse();
```

```
extern bool debugFlag;

LexTokCls :: LexTokCls(int LineNo, int Token, char *Lexeme) {
    if (debugFlag) cout &lt;&lt; "LexTokCls(LineNo, Token, Lexeme)" &lt;&lt; endl;
    line_no = LineNo;
    token   = Token;
    lexeme = new char[80];
    if (Lexeme) {
        strcpy(lexeme,Lexeme);
    }
}
// end_LexTokCls

PPTreeNodeCls prgm_node; //Global!
//Set by top of tree, ProgramCls.

ScanParseCls :: ScanParseCls() {
    if (debugFlag) cout &lt;&lt; "ScanParseCls()" &lt;&lt; endl;
    yyparse();
    parse_tree = new PTreeCls(prgm_node);
}
// end_ScanParseCls

void yyerror(char* s) {
        extern char *textline;
        cout &lt;&lt; "error has occurred..." &lt;&lt; s &lt;&lt; endl;
        cout &lt;&lt; textline &lt;&lt; endl;
        exit(1);
}
```

symtab.h

```
/*
 * symtab.h
 */

#include "p_tree.h"
```

```
class ControllerCls;
class SymtabCls;

typedef class ScopeCls *PScopeCls;
class ScopeCls {
  public:
ScopeCls()        {;}
static SymtabCls    *get_vista() {return vista;}
friend class ControllerCls;
  private:
static SymtabCls *vista;
};

typedef class SymtabEntryCls *PSymtabEntryCls;
class SymtabEntryCls   {
  public:
SymtabEntryCls() {;}
SymtabEntryCls(char *Name);
friend class SymtabCls;
virtual int emit();
  protected:
char *name;
};

typedef class SymbolCls *PSymbolCls;
class SymbolCls: public SymtabEntryCls {
 public:
  SymbolCls() {}
  SymbolCls(char *name, PDataTypeCls dataType, PAddressCls addr);
  PDataTypeCls getDataType();
  PAddressCls getAddress();

 private:
  PDataTypeCls data_type;
  PAddressCls address;
};
```

```
/* end-attributes-def */

/* symtab-def */
/* ************************************************************ */
/*
 * Symbol Table
 */
/* ************************************************************ */


typedef class SymtabCls *PSymtabCls;
class SymtabCls  {
   public:
SymtabCls();
int insert(PSymtabEntryCls);
PSymtabEntryCls lookup(char*);
int emit();
   private:
int tablesize;
int next_location;
int   *hashtable;
PSymtabEntryCls *symtab;  //N.B.:  pointers!
int  hash(char *);
};
/* end-symtab-def */
```

symtab.C

```
/*
 * symtab.C
 */
#include &lt;iostream.h&gt;
#include &lt;string.h&gt;

#include "symtab.h"

extern bool debugFlag;
```

```
PSymtabCls ScopeCls::vista = 0; // define static member

SymtabEntryCls :: SymtabEntryCls(char *Name) {
    if (debugFlag) cout &lt;&lt; "SymtabEntryCls()" &lt;&lt; endl;
    name = Name;
}

int SymtabEntryCls :: emit() {
    cout &lt;&lt; "SymtabEntryCls::emit() BASE CLASS!!!!" &lt;&lt; endl;
    return 0;
}

SymbolCls::SymbolCls(char *name, PDataTypeCls dataType, PAddressCls addr)
{
  this-&gt;name = name;
  data_type = dataType;
  address = addr;
}

PDataTypeCls SymbolCls::getDataType()
{
  return data_type;
}

PAddressCls SymbolCls::getAddress()
{
  return address;
}

/* end-attributes */

/* begin-symtab */
/* **************************************************************** */
/*
 * SymtabCls
 */
```

```
/* ****************************************************************** */
SymtabCls :: SymtabCls() {
    if (debugFlag) cout &lt;&lt; "SymtabCls::()"&lt;&lt; endl;
    tablesize = 17;     //Must be a prime for good performance
    hashtable = new int[tablesize];
    next_location = 1;  // sacrifice 0th spot - hashtable empty:NIL
    symtab = new PSymtabEntryCls[tablesize]; // Note "P" !!!!
    PSymtabEntryCls tmp = new SymtabEntryCls("      ");
    for (int i=0; i&lt;tablesize; i++) {
hashtable[i] = 0;
symtab[i] = tmp;
    }
}

int SymtabCls :: hash(char *s) {
    if (debugFlag) cout &lt;&lt; "SymtabCls :: hashing for " &lt;&lt; s ;
    char* ss = s;
    unsigned int h = 0, g;
    for (; *ss != '\0'; ss++) {
h = (h &lt;&lt; 4) + *ss;
if (g = h & 0xf0000000) {
    h ^= g &gt;&gt; 24; // fold top 4 bits onto ——X-
    h ^= g; // clear top 4 bits
}
    }
    return h % tablesize ;
}

int SymtabCls :: insert(PSymtabEntryCls info) {
    if (debugFlag) cout &lt;&lt; "SymtabCls::insert()" &lt;&lt; endl;
    //Return 0 if insert successful; else location in symtab.

    //Look for open slot in the hashtable....
    int Try, hash_try; //'try' can be a reserved word
    char *Name = info -&gt; name;
    if (debugFlag) cout&lt;&lt;"preparing to enter"&lt;&lt;Name&lt;&lt;"\n";
    Try = hash(Name);
```

```
    if (debugFlag) cout &lt;&lt; "preparing to go into hash ta-
ble at " &lt;&lt; Try &lt;&lt; endl;

    while (hash_try = hashtable[Try]) { //something's in hashtable
//Check to see if it's the same thing we want to insert...
if (!strcmp((symtab[hash_try] -&gt; name), Name)) {
    return hash_try; //it's already there!
} else if (++Try &gt;= tablesize) {
    //resolve collision by looking for open spot ...
    Try = 0; //wrap around
}
//Mature (growing) tables can be at most 2/3 full,
    }
    // So an open spot MUST be found
    hashtable[Try] = next_location;
   if (debugFlag) cout &lt;&lt; "entered current loc'n in table " &lt;&lt; Try &lt;&lt; endl;
    symtab[next_location++] = info; //Since they're both pointers
    return 0; // success!
}

PSymtabEntryCls SymtabCls :: lookup(char *Name) {
    if (debugFlag) cout &lt;&lt; "SymtabCls :: lookup for " &lt;&lt; Name ;
    int cur_table_size = tablesize;
    int Try, orig_try, hash_try;

    orig_try = Try = hash(Name);
    hash_try = hashtable[Try];
    while (hash_try) {
if (!strcmp(symtab[hash_try] -&gt; name, Name)) {
    return symtab[hash_try];
}
if (++Try &gt;= cur_table_size) Try = 0; // wrap around
if (Try == orig_try) {
    return symtab[0];
} else {
    hash_try = hashtable[Try];
}
```

```
    }
return 0; //Failure!
}
```

p_tree.h

```
#ifndef p_tree_h_
#define p_tree_h_

/*
 * p_tree.h
 */

class LexTokCls;
class SymtabCls;

typedef class LstSeqBldrCls *PLstSeqBldrCls;
class LstSeqBldrCls {
public:
   LstSeqBldrCls();
   virtual PLstSeqBldrCls  append(PLstSeqBldrCls);
   virtual PLstSeqBldrCls get_next()
      {return next;}
private:
   PLstSeqBldrCls next;
};
/* end-LstSeqBldrCls */

typedef class PTreeNodeCls *PPTreeNodeCls;
class PTreeNodeCls {
public:
   PTreeNodeCls();
   virtual int emit();
   virtual void print();
   virtual LexTokCls *get_lex_tok()
      {return lt;}
protected:
```

```
   LexTokCls *lt;
};
/* end-PTreeNodeCls */

typedef class PTreeCls *PPTreeCls;
class PTreeCls {
public:
   PTreeCls(PPTreeNodeCls Root);
   int emit();
   void print();
private:
   PPTreeNodeCls root;
   PPTreeNodeCls current;
};
/* end-PTreeCls */

typedef class DataTypeCls *PDataTypeCls;
class DataTypeCls: public PTreeNodeCls {
public:
   virtual int getSize()=0;
   virtual PDataTypeCls copy()=0;
};

typedef class DataItemCls *PDataItemCls;
class DataItemCls: public PTreeNodeCls, public LstSeqBldrCls {
public:
   DataItemCls(PPTreeNodeCls item, PPTreeNodeCls sub=NULL);
   int emit();

#ifdef MULT_ARR
   virtual PDataTypeCls typeEmit();
#endif

private:
   PPTreeNodeCls item;
   PPTreeNodeCls sub;
};
```

```
typedef class StatementCls *PStatementCls;
class StatementCls : public PTreeNodeCls, public LstSeqBldrCls {
public:
    StatementCls():stmt_text(NULL) {}

  StatementCls(char *StmtText);
  int emit();
protected:
  char *stmt_text;
};
/* end-StatementCls */

typedef class StatementSeqCls *PStatementSeqCls;
class StatementSeqCls : public PTreeNodeCls {
public:
  StatementSeqCls(PPTreeNodeCls Stmt);
  int emit();
  PPTreeNodeCls append(PPTreeNodeCls Stmt);
private:
  StatementCls  *seq_head;
  StatementCls  *seq_tail;
};
/* end-StatementSeqCls */

typedef class EmptyStmtCls *PEmptyStmtCls;
class EmptyStmtCls : public StatementCls {
public:
  EmptyStmtCls() {}
  int emit();
};
/* end-EmptyStmtCls */

typedef class WriteStmtCls *PWriteStmtCls;
class WriteStmtCls : public StatementCls {
public:
  WriteStmtCls(PPTreeNodeCls Expr, char* StmtText);
```

```
  /*
   SLICK Emitter.

   Emits the expression and the {\em write} opcode.

   return 0
  */
  int emit();

private:
  PPTreeNodeCls expr;
};
/* end-WriteStmtCls */

typedef class AssignmentStmtCls *PAssignmentStmtCls;
class AssignmentStmtCls : public StatementCls {
public:
    AssignmentStmtCls(PPTreeNodeCls assignSeq, PPTreeNodeCls Expr,
   char* StmtText);

  int emit();
private:
    PPTreeNodeCls assign_seq;    //lhs

  PPTreeNodeCls expr; //rhs
};
/* end-AssignmentStmtCls */

typedef class AssignmentSeqCls *PAssignmentSeqCls;
class AssignmentSeqCls: public PTreeNodeCls {
public:
  AssignmentSeqCls(PPTreeNodeCls item);
  int emit();
  PPTreeNodeCls append(PPTreeNodeCls item);
  int getLength();
```

```
private:
   DataItemCls *seq_head;
   DataItemCls *seq_tail;
   int length;
};

typedef class LoopStmtCls *PLoopStmtCls;
class LoopStmtCls: public StatementCls {
public:
   LoopStmtCls(PPTreeNodeCls stmts);
   int emit();

private:
   PPTreeNodeCls statements;
};

typedef class ExitStmtCls *PExitStmtCls;
class ExitStmtCls: public StatementCls {
public:
   /*
     SLICK Emitter.

     Emit {\em exit} opcode.

     @return 0
   */
   int emit();
};


typedef class IfStmtCls *PIfStmtCls;
class IfStmtCls: public StatementCls {
public:
   IfStmtCls(PPTreeNodeCls condition, PPTreeNodeCls statements, PPTreeNode-
Cls elseStmts);
   int emit();
```

```
private:
   /*
     Condition.
   */
   PPTreeNodeCls condition;

   /*
     True condition statements.
   */
   PPTreeNodeCls statements;

   /*
     False condition statements.
   */
   PPTreeNodeCls elseStmts;
};

typedef class ElseStmtCls *PElseStmtCls;
class ElseStmtCls: public StatementCls {
public:
   ElseStmtCls(): statements(NULL) {}
   ElseStmtCls(PPTreeNodeCls statements);
   int emit();

private:
   PPTreeNodeCls statements;
};

typedef class RelOpCls *PRelOpCls;
class RelOpCls: public PTreeNodeCls {
public:
   enum RelOp {
      ltOp,
      leOp,
      eqOp,
      neOp,
      geOp,
```

```
        gtOp
    };
    RelOpCls(RelOp op);
    int emit();

private:
    /*
      The relational operator.
    */
    RelOp op;
};

typedef class ConditionCls *PConditionCls;
class ConditionCls: public PTreeNodeCls {
public:
    ConditionCls(PPTreeNodeCls expr1, PPTreeNodeCls relOp, PPTreeNodeCls expr2);
    int emit();

private:
    PPTreeNodeCls expr1;
    PPTreeNodeCls relOp;
    PPTreeNodeCls expr2;
};

typedef class ExprCls  *PExprCls;
class ExprCls : public PTreeNodeCls {
public:
    ExprCls();
    virtual int    emit();
protected:
    int value;
};
/* end-ExprCls */

typedef class FactorCls *PFactorCls;
class FactorCls : public ExprCls {
public:
```

```
  FactorCls();
};
/* end-FactorCls */

typedef class NumFactorCls *PNumFactorCls;
class NumFactorCls : public FactorCls {
public:
  NumFactorCls(PPTreeNodeCls NumLit);
  int emit();
};
/* end-NumFactorCls */

typedef class VarAccessFactorCls *PVarAccessFactorCls;
class VarAccessFactorCls : public FactorCls {
public:
    VarAccessFactorCls(PPTreeNodeCls Item);

  int emit();
private:
    PPTreeNodeCls item;

};
/* end-VarAccessFactorCls */

typedef class ArithmCls *PArithmCls;
class ArithmCls: public PTreeNodeCls {
public:
  enum ArithmOp {
    PlusOp,
    MinusOp,
    MultOp,
    DivOp
  };
  ArithmCls(PPTreeNodeCls left, PPTreeNodeCls right, ArithmOp op);
  int emit();

private:
```

```
  PPTreeNodeCls left;
  PPTreeNodeCls right;
  ArithmOp op;
};

typedef class NumLiteralCls  *PNumLiteralCls;
class NumLiteralCls : public PTreeNodeCls {
public:
  NumLiteralCls();
  int      get_value()
    {return value;}

private:
  int      value;
};
/* end-NumLiteralCls */

typedef class IdentCls  *PIdentCls;
class IdentCls : public PTreeNodeCls, public LstSeqBldrCls {
public:
  IdentCls();
  char         *get_name();
};
/* end-IdentCls */

typedef class IdentSeqCls *PIdentSeqCls;
class IdentSeqCls: public PTreeNodeCls {
public:
  IdentSeqCls(PPTreeNodeCls ident);
  int emit();
  PPTreeNodeCls append(PPTreeNodeCls ident);
  PIdentCls get_first();

private:
  IdentCls *seq_head;
  IdentCls *seq_tail;
};
```

```
typedef class BasicTypeCls *PBasicTypeCls;
class BasicTypeCls: public DataTypeCls {
};

typedef class CompTypeCls *PCompTypeCls;
class CompTypeCls: public DataTypeCls {
};

typedef class IntTypeCls *PIntTypeCls;
class IntTypeCls: public BasicTypeCls {
public:
   int getSize() {return 1;}
   PIntTypeCls copy();
};

typedef class ArrayTypeCls *PArrayTypeCls;
class ArrayTypeCls: public CompTypeCls {
public:
   ArrayTypeCls(PPTreeNodeCls rangeFrom, PPTreeNodeCls rangeTo,
PPTreeNodeCls dataType);
   int getSize();
   int getItemSize();
   PNumLiteralCls getRangeFrom();

   PNumLiteralCls getRangeTo();
   PDataTypeCls getItemDataType();
   PArrayTypeCls copy();

private:
   PNumLiteralCls range_from;
   PNumLiteralCls range_to;
   PDataTypeCls data_type;
};

typedef class DeclarationCls *PDeclarationCls;
class DeclarationCls : public PTreeNodeCls, public LstSeqBldrCls {
```

```
public:
   DeclarationCls():decl_text(NULL) {}
   DeclarationCls(PPTreeNodeCls identSeq, PPTreeNodeCls dataType, char
  *declText);
   int emit();

protected:
   PIdentSeqCls ident_seq;
   PDataTypeCls data_type;
   char *decl_text;
};

typedef class DeclarationSeqCls *PDeclarationSeqCls;
class DeclarationSeqCls: public PTreeNodeCls {
public:
   DeclarationSeqCls():seq_head(NULL), seq_tail(NULL) {}
   DeclarationSeqCls(PPTreeNodeCls decl);
   int emit();
   PPTreeNodeCls append(PPTreeNodeCls decl);

private:
   DeclarationCls *seq_head;
   DeclarationCls *seq_tail;
};

typedef class AddressCls *PAddressCls;
class AddressCls {
public:
   AddressCls():addr(-1) {}
   virtual void setAddr(int a);
   virtual int getAddr();

protected:
   int addr;
};

typedef class BlockCls *PBlockCls;
```

```
class BlockCls : public PTreeNodeCls {
public:
   BlockCls(PPTreeNodeCls StmtSeq);
   int emit();
private:
   PPTreeNodeCls stmt_seq;
};
/* end-BlockCls */

typedef class ProgramCls *PProgramCls;
class ProgramCls : public PTreeNodeCls {
public:
   ProgramCls()  {}
   ProgramCls(PPTreeNodeCls Ident, PPTreeNodeCls Block);
   int emit();
   void print();
private:
   SymtabCls *std_table;
   PIdentCls ident;
   PPTreeNodeCls  block;
};
/* end-ProgramCls */

#endif
```

p_tree.C

```
/*
 * p_tree.C
 */

#include &lt;iostream.h&gt;
#include &lt;stdlib.h&gt;
#include &lt;string.h&gt;

#include "ctrl.h"
#include "scanparse.h"
```

```
#include "symtab.h"

#include "p_tree.h"

extern bool debugFlag;

LstSeqBldrCls :: LstSeqBldrCls() {
   if (debugFlag) cout &lt;&lt; "LstSeqBldrCls()" &lt;&lt; endl;
   next = 0;
}

PLstSeqBldrCls LstSeqBldrCls ::
append(PLstSeqBldrCls ToBeAdded) {
   if (debugFlag) cout &lt;&lt; "LstSeqBldrCls::append()" &lt;&lt; endl;
   return this -&gt; next = ToBeAdded;
}
/* end-LstSeqBldrCls */
PTreeNodeCls :: PTreeNodeCls() {
   if (debugFlag) cout &lt;&lt; "PTreeNodeCls" &lt;&lt; endl;
   extern LexTokCls *lex_tok;
   lt = lex_tok;
}

int PTreeNodeCls :: emit() {
   cout &lt;&lt; "PTreeNodeCls::emit()  BASECLASS !!!!!!!" &lt;&lt; endl;
   return 0;
}

void PTreeNodeCls :: print() {
   if (debugFlag) cout &lt;&lt; "PTreeNodeCls::print() ";
}
/* end-PTreeNodeCls */

PTreeCls :: PTreeCls(PPTreeNodeCls Root) {
   if (debugFlag) cout &lt;&lt; "PTreeCls()" &lt;&lt; endl;
   root = current = Root;
}
```

```
void PTreeCls :: print() {
   cout &lt;&lt; "PTreeCls " &lt;&lt; endl;
   root -&gt; print();
}
/* end-PTreeCls */


DataItemCls::DataItemCls(PPTreeNodeCls item, PPTreeNodeCls sub)
{
   this-&gt;item = item;
   this-&gt;sub = sub;

   // error checking would go here
   // lookup identifier
   // check if array iff sub
}

StatementCls :: StatementCls(char *StmtText) {
   if (debugFlag) cout &lt;&lt; "StatementCls()" &lt;&lt; endl;
   stmt_text = new char[strlen(StmtText)+1];
   strcpy(stmt_text,StmtText);
}

StatementSeqCls :: StatementSeqCls (PPTreeNodeCls Stmt) {
   if (debugFlag) cout &lt;&lt; "StatementSeqCls() " &lt;&lt; endl;
   seq_tail = seq_head = PStatementCls(Stmt);
}

PPTreeNodeCls StatementSeqCls :: append(PPTreeNodeCls Stmt) {
   if (debugFlag) cout &lt;&lt; "StatementSeqCls::append()" &lt;&lt; endl;
   if (!seq_tail) {
      cerr &lt;&lt; "StatementSeqCls::append() – LOGIC ERROR" &lt;&lt; endl;
   } else {
      seq_tail = PStatementCls(seq_tail -&gt;
       LstSeqBldrCls::append(PStatementCls(Stmt)));
   }
```

```
  return this;
}
/* end-StatementSeqCls */

AssignmentStmtCls :: AssignmentStmtCls(
      PPTreeNodeCls assignSeq,
      PPTreeNodeCls Expr,
      char* TextLine):
  StatementCls(TextLine) {
  if (debugFlag) cout &lt;&lt; "AssignmentStmtCls() " &lt;&lt; endl;
  if (!assignSeq || !Expr) {
    cerr &lt;&lt; "AssignmentStmtCls() - LOGIC ERROR " &lt;&lt; endl;
  }
  assign_seq = assignSeq;
  expr = Expr;
}


AssignmentSeqCls::AssignmentSeqCls(PPTreeNodeCls item)
{
  seq_head = seq_tail = PDataItemCls(item);
  length = 1;
}

PPTreeNodeCls AssignmentSeqCls::append(PPTreeNodeCls item)
{
  if (debugFlag) cout &lt;&lt; "AssignmentSeqCls::append()" &lt;&lt; endl;

  if (!seq_tail) {
    cerr &lt;&lt; "StatementSeqCls::append() – LOGIC ERROR" &lt;&lt; endl;
  } else {
    seq_tail = PDataItemCls(seq_tail-
&gt;LstSeqBldrCls::append(PDataItemCls(item)));
  }
  length++;
```

```
  return this;
}

int AssignmentSeqCls::getLength() {
  return length;
}

WriteStmtCls :: WriteStmtCls(PPTreeNodeCls Expr,
    char* TextLine):
  StatementCls(TextLine) {
  if (debugFlag) cout &lt;&lt; "WriteStmtCls() " &lt;&lt; endl;
  if (!Expr) {
    cerr &lt;&lt; "WriteStmtCls() - LOGIC ERROR " &lt;&lt; endl;
  }
  expr = Expr;
}
/* end-WriteStmtCls */

LoopStmtCls::LoopStmtCls(PPTreeNodeCls stmts) {
  statements = stmts;
}

IfStmtCls::IfStmtCls(PPTreeNodeCls condition, PPTreeNodeCls statements, PPTreeN-
odeCls elseStmts)
{
  this-&gt;condition = condition;
  this-&gt;statements = statements;
  this-&gt;elseStmts = elseStmts;
}

ElseStmtCls::ElseStmtCls(PPTreeNodeCls statements)
{
  this-&gt;statements = statements;
}

RelOpCls::RelOpCls(RelOp op) {
  this-&gt;op = op;
```

```
}

ConditionCls::ConditionCls(PPTreeNodeCls expr1, PPTreeNodeCls relOp, PPTreeN-
odeCls expr2)
{
  this-&gt;expr1 = expr1;
  this-&gt;relOp = relOp;
  this-&gt;expr2 = expr2;
}

ExprCls :: ExprCls() {
  if (debugFlag) cout &lt;&lt; "ExprCls" &lt;&lt; endl;
}

int ExprCls :: emit() {
  cout &lt;&lt; "ExprCls::emit()  BASE CLASS !!!!!!!" &lt;&lt; endl;
  return 0;
}

FactorCls :: FactorCls() {
  if (debugFlag) cout &lt;&lt; "FactorCls() " &lt;&lt; endl;
}
/* end-FactorCls */

NumFactorCls :: NumFactorCls(PPTreeNodeCls NumLit) {
  if (debugFlag) cout &lt;&lt; "NumFactorCls() " &lt;&lt; endl;
  if (!NumLit) {
    cerr &lt;&lt; "NumFactorCls - LOGIC ERROR" &lt;&lt; endl;
  }
  ExprCls::value = PNumLiteralCls(NumLit) -&gt; get_value();
  if (debugFlag) cout &lt;&lt; " value " &lt;&lt; ExprCls::value &lt;&lt; endl;
}

VarAccessFactorCls :: VarAccessFactorCls(PPTreeNodeCls Item) {
  if (debugFlag) cout &lt;&lt; "VarAccessFactorCls() " &lt;&lt; endl;
  if (!Item) {
    cerr &lt;&lt; "VarAccessFactorCls() - LOGIC ERROR" &lt;&lt; endl;
```

```
  }
  item = Item;
}
/* end-VarAccessFactorCls */

ArithmCls::ArithmCls(PPTreeNodeCls left, PPTreeNodeCls right, ArithmOp op)
{
  this-&gt;left = left;
  this-&gt;right = right;
  this-&gt;op = op;
}

NumLiteralCls :: NumLiteralCls() {
  if (debugFlag) cout &lt;&lt; "NumLiteralCls" &lt;&lt; endl;
  value = atoi(this -&gt; PTreeNodeCls::lt -&gt; get_lexeme());
}
/* end-NumLiteralCls */

IdentCls :: IdentCls() {
  if (debugFlag) cout &lt;&lt; "IdentCls" &lt;&lt; endl;
}

char *IdentCls :: get_name() {
  if (debugFlag) cout &lt;&lt; "IdentCls::get_name()" &lt;&lt; endl;
  return this -&gt; PTreeNodeCls::lt -&gt; get_lexeme();
}
/* end-IdentCls */

IdentSeqCls::IdentSeqCls(PPTreeNodeCls ident)
{
  if (debugFlag) cout &lt;&lt; "IdentSeqCls::IdentSeqCls()" &lt;&lt; endl;

  seq_head = seq_tail = PIdentCls(ident);
}

PPTreeNodeCls IdentSeqCls::append(PPTreeNodeCls ident)
{
```

```
  if (debugFlag) cout &lt;&lt; "DeclarationSeqCls::append()" &lt;&lt; endl;

  if (!seq_tail) {
    cerr &lt;&lt; "IdentSeqCls::append() – LOGIC ERROR" &lt;&lt; endl;
    exit(1);
  }
  else {
    seq_tail = PIdentCls(seq_tail-&gt;LstSeqBldrCls::append(PIdentCls(ident)));
  }

  return this;
}

PIdentCls IdentSeqCls::get_first() {
  return seq_head;
}

PIntTypeCls IntTypeCls::copy() {
  PIntTypeCls tmp = new IntTypeCls();

  return tmp;
}

ArrayTypeCls::ArrayTypeCls(PPTreeNodeCls rangeFrom, PPTreeNodeCls rangeTo,
  PPTreeNodeCls dataType)
{
  range_from = PNumLiteralCls(rangeFrom);
  range_to = PNumLiteralCls(rangeTo);
  data_type = PDataTypeCls(dataType);

  if (range_from-&gt;get_value() != 1) {
    cout &lt;&lt; "error: array must start at index 1" &lt;&lt; endl;
    exit(1);
  }

  if (range_to-&gt;get_value() &lt; range_from-&gt;get_value()) {
    cout &lt;&lt; "error: array size &lt;= 0" &lt;&lt; endl;
```

```
    exit(1);
  }
}

int ArrayTypeCls::getSize()
{
  return (range_to-&gt;get_value()-range_from-&gt;get_value()+1)*getItemSize();
}

int ArrayTypeCls::getItemSize()
{
  return PDataTypeCls(data_type)-&gt;getSize();
}

PArrayTypeCls ArrayTypeCls::copy()
{
  PArrayTypeCls tmp = new ArrayTypeCls(getRangeFrom(), getRangeTo(), NULL);
  tmp-&gt;data_type = data_type-&gt;copy();

  return tmp;
}

PNumLiteralCls ArrayTypeCls::getRangeFrom()
{
  return range_from;
}

PNumLiteralCls ArrayTypeCls::getRangeTo()
{
  return range_to;
}

PDataTypeCls ArrayTypeCls::getItemDataType()
{
  return data_type;
}
```

```
DeclarationCls::DeclarationCls(PPTreeNodeCls identSeq, PPTreeNodeCls dataType,
      char *declText)
{
   if (debugFlag) cout &lt;&lt; "DeclarationCls()" &lt;&lt; endl;

   ident_seq = PIdentSeqCls(identSeq);
   data_type = PDataTypeCls(dataType);
   decl_text = new char[strlen(declText)+1];
   strcpy(decl_text,declText);

   PSymtabCls scp = ScopeCls::get_vista();
   PIdentCls ident = ident_seq-&gt;get_first();

   while(ident) {
      char *name = ident-&gt;get_name();

      PSymtabEntryCls found_it = scp -&gt; lookup(name);
      if (!found_it) {
 PAddressCls address = new AddressCls();
 address-&gt;setAddr(MMCls::allocate(data_type-&gt;getSize()));

 PSymbolCls sym = new SymbolCls(name, data_type, address);
 scp-&gt;insert(sym);
      }
      else {
 cout &lt;&lt; "error: multiple declaration of symbol: " &lt;&lt; name &lt;&lt; endl;
 exit(1);
      }

      ident = PIdentCls(ident-&gt;get_next());
   }
}

DeclarationSeqCls::DeclarationSeqCls(PPTreeNodeCls decl)
{
   if (debugFlag) cout &lt;&lt; "DeclarationSeqCls()" &lt;&lt; endl;
   seq_tail = seq_head = PDeclarationCls(decl);
```

```
}

PPTreeNodeCls DeclarationSeqCls::append(PPTreeNodeCls decl)
{
  if (debugFlag) cout &lt;&lt; "DeclarationSeqCls::append()" &lt;&lt; endl;

  if (!seq_tail) {
    cerr &lt;&lt; "DeclarationSeqCls::append() – LOGIC ERROR" &lt;&lt; endl;
    exit(1);
  }
  else {
    seq_tail = PDeclarationCls(seq_tail-
&gt;LstSeqBldrCls::append(PDeclarationCls(decl)));
  }

  return this;
}

void AddressCls::setAddr(int a)
{
  addr=a;
}

int AddressCls::getAddr()
{
  return addr;
}

BlockCls :: BlockCls(PPTreeNodeCls StmtSeq) {
  if (debugFlag) cout &lt;&lt; "BlockCls" &lt;&lt; endl;
  stmt_seq = StmtSeq;
}
/* end-BlockCls */

extern PPTreeNodeCls prgm_node; //declared in scanparse.C
ProgramCls :: ProgramCls(PPTreeNodeCls Ident, PPTreeNodeCls Block) {
  if (debugFlag) cout &lt;&lt; "ProgramCls() " &lt;&lt; endl;
```

```
    ident = PIdentCls(Ident);
    block = PBlockCls(Block);
    std_table = ScopeCls::get_vista();
    prgm_node = this;
}

void ProgramCls::print() {
    cout &lt;&lt; "ProgramCls::print()" &lt;&lt; endl;
}
/* end-ProgramCls */
```

emit.C

```
/*
 * emit.C
 */

#include &lt;iostream.h&gt;

#include "symtab.h"

extern bool debugFlag;

int SymtabCls :: emit() {
    if (debugFlag) cout &lt;&lt; "SymtabCls::emit()" &lt;&lt; endl;

    cerr &lt;&lt; "Nothing to emit in symbol table!" &lt;&lt; endl;
    exit(1);

    return 0;
}

#include "p_tree.h"

int PTreeCls :: emit() {
    if (debugFlag) cout &lt;&lt; "PTreeCls::emit() " &lt;&lt; endl;
    if (! root) {
```

```
      cerr &lt;&lt; "PTreeCls::emit - LOGIC ERROR" &lt;&lt; endl;
      return 0;
   } else {
      return root -&gt; emit();
   }
}

#ifdef MULT_ARR
int DataItemCls::emit()
{
   typeEmit();

   return 0;
}

PDataTypeCls DataItemCls::typeEmit()
{
   if (debugFlag) cout &lt;&lt; "DataItemCls::typeEmit()" &lt;&lt; endl;

   if (dynamic_cast&lt;PIdentCls&gt;(item) != NULL) {  // basic type
      PSymtabCls scp = ScopeCls::get_vista();
      char *name = PIdentCls(item)-&gt;get_name();
      PSymtabEntryCls found_it = scp -&gt; lookup(name);
      if (!found_it) {
 cerr &lt;&lt; "error: undeclared variable: " &lt;&lt; name &lt;&lt; endl;
 exit(1);
      }

      PDataTypeCls type = PSymbolCls(found_it)-&gt;getDataType();
      PAddressCls addr = PSymbolCls(found_it)-&gt;getAddress();

      // Too many modifiers?
      if (type == NULL || sub != NULL) {
 cerr &lt;&lt; "Type and reference mismatch" &lt;&lt; endl;
 exit(1);
      }
```

```
      cout &lt;&lt; "push" &lt;&lt; endl
   &lt;&lt; addr-&gt;getAddr() &lt;&lt; endl;

      return type;
   }
   else {  // array type
      PDataTypeCls this_type = PDataItemCls(item)-&gt;typeEmit();

      // Too many modifiers?
      if (this_type == NULL || sub == NULL ||
  dynamic_cast&lt;PCompTypeCls&gt;(this_type) == NULL) {
 cerr &lt;&lt; "Type and reference mismatch" &lt;&lt; endl;
 exit(1);
      }

      sub-&gt;emit();
      cout &lt;&lt; "push" &lt;&lt; endl
   &lt;&lt; PArrayTypeCls(this_type)-&gt;getRangeFrom()-
&gt;get_value() &lt;&lt; endl
   &lt;&lt; "-" &lt;&lt; endl;

      cout &lt;&lt; "push" &lt;&lt; endl
   &lt;&lt; PArrayTypeCls(this_type)-&gt;getItemSize() &lt;&lt; endl
   &lt;&lt; "*" &lt;&lt; endl
   &lt;&lt; "+" &lt;&lt; endl;

      return PArrayTypeCls(this_type)-&gt;getItemDataType();
   }
}

#else /* !MULT_ARR */
int DataItemCls::emit()
{
   if (debugFlag) cout &lt;&lt; "DataItemCls::emit()" &lt;&lt; endl;

   PSymtabCls scp = ScopeCls::get_vista();
   char *name = PIdentCls(item)-&gt;get_name();
```

```
  PSymtabEntryCls found_it = scp -&gt; lookup(name);
  if (!found_it) {
    cerr &lt;&lt; "error: undeclared variable: " &lt;&lt; name &lt;&lt; endl;
    exit(1);
  }

  PDataTypeCls type = PSymbolCls(found_it)-&gt;getDataType();
  PAddressCls addr = PSymbolCls(found_it)-&gt;getAddress();
  cout &lt;&lt; "push" &lt;&lt; endl
&lt;&lt; addr-&gt;getAddr() &lt;&lt; endl;

  if (sub) {
    sub-&gt;emit();
    cout &lt;&lt; "push" &lt;&lt; endl
  &lt;&lt; PArrayTypeCls(type)-&gt;getRangeFrom()-&gt;get_value() &lt;&lt; endl
  &lt;&lt; "-" &lt;&lt; endl;

    cout &lt;&lt; "push" &lt;&lt; endl
  &lt;&lt; PArrayTypeCls(type)-&gt;getItemSize() &lt;&lt; endl
  &lt;&lt; "*" &lt;&lt; endl
  &lt;&lt; "+" &lt;&lt; endl;
  }

  return 0;
}
#endif /* MULT_ARR */

int StatementCls :: emit() {
  if (debugFlag) cout &lt;&lt; "! " &lt;&lt; stmt_text &lt;&lt; endl;
  return 0;
}

int StatementSeqCls :: emit() {
  if (debugFlag) cout &lt;&lt; "StatementSeqCls::emit() " &lt;&lt; endl;
  PPTreeNodeCls p = this -&gt; seq_head;
  while (p) {
    p -&gt; emit();
```

```
    p = PStatementCls(PStatementCls(p) -&gt; get_next());
  }
  return 0;
}

int EmptyStmtCls :: emit() {
  if (debugFlag) cout &lt;&lt; "EmptyStmtCls::emit()" &lt;&lt; endl;
  return 0;
}

int WriteStmtCls :: emit() {
  if (debugFlag) cout &lt;&lt; "WriteStmtCls::emit()" &lt;&lt; endl;
  StatementCls::emit();

  expr -&gt; emit();
  cout &lt;&lt; "write" &lt;&lt; endl;

  return 0;
}

int AssignmentStmtCls :: emit() {
  if (debugFlag) cout &lt;&lt; "AssignmentStmtCls::emit()" &lt;&lt; endl;
  StatementCls::emit();


  assign_seq-&gt;emit();
  expr-&gt;emit();

  for (int i=0; i&lt;PAssignmentSeqCls(assign_seq)-&gt;getLength(); i++)
    cout &lt;&lt; ":=" &lt;&lt; endl;

  cout &lt;&lt; "pop" &lt;&lt; endl;

  return 0;
}

int AssignmentSeqCls::emit() {
```

```
    if (debugFlag) cout &lt;&lt; "AssignmentSeqCls::emit() " &lt;&lt; endl;
    PPTreeNodeCls p = this-&gt;seq_head;
    while (p) {
        p -&gt; emit();
        p = PDataItemCls(PDataItemCls(p)-&gt;get_next());
    }

    return 0;
}

int LoopStmtCls::emit() {
    if (debugFlag) cout &lt;&lt; "LoopStmtCls::emit()" &lt;&lt; endl;

    cout &lt;&lt; "loop" &lt;&lt; endl;
    statements-&gt;emit();
    cout &lt;&lt; "endloop" &lt;&lt; endl;

    return 0;
}

int ExitStmtCls::emit() {
    if (debugFlag) cout &lt;&lt; "ExitStmtCls::emit()" &lt;&lt; endl;

    cout &lt;&lt; "exit" &lt;&lt; endl;

    return 0;
}

int IfStmtCls::emit()
{
    if (debugFlag) cout &lt;&lt; "IfStmtCls::emit()" &lt;&lt; endl;

    condition-&gt;emit();
    cout &lt;&lt; "test" &lt;&lt; endl;

    statements-&gt;emit();
    elseStmts-&gt;emit();
```

```
  cout &lt;&lt; "endif" &lt;&lt; endl;

  return 0;
}

int ElseStmtCls::emit()
{
  if (debugFlag) cout &lt;&lt; "ElseStmtCls::emit()" &lt;&lt; endl;

  if (statements) {
    cout &lt;&lt; "else" &lt;&lt; endl;
    statements-&gt;emit();
  }

  return 0;
}

int RelOpCls::emit()
{
  if (debugFlag) cout &lt;&lt; "RelOpCls::emit()" &lt;&lt; endl;

  switch(op) {
  case ltOp:
    cout &lt;&lt; "&lt;" &lt;&lt; endl;
    break;

  case leOp:
    cout &lt;&lt; "&lt;=" &lt;&lt; endl;
    break;

  case eqOp:
    cout &lt;&lt; "=" &lt;&lt; endl;
    break;

  case neOp:
    cout &lt;&lt; "&lt;&gt;" &lt;&lt; endl;
```

```
    break;

  case geOp:
    cout &lt;&lt; "&gt;=" &lt;&lt; endl;
    break;

  case gtOp:
    cout &lt;&lt; "&gt;" &lt;&lt; endl;
    break;

  default:
    cerr &lt;&lt; "error: unsupported relational operator: " &lt;&lt; op &lt;&lt; endl;
    exit(1);
  }

  return 0;
}

int ConditionCls::emit()
{
  if (debugFlag) cout &lt;&lt; "ConditionCls::emit()" &lt;&lt; endl;

  expr1-&gt;emit();
  expr2-&gt;emit();
  relOp-&gt;emit();

  return 0;
}

int NumFactorCls :: emit() {
  if (debugFlag) cout &lt;&lt; "NumFactorCls::emit()" &lt;&lt; endl;
  cout &lt;&lt; "push" &lt;&lt; endl
&lt;&lt; ExprCls::value &lt;&lt; endl;

  return 0;
}
```

```
int VarAccessFactorCls :: emit() {
  if (debugFlag) cout &lt;&lt; "VarAccessFactorCls::emit()" &lt;&lt; endl;

  item-&gt;emit();
  cout &lt;&lt; "getValue" &lt;&lt; endl;

  return 0;
}

int ArithmCls::emit()
{
  left-&gt;emit();
  right-&gt;emit();

  switch (op) {
  case PlusOp:
    cout &lt;&lt; "+" &lt;&lt; endl;
    break;

  case MinusOp:
    cout &lt;&lt; "-" &lt;&lt; endl;
    break;

  case MultOp:
    cout &lt;&lt; "*" &lt;&lt; endl;
    break;

  case DivOp:
    cout &lt;&lt; "/" &lt;&lt; endl;
    break;

  default:
    cerr &lt;&lt; "error: unsupported arithmetic operator: " &lt;&lt; op &lt;&lt; endl;
    exit(1);
  }

  return 0;
```

```
}

int IdentSeqCls::emit()
{
  return 0;
}

int DeclarationCls::emit()
{
  return 0;
}

int DeclarationSeqCls::emit()
{
  if (debugFlag) cout &lt;&lt; "DeclarationSeqCls::emit()" &lt;&lt; endl;
  PDeclarationCls p = this -&gt; seq_head;
  while (p) {
    p -&gt; emit();
    p = PDeclarationCls(p-&gt;get_next());
  }

  return 0;
}

int BlockCls :: emit() {
  if (debugFlag) cout &lt;&lt; "BlockCls::emit() " &lt;&lt; endl;
  return this -&gt; stmt_seq -&gt; emit();
}

int ProgramCls :: emit() {
  if (debugFlag) cout &lt;&lt; "ProgramCls::emit() " &lt;&lt; endl;
  if (block && std_table) {
    this -&gt; block -&gt; emit();

    cout &lt;&lt; "finish" &lt;&lt; endl;

    return 0;
```

```
  } else {
    cerr &lt;&lt; "ProgramCls::emit - LOGIC ERROR" &lt;&lt; endl;
    return 0;
  }
}
```

References

[1] Building Your Own Compiler With C++, Jim Holmes, 1995, 112 pages, 0-131-82106-7, Prentice Hall.