Overall Compiler Structure - Stage 0

Following is the overall framework for stage0, the first stage of the Pascallite compiler, including the main routine and the interfaces between that routine and its major components. All of the stages are organized as a translation grammar processor. The grammar given below is an LL(1) grammar so that the processor can generate a leftmost derivation of programs without backtracking. The grammar given below includes the *action* symbols needed to build the symbol table (with the selection sets omitted).

Pascallite Grammar Stage 0

1.	PROG	→	PROG_STMT CONSTS VARS BEGIN_END_STMT
2.	PROG_STMT	→	'program' $NON\_KEY\_ID_x$ ';'
			Insert(x,PROG_NAME,CONSTANT,x,NO,0)
3.	CONSTS	→	'const' CONST_STMTS
		→	ε
4.	VARS	→	'var' VAR_STMTS
		→	ε
5.	BEGIN_END_STMT	→	'begin' 'end' '.'
6.	CONST_STMTS	→	$NON\_KEY\_ID_x$ '=' ($NON\_KEY\_ID_y$ | LIT_y) ';'
			Insert(x,WhichType(y),CONSTANT,WhichValue(y),YES,1)
			(CONST_STMTS | ε)
7.	VAR_STMTS	→	IDS_x ':' $TYPE_y$ ';'
			Insert(x,y,VARIABLE,ε,YES,1)
			(VAR_STMTS | ε)
8.	IDS	→	NON_KEY_ID (',' IDS | ε)
9.	TYPE	→	'integer'
		→	'boolean'
10.	LIT	→	INTEGER | BOOLEAN | 'not' BOOLEAN | '+' INTEGER | '-' INTEGER
11.	BOOLEAN	→	'true' | 'false'

Note that in production 6, subscript *y* is used twice. This does not contradict restriction 7 on the form of valid productions of a translation grammar, however, since *y* is subscripting alternatives. Production 6 is actually an abbreviation for *two* productions; therefore *y* can never be the value of both alternatives simultaneously and no conflict can arise.

There are just three action routines called in this simple translation grammar:

1. `Insert(externalName,storeType,mode,value,allocate,units)`
2. `WhichType(externalName)`
3. `WhichValue(externalName)`

These routines are explained further in the following pages.

Following is the pseudo code for the main program for stage0. It is extremely simple, reflecting the fact that most of the actual processing is performed by the parser. The symbol table is defined because this data structure is so pervasively referenced throughout the compiler. To reference an entry in the table for the external name 'trivia', the pseudo code simply writes `symbolTable['trivia']`. If there is no entry under that index, then the value referenced is *undefined*. The detail of how the look-up of entries is handled is left to the programmer.

Pascallite Stage 0

```
const int MAX_SYMBOL_TABLE_SIZE = 256;
enum storeType {INTEGER, BOOLEAN, PROG_NAME};
enum allocation {YES,NO};
enum modes {VARIABLE, CONSTANT};
struct entry  //define symbol table entry format
{
  string internalName;
  string externalName;
  storeType dataType;
  modes mode;
  string value;
  allocation alloc;
  int units;
}
vector<entry> symbolTable;
ifstream sourceFile;
ofstream listingFile,objectFile;
string token;
char charac;
const char END_OF_FILE = '$'; // arbitrary choice

int main(int argc, char **argv)
{
  //this program is the stage0 compiler for Pascallite.  It will accept
  //input from argv[1], generating a listing to argv[2], and object code to
  //argv[3]

  CreateListingHeader();
  Parser();
  CreateListingTrailer();

  PrintSymbolTable();

  return 0;
}
```

Functions called from `main()`

```
void CreateListingHeader()
{
  print "STAGE0:", names, DATE, TIME OF DAY;
  print "LINE NO:", "SOURCE STATEMENT";
    //line numbers and source statements should be aligned under the headings
}

void Parser()
{
  NextChar();
      //charac must be initialized to the first character of the source file
  if(NextToken() != "program")
    process error: keyword "program" expected;
      //a call to NextToken() has two effects
      //  (1) the variable, token, is assigned the value of the next token
      //  (2) the next token is read from the source file in order to make
      //      the assignment.  The value returned by NextToken() is also
      //      the next token.
  Prog();
      //parser implements the grammar rules, calling first rule
}

void CreateListingTrailer()
{
  print "COMPILATION TERMINATED", "# ERRORS ENCOUNTERED";
}

void PrintSymbolTable()
{
  print symbol table to object file
}
```

Grammar Rules

`Prog()` - production 1

```
void Prog()  //token should be "program"
{
  if (token != "program")
     process error: keyword "program" expected
  ProgStmt();
  if (token == "const") Consts();
  if (token == "var") Vars();
  if (token != "begin")
    process error: keyword "begin" expected
  BeginEndStmt();
  if (token != END_OF_FILE)
    process error: no text may follow "end"
}
```

`ProgStmt()` - production 2

```
void ProgStmt()  //token should be "program"
{
  string x;
  if (token != "program")
    process error: keyword "program" expected
  x = NextToken();
  if (token != NON_KEY_ID)
    process error: program name expected
  if (NextToken() != ";")
    process error: semicolon expected
  NextToken();
  Insert(x,PROG_NAME,CONSTANT,x,NO,0);
}
```

`Consts()` - production 3

```
void Consts()  //token should be "const"
{
  if (token != "const")
     process error: keyword "const" expected
  if (NextToken() != NON_KEY_ID)
    process error: non-keyword identifier must follow "const"
  ConstStmts();
}
```

`Vars()` - production 4

```
void Vars()  //token should be "var"
{
  if (token != "var")
     process error: keyword "var" expected
  if (NextToken() != NON_KEY_ID)
    process error: non-keyword identifier must follow "var"
  VarStmts();
}
```

`BeginEndStmt()` - production 5

```
void BeginEndStmt()  //token should be "begin"
{
  if (token != "begin")
     process error: keyword "begin" expected
  if (NextToken() != "end")
     process error: keyword "end" expected
  if (NextToken() != ".")
    process error: period expected
  NextToken();
}
```

`ConstStmts()` - production 6

```
void ConstStmts()  //token should be NON_KEY_ID
{
  string x,y;
  if (token != NON_KEY_ID)
     process error: non-keyword identifier expected
  x = token;
  if (NextToken() != "=")
     process error: "=" expected
  y = NextToken();
  if (y != "+","-","not",NON_KEY_ID,"true","false",INTEGER)
    process error: token to right of "=" illegal
  if (y == "+","-")
  {
    if(NextToken() != INTEGER)
      process error: integer expected after sign
    y = y + token;
  }
  if (y == "not")
  {
    if(NextToken() != BOOLEAN)
      process error: boolean expected after not
    if(token == "true")
      y = "false"
    else
      y = "true";
  }
  if (NextToken() != ";")
      process error: semicolon expected
  Insert(x,WhichType(y),CONSTANT,WhichValue(y),YES,1);
  if (NextToken() != "begin","var",NON_KEY_ID)
      process error: non-keyword identifier,"begin", or "var" expected
  if (NextToken() == NON_KEY_ID)
    ConstStmts();
}
```

`VarStmts()` - production 7

```
void VarStmts()  //token should be NON_KEY_ID
{
  string x,y;
  if (token != NON_KEY_ID)
     process error: non-keyword identifier expected
  x = Ids();
  if (token != ":")
     process error: ":" expected
  if(NextToken() != "integer","boolean")
      process error: illegal type follows ":"
  y = token;
  if(NextToken() != ";")
      process error: semicolon expected
  Insert(x,y,VARIABLE,"",YES,1);
  if (NextToken() != "begin",NON_KEY_ID)
      process error: non-keyword identifier or "begin" expected
  if (token != NON_KEY_ID)
    VarStmts();
}
```

`Ids()` - production 8

```
string Ids()  //token should be NON_KEY_ID
{
  string temp,tempString;
  if (token != NON_KEY_ID)
     process error: non-keyword identifier expected
  tempString = token;
  temp = token;
  if(NextToken() == ",")
  {
    if (NextToken() != NON_KEY_ID)
      process error: non-keyword identifier expected
    tempString = temp + "," + Ids();
  }
  return tempString;
}
```

Parser

Starting in `main()` in the parser, the action calls have been inserted into the productions. In the coding, the art of "defensive" programming is practiced. In particular, each parser routine expects the current token to be among a certain set of values when that routine is called. If the parser is performing properly (i.e., has no bugs), then each routine's input will be what it should be. However if there are any errors in the compiler, a routine could be called under improper conditions; e.g., `Prog()` could be called with the current token something other than "`program`". Such an erroneous call could propagate errors indefinitely through any number of other routines until it were caught (if at all). Rather than assume the compiler is correct, you should presume it might very well have bugs and test whether each parser routine is being called under the right circumstances. If not, an error processing routine is called to handle the problem, otherwise, compilation continues unabated. The price paid for this additional check is the added cost to test the value of the current token against the set of expected tokens, a small price to pay during development of the additional error detection capability. If stage0 were installed as a working compiler, the compiler implementor could choose to remove these additional checks prior to installation if he felt the performance would be unduly limited by their inclusion.

Action Routines

Insert()

`Insert()` creates entries in the symbol table. It has six arguments:

1. a list of external names
2. the type of the list members
3. the mode of the list members
4. the value of the list members
5. whether or not storage will be emitted
6. the number of storage units to be emitted (if any)

Note that `Insert()` calls `GenInternalName()`, a function that has one argument, the type of the name being inserted. `GenInternalName()` returns a unique internal name each time it is called, a name that is known to be a valid symbolic name. As a visual aid, we use different forms of internal names for each data-type of interest. The general form is:

`dn`

where `d` denotes the data-type of the name ("`I`" for *integer*, "`B`" for *boolean*) and `n` is a non-negative integer starting at `0`. The generated source code for `001.dat` clearly shows the effects of calling `GenInternalName()`. The compiler itself will also need to generate names to appear in the object code, but since the compiler is defining these itself, there is no need to convert these names into any other form. The external and internal forms will be the same. The code for `Insert()` treats any external name beginning with an uppercase character as defined by the compiler.

```
void Insert(string externalName,storeType inType, modes inMode, string inValue,
            allocation inAlloc, int inUnits)
    //create symbol table entry for each identifier in list of external names
    //Multiply inserted names are illegal
{
  string name;
  while (name broken from list of external names and put into name != "")
  {
    if symbolTable[name] is defined
      process error: multiple name definition
    else if name is a keyword
      process error: illegal use of keyword
    else  //create table entry
    {
      if name begins with uppercase then
        symbolTable[name]=(name,inType,inMode,inValue,inAlloc,inUnits)
      else
        symbolTable[name]=(GenInternalName(inType),inType,inMode,inValue,
                           inAlloc,inUnits)
    }
}
```

WhichType(), WhichValue()

```
storeType WhichType(string name)        //tells which data type a name has
{
  if name is a literal
    if name is a boolean literal then data type = BOOLEAN
    else data type = INTEGER
  else      //name is an identifier and hopefully a constant
    if symbolTable[name] is defined then data type = type of symbolTable[name]
    else process error:  reference to undefined constant
  return data type;
}

string WhichValue(string name)        //tells which value a name has
{
  if name is a literal
    value = name
  else      //name is an identifier and hopefully a constant
    if symbolTable[name] is defined and has a value
      value = value of symbolTable[name]
    else
      process error:  reference to undefined constant
  return value;
}
```

Lexical Scanner

The lexical scanner, `NextToken()`, of stage0 is referenced repeatedly in functions which define the parser. `NextToken()` is a function which always returns the next token; in addition, it always assigns the value it returns to the variable `token`, so that the value is easily referenced after the call is completed. The scanner itself calls a routine which returns characters to it, called `NextChar()`; `NextChar()` also assigns the value it returns to a variable for each referencing, `charac`. `NextChar()` can be used to print the listing file as well as returning the current character to `NextToken()`.

NextToken(), NextChar()

```
string NextToken()        //returns the next token or end of file marker
{
  token = "";
  while (token == "")
  {
    switch(charac)
    {
      case '{'              : //process comment
                              while (NextChar() != END_OF_FILE || '}'};
                              if (charac==END_OF_FILE)
                                process error:  unexpected end of file
                              else
                                NextChar();

      case '}'              : process error:  '}' cannot begin token

      case whitespace       : NextChar();

      case special character : token = charac;
                              NextChar();

      case letter           : token = charac;
                              while (NextChar() == letter or digit or '_')
                                token+=charac;
                              if token ends in '_'
                                process error:  '_' cannot end token

      case digit            : token = charac;
                              while (NextChar() == digit) token+=charac;

      case END_OF_FILE      : token = charac;

      default               : process error:  illegal symbol
  }
  return token;
}

char NextChar()   //returns the next character or end of file marker
{
  read in next character
  if end of file
    charac = END_OF_FILE     //use a special character to designate end of file
  else
    charac = next character
  print to listing file (starting new line if necessary)
  return charac;
}
```