1 Overview

Generally speaking, the process of compiling can be devided into two parts. The first part reduce the source program into a intermediat representation; The second part translate the intermediat representation to machine dependent code. They are called as front-end and back-end seperately.

The HPF front-end is a front-end which takes HPF program as input and give intermediate representaion of such program as output. In this paper we will introduce technical considerations during our implementation of HPF front-end.

The project is divided into two parts,

- A parser for HPF language, which includes a full set of FORTRAN 90.
- A set of simple utility tools.

The parser is composed of a lexical analyzer, a grammar analyzer and basic functions to establish intermediate representations of source code. The utilities includes displaying these intermediate representations and translating them back to source program.

In the following sections, we will first introduce the background of our project, and then address how the above two parts were implemented in our front-end. We provide interface among deferent parts of the parser.

The intermediate representation serves as an interface between the two parts of the front-end, it is the most important data structure used in our project. So we also give as appendix the template used to represent statements and expressions of the source program.

2 HPF Language Parser

Before we go into details of implementing methods, let's compare several possible ways of constructing a parser.

2.1 The Main Implementation Techniques for Parser

To build a parser for computer language, we may use the following three methods.

We can program it totally in a certain high-level language (for example, C), which includes lexical, syntactic analysis and intermediate representation generation. This can improve the efficiency of code execution and the speed of compiling.

Nevertheless, given a current computer language, its syntax is so manyfolded and complex that it is almost impossible to set up syntax analyzer in high-level language directly. So no vendor adopts this way to analyze HPF syntax to our knowledge. Comparatively, the lexical analysis is simple and easy. It is quite possible to be programmed in C language, like it was done in Sage++.

The second means is to make use of UNIX application programs, such as Lex (lexical analysis program generator)[1] and Yacc (yet another compiler-compiler)[2]. After a long time of practice by UNIX user, they are regarded as the mature and reliable compiling tools with good performance. It is widely accepted in both academic and industrial world. For example, the popular Perl language's compiler is generated by Yacc.

On the other hand the program generated from Lex and Yacc may contain certain redundant codes, which will decrease the efficiency of the program. This is the inevitable cost caused by reducing developers' work load.

The third methods we know is a even higher integrated compiler tool, called "Eli", published by the Compiler Tools Group in Colorado University in 1994[3].

Eli is a tool used in developing compiler front-end specially. It's mechanism of implementation is similar to the above methods: once provided lexical and syntactic rules, it will generate analyzer automatically. More over, it will allow users to provide the structure of their intermediate representation and related definitions, then integrate them all together as a parser in C language.

In a word, Eli packs and modularizes the procedures further, which allow programmer concentrate on storing the information in a given program only. Compared with Lex and Yacc, Eli makes things more convenient. Maybe it will become a widely used compiler tool in the near future.

Comparing the above three methods, obviously the third one will greatly easy the whole task. But considering that Eli is a new product which is still in developing, we adopt the more mature techniques – Lex and Yacc instead of Eli.

Therefor the implementation process of our front-end parser can be divided into four steps:

- The generation of lexical analyzing program
- The generation of syntactic analyzing program
- Constructing intermediate representation
- Sementics checking

The following subsections introduce these steps in detail.

2.2 Lexical Analysis

The lexical analysis of HPF is divided into four parts:

- Recognizing label
- Recognizing special characters
- Recognizing identifier
- Recognizing digit

They are treated separately in our front-end.

i. Label

In the lexical analyzing label is recognized in two situations: one is at the head of a line, which is matched by the following regular expression:

```
^[ \t]*[0-9]{1,5}[ \t]+  { ......
  return(LABEL);
}
```

the other is the label located in the statement, for example,

```
goto 100
```

where the label 100 is recognized in the form of digit-string and reduced into label according to rules in the syntax analyzing.

ii. Special characters

The special characters of HPF includes operational sign, parenthesis, separating character and various delimiter. For every special character there is a Lex rule matching it. For example,

```
")" {  return(RIGHTBRACKET);
  }

"+" {  return(PLUS);
  }
```

iii. Identifier

The character-string beginnings with alphabets composed by the alphabets, digits and underscores is called identifier. It includes HPF keywords, internal function name and variable name which are all matched by identifier, that is {`NAME`} in lexical analysis. The procedure **`keynameDeal_m()`** distinguishes one from the other by looking up table `KEY_WORD_G` and `FUNC_TAB_G` .

iv. Digit

Besides the preceding items integer constant, real constant, double precision constant etc. are all recognized by regular expressions in Lex. For example,

```
{RealLiteralConstant}  {
   yylval.charp=copys(yytext);
   return(REAL_LITERAL_CONST);
}
```

```
{DoubleLiteralConstant}  {
   yylval.charp=copys(yytext);
   return(DOUBLE_LITERAL_CONST);
}
```

Conflicts may occur in Lex rule-set. For example, the I in format statement of HPF represents the input (output) format of a integer,whereas it is expected to be used as ordinary variable in other statements, so conflict comes out. It is desirable to have several sets of lexical rules to be applied at different time in the input. There are mainly two methods to cope with this case:

i. Using a flag

This may be a flag explicitly tested by the programmer's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all. For example in the common statement of Fortran 90, when common area name is omitted between the two virgule, it forms an unnamed common area, such as:

```
common //m,q(3,3),//a,b ;
```

while symbol // also represents connecting string symbol, for example "ab"//"cd". The rules using flag to distinct them are :

```
{NAME}  {
  ... ...
  case 3: common=1;  /* common */
  break;
  ( which is in function keynameDeal_m( ). )
......
}

"//"    {
  if(common==1) {
    yyless(yyleng-1);
    return(DIVIDE);
  }
  else
    return(CONCAT);
 }
```

ii. Using initial conditions

It is to have Lex not user remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It is only be recognized when Lex is in that start condition. To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

```
%start name1 name2...
```

The word `start` may be abbreviated to `s` or `S`. The rule `<name1>` *expression* represents the expression is only recognized when Lex is in the start condition `name1`. Executing the action statement

```
BEGIN name?;
```

will change the start condition to name?.

`BEGIN 0;` will reset the Lex automation interpreter to the initial condition. A rule may be active in several start conditions. So, `<name1,name2,name3>` is a legal prefix, and the relationship among them is `or`. Any rule not beginning with the `< >` prefix operator is always active.

Considering HPF supports free format and fixed format, our front-end actuates the process of fixed format by command option `-fixed`, which is implemented in lexical analyzing.

2.3 Syntactic analysis

When using Yacc to construct syntactic analyzing program, there are two important things: first, we should contain all the syntactic rule-set in the grammars of this language integrately; second, we have to try to avoid or reduce the conflicts between rules as possible as we can.

HPF[4] extends the data distribute and data paralleling statements on the base of Fortran 77 and Fortran 90[5]. It's grammar is so enormous that it is an extremely difficult task to write the Yacc source program. In our front-end, the grammars are based on the HPF1.0 syntactic rules published by HPPF in 1993. They are divided into five parts:

- basic frame of a program unit
- specification statements
- executable statements
- data distribute statements
- basic data type

The aim of dividing into five parts is to make the syntactic rules clear as well as make a basement for the next step – constructing AST.

Because of the ambiguity lying between the rules, Yacc detects and reports two kinds of conflicts: shift/reduce conflict and reduce/reduce conflict. Handling conflicts is a knotty problem faced by the developers of generating large analyzing program with Yacc.

In general there are three basic treating methods:

i. Using precedence and associativity

Programmer may declare the precedence and associativity of operators. They are attached to tokens in the declarations section. This is done by a series of lines beginning with a yacc keyword: `%left`, `%right`, or `%nonassoc`, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. The keyword `%prec` can change the precedence level associated with a particular grammar rule.

This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedence and associativities. However, this method is very limited: on one hand, because the declarations, `%right`, `%prec`, etc., are associated with special terminal symbols usually, they are suitable in the parsing of arithmetic expressions, while for the conflicts caused by the other rules it is powerless; On the other hand, the conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar, while they are unknown to user. So it is a good idea to be sparing with precedences, which is the suggestion in many Yacc manuals.

ii. Rewriting syntactic rules

This is the second method that can remove the conflicts with same input. Of course rewriting only involves the syntactic rules causing conflicts and related. In general this is a small part of the whole grammars. By the practice of constructing HPF s syntactic rule-set the author regards it as the most reliable method. The rewrite syntax ensures that the parser is actually doing what was intended. The basic foundation of rewriting is the principle of "look ahead one symbol" in Yacc, according to which the conflicts between the rules are removed. Using this way the author decrease the original 6272 reduce/reduce conflicts, 158 shift/reduce conflicts to present 25 shift/reduce conflicts and ensured that the input set recognized by the new syntax rule-set is same with the original.

iii. Depending on default rules provided by Yacc

It is not removing conflicts by user's intervene but approving the conflicts. Here, Yacc invokes two disambiguating rules by default:

- In a shift/reduce conflict, the default is to do the shift;
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule(in the input sequence).

With the two default rules whatever conflicts exist Yacc still generates an analyzing program. However, the reduce/reduce conflicts handled by the second rule will cause serious error in syntactic analyzing which may reduce an input sequence to an in correct syntax rule. For this reason the second default rule is not approved by the Yacc users. While the rule is often feasible to shift/reduce conflicts. The existence of this kind of conflict carefully checked by user doesn't affect the correctness of syntactic analysis. Being tested by a suite of examples the left six shift/reduce conflicts following the first default rule don't affect the validity of analyzing program in our front-end.

Another extremely difficult and important area in syntactic analysis is error handing. Yacc provides a simple, but reasonably general being able to recover the analyzing program, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. Such as

```
specification...
   : specification cmnt
   | specification... spcification cmnt
   | error {
     errResynch ( ) ;
   }
   ;
```

Here, when there is an error the parser attempts to skip over the statement. At the same time this rule will be reduced and any error handing action associated with it performed. In general it is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

In Function

```
errResynch( )  {
   throw_away();
   yyerrok;
   yyclearin;
   }
```

`throw_away( )` is the basic resynchronous function in which when the error occurs all tokens after the error and before the next carriage are discarded to input from the next legal statement.

The function `yyerrok` be used to force the parser to believe that an error has been fully recovered from and resets the parser to its normal mode.

The function `yyclearin` find the correct place to resume input and cleared the previous lookahead token.

The followings error-handing program is called automatically by Yacc:

```
yyerror(str)
   char *str;
   {
       globeErr_g=1;
       printf("line(%d) syntax error:%d %s\n", lineNum_g,
       yychar, lxyError_m(yychar) );
   }
```

The external integer variable yychar contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Function lxyError_m( ) change the token number to correspondent word (the table is listed in file err.file) and return a character-string. So yyerror reports the concrete information of error diagnostic: line number and error attribute.

2.4 Constructing AST

The intermediate representation selected by our front-end should reflect all information in a source program. So it is very important to define the data structure of the intermediate representation. Here we adopted the internal definition of Sage++ and made some extensions and modifications.

Sage++ is upgraded from Sigma in 1994. It establishes an AST after analyzing the source program of pC++ and Fortran , then stores it into the file whose name is same as the source file but with the extent name .dep. It also provides some auxiliary libs for the interface to back-end.

As to the concrete defination, there are descriptions in documents HPFfe: a Front-end for HPF and HPFfe: User's Guide.

Here we only address how AST is built up on these data structures. The abstract syntax tree (AST) of a HPF program is constructed by inserting functions, such as makenode(), behind the relative rule in Yacc source file. When the source code matches certain rule, functions behind this rule will be executed and build some nodes in the tree.So these routines are important to construct AST. According to the classification of nodes, the routines are divided into five groups: building statement list(BFND list); building expression list(LLND list); building symbol table; building type list; building lable list.

The following are some basic routines in each group:

- Building statement list

```
PTR_BFND makeBfnd(PTR_FILE fi, int node_type)
  Make a new statement node;
PTR_BFND setBfndList(PTR_BFND old_list, PTR_BFND stat)
  Link together an old BIF node list with a new one;
```

- Building expresion list

```
PTR_LLND makeLlnd(PTR_FILE fi, int node_type,PTR_LLND ll1,PTR_LLND ll2,
PTR_SYMB symb_ptr)
  Make a new expresion node;
PTR_LLND set_ll_list(PTR_LLND old_list,PTR_LLND node,int ll_type)
  Attache a new node to a given old_list and make sure that the old_list
  is of the same type as ll_type;
```

- Building symbol table

```
PTR_SYMB lookupSymb(char *string, PTR_SYMB symb)
  Look up a symbol called *string from "symb" in the symbol table;
PTR_SYMB makeSymb (PTR_FILE fi,PTR_BFND scope,int node_type,char *string,
int decl)
  Make a new symbol node in the symbol table if it doesn't exist,
  otherwise return the location of the node in the ST (it calls lookupSymb());
PTR_SYMB makeSymbVar(PTR_FILE fi,PTR_BFND scope,int local, char *string,
int decl)
  Make a symbol node as a variable (it calls makeSymb());
PTR_SYMB set_id_list(PTR_SYMB old_list,PTR_SYMB id)
  Take an old id list and run down the list attaching new id at the end;
```

- Building type list

```
PTR_TYPE make_type (PTR_FILE fi, int node_type)
  Make a new type node;
```

- Building lable list

```
PTR_LABEL make_label (PTR_FILE fi,long l)
  Make a new label node for Fortran;
PTR_LLND make_llnd_label (PTR_FILE fi,int node_type,PTR_LABEL lab)
  Make a new low level node for label;
```

The whole AST is constructed on the base of these series of routines.

2.5 Sementics Checking

Sementics checking aims at getting the comprehensive AST in sementics by appending and fixing the information in AST and ST genetated in the syntacis analysis and check if the HPF source program is correct in sementics. This procedure is static.

Basic concern in sementics checking is:

- Whether a symbol is refered legally according to its scope and attribute;

- Whether a variable is refered legally according to its scope, attribute and type;
- Whether a lable is refered legally according its scope.

Due to the usage of module, interface and contained subroutine in HPF,there are some knotty problems in sementics checking. For example, in the case of module, a memory unit may be shared by several routines, at the same time USE statment allows rename of the shared variable in the current routine and makes some limitation on shared entities in the module. So the management of the ST is complex in this case.

In syntactic analysis we employ stack to identify the scope of a symbol. At the beginning of a program unit, the head-statment is pushed into the stack. It will not be popped until the end of the program unit. This statement in stack indicates the scope of current symbols.But once module occurs, we would have to do some special operations:

- For statements as

  ```
  USE module_name,ONLY: [local_name=>]use_name,
  ```

  a new symbol, whose type and attribute inherits use_name and whose name is "use_name", is inserted in the ST within current scope. If the use_name is renamed in USE statement with local_name, it will get the new name "local_name"; otherwise it will be "use_name".

- For statements as

  ```
  USE module_name [local_name=>use_name],
  ```

  we need push the head-statement of the module "module_name". Each module would have a use_list which indicates the modules which are used, so we trace the use_list and push all the mudules. If some variables are renamed, the new symbols need to be added in ST as the above way. Thus, when the end statement comes, pop the stack until the first non-module head-statement.

Sementics Checking supports all internal functions and procedures in HPF 1.0. These routines are stored in file "intrinsic.f" as the liberary fuctions. When certain internal funtion is called, sementics checking finds the its interface name in file "intrinsic.f" and locate at the original with proper number and type of arguments, at the same time the matching between dummy arguments and actual arguments is checked.

Due to the employment of module in HPF Specification, it is important to keep the integration of a program in the condition of not recompiling the whole program. Dependent compiling is adopted to support the partitional compiling technique. That is, if the source program f1 uses module m, while m is defined in program f2, f1 is dependent on f2. So when compiler does sementic checking for program f1, it would check f2 first and address the dependent relationship between f1 and f2.

In a word sementic checking enforces the AST and ST and returns enough information to the programmer to correct his program.

3 Utility Tools

Like Sage++, we allow the generated intermediate representations to be written onto disk file for future use. So we also provide simple utility tools for user to check the generated intermediate representations of a source program.

There are two utilities avalible to serve this purpose. One is similar with `dumpdep` in Sage++, which displays the intermediate representations in a plain text mode.The other one is used to *unparse* the intermediate code back to the source program.

3.1 Display the Intermediate Representation

To show AST to users in a readable format, we develop a tool displaying the IR– dumpdep. Dumpdep displays the contents of the AST in the form of some two dimentional tabels, such as statement table, expression table and so on and show the relationship among the data in these tables. As to how to read the output of dumpdep, please refer to "HPFfe: User's Guide". Here we just introduce the implementation of dumpdep briefly.

Firstly the intermediate representation is recovered to the memory from the disk file, then some conversion is done on the AST and results are written into a new readable file. Though there are different tables in the AST, their methods of convertion are similar. For instance, in the case of BFND table, we access the node from the beginning along the thread till a null node. Thus we get all BIF nodes. Then we convert the fields with different types in each node into the readable form.

- For fields with type int and char, just show their value;
- For fields with type node pointer, if the pointer is null, convert it's value to b e -1; otherwise, show it's node ID. To distinguish different node pointer, a flag is added after the ID number. "-B" indicates BIF (statement) node; "-E" indicates LOW LEVEL(expression) node; "-S" indicates SYMBOL node; "-T" indicates type node; "-L" indicates LABLE node; "-C" indicates COMMENT node. In the case of value -1, "– " is shown.

3.2 Unparse

We use table driven method, the same technique as in Sage++, to unparse the intermediate representation. Each kind of bfnd node and llnd node is predefined as a series of commands in a global table. An executor written in C will interpret these commands one by one.

The followings are several commands for unparsing a bfnd,

```
%ERROR      : Error ; syntax : %ERROR'message'
%CMNT       : the comment attached to a bif node
%NL         : NewLine
%%          : '%' (Percent Sign)
%TAB        : Tab
```

%IF	: If: %IF (condition) then_bloc [%ELSE else_bloc] %ENDIF
%ELSE	: Else
%ENDIF	: End of If
%SYMBID	: Symbol identifier
%LL1	: Low Level Node 1
%LL2	: Low Level Node 2
%LL3	: Low Level Node 3
%L2L2	: Low Level Node 2 of Low Level Node 2
%BLOB1	: All Blob 1
%BLOB2	: All Blob 2

The commands series for ASSIGN statement is defined as following:

```
%CMNT%IF(%LABEL != %NULL)%LABEL%ENDIF%PUTTAB%LL1 = %LL2%NL
```

which means the executor will first unparse the comment node attached to the statement, and then, if label exist, unparse the label. Finally, unparse the left hand side of the statement and the right hand side of the statement seperately.

The definition of the command series will be corresponding to the definition of each bfnd and llnd.

The advantage of doing so is that once the definition of the bfnd or llnd is changed, only the command series need to be changed, with the executor program untouched.

4 Conclusion

In the implementation of our front-end we try to make it complete and stable, so that it could serve for the further work efficiently and enduringly. At the same time, we try to accomplish the work with minimum cost, so many features in previous systems were inherited.

We install this suite of front end on 7 platforms successfully with various conbination of cc, gcc, lex, flex(2.5.2), yacc, bison. (The 7 platforms are descripted in "HPFfe: a Front-end for HPF".)

A Internal Definition

References

[1] Lesk,M.E. Lex - a Lexical Analyzer Generator. AT&T Bell Laboratories, Murray Hill, New Jersey,1975.

[2] Jhonson,S.C. Yacc - Yet Another Compiler Compiler. AT&T Bell Laboratories, Murray Hill, New Jersey,1975.

[3] Compiler Tools Group, Guide for New Eli Users. University of Colarado, Boulder, CO, 1994.

[4] Joint Technical Committee ISO/IEC JTC1, Information technology, ISO/IEC 1539, Fortran 90 Language, 1991.

[5] H.P.F.Forum. High Performance Fortran Language Specification. Rice University, Houston, Texas, May 1993, Version 1.0.