TITLE: Parser for sample language using YACC.

THEORY:

YACC-Yet Another Compiler-Compiler

Yacc is the Utility which generates the function ’yyparse’ which is indeed the Parser. Yacc describes a context free, LALR (1) grammar and supports both bottom-up and top-down parsing. The general format for the YACC file is very similar to that of the Lex file.

1. Declaration

2. Grammar Rules

3. Subroutines

Declarations

%%

Grammar Rules

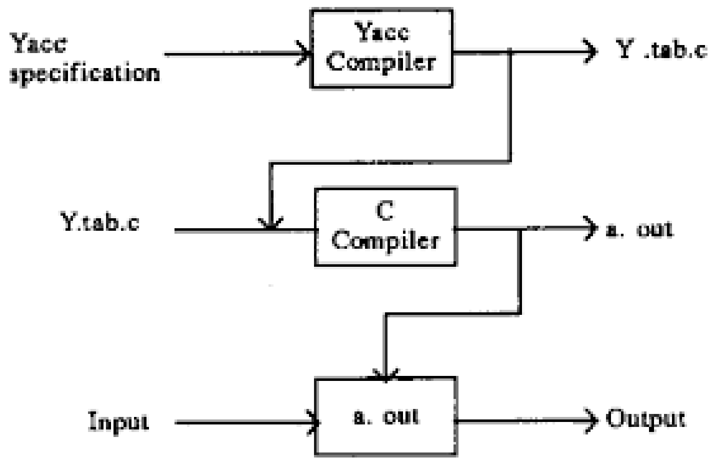
%%

Subroutines

In Declarations apart from the legal ’C’ declarations there are few Yacc specific declarations which begins with a %sign.

1. %union: It defines the Stack type for the Parser. It is a union of various data’s/structures/objects.
2. %token: These are the terminals returned by the yylex function to the yacc. A token can also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as %token <stack member> tokenName.
3. %type: The type of a non-terminal symbol in the Grammar rule can be specified with this. The format is %type <stack member> non-terminal.
4. %noassoc: Specifies that there is no associativity of a terminal symbol.
5. %left: Specifies the left associativity of a Terminal Symbol.
6. %right: Specifies the right associativity of a Terminal Symbol.
7. %start: Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules.
8. %prec: Changes the precedence level associated with a particular rule to that of the following token name or literal.

A YACC program consists of three parts:



The general style for coding the rules is to have all Terminals in upper-case and all non-terminals in lower-case. To facilitates a proper syntax directed translation the Yacc has something called pseudo-variables which forms a bridge between the values of terminal/non-terminals and the actions. These pseudo variables are $$, $1, $2, $3...... The $$ is the L.H.S value of the rule whereas $1 is the first R.H.S value of the rule and so is $2 etc. The default type for pseudo variables is integer unless they are specified by %type, %token <type> etc.

How are symbol table or data structures built in the actions?

For a proper syntax directed translation it is important to make full use of the pseudo variables. One must have structures/classes defined of all the productions which can form a proper abstraction. The yystack should then be a union of pointers of all such structures/classes. The reason why pointers should be used instead of structures is to save space and to avoid copying structures when the rule is reduced. Since the stack is always updated i.e. on reduction the stack elements are popped and replaced by L.H.S so any data that was referred gets lost.

<u>ALGORITHM:</u>

1) Start
2) Input file name and valid sample C file
3) Open and Read file
4) Match and return the tokens using lex
5) If token is variable/identifier, make entry in symbol table with corresponding datatype and default value
6) Check syntax of sample program against the Grammar mentioned
7) If Grammar rule not matched generate syntax error and goto step 10)
8) Repeat step 5) 6) till Grammar is matched completely
9) Display symbol table
10) Stop

```
/***********************OUTPUT************************

$ yacc -d -v parserfile.y
parserfile.y: warning: 168 shift/reduce conflicts [-Wconflicts-sr]
$ lex lexfile.l
$ gcc lex.yy.c y.tab.c
$ ./a.out sample.c

Program is Syntactically correct!!
Parsed successfully

Symbol Table:

VARIABLE        VALUE                           DATATYPE
------------------------------------------------------------
s1              Uninitialized                   float
r               2.5                     float
i               4                       float
p               100                     int
n               3                       int
x               9                       int
c               Uninitialized                   char

**************************************************************/
```