# Practical 7

Name: Tushar Harsora

Roll No: 19BCE509

## Introduction

The aim of this practical is to parse the expression and control statement in the given source code. This practical consists of 2 files *practical.l* and *practical.yy* the first file declares tokens that will be generated during lexing process. It declares the identifiers like types of braces and parentheses including some keywords, datatypes etc. second file is .yy file which is given to bison the bison converts the context free grammar to deterministic LR and generates the parser. It contains the precedence, start symbol, tokens and grammar rules. It also includes the symbol table to check if variable is already declared or not. And also check if the types are compatible to compare or not.

## Code

Practical.l

```
%{
   #include "practical.tab.hh"
   #include <stdlib.h>
%}

%option yylineno

%%

"<"|">"|"<="|">="|"||"|"&&"|"=="|"!="    { return REL; }
"int"|"float"       { yylval.name = strdup(yytext); return DATATYPE; }
"if"               { return IF; }
"else"              { return ELSE; }
"="                { return ASSIGN; }
"{"                { return LBRACES; }
"}"                { return RBRACES; }
"("                { return LPAREN; }
")"                { return RPAREN; }
[0-9]+.[0-9]+         { yylval.fvalue = atof(yytext); return FCONST; }
[0-9]+             { yylval.ivalue = atoi(yytext); return ICONST; }
[a-zA-Z][a-zA-Z0-9]*   { yylval.name = strdup(yytext); return ID; }
```

```
"+"|"-"|"/"|"*"       { return SIGN; }
";"                 { return SEPERATOR; }
[ \n\t]+             { }
.                   { return yytext[0]; }
%%
```

Practical.yy

```
%{
   #include "practical.tab.hh"
   // #include <stdio.h>
   #include <cstring>
   #include <map>
   #include <string>
   using namespace std;

   extern int yylineno;
   extern char* yytext;
   int yylex();
   void yyerror(char*);
   void yyerror(const string&);
   void assert_is_compatible(const char* lhs, const char* rhs);

   map<string, pair<string, string>> symb_tab;    // maps from variable
name to
                                    // variable datatype and value
%}

%union{
   char* name;
   int ivalue;
   float fvalue;
}

%define parse.lac full
%define parse.error detailed
%define parse.trace
%right '='
%left '+''-'
%left '*''/'
%left IF_ELSE_PREC
%token SEPERATOR REL SIGN IF ELSE ASSIGN LPAREN RPAREN LBRACES RBRACES
%token <ivalue> ICONST
%token <fvalue> FCONST
%token <name> ID DATATYPE
%type Start Decl BooleanThing IfStat ElseStat EXPR MATH

%%

Start : /* EMPTY */
```

```
      | EXPR Start
      | IfStat Start

Decl : DATATYPE ID SEPERATOR    { symb_tab[$2] = make_pair($1, ""); }

BooleanThing : ID REL ID
           {
              assert_is_compatible($1, $3);
           }

IfStat : IF LPAREN BooleanThing RPAREN LBRACES EXPR RBRACES ElseStat

ElseStat : /* EMPTY */ %prec IF_ELSE_PREC
         | ELSE LBRACES EXPR RBRACES

EXPR :  Decl
      | ID ASSIGN MATH SEPERATOR
         {
            /* Should check for datatype */
         }

MATH :   FCONST
      | ICONST
      | ICONST SIGN ICONST
      | FCONST SIGN FCONST
      | ID SIGN ID
         {
            assert_is_compatible($1, $3);
         }

%%

void assert_is_compatible(const char* lhs, const char* rhs){
   auto f = symb_tab.find(lhs);
   auto s = symb_tab.find(rhs);
   if(f == symb_tab.end()){
      yyerror((string("Identifier ") + lhs + string(" Not Declared")));
      exit(-1);
   }
   else if(s == symb_tab.end()){
      yyerror((string("Identifier ") + rhs + string(" Not Declared")));
      exit(-1);
   }

   if(f->second.first != s->second.first){
      yyerror((string("Incompatible Datatypes ") + string(f-
>second.first) + " " + string(s->second.first)));
      exit(-1);
```

```
   }
}

void yyerror(const string& s){
   char* temp = (char*) malloc(sizeof(char) * s.size());
   strcpy(temp, s.c_str());
   yyerror(temp);
   free(temp);       // i don't think it reaches here
   // exit(-1);
}
void yyerror(char* s) {
   printf("ERROR: %s at line %d\n", s, yylineno, yytext);
}

int main(){
   int failure = yyparse();
   if(!failure){
      puts("Program Accepted");
   }else{
      puts("Program Rejected");
   }
   return 0;
}
```

# Outputs

Compilation using flex and bison

```
→  Practical 7 flex practical.l
→  Practical 7 bison -d -Wcounterexamples practical.yy
→  Practical 7 g++ practical.tab.cc lex.yy.c -ll -o Pract
practical.tab.cc: In function ‘int yyparse()’:
practical.tab.cc:1559:16: warning: ISO C++ forbids converting a string constant to ‘char*’
 1559 |   yyerror (YY_("memory exhausted"));
      |                ^~~~~~~~~~~~~~~~~~
practical.tab.cc:274:22: note: in definition of macro ‘YY_’
  274 | #  define YY_(Msgid) Msgid
      |                      ^~~~
```

Source code is correct and types are compatible so accepted

```
→  Practical 7 cat test1
int a;
int b;
a = 10;
b = 20;
if(a == b){
    a = 20;
}
→  Practical 7 cat test1 | ./Pract
Program Accepted
```

The If statement is missing closing bracket.

```
→  Practical 7 cat test2
int a;
int b;
a = 10;
b = 20;
if(a == b){
    a = 20;
→  Practical 7 cat test2 | ./Pract
ERROR: syntax error, unexpected end of file, expecting RBRACES
Program Rejected
```

The declaration of identifier *b* is missing

```
→  Practical 7 cat test3
int a;
a = 10;
if(a == b){
    a = 20;
}
→  Practical 7 cat test3 | ./Pract
ERROR: Identifier b Not Declared
```

The types of variables *a* and *b* are incompatible

```
→  Practical 7 cat test4
int a;
float b;
a = 10;
b = 20.20;
if(a == b){
    a = 20;
}
→  Practical 7 cat test4 | ./Pract
ERROR: Incompatible Datatypes int float at line 5
→  Practical 7
```

## Conclusion

In this practical we implemented how identifiers works and how to create CFG grammar for language incorporating statements, assignments and comparisons. We included basic symbol table that can be used to perform conversion and comparison in type-safe way.