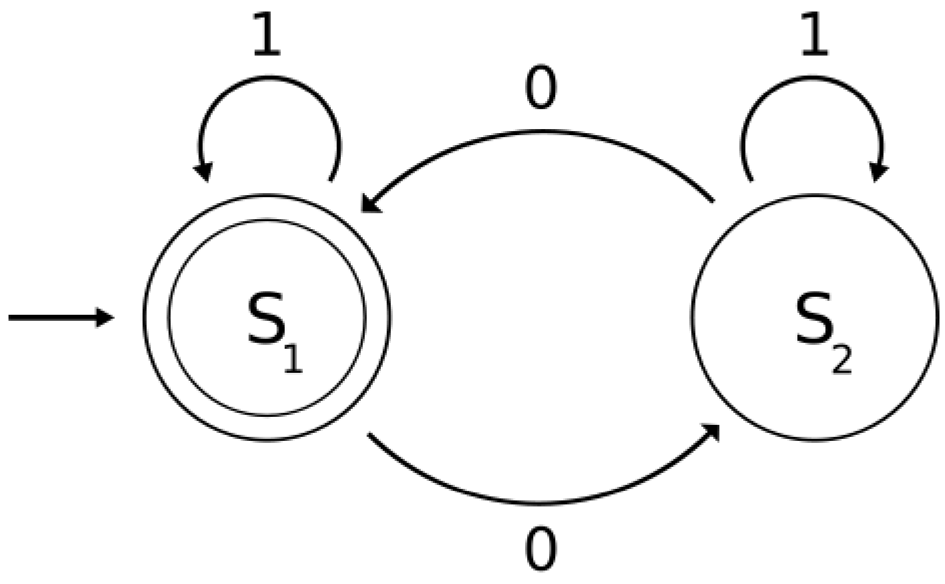
Lab Sheet Report

Compiler Design Lab [EC – 372]

Lokesh Chandra Basu 10114026

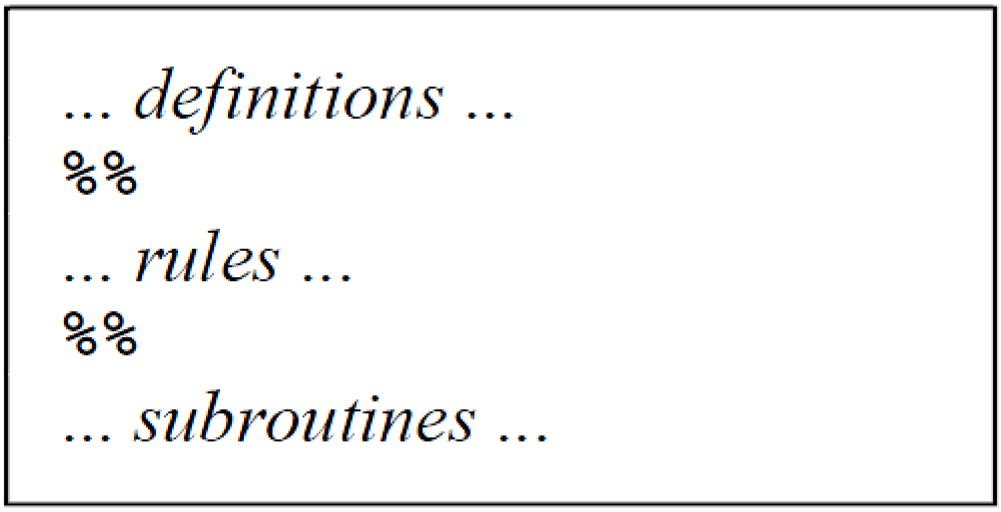
Lab Sheet 1

Logic for question 1, 2 and 3:



Lab Sheet 2

Logic for question 1 to 7:



```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

The following is an example Lex file for the flex version of Lex. It recognizes strings of numbers (integers) in the input, and simply prints them out.

```
/*** Definition section ***/
%{
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells flex to read only one input file */
%option noyywrap

%%


 /*** Rules section ***/
 /* [0-9]+ matches a string of one or more digits */
[0-9]+  {
            /* yytext is a string containing the matched text. */
```

```
          printf("Saw an integer: %s\n", yytext);
       }

.|\n    {   /* Ignore all other characters. */   }

%%

/*** C Code section ***/

int main(void)
{
   /* Call the lexer, then quit. */
   yylex();
   return 0;
}
```

INPUT :
abc123z.!&*2gj6

OUTPUT:

```
Saw an integer: 123
Saw an integer: 2
Saw an integer: 6
```

name	*function*
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN condition`	switch start condition
`ECHO`	write matched string

Lex Predefined Variables

Lab Sheet 3

Logic for question 1:

FIRST & FOLLOW

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during panic-mode error recovery.

FIRST (α)

If α is any string of grammar symbols, let FIRST(α) be the set of terminals that begin the strings derived from α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST(α).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set:

1. If X is terminal, then FIRST(X) is {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
3. If X is nonterminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$. is a production, then place *a* in FIRST(X) if for some *i*, *a* is in FIRST(Y_i), and ε is in all of FIRST(Y_1), ... , FIRST(Y_{i-1}); that is, $Y_1, \ldots, Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ε to FIRST(X). For example, everything in FIRST(Y_1) is surely in

FIRST(X). If Y_1 does not derive ε, then we add nothing more to FIRST(X), but if $Y_1 \Rightarrow \varepsilon$, then we add FIRST($Y_2$) and so on.

Now, we can compute FIRST for any string $X_1X_2 \ldots X_n$ as follows. Add to FIRST($X_1X_2 \ldots X_n$) all the nonε symbols of FIRST(X_1). Also add the non-ε symbols of FIRST(X_2) if ε is in FIRST(X_1), the non-ε symbols of FIRST(X_3) if ε is in both FIRST(X_1) and FIRST(X_2), and so on. Finally, add ε to FIRST($X_1X_2 \ldots X_n$) if, for all *i*, FIRST(X_i) contains ε.

FOLLOW(A)

Define FOLLOW(A), for nonterminal A, to be the set of terminals *a* that can appear immediately to the right of A in some sentential form, that is, the set of terminals *a* such that there exists a derivation of the form S⇒αA*a*β for some α and β. Note that there may, at some time during the derivation, have been symbols between A and *a*, but if so, they derived ε and disappeared. If A can be the rightmost symbol in some sentential form, then $, representing the input right endmarker, is in FOLLOW(A).

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set:

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right endmarker.
2. If there is a production A ⇒ αBβ, then everything in FIRST(β), except for ε is placed in FOLLOW(B).
3. If there is a production A ⇒ αB, or a production A ⇒ αBβ where FIRST(β) contains ε (i.e., β ⇒ε) then everything in FOLLOW(A) is in FOLLOW(B).

EXAMPLE

Consider the expression grammar (4.11), repeated below:

E → T E′

E′→ + T E′ | ε

T → F T′

T′→ * F T′ | ε F → (E) | **id**

Then:

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}

FIRST(E′) = {+, ε}

FIRST(T′) = {*, ε}

FOLLOW(E) = FOLLOW(E′) = {) , $}

FOLLOW(T) = FOLLOW(T′) = {+,), $}

FOLLOW(F) = {+, *,), $}

Logic for question 2:

```
for each production $n \rightarrow \alpha$
    for each $a \ \varepsilon$ first($\alpha$)
        add $n \rightarrow \alpha$to T[n, a]
    if $\varepsilon \varepsilon$ first($\alpha$) then
        for each $b \ \varepsilon$ follow(n)
            add $n \rightarrow \alpha$to T[n, a]
```

Logic for question 3:

$factor \longrightarrow (\ exp \ ) \mid$ **number**

Grammar rule	Pass 1	Pass 2	Pass 3
$exp \longrightarrow exp \ addop \ term$			
$exp \longrightarrow term$			First(exp) = {(, **number**}
$addop \longrightarrow +$	First($addop$) = {+}		
$addop \longrightarrow -$	First($addop$) = {+, −}		
$term \longrightarrow term \ mulop \ factor$			
$term \longrightarrow factor$		First($term$)= {(, **number**}	
$mulop \longrightarrow *$	First($mulop$) = {∗}		
$factor \longrightarrow (\ exp \ )$	First($factor$) = { (}		
$factor \longrightarrow$ **number**	First($factor$) = {(, **number**}		

Logic for question 4 and 5:

One big difference between simple precedence and operator precedence is that in simple precedence parsing, the non-terminal symbols matter. In operator precedence parsing all non-terminal symbols are treated as one generic non-terminal, N. Thus, a grammar such as

1.	E	E + T
2.		| T
3.	T	T * P
4.		| P
5.	P	(E)
6.		| V

will look like, in operator precedence terms, the following:

1.	N	N + N
2.		| N
3.	N	N * N
4.		| N
5.	N	(N)
6.		| V

Note: We no longer have specific non-terminal symbols, but one generic non-terminal, N.

Operator Precedence Parsing Algorithm

Input: an operator precedence matrix or precedence functions modified grammarsentence with ppended to the end.

Assumptions: < Vi > where Vi Vt, V

Algorithm:

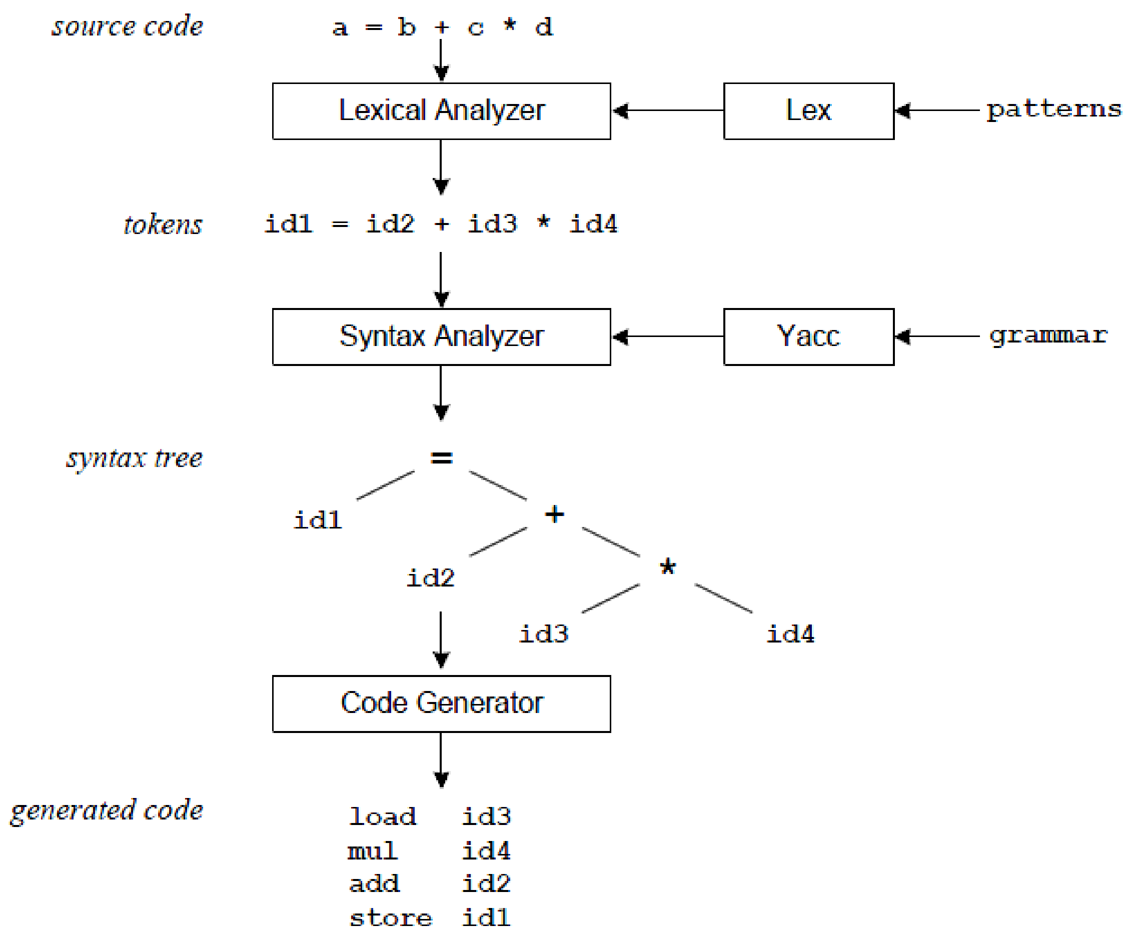
```
1.    Push  onto stack
2.    Read first input symbol and push it onto stack
3.    Do
      3.1   Obtain OP relation between the top terminal symbol on the stack and the next input symbol
      3.2   If the OP relation is  <o  or  =o
      Then
            3.2.1   Stack input symbol.
            Else  { relation is >o }
            3.2.2   Pop top of the stack into handle, include non-terminal symbol if appropriate
            3.2.3   Obtain the relation between the top terminal symbol on the stack and the leftmost   terminal symbol in the handle
            3.2.4   While the OP relation between terminal symbols is =o
            Do
                  3.2.4.1  Pop top terminal symbol and associated non-terminal symbol on stack into handle
                  3.2.4.2  Obtain the OP relation between the top terminal symbol on the stack and the leftmost terminal symbol in the handle
            3.2.5   Match the handle against the RHS of all productions
            3.2.6   Push N onto the stack
4.    Until  end-of-file  and only   and N are on the stack
```

Note: The algorithm above does not detect any syntax errors which we will talk about later.

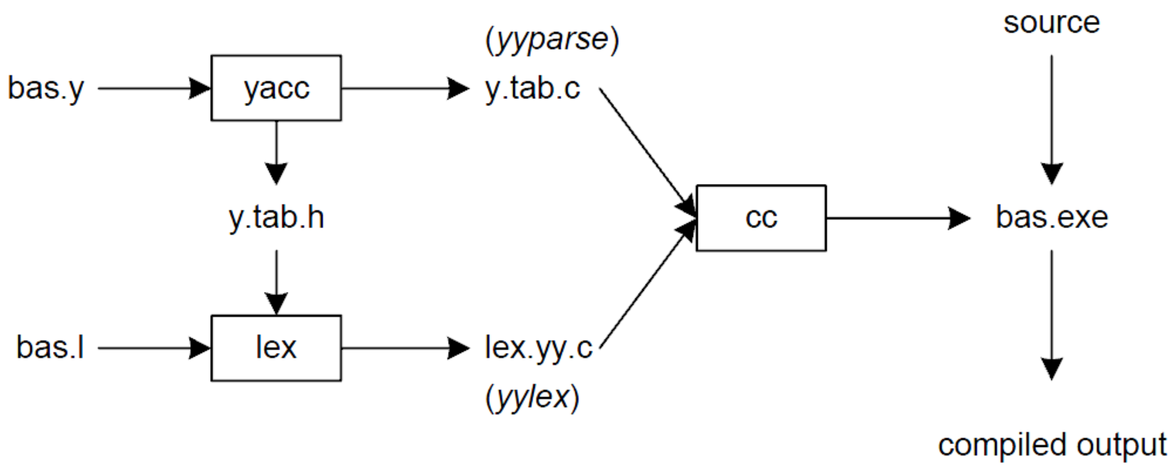
Def: A operator precedence parse configuration triple is (S, ,) where S is the stack, is the input string, and is the action sequence.

Lab Sheet 4

Logic for question 1-8:



Compilation Sequence



Building a Compiler with Lex/Yacc

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. Declarations
2. Grammar Rules
3. 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 atas/ 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 assocoativity 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.

The grammar rules are specified as follows:

Context-free grammar production:	p->AbC
Yacc Rule:	p : A b C { /* 'C' actions */}

The general style for coding the rules is to have all Terminals in upper-case and all non-terminals in lower-case.

To facilitate 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 Recursions handled in the grammar rule ?

Recursions are of two types left or right recursions. The left recursion is of form

```
    list :  item    { /* first item */ }
          |  list  item  { /* rest of the items */ }
```

The right recursion is of form

```
    list  : item  { /* first item */ }
          | item list  { /* rest of items */ }
```

In right Recursion the Parser is a bit bigger than that of left recursion and the items are matched from right to left.

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 also 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.