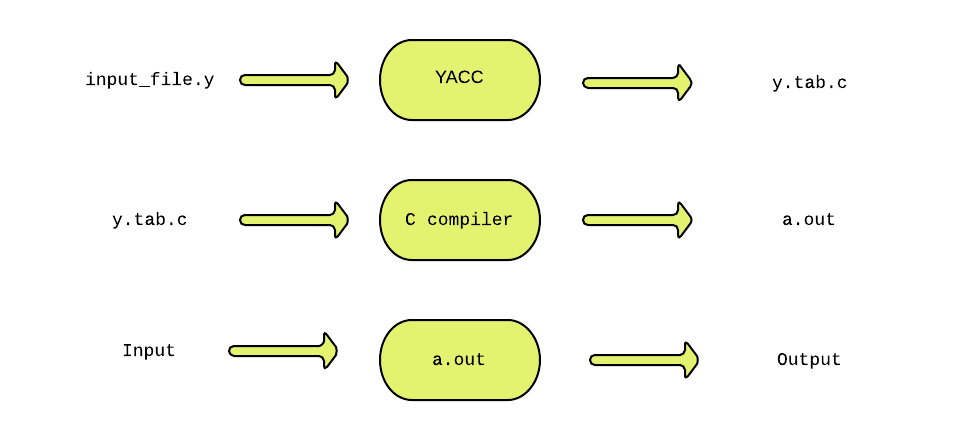
**Introduction to YACC**

YACC (Yet Another Compiler Compiler) is a tool used to generate a parser. This document is a tutorial for the use of YACC to generate a parser for ExpL. YACC translates a given [Context Free Grammar (CFG)](https://silcnitc.github.io/yacc.html#navcfg) specifications (input in input\_file.y) into a C implementation (y.tab.c) of a corresponding [push down automaton](http://en.wikipedia.org/wiki/Pushdown_automaton)(i.e., a finite state machine with a stack). This C program when compiled, yields an executable parser.



The source SIL program is fed as the input to the generated parser ( a.out ). The *parser* checks whether the program satisfies the syntax specification given in the input\_file.y file.

[YACC](https://en.wikipedia.org/wiki/Yacc) was developed by [Stephen C. Johnson](https://en.wikipedia.org/wiki/Stephen_C._Johnson) at [Bell labs](https://en.wikipedia.org/wiki/Bell_Labs).

**Parser**:

A parser is a program that checks whether its input (viewed as a stream of tokens) meets a given grammar specification. The syntax of SIL can be specified using a Context Free Grammar. As mentioned earlier, YACC takes this specification and generates a parser for SIL.

**Context Free Grammar (CFG)**:

A context free grammar is defined by a four tuple (N,T,P,S) - a set N of non-terminals, a set T of terminals (in our project, these are the [tokens](https://silcnitc.github.io/lex.html#navintro) returned by the lexical analyzer and hence we refer to them as tokens frequently), set P of productions and a start variable S. Each production consists of a non-terminal on the left side (head part) and a sequence of tokens and non-terminals (of zero or more length) on the right side (body part). We will explore productions further in detail [later](https://silcnitc.github.io/yacc.html#navprod) in this documentation. For more about context free grammars refer to this[wiki](https://en.wikipedia.org/wiki/Context-free_grammar).

**Example:** This example is an Infix to Postfix converter implemented using YACC. The rules part of the YACC program has been shown below:

start: expr '\n' {exit(1);}

;

expr: expr '+' expr {printf("+ ");}

| expr '\*' expr {printf("\* ");}

| '(' expr ')'

| DIGIT {printf("NUM%d ",pos);}

;

In this example:  
The set of non-terminals are N = {start, expr}  
The set of terminals are T = {'\n', '+', '\*', '(', ')' , DIGIT }  
The start symbol S = start.

Sample Input/Output :

I:1+5  
O: NUM1 NUM2 +

When the input expression 1+5 is given to the parser generated by YACC, the parser prints a [postfix](http://en.wikipedia.org/wiki/Reverse_Polish_notation) form of the original expression 1+5 as NUM1 NUM2 + where, NUM1 represents the first number ( 1 ) in the input and NUM2 represents the second number ( 5 ) in the input.

Sample Input/Output :

I: 3+(1\*9)+5  
O: NUM1 NUM2 NUM3 \* + NUM4 +  
  
I: 5$  
O: NUM1 error

The [example](https://silcnitc.github.io/yacc.html#navgramex1) above demonstrates the specification of rules in YACC. In this example there are five rules. Each rule has a *production part* and an *action part*.The action part consists of C statements enclosed within a { and }. Each production part has a *head* and a *body* separated by a '**:**'. For example, the first rule above has production part with start as the head and expr '\n' as the body. The action part for the rule is {exit(1);}. The parser reads the input sequentially and tries to find a pattern match with the body part of each production. When it finds a matching production, the action part of the corresponding rule is executed. The process is repeated till the end of the input.

In the above example, when the input 1+5 is given to the parser, it attempts to match the input with the body of the production of the first rule. When the input has been parsed completely and correctly matched with the start production start: expr '\n' the parser executes the action exit(1);. The statements printf("NUM "); and printf("+ "); are executed as result of the input being matched with the productions expr: DIGIT and expr: expr '+' expr respectively. If the parser fails to find any matching body part, it invokes a special yyerror() function. In our example, the yyerror() function is programmed to print the message “error”.

**yyparse()**

* The y.tab.c file contains a function yyparse() which is an implementation (in C) of a [push down automaton](http://en.wikipedia.org/wiki/Pushdown_automaton).
* yyparse() is responsible for parsing the given input file.
* The function yylex() is invoked by yyparse() to read tokens from the input file.
* Note that the yyparse() function is automatically generated by YACC in the y.tab.c file.
* Although YACC declares yylex() in the y.tab.c file, it does not generate the definition for yylex().
* Hence the yylex() function definition has to be supplied by you (either directly by defining yylex() in the *auxiliary functions* section (explained in the next section) or using a lexical analyzer generator like LEX).
* Each invocation of yylex() must return the next token (from the input steam) to yyparse().
* The action corresponding to a production is executed by yyparse() only after sufficient number of tokens has been read (through repeated invocations of yylex()) to get a complete match with the body of the production.

## The structure of YACC programs

A YACC program consists of three sections: Declarations, Rules and Auxiliary functions. (Note the similarity with the structure of LEX programs).

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

### 2.1 Declarations

The declarations section consists of two parts: (i) C declarations and (ii) YACC declarations. The C Declarations are delimited by **%{** and **%}**. This part consists of all the declarations required for the C code written in the *Actions* section and the *Auxiliary functions* section. YACC copies the contents of this section into the generated y.tab.c file without any modification.  
The following example shows an abstract outline of the structure of the declarations part of a YACC program:

**Example:**

/\* Beginning of Declarations part \*/

%{

/\*Beginning of C declarations\*/

/\*End of C declarations\*/

%}

/\*Beginning of YACC declarations \*/

/\*End of YACC declarations \*/

/\* End of Declarations Part \*/

%%

The YACC declarations part comprises of declarations of tokens (usually returned by the lexical analyzer). The parser reads the tokens by invoking the function yylex() (To be discussed in detail later).

### 2.2 Rules

A rule in a YACC program comprises of two parts (i) the production part and (ii) the action part. In this project, the syntax of SIL programming language will be specified in the form of a context free grammar. A rule in YACC is of the form:

production\_head : production\_body {action in C } ;

The following example shows an abstract outline of the structure of the rules part of a YACC program:

%%  
/\* Rules Section begins here \*/  
  
/\* Rules Section ends here \*/  
%%

The rules in our example can be found [here](https://silcnitc.github.io/yacc.html#navexy0r)

**2.2.1 Productions**

Each production consists of a production head and a production body. Consider a production from our [example](https://silcnitc.github.io/yacc.html#navexy0r):

expr : expr '+' expr

The expr on the LHS of the **:** in the production is called the *head* of the production and the expr '+' expr on the RHS of the : is called the *body* of the production. In the above example, '+' is a terminal (token) and expr is a non-terminal. Users can give name to a token. (for instance we can give the name 'PLUS' to the token '+'). In such cases, the names must be defined in the declarations section. For example have a look at the definition of the token DIGIT [here](https://silcnitc.github.io/yacc.html#navexy0yd). The head of a production is always a non-terminal. Every non-terminal in the grammar must appear in the head part of at least one production.

Note that a non-terminal in the head part of a production may have one or more production bodies separated by a “|”. Consider the non-terminal expr in our [example](https://silcnitc.github.io/yacc.html#navexy0). The non-terminal has four production bodies expr '+' expr , expr '\*' expr , '(' expr ')' and DIGIT. The first production body has an associated print action op\_printf("+") and the second production body has an associated action op\_print("\*"). yyparse() executes the action only when the body expr '+' expr has been matched with the input. The action part of a single production may have several statements of C code.

**2.2.2 Actions**

The action part of a rule consists of C statements enclose within a '{' and '}'. These statements are executed when the input is matched with the body of a production and a *reduction* takes place. The notion of a *reduction* will be explained later. From the [example](https://silcnitc.github.io/yacc.html#navexy0r) below, consider the following rule:

expr: DIGIT {printf("NUM%d ",pos);}

In this rule, when the input matches with the body of the production DIGIT, it is *reduced* to expr and the action {printf("NUM%d ",pos);} is executed.

**2.2.3 Auxiliary Functions**

The Auxiliary functions section contains the definitions of three mandatory functions main(), yylex() and yyerror(). You may wish to add your own functions (depending on the the requirement for the application) in the y.tab.c file. Such functions are written in the auxiliary functions section. The [main()](https://silcnitc.github.io/yacc.html#navexy0al) function must invoke [yyparse()](https://silcnitc.github.io/yacc.html" \l "navexy0al) to parse the input. You will need to write your supporting functions later in this project.

**Example: intopost.y**

%{

**/\*\*\* Auxiliary declarations section \*\*\*/**

#include<stdio.h>

#include<stdlib.h>

/\* Custom function to print an operator\*/

void print\_operator(char op);

/\* Variable to keep track of the position of the number in the input \*/

int pos=0;

%}

**/\*\*\* YACC Declarations section \*\*\*/**

%token DIGIT

%left '+'

%left '\*'

%%

**/\*\*\* Rules Section \*\*\*/**

start : expr '\n' {exit(1);}

;

expr: expr '+' expr {print\_operator('+');}

| expr '\*' expr {print\_operator('\*');}

| '(' expr ')'

| DIGIT {printf("NUM%d ",pos);}

;

%%

**/\*\*\* Auxiliary functions section \*\*\*/**

void print\_operator(char c){

switch(c){

case '+' : printf("PLUS ");

break;

case '\*' : printf("MUL ");

break;

}

return;

}

yyerror(char const \*s)

{

printf("yyerror %s",s);

}

yylex(){

char c;

c = getchar();

if(isdigit(c)){

pos++;

return DIGIT;

}

else if(c == ' '){

yylex(); /\*This is to ignore whitespaces in the input\*/

}

else {

return c;

}

}

main()

{

yyparse();

return 1;

}

*y.tab.c* file can be generated using the command

yacc intopost.y

*y.tab.c* is compiled using C compiler

gcc y.tab.c

NOTE: *%left*option is used to resolve shift/reduce conflicts. It is explained in detail later.

Sample Input/Output:

I: 2+2  
O: NUM1 NUM2 PLUS

When yyparse() matches the input 2+2 with the production body expr '+' expr, it executes the action op\_print('+'); and as a result prints “PLUS” in place of '+' as per the definition of op\_print().

**NOTE**: op\_print() is used in the example just to show an example of the declaration, definition and usage of a user defined auxliary function. Generally in this project, we use printf() to display content.