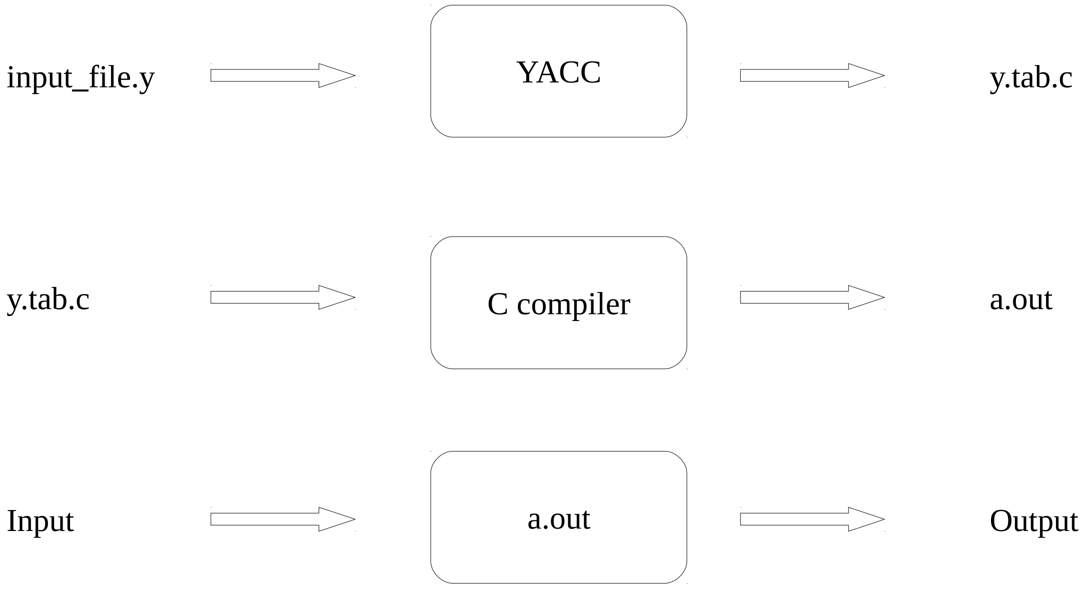
1 Introduction to YACC

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 SIL. YACC translates a given Context Free Grammar (CFG) specifications (input in input_file.y) into a C implementation (y.tab.c) of a corresponding push down 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.

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.

Recall that 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 returned by the lexical analyzer and hence we may refer to them as *tokens* occasionally), 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). For more about context free grammars refer to this wiki.

Example: This example [Link to eg_1n2post_no-att.y] 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 N = {`start, expr`}, the set of terminals T = {`'\n', '+',  '*', '(', ')' , DIGIT` } and the start symbol S = `start`.

Sample Input/Output :

```
I:   1+5
O:   NUM1 NUM2 +
```

When the input 1+5 is given to the parser (object file) generated by YACC [Link to yacc_run_inst.txt] , the parser prints a *postfix form* of the original expression `1+5` as `NUM1 NUM2 +` where, `NUM1` represents the first number in the input expression i.e. `1` and `NUM2` represents the second number in the input expression i.e. `5`.

```
I:   3+(1*9)+5
O:   NUM1 NUM2 NUM3 * NUM4 + +

I:   5$
O:   NUM1 error
```

This example 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`". [Link to yyerror() of eg_in2post_no-att.y]

2 The structure of YACC programs

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

```
DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS
```

2.1 Declarations

The Declarations section consists of two parts, C declarations and YACC Declarations.

The C Declarations are delimited by `%{` and `%}`. This part consists of all the declarations required for the C code you write 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:

```
/* 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 [Link to LEX document]). 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 [Link to Rules section in eg_in2post_no-att.y]

2.2.1 Productions

Each production consists of a production *head* and a production *body*. Consider a production from our example [Link to productions of eg_in2post_no-att.y]:

```
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 name to a tokens. (for instance we can give the name 'PLUS' to the token '+'). **In such cases, the names must be defined in the declarations section. (example)** 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.

2.2.2 Actions

The action part of a rule consists of C statements which are executed when the input is matched with the body of a production. ([Link to Actions section of eg_in2post_no-att.y])

The y.tab.c file contains a function `yyparse()` which is an implementation (in C) of a push down automaton. `yyparse()` is responsible for parsing the given input file. The function `yylex()` is invoked by `yyparse()` to read tokens from the input file. [Link to yylex() in eg_in2post_no-att.y]. 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* 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.

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 [Link to body of expr in eg_in2post_no-att.y]. The non-terminal has four production bodies `expr '+' expr`, `expr '*' expr`, `'(' expr ')'` and `DIGIT`. The first production body has an associated print action `printf("+ "), [ add one more ]`. `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.

2.2.3 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()` [Link to main() in eg_in2post_no-att.y] function must invoke `yyparse()` to parse the input.

The auxiliary functions section of our example [Link to Auxiliary functions section of eg_in2_post_no-att.y] program uses no user defined functions. You will need to write your supporting functions later in this project.

```
expr:  expr '+' expr   {op_print('+');}
    | expr '*' expr    {op_print('*');}
    | '(' expr ')'
    | DIGIT            {printf("NUM%d ",pos);}
    ;

%%
/*** Auxiliary functions part ***/

void op_print(char op)
{
    if(op == '+')
     printf(“PLUS ”);
    else if(op == '*')
     printf(“MUL ”);
}

yyerror()
{
    printf("error");
    return;
}

yylex()
{
    int c;
    c = getchar();
    if(isdigit(c))
    {
     pos++;
     return DIGIT;
    }
    return c;
}

main()
{
    yyparse();
    return 1;
}
```

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 the user defined auxiliary function `op_print()`.

3. A WORKING INTRODUCTION TO SHIFT-REDUCE PARSING

The shift-reduce parser is essentially a push down automaton. Hence it consists of a finite state machine with a stack. The stack is used to hold terminal and/or non-terminal symbols. The following is a gentle introduction to shift-reduce parsing.

Let us make the following assumptions:
- The input to be parsed, which is a sequence of terminal symbols, is stored in an input buffer.
- $ is used as an end-marker to mark the end of the input buffer.
  The stack is initialized to contain just the symbol $.
- Once a terminal has been read from the input, it is removed from the input buffer.

Consider the following context free grammar:

```
(1) expr :    expr '+' expr
(2)  |    '(' expr ')'
(3)  |    '0' | '1' |   '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
(4)  ;
```

The terminal set is `{+,(,),0,1,2,3,4,5,6,7,8,9}`. The only non-terminal is '`expr`'.

Let us consider parsing of the input 2+2 using this grammar. When the parsing process begins, the contents of the stack and the input buffer would be as follows:

STACK: $ I/P BUFFER: 2 + 2 $

That is, the stack is initialized with the $ symbol. The input buffer is initialized with the input (here 2+2) terminated with a $.

The contents of the stack and the contents of the input buffer together define the *configuration of the parser*.

The parsing process takes place in several steps. On successful completion of parsing, the configuration would be:

STACK: $ expr I/P BUFFER: $

where expr is the start variable of the parser's context free grammar. Thus at the end of successful parsing, the start symbol of the grammar followed by $ will appear on the stack whereas, the input buffer will contain nothing except the $.

At each step of parsing, the parser takes an *action* resulting in a configuration change. A shift-reduce parser can take four possible *parser-actions*:

1. **Shift** is the parser-action of shifting the next unread terminal from the input buffer to the stack. Once an input character is read, it is removed from the buffer and it is *pushed* onto the stack.

2. **Reduce** is the parser-action of replacing one or more grammar symbols from the top of the stack that matches a body of a production, with the corresponding production head. The contents on top of the stack which matches the right side of a production is called a *handle*. The process of replacing a handle with the corresponding production

head is called a *reduction*.

3. **Accept** is the parser-action indicating that the entire input has been parsed successfully. The parser executes an accept action only if the input buffer is empty and the stack consists of the start variable terminated by a $.

Accepting configuration: STACK: $ <start_variable>   I/P BUFFER: $

4. **Error** indicates that an error was encountered while parsing the input. In our example, there was no error. We will see error conditions later.

EXPLANATION:

At each step of parsing, the shift-reduce parser decides on an action depending on the configuration of the parser.

(1)   STACK: $ I/P BUFFER: 2 + 2 $

At this configuration, the parser executes a *shift* action i.e. 2 is pushed onto the stack resulting in the configuration:

(2)   STACK: $ 2   I/P BUFFER: + 2 $

At configuration (2), the parser executes a reduce action as the contents on top of the stack match the body of the production expr -> '2' i.e., 2 is the handle in this case and a reduction takes place replacing the handle with the production head expr.

(3) STACK: $ expr I/P BUFFER: + 2 $

Following the reduction, the parser shifts '+' to the stack.

(4) STACK: $ expr + I/P BUFFER: 2 $

The parser shifts once again, resulting in the configuration:

(5) STACK: $ expr + 2 I/P BUFFER: $

The parser executes a reduce action *reducing* the handle '2' to the grammar symbol expr.

(6) STACK: $ expr + expr I/P BUFFER: 2 + 2 $

The handle expr + expr is further reduced to expr.

(7) STACK: $ expr I/P BUFFER: $

Since the I/P BUFFER is empty and the stack contains only the start variable, the parser executes an accept action, indicating that the input has been parsed successfully.

The action of the parser is summarized in the following table (link). The logic behind the parsing process will be discussed subsequently.

The following table summarizes the step-by-step change in the parser's configuration after each *action* taken by parser.

STACK	**I/P BUFFER**	**ACTION TAKEN**
(1) $	2 + 2 $	_
(2) 2 $	+ 2 $	SHIFT
(3) expr $	+ 2 $	REDUCE
(4) + expr $	2 $	SHIFT
(5) 2 + expr $	$	SHIFT
(6) expr + expr $	$	REDUCE
(7) expr $	$	REDUCE
(8) expr $	$	ACCEPT

The following parsing algorithm exhibits the behavior of the parser.

```
Initialize the stack with $
while (true)
{
     next = next_token();  /* look at next input token, don't remove it from input*/
     if ( next == $ and stack contains “start_symbol $” ) then
     return ACCEPT;
     if (valid_shift(next) then
         parser-shift(next); /* remove token from input and push it to stack */
     else if (valid_reduce(next))
         parser-reduce();  /* don't remove token from input, but do a reduction */
    else
     return ERROR /* no valid shift or reduce possible */
}

parser_shift(next)
{
    Remove the token from the input and advance the input to the next token.
     Push the current token into the stack.
}

parser_reduce()
{
    Pop out the handle from the top of the stack.
    Push of the handle's production to the stack.
}
```

There are several variants of shift-reduce parsing like the LR(1), SLR(1) and LALR(1) parsing methods. The notion of what is a valid shift or a valid reduce depends on the particular parsing method and can be fairly involved. Hence, we do not explain here how the functions valid_shift(next) and valid_reduce(next) operate. YACC uses an LALR parsing method. (See ... References and links). However, an understanding of the general principles of shift-reduce parsing at this level will be sufficient for most part of this project.

4. Working of the Infix to Postfix program

When input_file.y is fed to YACC, it generates a y.tab.c file. When compiled, this program yields a parser. [Link to Introduction to Yacc]. The generated parser uses shift-reduce parsing to parse the given input. Yacc copies the C declarations (in the Declaration section of input_file.y) and all the auxiliary functions (in the Auxiliary functions section of input_file.y) directly into y.tab.c without any modification. In addition to these, YACC generates the definition of yyparse() in y.tab.c.

It is important to understand that, y.tab.c contains the following :

- The C declarations from the input_file.y file [Link to part in y.tab.c]
- Generated yyparse() definition [Link to part in y.tab.c]
- All the auxiliary functions from the input_file.y [Link to part in y.tab.c]

Recall our infix to postfix program (link)
Here is a Sample Input and Output:

```
I: 2+3
O: NUM1 NUM2 +
```

When the expression `2+3` is fed as the input to the generated parser, the main() function in the auxiliary functions section invokes yyparse() as below: (The code for main() from the example is copied below)

```
main()
{
    yyparse();
    return 1;
}
```

As noted earlier, yyparse() invokes yylex() to read tokens from the input. For example, yylex() reads the input 2 and returns the token DIGIT (code of yylex() shown below)

```
yylex()
{
    int c;
    c = getchar();
    if(isdigit(c))
    {
     pos++;
     return DIGIT;
    }
    return c;
}
```

/* Comment: Every time a number is found in the input stream, `yylex()` increments pos and returns a token `DIGIT` to `yyparse()`. If any character other than a number is found, `yylex()` simply returns the character itself to `yyparse()`. In the process of returning `DIGIT` to yyparse(), yylex() increments the value of pos. As it was initialized to 0, after returning DIGIT, `pos` holds the value 1. */

The input is copied into an input buffer terminated by an end-marker (Say for example, '$') and the stack is initialized with the end-marker. Initial configuration of the parser:

STACK: $ I/P BUFFER: 2+3 $

The parser uses shift-reduce parsing methodology to arrive at the final accepting configuration (Note that an accepting configuration indicates successful parsing):

STACK: $ start I/P BUFFER: $

At every step of parsing, yyparse() calls yylex() for reading the input buffer.

STACK: $
I/P BUFFER: 2+3 $

yylex() reads 2 and returns DIGIT to yyparse(). Yyparse() shifts DIGIT to the stack.

STACK: $ DIGIT
I/P BUFER: +3 $

Now, the parser reduces the handle DIGIT to expr, as DIGIT matches the body of the production expr: DIGIT.

STACK: $ expr

I/P BUFFER: +3 $

For every handle reduced by the parser, the corresponding action of the handle's production is executed. As a result the parser executes `printf("NUM%d ",pos)` and prints NUM1 on the screen. At the second call, yylex() returns '+' (Note that yyparse() invokes yylex() repeatedly till the end of input is encountered i.e. $ is encountered in the input). yyparse() then shifts '+' to the stack.

STACK: $ expr +
I/P BUFFER: 3 $

At the third, invocation of yylex() by yyparse(), yylex() returns DIGIT on reading 3 from the buffer. yyparse() shifts DIGIT resulting in the configuration:

STACK: $ expr + DIGIT
I/P BUFFER: $

Now the handle DIGIT is reduced to expr, and as a result the action `printf("NUM%d ",pos)` is executed, printing NUM2 on the screen. Following the previous reduction, the configuration would be:

STACK: $ expr + expr
I/P BUFFER: $

The parser further reduces the handle expr + expr to expr, as the handle matches the body of the production expr: expr '+' expr. The corresponding action printf("+ ") is executed, thus printing + on the screen.

```
STACK:         $ expr
I/P BUFFER:    $
```

The handle expr is further reduced to start, resulting in the accepting configuration.

```
STACK:         $ start
I/P BUFFER:    $
```

And as the parser has reached the accepting configuration, yyparse() flags successful parsing by returning 0 to the `main()` function.

A generalized algorithm of yyparse() would look like:

```
Initialize the stack with the end-marker $
new_symbol = yylex()
while (true)
    switch(decision by shift-reduce parser)
    case 'reduce':
        pop handle from stack
        push production head of the handle to stack
        execute action corresponding to the handle's production
    case 'shift':
        push new_symbol to stack
        new_symbol = yylex()
    case 'accept':
        return 0
    case 'error':
        return 1
```

5. The Infix to Postfix program, revised

The previous infix to postfix program prints the structure of the postfix expression and not the postfix expression itself. For the parser to print the postfix expression it would need the value associated with every `DIGIT` token. For example, the value associated with the token `NUM` in the first sample input/output is 2. The value associated with a token is called an *attribute.*

In the previous program `yylex()` simply returns the token DIGIT to `yyparse()` and does not return any value associated with it. In order to access the value of the token DIGIT, there must be some method to return an attribute along with the token from `yylex()` to `yyparse()`. This can be achieved using a variable called `yylval`. The usage of yylval has been demonstrated in the following program.

```
%{

#include <stdio.h>

%}

%token DIGIT

%%

start : expr '\n'          {printf("\nComplete");exit(1);}
```

```
    ;

expr:  expr '+' expr        {printf("+ ");}
    | expr '*' expr         {printf("* ");}
    | '(' expr ')'
    | DIGIT                 {printf("%d ",$1);}
    ;

%%


yyerror()
{
    printf("Error");
}

yylex()
{
    int c;
    c = getchar();
    if(isdigit(c))
    {
     yylval = c - '0';
     return DIGIT;

    }
    return c;
}

main()
{
    yyparse();
    return 1;

}
```

Sample I/O:

```
I:  1+2*3
O:  1 2 3 * +

I:  (2+7)*4
O:  2 7 + 4 *
```

Attributes can be implemented by using `yylval.` `yylval` is a global variable of the type `YYSTYPE` declared in y.tab.c. By default, `YYSTYPE` is of the type `int`. This is evident from the following code segment found in y.tab.c

```
typedef int YYSTYPE;
```

As a result, `yylval` (which is originally of the type `YYSTYPE`), has an *inferred* type `int.` It is used to return additional information about the lexeme found to the parser i.e., `yylval` is used to return an attribute in addition to the token to the parser.

In the above example, the `yylex()` returns the token `DIGIT` and the value of the token in the following code segment under definition of `yylex()`:

```
yylval = c - '0';
return DIGIT;
```

The attribute of a grammar symbol (i.e., the value of yylval associated with the grammar symbol) can be accessed in the action of a YACC rule using `$i` (where `i` is the position of the grammar symbol in the body of a production). `$$` refers to the attribute value associated with the head of a production.

Example:

```
    expr: DIGIT        {printf("%d",$1);}
```

The action prints the attribute associated with the token `DIGIT` obtained through `$1`.

`YYSTYPE` can be defined to be of any data type by the programmer. To return an attribute of a type other than int, `yylval` maybe overridden by a user defined yylval in the auxiliary declarations section. In order to return multiple attribute values for a token, it may be declared to be of the type `union`.

Example:

```
%{
    #include<stdio.h>
    typedef union
    {
     int value;
     int number_of_digits;
    }YYSTYPE;
    YYSTYPE yylval;
%}

%%

    /* Rules */
%%
    /* Auxiliary functions */
```

6. ATTRIBUTE STACK

Similar to the parse stack, YACC also maintains an attribute stack. The attribute stack is used to hold the attributes of the grammar symbols in the body of a production. The elements of the stack are of type `YYSTYPE`. The contents of yylval are pushed to the attribute stack when when the parser executes a shift action on the parse stack.