Using YACC: a Small Introduction

1 The Content of this Document

Subject of this tutorial is Yacc: using it, writing programs in it, its output, how ambiguities are resolved, how lex and yacc communicate, etc.

2 Running Yacc

Given a LALR(1) grammar G, Yacc generates an automaton (in fact a Push-Down Automaton) that parses the language over G. The given grammar G may be ambiguous, but specifying precedence rules breaks the ambiguities. The output of Yacc is a file named `y.tab.c` which should be compiled to produce a function called `yyparse()`. Loading `yyparse()` with

- the scanner `yylex()` generated by Lex,
- a function `main()`,
- `yyerror()` – we will see later what this file, an error handling routine, means
- suitable libraries (`ly` and `ll`)

will produce a working parser for the source language you are writing a compiler for. Notice that it is your responsibility to provide Yacc with these routines. You are lucky for this class: the professor has provided you with some of them!

Figure 1 show you how to construct a parser using Yacc. Basically, follow the sequence

1. save your Yacc source program (that is, the specification of the given Grammar you have written in the Yacc language) in a file, say `csc488.y`.

2. use the UNIX command `yacc csc488.y` to get an output called `y.tab.c` which is a C program representing an LALR parser written in C, together with some routines you may have already inserted in the specification file `csc488.y`; Note that the parsing table in `y.tab.c` is in a compact form.

3. compile `y.tab.c` together with the libraries `ll` and `ly`, and with your functions `main()`, `yylex()`, and `yyerror()`. To do this, write an appropriate Makefile.

From the manual page, you can get the most useful options to the Yacc command. There you also will find what files are produced by the command. A more detailed explanation can be found in T. Mason and D. Brown's book.

The generated files are:

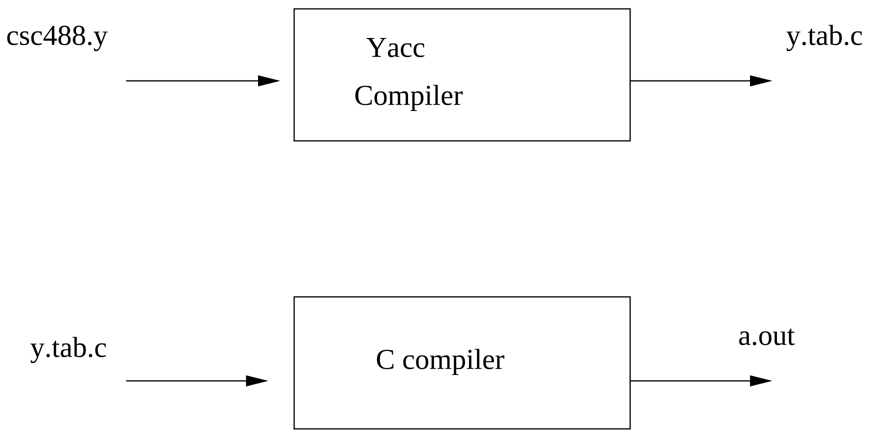

Figure 1: Constructing a parser using Yacc

- `y.output`: This file contains a description of the parsing tables.
- `y.tab.c`: C program representing an LALR parser written in C, together with some routines you may have already inserted in the Yacc program.
- `y.tab.h`: These are "defines" for token names. This is a header file that makes the token declarations available to other source files outside of the file with the parser routine.
- `yacc.acts`: A temporary file.
- `yacc.debug`: A temporary file for debugging purposes.
- `yacc.tmp`: A temporary file.
- `yaccpar`: A parser prototype for C programs.

Please note a difference in the naming convention between Yacc and Bison, a newer version of Yacc. Bison uses the (base) file name of your Yacc program and append `.tab.c` to it. Thus if your file name is `csc488.y`, then the output of Bison will have the name `csc488.tab.c` instead of `y.tab.c`.

Here are some options I think are important:

- `-d` Generates the file `y.tab.h` with the #define statements associating the user-supplied token specifications with the declared token names.
- `-t` Compiles runtime debugging code by default. Whether or not you use `-t`, the runtime debugging code is under the control of a preprocessor variable named YYDEBUG. If the value of YYDEBUG is a non-zero number, then Yacc includes the debugging code. If its value is zero, then it does not.

- `-v` Prepares the file `y.output` and a report on conflicts due to ambiguities.

- `-p <driver-file>` Allows you to specify a parser skeleton of your choice instead of the default parser prototype `yaccpar` which is to find in the directory /usr/ccs/bin/ where you may take a look on it. For example, if you have a skeleton `csc488parser` stored in your home directory, you can specify: `yacc -p /csc488parser parser.y` to obtain a parser `parser.y`.

3 Structure of a Yacc Program

A Yacc source program has three parts:

```
declarations

  %%

translation rules

  %%

user-specified routines
```

The declaration part has two (optional) subparts:

- First subpart contains C declarations delimited by `%{` and `%}`. What you declare here will be directly passed through to the output of Yacc, that is, to your parser program written in C.

- Second subpart contains declarations of grammar tokens. These are shorthands that will be used later in other parts of the Yacc program. the second subpart also contains statement for enforcing precedence and associativity of operators.

The translation rules part is everything that comes between the first and second occurrences of `%%`. It basically contains the translation of the rules of the given LALR(1) grammar into a suitable syntax. Rules have the form:

```
<left hand side> : <alternative_1>    {semantic action_1}
                 | <alternative_2>    {semantic action_2}
               . . .
                 | <alternative_n>    {semantic action_n}
                 ;
```

which is a Yacc translation of a set of grammar production written as

```
<left hand side> : <alternative_1>
                 | <alternative_2>
               . . .
                 | <alternative_n>
```

Notice the appearance of something new here, semantic actions. This is the the next big step in the study of compiler construction. Semantic actions are C routines that are performed once the corresponding production has been processed.

The user routines part is everything that follows the last occurrence of `%%`. Normally, the routines `yylex()`, `main()`, and `yyerror()` go here. Notice that the scanner must have the name `yylex()`. Notice also that you are luckily provided with a scanner and some parts of the main routine.

4 Learning by Examples

The first example, adapted from Aho *et al.*, is a Yacc program specifying the following grammar.

```
E -> E+E
  |  E-E
  |  E*E
  |  E/E
  |  (E)
  |  -E
  |  number
```

If you run Yacc on this grammar with the option `-v` , you will get a report on conflicts due to the obvious ambiguities that are in the grammar. The following is the resulting Yacc source program:

```
%{
#include  <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* Used for Yacc stack */
%}


%token NUMBER
%left '+' '-'
%left '*' '/'
%left UMINUS


%%
```

```
lines   :  lines expr '\n' { printf("%g\n", $2); }
        |  lines '\n'
        |    /* this is a production of the empty word */
        ;

expr    :  expr '+' expr            { $$ = $1 + $3; }
        |  expr '-' expr            { $$ = $1 - $3; }
        |  expr '*' expr            { $$ = $1 * $3; }
        |  expr '/' expr            { $$ = $1 / $3; }
        |  '(' expr ')'             { $$ = $2; }
        |  '-' expr   %prec UMINUS  { $$ = - $2; }
        | NUMBER
        ;

%%

yylex() {
    int c;
    while ((c = getchar()) == '');
    if ((c == '.') || (isdigit(c))) {
       ungetc(c, stdin);
       scanf("%lf", &yylval);
       return NUMBER
    }
    return c;
}
```

Let us see what is done in this example. The declarations between `%{` and `%}` are just passed through to `y.tab.c`. With the line `%token NUMBER`, we declare the symbol NUMBER to be a token. Notice that `%token` is a keyword of Yacc. Other keywords are:

- `%left` Defines left-associative operators.
- `%right` Defines right-associative operators.
- `%nonassoc` Defines operators that may not associate with themselves.
- `%type` Define the type of nonterminal.
- `%union` Declare multiple data types for semantic values.
- `%start` declare the start symbol. Default is first symbol in rules section
- `%prec` Assign precedence to a rule.

Thus, with `%left '*' '/'`, we are declaring + and - to be of the same precedence and be left associative. By declaring first the pair `<+,->` and then the pair `<*,/>`, we implicitly assign to *,/ a higher precedence than to `<+,->`. Thus token are given precedence in the order they appear in declarations, in the lowest-first logic.

With instructions like `%left`, `%right`, etc, you tell Yacc to resolve conflict in a certain manner. However, without these mechanisms, Yacc resolve itself all parsing action conflicts using two basic rules:

- A reduce/reduce conflict is resolved by choosing the conflict rule that is put first in the Yacc program.
- In case of a shift/reduce conflict, Yacc resolve it always in favor of shift.

As a rule of thumb, do not have an entire confidence in this mechanism. This should be a kind of helper that save you from a unknown conflict.

Once Yacc finds a conflict, use the option `-v` to generate a file `y.output`. This will tell you what kind of conflicts occurred, and a readable version of the parsing table telling you how Yacc resolved the conflicts. This is important to see if the conflicts were resolved as you would like. This is why I told you not to have an entire confidence in Yacc's mechanism for conflict resolution.

Let us move to the production rules part. Terminals are quoted. Alternatives in the right hand side of productions are separated by a vertical bar. A semicolon ends a set of production rules. The semantic actions are sequences of C statements. Here the symbol $$ refers to value of the symbol on the left hand side of the rule, and $ refers to the value of the *i*th symbol on the right hand side. Thus `$$ = $1 + $3;` is associated to `expr : expr '+' expr`, with $$, $1, and $3 referring to the first, the second, and the third occurrence of `expr`, respectively.

The criptic line `| '-' expr %prec UMINUS  $$ = - $2;` means that the unary minus operator appearing in this production is given a precedence level higher than that of all other preceding operators. UMINUS is just a placeholder in order to declare a precedence for a production. We did this by putting the declaration `%left UMINUS` in the declaration part.

Finally, the user supplied routine is a scanner with the mandatory name `yylex()`. What it does is easily understandable.

Note that Yacc does not report shift/reduce conflicts resolved with the above explicit mechanisms.

5 Compilation

Note: This will be explained in the next tutorial with more details. I will assume that the content of this introduction is known.

Suppose `csc488.l` is the name of the lex file. You should now create a C code by entering the command

```
$ lex csc488.l
```

If csc488.y is your Yacc grammar, then enter

```
$ yacc -d csc488.y
```

to compile it. Finally, compile the output of Lex and Yacc with a C compiler by typing

```
$ cc -o csc488 y.tab.c  -ly -ll
```

Note that in order for the later command to work, you should have a #include `#include "lex.yy.c"` in the third part of the Yacc program (see below).

Notice that for the above small example, we have only to use Yacc and C since the scanner is already integrated in the Yacc specification as a user provided routine. We could however have a separate Lex program to generate our scanner. Let it be the following

```
number [0-9]+\.?|[0-9]*\.[0-9]+

%%

[ ]          {/* to skip blanks */}
{number}     { sscanf(yytext, ``%lf'', &yylval);
               return NUMBER;}
\n|.         {return yytext[0];}
```

The Yacc program is now

```
%{
#include  <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* Used for Yacc stack */
%}


%token NUMBER
%left '+' '-'
%left '*' '/'
%left UMINUS


%%

lines  :  lines expr '\n' { printf("%g\n", $2); }
       |  lines '\n'
```

```
        |    /* this is a production of the empty word */
        ;

expr    :  expr '+' expr             { $$ = $1 + $3; }
        |  expr '-' expr             { $$ = $1 - $3; }
        |  expr '*' expr             { $$ = $1 * $3; }
        |  expr '/' expr             { $$ = $1 / $3; }
        |  '(' expr ')'              { $$ = $2; }
        |  '-' expr   %prec UMINUS   { $$ = - $2; }
        | NUMBER
        ;

%%

 #include ``lex.yy.c''
```

Here is a sample session with the above specifications

```
qew.cs> lex calc.l
qew.cs> yacc calc.y
qew.cs> ls
calc.l          calc.y          lex.yy.c        y.tab.c
qew.cs> cc -o calc y.tab.c  -ly -ll
qew.cs> ls
calc            calc.l          calc.y          lex.yy.c        y.tab.c
qew.cs> calc
3 - 2
1
456 * 5
2280
1234 + 1234
2468
78/9
8.66667
^C
qew.cs>
```

6 how lex and yacc communicate

Figure 2, adapted from Mason and Brown's book, show how lex and yacc communicate.

Yacc usually uses scanners produced by Lex in the following way. The Lex library `ll` has a scanner skeleton (a driver) named `yylex()` that is used to produce a working scanner. This name is MANDATORY. You should have in the third part of your Yacc program a #include statement

```
#inlude ``lex.yy.c''
```

With this statement, each Lex action return a terminal (i.e. a token) that is known to Yacc. Thus, the scanner `yylex()` has now access to token names used by Yacc.

You have seen those criptic names such as `yytext`, or `yylval`. Let us now see what they mean:

- `yylval` This an external variable that the parser sets up. The scanner uses it to pass the value of the token. Note that a literal character may also be passed to the parser as a token that has its ASCII value as value.

- `yytext` This external variable is used by the parser to store the token matched by the scanner.

7 Debugging and errorhandling

(Incomplete)

You should check the output files from Lex and Yacc to see where your code has been inserted. Yacc mention explicitely the original lines of the C code it takes from Yacc specifications. Thus it is obvious that when you are compiling the output of Lex and Yacc with a C compiler, the only place where you have to check for C syntax and/or logical errors is the source code you provided. This is easy to find.

Now, what about conflicts in yacc programms? Let us change the above Yacc program to

```
%{
#include  <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* Used for Yacc stack */
%}


%token NUMBER
%left '+' '-'
%left UMINUS


%%
```

```
lines   :  lines expr '\n' { printf("%g\n", $2); }
        |  lines '\n'
        |    /* this is a production of the empty word */
        ;

expr    :  expr '+' expr              { $$ = $1 + $3; }
        |  expr '-' expr              { $$ = $1 - $3; }
        |  expr '*' expr              { $$ = $1 * $3; }
        |  expr '/' expr              { $$ = $1 / $3; }
        |  '(' expr ')'               { $$ = $2; }
        |  '-' expr   %prec UMINUS    { $$ = - $2; }
        | NUMBER
        ;

%%

 #include ``lex.yy.c''
```

That is, the line with `%left '*' '/'` has been removed. I you run this new Yacc program, you get

```
qew.cs> yacc calc-err.y

conflicts: 14 shift/reduce
```

8 Built-in variables

(Incomplete)

9 An example similar to, though different from our csc488 compiler

(Incomplete)

This will be given in a separate document where I want to put together everything from Lex and Yacc specification to code generation.

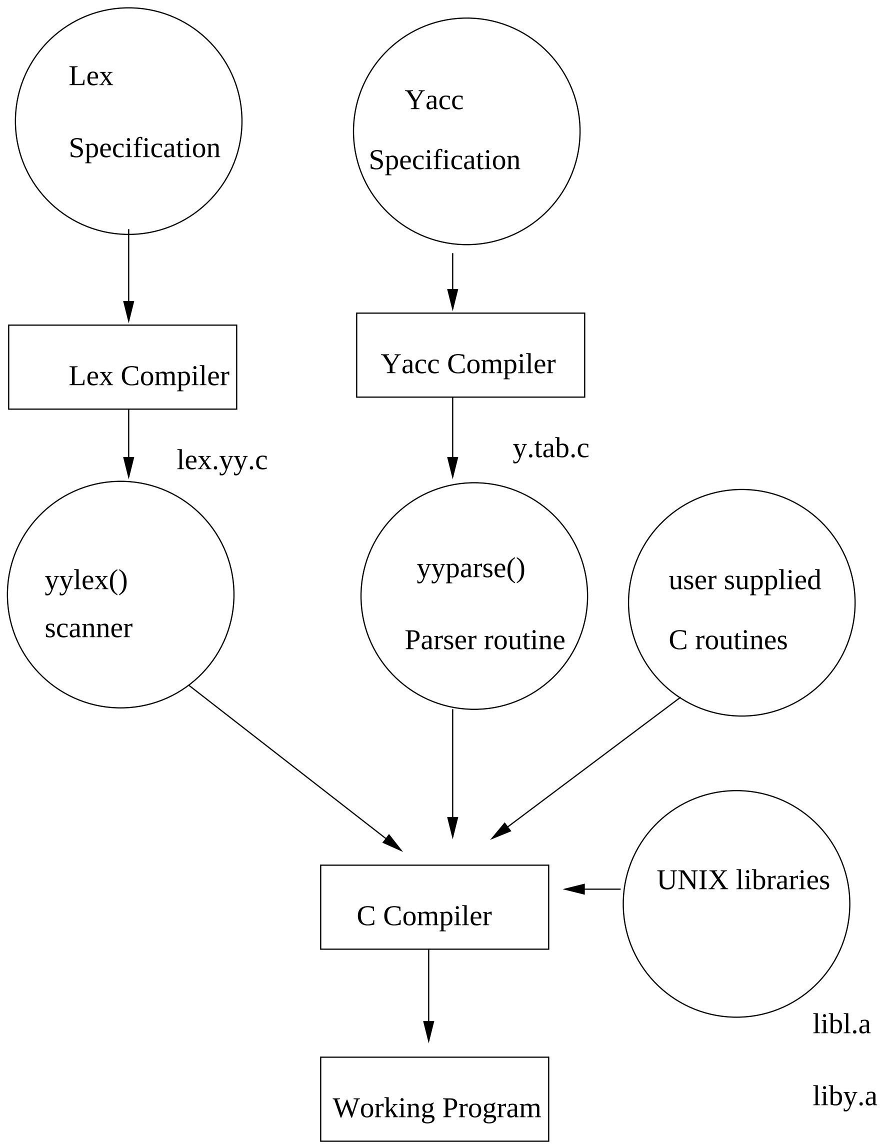

Figure 2: How lex and yacc communicate