yacc and parsing

- `yacc` (yet another compiler compiler) allows us to describe the parsing rules of a language using context free grammars
- It also allows us to specify what information to record and other actions to take once a rule is applied in a derivation
- The actions taken and information recorded is done using C (e.g. we can use C functions, structs, data types, etc)
- Together with `lex` tokenizing code, `yacc` allows us to generate a full analysis tool for a specified language

lex/yacc sequence

- To use lex/yacc to produce a program to analyze source code in language L, the steps are as follows:
  - Create a file, e.g. L.lex, containing L’s tokenization rules
  - Create a file, e.g. L.yacc, containing L’s parsing rules
  - Run lex on L.lex to produce lex.yy.c
  - Run yacc on L.yacc to produce y.tab.c
  - Compile the two .c files to produce our tool
  - Gcc y.tab.c lex.yy.c -o analyzeL
- We can now use the tool on source code written in L, e.g.

```
./analyzeL < someLsourcecode
```

yacc file organization

- A `yacc` file uses the extension `.yacc`, and is divided into three core sections, separated from one another with %%
  - The first section is used to define token types, and for C library includes, prototypes, global vars, and typedefs
  - The second section is used for the actual parsing rules and code to be applied when a derivation rule is applied
  - the final section contains regular C code, typically the implementations for any functions prototyped in the first section
- Syntax gotcha: when using C-style comments in your lex file, don’t start them on the first character of the line, put space(s) first

just the .yacc file layout...

```
 /* declarations  */
%{
#include <stdio.h>
#include “y.tab.h”
int yylex(void);
int yywrap();
int yyerror(char *s);
%}
 /* tokens go here */
%%
```

```
/* parser rules go here */
%%
 /* C function bodies,
    main runs yyparse */
int main()
{
   int res = yyparse();
   return res;
}
```

declarations section: C portions

- `yylex` is auto-generated for us, while `yyerror` and `yywrap` are defined in the .lex file
- If we wish to access variables defined in the .lex file they we need to declare them as externals in the %{ ... %}, e.g.

```
extern int row;
```

- We can prototype additional C functions, as long as the full implementation is in the bottom section of the `.yacc` file

declarations section: types, tokens

- We need to identify the available yylval data types and token types, and associate a data type with each token type
- yylval is a union of all the possible yylval types with a name for each, e.g. an int, a char*, or a struct of two ints:

```
%union { int id; char* str;
         struct Finfo { int num, den; } fract; }
```

- We list each type of token with its associated type, e.g.

```
%token<int> FOOTOKEN, INTEGER
%token<char*> IDENTIFIER
%token<struct Finfo> FRACTION
```

Aside: accessing the token data

- Access to the token's data will be different in the .lex file than in the `.yacc` file
- In the `.lex` file, we can store/access the token information using `yylval`, e.g. `yylval = x;` for a simple data type like the `id`, or `yylval.num = x;` for a struct type like `Finfo`
- In the `.yacc` file we'll access the token information using the union name, e.g. `id` or `fract.num` (though we'll wind up embedding the access in something like `$< >`)
- More details later!

Declarations: non terminals

- We also need to provide a type and name for each non-terminal in our context free grammar
- The types get included in the union we declared earlier
- The non-terminals are usually denoted in lowercase (as opposed to the terminals in uppercase), e.g.

```
%type<struct nonterminfo> program statementlist
                statement expression multexpr addexpr
```

- We also need to identify the “top level” non-terminal:

```
%start program
```

Declarations: associativity

- Finally, in the declarations section we can specify the associativity of operators (left-to-right or right-to-left)
- We give the list of tokens corresponding to the operators

```
%right ASSIGN
%left ADD MULT DIV SUB
```

- That ends the declarations section, the next section is the list of parsing rules (the two sections being separated by a line that is just a %%)

(augmented) CFG parsing rules

- The second section (after the %%) gives the CFG rules
- Each rule will be a non-terminal, a colon, the sequence of terminals/non-terminals it is made up of, and then a code block to be executed when applying the rule, and a final semi-colon, e.g. two possible rules for a list of statements

```
statement_list: statement
    { printf("one statement\n"); }
    ;
statement_list: statement statement_list
    { printf("more statements to go...\n"); }
    ;
```

Accessing the associated data

- %union, %token, %type identified a data type for each token and terminal
- In a rule, each token/terminal on the RHS of : is numbered, from 1 to whatever, then accessed using $<unionname>1

```
statement: IDENTIFIER ASSIGNOP expression { } ;
```

- `IDENTIFIER`'s type name was `str` (of data type `char*`) thus we access `IDENTIFIER`'s content using `$<str>1`
- Similarly `ASSIGNOP`'s data would be in `$<`*`whatever`*`>2` and `expression`'s would be in `$<`*`whatever`*`>3`

The C code associated with rules

- The tool we're creating may be a compiler, a static analysis tool like lint, a code formatter or optimizer, anything we like
- Whatever actions our tool takes will be performed in the C code sections associated with our rules: e.g. translating the C++ source code we're reading into machine code
- As with the .lex files, the code can make use of the prototypes, data types, and variables identified in our declarations section, and the actual code possibilities are endless

The final section, C functions

- Implement and C functions prototyped above (regular C syntax, usual gotcha about indenting your comments)
- main routine needs to be defined here, and must call to yyparse() (auto-generated) to begin the parsing process:

```
int main() {
   /* do whatever before */
   int R = yyparse();
   /* do whatever after */
   return R;
}
```