Appendix B.2

Yacc

- Yacc takes a description of a grammar as its input and generates the table and code for a LALR parser.
- Input specification file is in 3 parts
  - Declarations and Definitions
  - Grammar and Actions
  - User-Written code
- Parts are separated by %%

1. Grammar

- We will start with the grammar section, since this is the easiest to relate to what you are learning in Chapters 3 and 4.
- **Productions**
  - Grammars are defined in near-BNF form. The differences can be summarized as follows:
  - 1. Single characters used as terminals are put into single quote, but nonterminals are written out by name.

- 2. Terminals that are keywords, or tokens like ***id*** are declared as such in the declarations section.
- 3. Instead of —> in the production, Yacc uses a colon, but alternatives are separated by a | as usual.
- 4. Yacc uses a blank to represent an epsilon production.

- Thus a grammar like
  - E -> E+T | E-T | T
  - T -> T*F | T/F | F
  - F -> (E) | I
- can be written as:
  - expr : expr '+' term
  - | expr '-' term
  - | term
  - ;

- term : term '*' fact
- | term '/' fact
- | fact
- ;

- fact : '(' expr ')'
- | ID
- ;

- In this example, ID will have been declared a token in the declarations part.

- **Semantic Actions**
  - Inside of { } you can have code segments.
  - Each item in the production has a semantic value.
    - $$ is the left hand side,
    - things on the RHS are numbered from $1 on.
- Thus
  - expr : expr '+' term { $$ = $1 + $3; }
- If the attributes are structs we can have
  - expr : ID { $$.loc = $1.loc; }

2. Declarations and Definitions

- In the declarations section we identify all tokens except the single-character operators (unless they are also returned as a token).
- To declare a token we write:
  - %token ID
  - %token NUMBER

- Yacc assigns a numerical code to each token, and expects these codes to be returned to it by the lexical analyzer.
  - This assignment is placed in yytab.h
  - you can get Lex to use these by placing
    #include "yytab.h"
    inside the %{ %} at the beginning of your Lex specification.
- Notice you do not have to declare non-terminals. Their appearances on the left-hand side of productions in the grammar section declares them automatically.

- You can declare precedence, and associativity.
  - Most of the time this is unnecessary, since precedence and associativity are built into the grammar **IF** it is unambiguous.
- Finally we must identify the starting symbol of the grammar.
  - %start statement

- The data type for attributes has the predefined name YYSTYPE, and we must define what it means.
  - %{
  - #include <stdio.h>
  - #typedef int YYSTYPE;
  - %}
- Note: this allows you to change what YYSTYPE is by declaring a struct and using it.

3. User Written Code

- The user written code contains (at a minimum) the main program (that invokes the parser) and an error handler.
  - main(){
  - yyparse();
  - }
  - void yyerror(char * msg){
  - printf("%s\n", msg);
  - }

4. A Sample Yacc Specification

- Examples...