Using an LALR(1) Parser Generator

- Yacc is an LALR(1) parser generator
  - Developed by S.C. Johnson and others at AT&T Bell Labs
  - **Yacc** is an acronym for **Yet another compiler compiler**
  - Yacc generates an integrated parser, not an entire compiler
- One popular compatible version of Yacc is **Bison**
  - Part of the Gnu software distributed by the Free Software Foundation
- Input to yacc is called **yacc specification**
  - The **.y** extention is a convention for yacc specification (example: parser.y)
- Yacc produces an entire parser module in C
  - Parser module can be compiled and linked to other modules
  - **`yacc parser.y`** (command produces **`y.tab.c`**)
  - **`cc –c y.tab.c`** (command produces **`y.tab.o`**)
  - **`y.tab.o`** can be linked to other object files

Yacc Basics

- The parser generated by yacc is a C function called **yyparse( )**
- **yyparse( )** is an LALR(1) parser
- **yyparse( )** calls **yylex( )** repeatedly to obtain the next input token
  - The function **yylex( )** can be hand-coded in C or generated by lex
- **yyparse( )** returns an integer value
  - 0 is returned if parsing succeeds and end of file is reached
  - 1 is returned if parsing fails due to a syntax error
- A yacc specification file has three sections:

declarations
%%
productions
%%
user subroutines

The **%%** *separates between sections*

Example of a Yacc Specification

The following is a yacc specification for an expression sequence

Token Declaration

```
%token NUMBER 300
%%
ExprSeq   : ExprSeq ',' Expr
          | Expr
          ;
Expr      : Expr '+' Term
          | Expr '-' Term
          | Term
          ;
Term      : Term '*' Factor
          | Term '/' Factor
          | Factor
          ;
Factor    : '(' Expr ')'
          | NUMBER
          ;
```

Production Rules

Yacc Declarations

- The first section can include a variety of declarations
- A **literal block** is C code delimited by **%{** and **%}**
  - Ordinary C declarations and **#include** directives are placed in literal block
  - Declarations made in literal block can be used in second and third sections
- Tokens should be declared in the first section
  - Tokens can either be named or quoted character literals (example, `'+'`)
  - Named tokens must be declared to distinguish them from non-terminals
- A token declaration is of the form:

```
%token token1 value1 token2 value2 . . .
```

  - The integer values define the token codes used by the scanner
  - All declared tokens should have positive code values
  - Assignment of code values to tokens is optional
  - Tokens not assigned an explicit code receive an implicit code value

Production Rules

- The productions section defines the grammar that will be parsed
- Productions are of the form:

  A : X_1 . . . X_n ;

  A is a non-terminal on the left-hand side of the production

  X_1 . . . X_n are zero or more grammar symbols on the right-hand side

  A production may span multiple lines and should terminate with a semicolon

- A sequence of productions with same LHS may be written as:

```
A : X1 . . . Xl ;
A : Y1 . . . Ym ;
A : Z1 . . . Zn ;
```

Equivalently, it may be written as:

```
A : X1 . . . Xl
  | Y1 . . . Ym
  | Z1 . . . Zn ;
```

Start Symbol and Auxiliary Code

- The LHS of first production is assumed to be the start symbol
- You may also declare the start symbol in the declaration section

  ```
  %start name
  ```

  - This will make **name** as the start symbol
  - Required when start symbol does not appear on LHS of first production
- Additional code can be provided in third section as necessary
- Example: **yylex()** can be implemented in third section
- Alternatively, **yylex()** can be generated by lex
- Error reporting and recovery routines may be added as well

Attribute Values and Semantic Actions

- Every grammar symbol has an associated **attribute value**
- An attribute value can represent anything we choose
  - The value of an expression
  - The data type of an expression
  - The translated code
- Yacc associates an attribute with every token and non-terminal
  - Token attributes are returned by the scanner in the **yylval** variable
  - Non-terminal attributes are computed while parsing
- Attribute values are pushed an popped on a **semantic stack**
  - The semantic stack operates in parallel with the parser stack
- A **semantic action** in Yacc is a code fragment delimited by { }
  - Executed when yacc matches a rule in the grammar
  - Semantic Actions can be used to make calls to semantic routines

Yacc Specification for a Simple Calculator

```
%{
#include <stdio.h>
extern int yylex();
%}
%token NUMBER 300
%%
ExprSeq : ExprSeq ',' Expr      {printf("%d\n",$3);}
        | Expr                  {printf("%d\n",$1);}
        ;
Expr    : Expr '+' Term         {$$ = $1 + $3;}
        | Expr '-' Term         {$$ = $1 - $3;}
        | Term                  {$$ = $1;}
        ;
Term    : Term '*' Factor       {$$ = $1 * $3;}
        | Term '/' Factor       {$$ = $1 / $3;}
        | Factor                {$$ = $1;}
        ;
Factor  : '(' Expr ')'          {$$ = $2;}
        | NUMBER                {$$ = $1;}
        ;
```

Literal Block

Production Rules and Semantic Actions

The $ Notation

- The **$ notation** in Yacc is used to represent the attribute values
- **$$** is the attribute value of the left-hand side nonterminal
- **$1**, **$2**, ... are the attribute values of right-hand side symbols
  - $1 is attribute value of first symbol, $2 is attribute of second, … etc
- Yacc uses the **$ notation** to locate attributes on **semantic stack**
- Consider **`A : X`**$_1$ **`. . . X`**$_n$ **`;`**
- Just before reducing the above production …
  - $1 = *stack* [*top*–*n*+1] , $2 = *stack* [*top*–*n*+2] , … , $*n* = *stack* [*top*]
- When reducing the above production …
  - $*n* , … , $2 , $1 are popped from semantic stack
  - **$$** is pushed on top of semantic stack in place of **$1**
  - *top* = *top* – *n* + 1 ; *stack* [*top*] = **$$**

Attribute Data Types and %union

- Unless explicitly specified, the default type of attributes is **int**
  - The types of **$$**, **$1**, **$2**, … is integer by default
- Suppose, we want floating-point values for numbers
  - We can change the types of **$$**, **$1**, **$2**, … to **double** by placing

    `#define YYSTYPE double` **in the literal block**
- The elements of the semantic stack are of type *YYSTYPE*
- In general, we may associate different types to different attributes
- The %union declaration identifies all possible attribute types

  `%union {` . . . *field declarations* . . . **`}`**
- The fields of a %union declaration are copied into a C union
  - *YYSTYPE* is defined to be the C union type
- Yacc puts the generated C union in the generated output file

%union and %type Declarations

- Example of a %union declaration

```
%union {
  Operator  op;       char*   name;
  Treenode* tree;     Symbol* sym;
}
```

- We associate the fields in %union with tokens and nonterminals
- A %token declaration may specify the attribute type of a token

```
%token <name> ID
%token <op>   ADDOP MULOP
```

  - The attribute of **ADDOP** and **MULOP** is an **op** of type **Operator**
  - The attribute of **ID** is a **name** of type **char***
- Type of a nonterminal is specified with a %type declaration

```
%type <tree> Expr Term Factor
```

  - The attribute of **Expr**, **Term**, and **Factor** is a **tree** of type **Treenode***

Generating Syntax Trees for Expressions

```
%union {
  Operator  op;       char*   name;
  Treenode* tree;     Symbol* sym;
}
%token <op>   ADDOP MULOP
%token <name> ID
%token <sym>  NUMBER
%type  <tree> E T F
%%
E : E ADDOP T  {$$ = new Treenode($2, $1, $3);};
E : T          {$$ = $1;};
T : T MULOP F  {$$ = new Treenode($2, $1, $3);};
T : F          {$$ = $1;};
F : '(' E ')'  {$$ = $2;};
F : ID         {$$ = (Treenode*) idTable.lookup($1);};
F : NUMBER     {$$ = (Treenode*) $1;};
```

Ambiguity and Conflicts in Yacc

- Parser generators of all varieties reject ambiguous grammars
- Ambiguous grammars fail to be LR(*k*) for any value of *k*
- Yacc will report conflicts: **shift-reduce** and **reduce-reduce**
- In some cases, a conflict is due to ambiguity in the grammar
- In other cases, a conflict is a limitation of the LALR(1) method
  - Only one token of lookahead is used by Yacc
  - A grammar may require more than one token of lookahead
- A **shift-reduce** conflict occurs when two parses exist
  - One of the parses completes a production rule - **the reduce action**
  - A second parse shifts a token - **the shift action**
- A **reduce-reduce** conflict occurs when …
  - Same lookahead token could complete two different productions

Ambiguity and Conflicts – cont'd

- Example of a **shift-reduce** conflict:

```
E : E  '+'  E
  | id  ;
```

- For the input **id + id + id** there are two parses:
  - **(id + id) + id** that uses the reduce action, and
  - **id + (id + id)** that uses the shift action
- Yacc always chooses the **shift action** in a shift-reduce conflict
- Example of a **reduce-reduce** conflict:

```
S : X | Y ;
X : A ;
Y : A ;
```

- Reduce-reduce conflicts represent mistakes in the grammar
- Yacc reduces the **first production** in a reduce-reduce conflict

More on Conflicts

- Having two productions with the same right-hand side **does not imply** a reduce-reduce conflict
- the following example does not cause any conflict
  - Lookahead token uniquely determines the production to be reduced

```
S : X b | Y c ;
X : A ;
Y : A ;
```

- Some reduce-reduce conflicts are due to the limitations of Yacc
- Reduce-reduce conflict caused by the limitation of LALR(1)

```
S : X B c | Y B d ;
X : A ;
Y : A ;
```

Using Yacc with Ambiguous Grammars

- Ambiguity, if controlled, can be of value
- An ambiguous grammar provides a shorter specification
  - Can be more natural than any equivalent unambiguous grammar
  - Produces more efficient parsers for real programming languages
- For language constructs like expressions …
  - An ambiguous grammar is more natural and more efficient

```
E : E ADDOP E | E MULOP E | … | '(' E ')' | ID
```

- Operator precedence and associativity eliminate the ambiguity
- Most binary operators, like +, – , *, and /, are left-associative
- Few, such as the exponentiation operator, are right-associative
- Few, typically the relational operators, do not associate at all
  - Two relational operators cannot be combined at all

Operator Precedence and Associativity

- Yacc provides operator precedence and associativity rules for …
  - Eliminating ambiguity and resolving shift-reduce conflicts
- Example on precedence and associativity of operators:

```
%nonassoc RELOP
%left   ADDOP
%left   MULOP
%right  EXPOP
```

- The order of declarations defines precedence of operators
  - **RELOP** has least precedence and **EXPOP** has the highest
  - ADDOP has higher precedence than RELOP
- %left declarations means **left-associative**
- %right declarations means **right-associative**
- %nonassoc declarations means **non-associative**

Resolving Conflicts

- The operator precedence and associativity resolve conflicts
- Given the two productions:

```
E : E op1 E ;
E : E op2 E ;
```

- Suppose *E* **op1** *E* is on top of parser stack and next token is **op2**
- If **op2** has a **higher precedence** than **op1**, we **shift**
- If **op2** has a **lower precedence** than **op1**, we **reduce**
- If **op2** has an **equal precedence** to **op1**, we **use associativity**
  - If **op1** and **op2** are **left-associative**, we **reduce**
  - If **op1** and **op2** are **right-associative**, we **shift**
  - If **op1** and **op2** are **non-associative**, we have a **syntax error**