Yacc and Bison

- *Yacc* (Yet Another Compiler Compiler)
  - Generates LALR(1) parsers
- *Bison*
  - Improved version of Yacc
Creating an LALR(1) Parser with Yacc/Bison

Bison Specification

- A **Bison specification** consists of three parts:
  - **Bison declarations, and C declarations within %{ %}**
  - **translation rules**
  - **user-defined auxiliary procedures**
- The **translation rules** are CFG productions with actions:
  - `production_1 { semantic action_1 }`
  - `production_2 { semantic action_2 }`
  - ...
  - `production_n { semantic action_n }`
Writing a Grammar in Yacc

- Productions in Yacc are of the form
  
  \[
  \text{Nonterminal} : \text{tokens/nonterminals } \{ \text{action} \} \\
  | \text{tokens/nonterminals } \{ \text{action} \} \\
  \ldots \\
  ;
  \]

- Tokens that are single characters can be used directly within productions, e.g. ‘+’
- Named tokens must be declared first in the declaration part using
  
  `%token TokenName`

Synthesized Attributes

- Semantic actions may refer to values of the synthesized attributes of terminals and nonterminals in a production:
  
  \[
  X : Y_1 \ Y_2 \ Y_3 \ldots Y_n \ \{ \text{action} \} \\
  - $$ refers to the value of the attribute of X
  - $i \text{ refers to the value of the attribute of } Y_i
  \]

- Example:
  
  `factor : ‘(’ expr ‘)’ \{ $$=$2; \}`

\[
\begin{align*}
\text{factor.val}&=x \\
( \text{expr.val}&=x ) & \Rightarrow $$=$2
\end{align*}
\]
Example 1: Incomplete Code

```c
%{ #include <ctype.h> %}
%token DIGIT
%
line : expr \n
expr : expr + term
     | term

term : term * factor
     | factor

factor : ( expr )
       | DIGIT

%%

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

Example of a very crude lexical analyzer invoked by the parser

Example 1: Complete Code

```c
%{ #include <ctype.h>
 #include <stdio.h>
 %}
%token DIGIT
%
line : expr \n
expr : expr + term
     | term

term : term * factor
     | factor

factor : ( expr )
       | DIGIT

%%

int yylex()
{ int c = getchar();
  if (isdigit(c))
    { yylval = c - '0';
      return DIGIT;
    }
  return c;
}
```
Example 1: Complete Code

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

int main() {
    if (yyparse() != 0 )
        fprintf(stderr, "Abnormal exit\n");
    return 0;
}

int yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

bison calculator.y
gcc calculator.tab.c
./a.out
5+8*(9+2)*3
269
^D

Tokens

- Two types of tokens: literal and symbolic
- **Literal** tokens represented using the corresponding C character constant (ASCII code)
- **Symbolic** tokens represented as numbers higher than any possible character’s code, so they will not conflict with any literal tokens
- Numbers can be forced by declaration
  ```c
  %token NUMBER 621
  ```
Tokens

• When using symbolic tokens, run Bison with `–d` option to create a C header file with definitions
• If Bison is combined with Flex, add
  `#include xxx.tab.h` in lexer file, where `xxx.y` is the source file

Symbol Values

• Both tokens and nonterminals have an associated (semantic) value
  – For token, the value is stored in the C variable `yyvval`
  – For nonterminals, the value is stored in the $$, $1, … pseudo-variables
• The associated semantic value is of type `YYSTYPE`, declared as `int` by default
• `YYSTYPE` can be redefined using the C instruction `#define`
Dealing with Ambiguous Grammars

• A description of parsing action conflicts can be obtained using the \texttt{-v} option, which produces an additional file \texttt{y.output}
• Reduce/reduce conflicts solved by using the conflicting production listed first
• Shift/reduce conflicts resolved in favor of shift

Dealing with Ambiguous Grammars

• We can also deal with ambiguous grammars by defining operator precedence levels and left/right associativity of the operators
• Example:

\begin{verbatim}
%left '+' '-'
%left '*' '/'
%right UMINUS
\end{verbatim}
Dealing with Ambiguous Grammars

• Productions are also given precedence and associativity, inherited from their rightmost nonterminal
• Example: item \( E \rightarrow E + E \) and lookahead + resolved with reduction (+ left associative)
• Example: item \( E \rightarrow E + E \) and lookahead * resolved with shift (* higher precedence)

Dealing with Ambiguous Grammars

• Can force different precedence by attaching to a production \(*\text{prec} \langle \text{terminal} \rangle\)
• Symbol \( \langle \text{terminal} \rangle \) can be a placeholder: this terminal is never used by lexical analyzer, but indicates a precedence (see Example 2 later)
Combining Bison with Flex

Yacc or Bison specification

```
yacc.y
```

Lex or Flex specification

```
lex.l
```

Yacc or Bison compiler

```
y.tab.c
y.tab.h
```

Lex or Flex compiler

```
x.tab.c
lex.yy.c
```

C compiler

```
a.out
```

input stream

```
lex.yy.c
y.tab.c
```

output stream

```
a.out
```

Example 2: Bison

```%
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines : lines expr \n\n { printf("%g\n", $2); }
| lines \n\n | /* empty */
expr : expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
| '-' expr %prec UMINUS { $$ = -$2; }
| NUMBER
%;```
Example 2: Bison (cont’d)

```c
/*
 * int main()
 * { if (yyparse() != 0)
 *     fprintf(stderr, "Abnormal exit\n");
 *     return 0;
 * }
 * int yyerror(char *s)
 * { fprintf(stderr, "Error: %s\n", s);
 * }
 */
```

Run the parser  
Invoked by parser to report parse errors

Example 2: Flex

```flex
%option noyywrap
%
#include "example2.tab.h"
extern YYSTYPE yylval;
%
number [0-9]+\.?|[0-9]*\.?[0-9]+  
%
[ \t]  { /* skip blanks */ }
{number}  { sscanf(yytext, "%lf", &yylval);  
          return NUMBER; }
\n|.    { return yytext[0]; }
```

Generated by Bison, contains  
Defined in `example2.tab.c`

```bash
bison –d example2.y
flex example2.1
gcc example2.tab.c lex.yy.c
./a.out
2.4*(6+13)
45.6
```

Defined in `example2.tab.c`
Error Recovery in Yacc

```c
%{
...;
%
}

lines : lines expr \n | lines \n | /* empty */
| error \n { printf("%g\n", $2; }
| /* empty */
| error \n { yyerror("reenter last line: ");
| yyerrok;
| }

; /* empty */

Error production: set error mode and skip input until newline

Reset parser to normal mode
```

Symbol Values

- If several types are needed for grammar symbols, a union type must be defined.
- The `%union` declaration identifies all of the possible C types that a symbol value can have.
- The field declarations are copied verbatim into a C union declaration of the type `YYSTYPE`.
- In the absence of a `%union` declaration, Bison defines `YYSTYPE` to be `int`.
Symbol Values

- Associate the types declared in `union` with specific grammar symbols using
  - the `type` declaration for nonterminal
  - the `token` declaration for tokens

Example

```plaintext
%union {
    double dval;
    char *sval;
}
...
%token <dval> REAL    // token
%token <sval> STRING
%type <dval> expr    // nonterminal
```