Lesson 10

- LR parsing with ambiguous grammars
- Error recovery in LR parsing
- Parser generators: yacc/bison
- Handling ambiguous grammars in yacc/bison
Using Ambiguous Grammars

• All grammars used in the construction of LR-parsing tables must be un-ambiguous.

• Can we create LR-parsing tables for ambiguous grammars?
  – Yes, but they will have conflicts.
  – We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  – At the end, we will have again an unambiguous grammar.
Using Ambiguous Grammars

• Why we want to use an ambiguous grammar?
  – Some of the ambiguous grammars are much natural, and a corresponding unambiguous grammar can be very complex
  – Usage of an ambiguous grammar may eliminate unnecessary reductions (single productions)
  – We may want to postpone/possibly change decisions about associativity/precedence of operators

• Example grammar:

\[
E \rightarrow E+E \mid E*E \mid (E) \mid \text{id} \quad \rightarrow
\]

\[
E \rightarrow E+T \mid T
\]

\[
T \rightarrow T*F \mid F
\]

\[
F \rightarrow (E) \mid \text{id}
\]
Sets of LR(0) Items for Ambiguous Grammar

I₀: $E' \rightarrow .E$
$E \rightarrow .E+E$
$E \rightarrow .E*E$
$E \rightarrow .(E)$
$E \rightarrow .id$

I₁: $E' \rightarrow E.$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

I₂: $E \rightarrow (\cdot E)$
$E \rightarrow \cdot E$
$E +E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

I₃: $E \rightarrow id.$

I₄: $E \rightarrow E + .E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

I₅: $E \rightarrow E \cdot *E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot id$

I₆: $E \rightarrow (E \cdot)$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

I₇: $E \rightarrow E+E.$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

I₈: $E \rightarrow E*E.$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

I₉: $E \rightarrow (E \cdot)$
SLR-Parsing Tables for Ambiguous Grammar

<table>
<thead>
<tr>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

1. $E \rightarrow E+E$
2. $E \rightarrow E*E$
3. $E \rightarrow (E)$
4. $E \rightarrow id$

FOLLOW($E$) = \{ $, +, *, )$ \}
Resolving conflicts

State \( I_7 \) has shift/reduce conflicts for symbols + and *.

After reading \text{id + id}:

\[
I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7
\]

when current token is +
- shift \( \xrightarrow{\rightarrow} \) + is right-associative
- reduce \( \xrightarrow{\rightarrow} \) + is left-associative

when current token is *
- shift \( \xrightarrow{\rightarrow} \) * has higher precedence than +
- reduce \( \xrightarrow{\rightarrow} \) + has higher precedence than *
Resolving conflicts

Also state $I_8$ has shift/reduce conflicts for symbols $+$ and $\ast$. After reading $\text{id} \ast \text{id}$:

$$
I_0 \xrightarrow{E} I_1 \xrightarrow{\ast} I_5 \xrightarrow{E} I_8
$$

- **when current token is $\ast$**
  - **shift** $\Rightarrow$ $\ast$ is right-associative
  - **reduce** $\Rightarrow$ $\ast$ is left-associative

- **when current token is $+$**
  - **shift** $\Rightarrow$ $+$ has higher precedence than $\ast$
  - **reduce** $\Rightarrow$ $\ast$ has higher precedence than $+$
Disambiguated SLR-Parsing Tables

<table>
<thead>
<tr>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>s3</td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
</tr>
</tbody>
</table>

1. $E \rightarrow E+E$
2. $E \rightarrow E*E$
3. $E \rightarrow (E)$
4. $E \rightarrow id$

$FOLLOW(E) = \{$, $+, *, )$ $\}$
Error Recovery in LR Parsing

• An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
• Errors are never detected by consulting the **goto** table.
• An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
• A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
• The SLR and LALR parsers may make several reductions before announcing an error.
• But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.
Panic Mode Error Recovery in LR Parsing

• Scan down the stack until a state s with a goto on a particular nonterminal A is found. (Get rid of everything from the stack before this state s).
• Discard zero or more input symbols until a symbol a is found that can “legitimately follow” A.
  – The symbol a is simply in FOLLOW(A), but this may not work for all situations.
• The parser stacks the nonterminal A and the state goto[s,A], and it resumes the normal parsing.
• This nonterminal A is normally is a basic programming block (there can be more than one choice for A).
  – stmt, expr, block, ...
• Symbol a can be typically ‘;’, ‘}’
Phrase-Level Error Recovery in LR Parsing

• Each empty entry in the action table is marked with a specific error routine.
• An error routine reflects the error that the user most likely will make in that case.
• An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  – missing operand
  – unbalanced right parenthesis
**Phrase-Level Error Recovery: intuition**

- Suppose $abEc$ is popped and there is no production right hand side that matches $abEc$
- If there were a rhs $aEc$, we might issue message “illegal $b$ on line $x$”
- If the rhs is $abEdc$, we might issue message “missing $d$ on line $x$”
- If the found rhs is $abc$, we might issue message “illegal $E$ on line $x$” where $E$ stands for an appropriate syntactic category represented by non-terminal $E$
### Disambiguated SLR-Parsing Tables

<table>
<thead>
<tr>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s2</td>
</tr>
<tr>
<td>1</td>
<td>s5</td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s2</td>
</tr>
<tr>
<td>5</td>
<td>s2</td>
</tr>
<tr>
<td>6</td>
<td>s9</td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>(</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

### FOLLOW(E) = \{ $, +, *, ) \}
Disambiguated SLR-Parsing Tables with error routines

<table>
<thead>
<tr>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+    *</td>
</tr>
<tr>
<td>0</td>
<td>s3   E1 E1 s2 E2 E1 1</td>
</tr>
<tr>
<td>1</td>
<td>E3   s4 s5 E3 E2 acc</td>
</tr>
<tr>
<td>2</td>
<td>s3   E1 E1 s2 E2 E1 6</td>
</tr>
<tr>
<td>3</td>
<td>r4   r4 r4 r4 r4 r4</td>
</tr>
<tr>
<td>4</td>
<td>s3   E1 E1 s2 E2 E1 7</td>
</tr>
<tr>
<td>5</td>
<td>s3   E1 E1 s2 E2 E1 8</td>
</tr>
<tr>
<td>6</td>
<td>E3   s4 s5 E3 s9 E4</td>
</tr>
<tr>
<td>7</td>
<td>r1   r1 s5 r1 r1 r1</td>
</tr>
<tr>
<td>8</td>
<td>r2   r2 r2 r2 r2 r2</td>
</tr>
<tr>
<td>9</td>
<td>r3   r3 r3 r3 r3 r3</td>
</tr>
</tbody>
</table>

1. $E \rightarrow E+E$
2. $E \rightarrow E*E$
3. $E \rightarrow (E)$
4. $E \rightarrow id$

FOLLOW($E$) = \{ $, +, *, )$ \}
Phrase-Level Error Recovery: example

• **E1: */* called when operand expected: ‘(‘ or ‘id’, but ‘+’, ‘*’ or ‘$’ found */
  – push ‘id’ (state 3) onto the stack
  – print “missing operand”

• **E2: */* called when unexpected ‘)’ is found */
  – delete ‘)’ from the input
  – print “unbalanced right parenthesis”

• **E3: */* called when expecting an operator, but ‘id’ or ‘(‘ found/
  – push ‘+’ (state 4) onto the stack
  – print “missing operator”

• **E4: */* called from state 6 when end of input is found */
  – print “missing right parenthesis”
PARSER GENERATORS
Parser Generators: ANTLR, Yacc, and Bison

• **ANTLR tool**
  – Generates LL($k$) parsers

• **Yacc (Yet Another Compiler Compiler)**
  – Generates LALR parsers

• **Bison**
  – Improved version of Yacc (GNU project)
Creating an LALR(1) Parser with Yacc/Bison

- **yacc specification** `spec.y`
- **a.out**
- **y.tab.c or spec.tab.c**
- **input stream**
- **output stream**
- **C compiler**
- **yacc or bison**

Diagram:
- **yacc specification** `spec.y` to **yacc or bison**
- **y.tab.c or spec.tab.c** to **C compiler**
- **a.out** to **output stream**
- **a.out** to **output stream**
- **input stream** to **a.out**
- **y.tab.c or spec.tab.c** to **a.out**
Yacc Specification

• A **yacc specification** consists of three parts:

• *yacc declarations*, and C declarations within %{   %}
  
  %%
  *translation rules* (productions + semantic actions)
  
  %%
  *user-defined auxiliary procedures*

• The *translation rules* are productions with actions:
  
  production$_1$   { semantic action$_1$ }
  
  production$_2$   { semantic action$_2$ }

  ...

  production$_n$   { semantic action$_n$ }
Writing a Grammar in Yacc

• Productions  \( head \rightarrow body_1 \mid body_2 \mid \ldots \mid body_n \mid \varepsilon \)
  becomes in Yacc

\[
head : \quad body_1 \{ \text{semantic action}_1 \} \\
| \quad body_2 \{ \text{semantic action}_2 \} \\
| \quad \ldots \\
| \quad /* \text{empty} */ \\
; 
\]

• Tokens (terminals) can be:
  – Quoted single characters, e.g. ‘+’, with corresponding ASCII code
  – Identifiers declared as tokens in the declaration part using
    \%token TokenName

• Nonterminals:
  – Arbitrary strings of letters and digits (not declared as tokens)
Semantic Actions and Synthesized Attributes

- **Semantic actions** are sequences of C statements, and may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:

\[
X : Y_1 \ Y_2 \ Y_3 \ ... \ Y_n \ \{ \ action \ \}
\]

- $$ refers to the value of the attribute of \( X \)
- \$i refers to the value of the attribute of \( Y_i \)

- For example

\[
\text{factor : ‘(’ expr ‘)’} \ \{ \ $$=$$2; \ \}
\]

- The values associated with tokens (terminals) are those returned by the lexer
An S-attributed Grammar for a simple desk calculator

The grammar
line → expr ‘\n’
expr → expr + term | term
term → term * factor | factor
factor → (expr) | DIGIT

%token DIGIT
%%
line : expr ‘\n’ { printf(“= %d\n”, $1); }
;
expr : expr ‘+’ term { $$ = $1 + $3; }
   | term { $$ = $1; }
   ;
term : term ‘*’ factor { $$ = $1 * $3; }
   | factor { $$ = $1; }
   ;
factor : ‘(’ expr ‘)’ { $$ = $2; }
   | DIGIT { $$ = $1 };
%%

Also results in definition of
#define DIGIT xxx

Attribute of token
Attribute of term (parent)
Attribute of term (child)
A simple desk calculator

```c
#include <ctype.h>

%token DIGIT

line : expr '\n'   { printf("= %d\n", $1); }
     ;
expr  : expr '+' term   { $$ = $1 + $3; }
      | term      { $$ = $1; }
     ;
term  : term '*' factor { $$ = $1 * $3; }
      | factor     { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
        | DIGIT      { $$ = $1; }
     ;

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

Attribute of token (stored in `yylval`)

Very simple lexical analyzer invoked by the parser

The grammar

- `line → expr \n`  
- `expr → expr + term | term`  
- `term → term * factor | factor`  
- `factor → ( expr ) | DIGIT`
Bottom-up Evaluation of S-Attributed Definitions in Yacc

<table>
<thead>
<tr>
<th>Stack</th>
<th>val</th>
<th>Input</th>
<th>Action</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>_</td>
<td>3*5+4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ 3</td>
<td>3</td>
<td>*5+4n$</td>
<td>reduce $F \rightarrow \text{digit}$</td>
<td>$$$ = $1</td>
</tr>
<tr>
<td>$ F</td>
<td>3</td>
<td>*5+4n$</td>
<td>reduce $T \rightarrow F$</td>
<td>$$$ = $1</td>
</tr>
<tr>
<td>$ T</td>
<td>3</td>
<td>*5+4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ T *$</td>
<td>3 _</td>
<td>5+4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ T * 5$</td>
<td>3 _ 5</td>
<td>+4n$</td>
<td>reduce $F \rightarrow \text{digit}$</td>
<td>$$$ = $1</td>
</tr>
<tr>
<td>$ T * F$</td>
<td>3 _ 5</td>
<td>+4n$</td>
<td>reduce $T \rightarrow T * F$</td>
<td>$$$ = $1 * $3</td>
</tr>
<tr>
<td>$ T $</td>
<td>15</td>
<td>+4n$</td>
<td>reduce $E \rightarrow T$</td>
<td>$$$ = $1</td>
</tr>
<tr>
<td>$ E $</td>
<td>15</td>
<td>+4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ E +$</td>
<td>15 _</td>
<td>4n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ E + 4$</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce $F \rightarrow \text{digit}$</td>
<td>$$$ = $1</td>
</tr>
<tr>
<td>$ E + F$</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce $T \rightarrow F$</td>
<td>$$$ = $1</td>
</tr>
<tr>
<td>$ E + T$</td>
<td>15 _ 4</td>
<td>n$</td>
<td>reduce $E \rightarrow E + T$</td>
<td>$$$ = $1 + $3</td>
</tr>
<tr>
<td>$ E $</td>
<td>19</td>
<td>n$</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>$ E \ n$</td>
<td>19 _</td>
<td>$</td>
<td>reduce $L \rightarrow E \ n$</td>
<td>print $1$</td>
</tr>
<tr>
<td>$ L $</td>
<td>19</td>
<td>$</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>
Dealing With Ambiguous Grammars

• By defining operator precedence levels and left/right associativity of the operators, we can specify ambiguous grammars in Yacc, such as

E → E+E | E–E | E*E | E/E | (E) | –E | num

• Yacc resolves conflicts, by default, as follows:
  – **Reduce/reduce** conflict: precedence to first production in the specification
  – **Shift/reduce** conflict: precedence to shift
    • ok for *if-then-else*
    • infix binary operators are handled as **right-associative!**
%token NUMBER

lines : expr 'n' { printf("= %g\n", $1); }
expr : expr '+' expr { $$ = $1 + $3; }
| expr '*' expr { $$ = $1 * $3; }
| NUMBER ;

• bison’s warning: conflicts: 4 shift/reduce

> ./PlusTimesCalculator-flat
1+2*3+4*5
= 47 /* right associate, no precedence */
Ambiguous Grammars in bison

• To define precedence levels and associativity in Yacc’s declaration part, list tokens in order of increasing precedence, prefixed by %left or %right:

  %left ‘+’ ‘–’ //same precedence, associate left
  %left ‘*’ ‘/’
  %right UMINUS

• If tokens have precedence, productions also have, equal to that of the rightmost terminal in the body. In this case:

  – **Shift/reduce** conflict are resolved with **reduce** if the production has higher precedence than the input symbol, or if they are equal and are left-associative.
Example: PlusTimesCalculator

```c
%token NUMBER    /* tokens listed in increasing order of precedence */
%left '+'
%left '*'
%
lines : expr '\n' { printf("= %g\n", $1); }
expr : expr '+' expr { $$ = $1 + $3; }
| expr '*' expr { $$ = $1 * $3; }
| NUMBER ;
%
```

- No warnings by bison

> ./PlusTimesCalculator-flat
1+2*3+4*5
= 27 /* correct precedence */

state 8

2 expr: expr .'+' expr
2 | expr .'+' expr .
3 | expr .'*' expr

'.' shift, and go to state 6

$default reduce using rule 2 (expr)
A more advanced calculator

{%
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%
%token NUMBER /* tokens listed in increasing order of precedence */
%left '+', '-'
%left '*', '/'
%right UMINUS /* fake token with highest precedence, used below */
%%
lines : lines expr \n \n { printf("= %g\n", $2); } 
 | lines \n
 | /* empty */

;expr : expr '+' expr { \n \n $ = $1 + $3; } 
 | expr '-' expr { \n \n $ = $1 - $3; } 
 | expr '*' expr { \n \n $ = $1 * $3; } 
 | expr '/' expr { \n \n $ = $1 / $3; } 
 | '(' expr ')'. { \n \n $ = $2; } 
 | '-' expr %prec UMINUS { \n \n $ = -$2;} /* rule with highest precedence */ 
 | NUMBER 

;%

Double type for attributes and yylval
A more advanced calculator (cont’d)

```c
%%
int yylex()
{ int c;
  while (((c = getchar()) == ' ')
  ;
  if ((c == '.') || isdigit(c))
  { ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
int main()
{ if (yyparse() != 0)
    fprintf(stderr, "Abnormal exit\n");
    return 0;
}
int yyerror(char *s)
{ fprintf(stderr, "Error: %s\n", s);
}
```

Crude lexical analyzer for fp doubles and arithmetic operators

Run the parser

Invoked by parser to report parse errors
Dealing With Ambiguous Grammars (summary)

• Yacc does not report about conflicts that are solved using user-defined precedences
• It reports conflicts that are resolved with default rules
• To visit the automaton and the LALR parsing table generated, execute Bison/Yacc with option \(-v\), and read the `<filename>.output` file
• This allows to see where conflicts were generated, and if the parser resolved them correctly
• Graphical representation of the automaton using Bison/Yacc with option \(-g\). Output should be in `dot` format
Combining Lex/Flex with Yacc/Bison

- **yacc specification**: `yacc.y`
- **Lex specification**: `lex.l`
  - and token definitions
  - `y.tab.h`
- **Lex or Flex compiler**
  - `lex.yy.c`
- **C compiler**
  - `y.tab.h`
  - `y.tab.c`
  - `a.out`
  - `lex.yy.c`
  - `y.tab.c`
- **Yacc or Bison compiler**
  - `y.tab.c`
  - `y.tab.h`
- **Output stream**
- **Input stream**
Lex Specification for Advanced Calculator

```c
%option noyywrap
%
#define YYSTYPE double
#include "y.tab.h"

extern double yylval;
%
number [0-9]+\.|[0-9]*\.?\d*\d+ %
[ ] { /* skip blanks */ }
{number} { sscanf(yytext, "%lf", &yylval);
  return NUMBER;
}

| Generated by Yacc, contains
| #define NUMBER xxx

| Defined in y.tab.c
```

```bash
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```bash
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```
Error Recovery in Yacc

- Based on error productions of the form \[ A \rightarrow \text{error } \alpha \]

```
{%
...%
%

lines : lines expr '\n' { printf("%g\n", $2; }
  | lines '\n'
  | /* empty */
  | error '\n'
  | yyperrokok;
%
```

Error production:
- set error mode and skip input until newline

Reset parser to normal mode