Bottom-up Parsing:

- *Table-driven* using an explicit stack (no recursion!).
- Stack can be viewed as containing both terminals and nonterminals.
- Basic operation is to *shift* terminals from the input to the stack until the right-hand side of an appropriate grammar rule is seen, and then to *reduce* the stuff on the stack that matches the rhs to the single nonterminal of the rule. Hence, bottom-up parsers are often called *shift-reduce parsers.*
Example

Grammar:

\[ E \rightarrow E + n \mid n \]

Input: \( 2 + 3 \), or \( n + n \)

Parse: (\$ is EOF in input, also bottom of stack)

<table>
<thead>
<tr>
<th>Parsing stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $</td>
<td>( n + n ) $</td>
<td>shift</td>
</tr>
<tr>
<td>2 $ ( n )</td>
<td>( + n ) $</td>
<td>reduce ( E \rightarrow n )</td>
</tr>
<tr>
<td>3 $ ( E )</td>
<td>( + n ) $</td>
<td>shift</td>
</tr>
<tr>
<td>4 $ ( E + )</td>
<td>( n ) $</td>
<td>shift</td>
</tr>
<tr>
<td>5 $ ( E + n )</td>
<td>$</td>
<td>reduce ( E \rightarrow E + n )</td>
</tr>
<tr>
<td>6 $ ( E )</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

5/15/2003
Notes:

Left recursion is not a problem in bottom-up parsing. Indeed, as we shall see, lookahead is not as serious an issue.

Keeping track of what is on the stack, however, *is* an issue (note the difference in the grammar rule reductions at lines 2 and 5 of the previous example). See later discussion on stack state.

Right recursion is actually a bit of a problem, because it makes the stack grow large (see next example).
Example

Grammar:

\[ E \rightarrow n + E \mid n \]

Input: 2 + 3, or \( n + n \)

Parse:

<table>
<thead>
<tr>
<th>Parsing stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $</td>
<td>$</td>
<td>shift</td>
</tr>
<tr>
<td>2 $ n</td>
<td>+ n $</td>
<td>shift</td>
</tr>
<tr>
<td>3 $ n +</td>
<td>n $</td>
<td>shift</td>
</tr>
<tr>
<td>4 $ n + n</td>
<td>$</td>
<td>reduce ( E \rightarrow n )</td>
</tr>
<tr>
<td>5 $ n + E</td>
<td>$</td>
<td>reduce ( E \rightarrow n + E )</td>
</tr>
<tr>
<td>6 $ E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

5/15/2003
Decision Problems in Bottom-up Parsing (parsing conflicts):

- Shift-reduce conflicts: almost always come from ambiguities, and almost always the right disambiguating rule is to shift (dangling-else).
- Reduce-reduce conflicts are more difficult; bottom-up parsers try to resolve them using Follow contexts.
- There are no shift-shift conflicts.
Dangling-else Example:

Grammar: \[ S \rightarrow I \mid o \]
\[ I \rightarrow iS \mid iSeS \]

Input: i i o e o

Parse:

<table>
<thead>
<tr>
<th>Parsing stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  $</td>
<td>i i o e o$</td>
<td>shift</td>
</tr>
<tr>
<td>2  $i$</td>
<td>i o e o$</td>
<td>shift</td>
</tr>
<tr>
<td>3  $ii$</td>
<td>o e o$</td>
<td>shift</td>
</tr>
<tr>
<td>4  $iio$</td>
<td>e o$</td>
<td>reduce $ S \rightarrow o$</td>
</tr>
<tr>
<td>5  $iiS$</td>
<td>e o$</td>
<td>shift/reduce (shift)</td>
</tr>
<tr>
<td>6  $iiSe$</td>
<td>o$</td>
<td>shift</td>
</tr>
<tr>
<td>7  $iiSeo$</td>
<td>$</td>
<td>reduce $ S \rightarrow o$</td>
</tr>
<tr>
<td>8  $iiSeS$</td>
<td>$</td>
<td>reduce $ I \rightarrow iS eS$</td>
</tr>
<tr>
<td>9  $iI$</td>
<td>$</td>
<td>reduce $ S \rightarrow I$</td>
</tr>
<tr>
<td>10 $iS$</td>
<td>$</td>
<td>reduce $ I \rightarrow iS$</td>
</tr>
<tr>
<td>11 $I$</td>
<td>$</td>
<td>reduce $ S \rightarrow I$</td>
</tr>
<tr>
<td>12 $S$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
# Reduce-reduce Example

**Grammar:** \( S \rightarrow A \, B \)

\[
A \rightarrow x \\
B \rightarrow x
\]

**Input:** \( x \, x \)

**Parse:** \((\text{Follow}(A) = \{x\}, \text{Follow}(B) = \{$})\)

<table>
<thead>
<tr>
<th>Parsing stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $</td>
<td>x x $</td>
<td>shift</td>
</tr>
</tbody>
</table>
| 2 $ x         | x $         | reduce \( A \rightarrow x \)  
              |              | (reduce \( B \rightarrow x \)) |
| 3 $ A         | x $         | shift         |
| 4 $ A x       | $           | reduce \( B \rightarrow x \)  |
| 5 $ A B       | $           | reduce \( S \rightarrow A \, B \) |
| 6 $ S         | $           | accept        |
Shift-reduce parsers differ in their use of Follow information:

- **LR(0)** parsers never consult the lookahead at all.
- **SLR(1)** parsers use the Follow sets as previously constructed.
- **LR(1)** parsers use context to split the Follow sets into subsets for different parsing paths (huge, inefficient parsers).
- **LALR(1)** parsers: like LR(1) but coarser subsets are used (achieves most of the benefit, but much smaller and faster).
Shift-reduce parsers have trouble figuring out when to accept, so acceptance is turned into a reduction by a new rule $S' \to S$ with a new start symbol $S'$. Adding this rule is called augmenting the grammar:

<table>
<thead>
<tr>
<th>Parsing stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;previous example&gt;</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5 $A B$</td>
<td>$</td>
<td>reduce $S \to A \ B$</td>
</tr>
<tr>
<td>6 $S$</td>
<td>$</td>
<td>reduce $S' \to S$</td>
</tr>
<tr>
<td>7 $S'$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

5/15/2003
Yacc

- “Yet another compiler compiler” (historical term for parser generator)
- Written by Steve Johnson at Bell Labs 1975.
- Follows same basic conventions as Lex/Flex.
- Complete Bison documentation at http://www.gnu.org/manual/bison-1.25/
Format of a Yacc/Bison definition file

{definitions}
%%
{rules}
%%
{auxiliary C functions}

Typical file extensions: .y .yacc .bison
Yacc Example

%token NUMBER
%

command : exp { printf("%d\n",$1);}  
; /* allows printing of the result */

exp : exp '+' term {$$ = $1 + $3;}  
| exp '-' term {$$ = $1 - $3;}  
| term {$$ = $1;}  
;

term : term '*' factor {$$ = $1 * $3;}  
| factor {$$ = $1;}  
;

factor : NUMBER {$$ = $1;}  
| '(' exp ')' {$$ = $2;}  
;

Yacc insists on defining tokens itself (except single chars can be matched directly).

Actions can use a “value” stack to compute results (yylval); number is position.

The value of a token must be assigned to yylval by the scanner.
Yacc Example, continued

%%

main()
{
    return yyparse();
}

int yylex(void)
{
    int c;
    while((c = getchar()) == ' ');
    if (isdigit(c)) {
        ungetc(c, stdin);
        scanf("%d", &yylval);
        return(NUMBER);
    }
    if (c == '\n') return 0; /* makes the parse stop */
    return(c);
}

void yyerror(char * s) /* prints an error message */
{
    fprintf(stderr,"%s\n", s);
}

5/15/2003
Interfacing Yacc/Bison

- Yacc generates a C file named y.tab.c (Bison: <filename>.tab.c)
- Yacc/Bison will generate a header file with token information for a scanner with the -d option: bison -d tiny.y produces tiny.tab.c and tiny.tab.h
- The .tab.h file for the above grammar looks as follows:

```c
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define NUMBER 258
extern YYSTYPE yylval;
```
Yacc/Bison Parsing Tables

With the -v option ("verbose") Yacc generates a file y.output (Bison: <filename>.output) describing its parsing actions. For example, for the grammar

\[
S \rightarrow A \ B \\
A \rightarrow x \\
B \rightarrow x
\]

the output file looks as on the next slide.
<table>
<thead>
<tr>
<th>State</th>
<th>Action/Rule</th>
</tr>
</thead>
</table>
| 0     | 'x' shift, and go to state 1 | 3
|       | S go to state 5               | 2
|       | A go to state 2               | 1
| 1     | A → 'x' (rule 2)             | 4
|       | $default reduce using rule 2 (A) |
| 2     | S → A . B (rule 1)           | 5
|       | 'x' shift, and go to state 3 | 6
|       | B go to state 4               | 7
| 3     | B → 'x' (rule 3)             | 8
|       | $default reduce using rule 3 (B) |
| 4     | S → A B (rule 1)             | 9
|       | $default reduce using rule 1 (S) |
| 5     | $ go to state 6               | 10
| 6     | $ go to state 7               | 11
| 7     | $default accept              | 12
Stack states and the description of shift-reduce parsing tables

- Represent the state of a parse by a position in a grammar rule (indicated by some symbol - a period in the text).
- Track positions using a DFA, with transitions labeled by symbols (terminals and nonterminals).
- Transitions on terminals represent shifts
- Transitions on nonterminals represent reductions ("gotos")

To be continued in next slide set….

5/15/2003
Bison Parsing Conflicts in C-Minus?

Only the dangling else:

state 95

\[\text{sel_stmt} \rightarrow \text{IF '(' expr ')'} \text{ stmt } \] (rule 29)
\[\text{sel_stmt} \rightarrow \text{IF '(' expr ')'} \text{ stmt . ELSE stmt} \] (rule 30)

ELSE shift, and go to state 98

ELSE [reduce using rule 29 (sel_stmt)]
$default$ reduce using rule 29 (sel_stmt)
Bison and TINY

- No parsing conflicts at all (no dangling else)
- Tokens are communicated to scanner by including tiny.tab.h in globals.h
- tokenString is communicated to parser by including scan.h in tiny.y
- yylval and YYSTYPE not used by scanner
#ifndef _SCAN_H_
#define _SCAN_H_

/* MAXTOKENLEN is the maximum size of a token */
#define MAXTOKENLEN 40

/* tokenString array stores the lexeme of each token */
extern char tokenString[MAXTOKENLEN+1];

/* function getToken returns the 
 * next token in source file 
 */
TokenType getToken(void);

#endif
globals.h

. . .
 ifndef YYPARSER
 /* the name of the following file may change */
#include "tiny.tab.h"
 /* ENDFILE is implicitly defined by Yacc/Bison,
 * and not included in the tab.h file
 */
#define ENDFILE 0
 endif
 . . .
 /* Yacc/Bison generates its own integer values
 * for tokens
 */
 typedef int TokenType;
 . . .

5/15/2003
tiny.tab.h

 ifndef BISON_TINY_TAB_H
 define BISON_TINY_TAB_H
 ifndef YYSTYPE
 define YYSTYPE int
 define YYSTYPE_IS_TRIVIAL 1
 endif
 define IF 257
 define THEN 258
 define ELSE 259
 define END 260
 . . .
define RPAREN 275
define SEMI 276
define ERROR 277
extern YYSTYPE yylval;
 endif /* not BISON_TINY_TAB_H */
tiny.y (part 1)

{%
#define YYPARSER /* distinguishes Yacc output
   from other code files */

#include "globals.h"
#include "util.h"
#include "scan.h"
#include "parse.h"

#define YYSTYPE TreeNode *
static char * savedName;  /* for use in assignments */
static int savedLineNo;   /* ditto */
static TreeNode * savedTree;  /* stores syntax tree
   for later return */
%
%
5/15/2003
%token IF THEN ELSE END REPEAT UNTIL READ WRITE
%token ID NUM
%token ASSIGN EQ LT PLUS MINUS TIMES OVER
%token LPAREN RPAREN SEMI ERROR
%
/* Grammar for TINY */
program : stmt_seq { savedTree = $1;}
 |
stmt_seq : stmt_seq SEMI stmt
       { YYSTYPE t = $1;
         if (t != NULL)
         { while (t->sibling != NULL) t = t->sibling;
           t->sibling = $3;
           $$ = $1; }
         else $$ = $3;
       }
       |
          stmt   { $$ = $1; }
       ;
stmt : if_stmt { $$ = $1; } 
| repeat_stmt { $$ = $1; } 
| assign_stmt { $$ = $1; } 
| read_stmt { $$ = $1; } 
| write_stmt { $$ = $1; } 
| error { $$ = NULL; } 
;

if_stmt : IF exp THEN stmt_seq END
{ $$ = newStmtNode(IfK);
  $$->child[0] = $2;
  $$->child[1] = $4;
}

| IF exp THEN stmt_seq ELSE stmt_seq END
{ $$ = newStmtNode(IfK);
  $$->child[0] = $2;
  $$->child[1] = $4;
}
assign_stmt  :  ID
    {  savedName = copyString(tokenString);  
      savedLineNo = lineno;  }
    ASSIGN  exp
    {  $$ = newStmtNode(AssignK);  
      $$->child[0] = $4;  
      $$->attr.name = savedName;  
      $$->lineno = savedLineNo;  }
    ;

    ...

factor      :  . . .
    |  NUM
    {  $$ = newExpNode(ConstK);  
      $$->attr.val = atoi(tokenString);  
    }  . . .  /* also an error production */
%%

int yyerror(char * message)
{ fprintf(listing,"Syntax error at line %d: %s\n", 
        lineno,message);
    fprintf(listing,"Current token: ");
    printToken(yychar,tokenString);
    Error = TRUE;
    return 0;
}

int yylex(void)
{ return getToken(); }

TreeNode * parse(void)
{ yyparse();
    return savedTree;
}
# Yacc/Bison internal names

<table>
<thead>
<tr>
<th>Yacc internal name</th>
<th>Meaning/Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>y.tab.c</td>
<td>Yacc output file name</td>
</tr>
<tr>
<td>y.tab.h</td>
<td>Yacc-generated header file containing token definitions</td>
</tr>
<tr>
<td>yyparse</td>
<td>Yacc parsing routine</td>
</tr>
<tr>
<td>yylval</td>
<td>value of current token in stack</td>
</tr>
<tr>
<td>yyerror</td>
<td>user-defined error message printer used by Yacc</td>
</tr>
<tr>
<td>error</td>
<td>Yacc error pseudotoken</td>
</tr>
<tr>
<td>yynerror</td>
<td>procedure that resets parser after error</td>
</tr>
<tr>
<td>yynchar</td>
<td>contains the lookahead token that caused an error</td>
</tr>
<tr>
<td>YYSTYPE</td>
<td>preprocessor symbol that defines the value type of the parsing stack</td>
</tr>
<tr>
<td>yydebug</td>
<td>variable which, if set by the user to 1, causes the generation of runtime information on parsing actions</td>
</tr>
</tbody>
</table>
# Yacc/Bison definition mechanisms

<table>
<thead>
<tr>
<th>Yacc definition mechanism</th>
<th>Meaning/Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%token</td>
<td>defines token preprocessor symbols</td>
</tr>
<tr>
<td>%start</td>
<td>defines the start nonterminal symbol</td>
</tr>
<tr>
<td>%union</td>
<td>defines a union <code>YYSTYPE</code>, allowing values of different types on parser stack</td>
</tr>
<tr>
<td>%type</td>
<td>defines the variant union type returned by a symbol</td>
</tr>
<tr>
<td>%left %right %nonassoc</td>
<td>defines the associativity and precedence (by position) of operators</td>
</tr>
</tbody>
</table>