

CS2510

Modern Programming Languages

Lecture 4

Lexical and Syntactical Analysis

Topics

- Introduction
- Lexical Analysis
- The Parsing Problem
- Recursive-Descent Parsing
- Bottom-Up Parsing

Introduction

- Language implementation systems must analyse source code, regardless of the specific implementation approach
- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF)

Syntax Analysis

The syntax analysis portion of a language processor nearly always consists of two parts:

1. A low-level part called a *lexical analyser*
 - A finite automaton based on a regular grammar
2. A high-level part called a *syntax analyzer* (parser)
 - A push-down automaton based on a context-free grammar

Advantages of BNF to describe syntax

- Provides a clear and concise syntax description
- The parser can be based directly on the BNF
- Parsers based on BNF are easy to maintain

Why separate lexical & syntactic analysis

Simplicity

- Less complex approaches can be used for lexical analysis; separating them simplifies the parser

Efficiency

- Separation allows optimisation of the lexical analyzer

Portability

- Parts of the lexical analyser may not be portable, but the parser always is portable

Lexical Analysis

- Lexical analyser is a pattern matcher for character strings
- A lexical analyser is a “front-end” for the parser
- Identifies substrings of the source program that belong together (*lexemes*)
 - Lexemes match a character pattern, which is associated with a lexical category called a *token*
 - `sum` is a lexeme; its token may be `IDENT`

Lexical Analysis (continued)

- The lexical analyser is usually a functionality the parser invokes when it needs a token
- 3 approaches to building a lexical analyzer:
 1. Write a formal description of tokens and use software to create table-driven lexical analyzer from description
 2. Design state diagram that describes tokens and write a program that implements the state diagram
 3. Design state diagram that describes tokens and manually code a table-driven implementation of the state diagram

State Diagram Design

- A naïve state diagram would have a transition from every state on every character in the source language
 - Such a diagram would be very large!

Lexical Analysis (continued)

- In many cases, transitions can be combined to simplify the state diagram
 - When recognizing an identifier, all uppercase and lowercase letters are equivalent
 - Use a character class that includes all letters
 - When recognizing an integer literal, all digits are equivalent (use a digit class)

Lexical Analysis (continued)

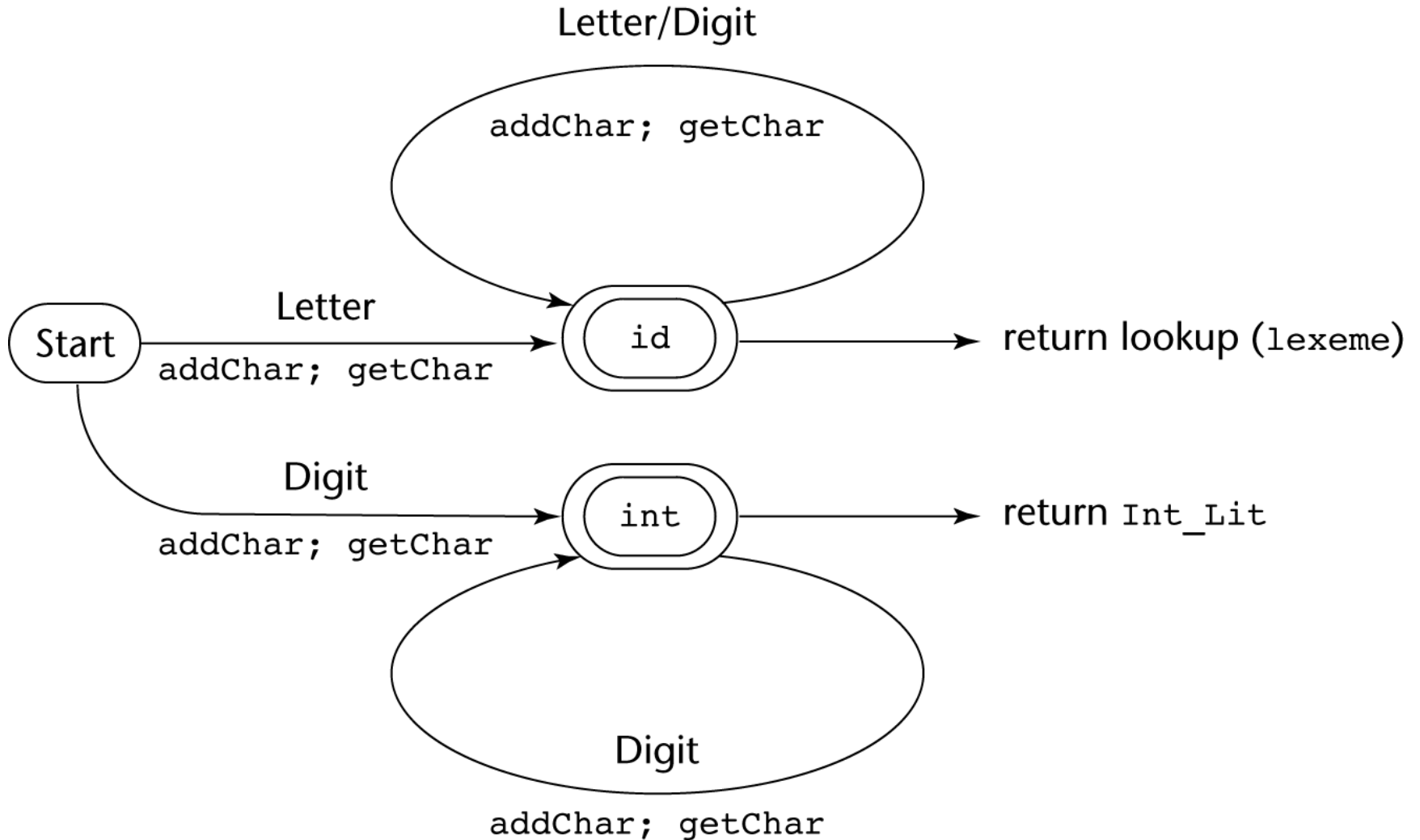
- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
 - Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analysis (continued)

Convenient utility subprograms:

- **getChar**
 - Gets the next character of input, puts it in **nextChar**, determines its class and puts the class in **charClass**
- **addChar**
 - puts the character from **nextChar** into the place the lexeme is being accumulated, **lexeme**
- **lookup**
 - determines whether the string in **lexeme** is a reserved word (returns a code)

State Diagram



Lexical Analyzer

- Implementation: `front.c` (pp. 172-177)
- Output of `front.c` when used on

`(sum + 47) / total`

Next token is: 25 Next lexeme is (

Next token is: 11 Next lexeme is sum

Next token is: 21 Next lexeme is +

Next token is: 10 Next lexeme is 47

Next token is: 26 Next lexeme is)

Next token is: 24 Next lexeme is /

Next token is: 11 Next lexeme is total

Next token is: -1 Next lexeme is EOF

The Parsing Problem

- The goals of the parser, given an input program:
 - Find all syntax errors; for each error, produce an appropriate diagnostic message and recover quickly
 - Produce the parse tree, or at least a trace of the parse tree, for the program

The Parsing Problem (continued)

Two categories of parsers

- *Top down*: produces parse tree from the root
 - Order is that of a *leftmost derivation*
 - Traces or builds the parse tree in *preorder*
- *Bottom up*: produces parse tree from leaves
 - Order is that of the reverse of a rightmost derivation

Useful parsers look ahead only one token!

The Parsing Problem (continued)

Top-down parsers

- Given a sentential form, $xA\alpha$, the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms:
 - Recursive descent – a coded implementation
 - LL parsers – table-driven implementation

The Parsing Problem (continued)

Bottom-up parsers

- Given a right sentential form, α , determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
- The most common bottom-up parsing algorithms are in the LR family

The Parsing Problem (continued)

Complexity of Parsing

- Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
- Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time ($O(n)$, where n is the length of the input)

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing (cont'd)

- A grammar for simple expressions:

`<expr> → <term> { (+ | -) <term> }`

`<term> → <factor> { (* | /) <factor> }`

`<factor> → id | int_constant | (<expr>)`

Recursive-Descent Parsing (cont'd)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`
- The process when there is only one RHS:
 - For each terminal symbol in the RHS,
 - compare it with the next input token;
 - if they match, continue, else there is an error
 - For each nonterminal symbol in the RHS,
 - call its associated parsing subprogram

Recursive-Descent Parsing (cont'd)

```
/* Function expr
   Parses strings in the language generated by the rule:
   <expr> → <term> { (+ | -) <term> }
*/

void expr() {

/* Parse the first term */
    term();

/* As long as the next token is + or -, call lex to get
   the next token and parse the next term */
    while (nextToken == ADD_OP || nextToken == SUB_OP){
        lex();
        term();
    }
}
```

Recursive-Descent Parsing (cont'd)

- This particular routine does not detect errors
- Convention:
 - Parsing routine leaves the next token in **nextToken**

Recursive-Descent Parsing (cont'd)

- Nonterminals with more than one RHS require initial process to determine which RHS to use
 - The correct RHS is chosen on the basis of the next token of input (the look-ahead)
 - The next token is compared with the first token that can be generated by each RHS until a match is found
 - If no match is found, it is a syntax error

Recursive-Descent Parsing (cont'd)

```
/* term
```

Parses strings in the language generated by the rule:

```
<term> -> <factor> { (* | /) <factor> }
```

```
*/
```

```
void term() {
```

```
/* Parse the first factor */
```

```
    factor();
```

```
/* As long as the next token is * or /, next token and parse the  
   next factor */
```

```
while (nextToken == MULT_OP || nextToken == DIV_OP) {
```

```
    lex();
```

```
    factor();
```

```
}
```

```
} /* end of function term */
```

Recursive-Descent Parsing (cont'd)

```
/* Function factor
   Parses strings in the language generated by the rule:
   <factor> -> id | (<expr>)
*/

void factor() {

    /* Determine which RHS */
    if (nextToken) == ID_CODE || nextToken == INT_CODE)

    /* For the RHS id, just call lex */
        lex();

    /* If the RHS is (<expr>) call lex to pass over the left parenthesis,
       call expr, and check for the right parenthesis */
    else if (nextToken == LP_CODE) {
        lex();
        expr();
        if (nextToken == RP_CODE)
            lex();
        else
            error();
    } /* End of else if (nextToken == ... */
    else error(); /* Neither RHS matches */
}
```

Recursive-Descent Parsing (cont'd)

Trace of the lexical and syntax analyzers on (sum + 47) / total

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
```

```
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

Recursive-Descent Parsing (cont'd)

The LL Grammar Class

- The Left Recursion Problem

- If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
- A grammar can be modified to remove direct left recursion as follows:

For each nonterminal A

1. Group the A-rules as $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

where none of the β 's begins with A

2. Replace the original A-rules with

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Recursive-Descent Parsing (cont'd)

- Another characteristic of grammars that disallows top-down parsing: lack of pairwise disjointness
 - The inability to determine the correct RHS on the basis of one token of lookahead
 - $\text{FIRST}(\alpha) = \{a \mid \alpha \rightarrow^* a\beta\}$ (If $\alpha \rightarrow^* \varepsilon$, ε is in $\text{FIRST}(\alpha)$)

Recursive-Descent Parsing (cont'd)

Pairwise Disjointness Test:

- For each nonterminal A in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$$

- Example:

$A \rightarrow a \mid bB \mid cAb$

$A \rightarrow a \mid aB$

Recursive-Descent Parsing (cont'd)

- Left factoring may solve problem

Replace

$\langle \text{variable} \rangle \rightarrow \text{identifier} \mid \text{identifier} [\langle \text{expression} \rangle]$

with

$\langle \text{variable} \rangle \rightarrow \text{identifier} \langle \text{new} \rangle$

$\langle \text{new} \rangle \rightarrow \varepsilon \mid [\langle \text{expression} \rangle]$

or

$\langle \text{variable} \rangle \rightarrow \text{identifier} [[\langle \text{expression} \rangle]]$

(the outer brackets are metasymbols of EBNF)

Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

Bottom-up Parsing (cont'd)

- Intuition about handles:

- Def: β is the *handle* of the right sentential form $\gamma = \alpha\beta w$
if and only if $S \rightarrow_{rm}^* \alpha A w \rightarrow_{rm} \alpha\beta w$
- Def: β is a *phrase* of the right sentential form γ
if and only if $S \rightarrow^* \gamma = \alpha_1 A \alpha_2 \rightarrow^+ \alpha_1 \beta \alpha_2$
- Def: β is a *simple phrase* of the right sentential form γ
if and only if $S \rightarrow^* \gamma = \alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$

Bottom-up Parsing (cont'd)

Intuition about handles (continued):

- The handle of a right sentential form is its leftmost simple phrase
- Given a parse tree, it is now easy to find the handle
- Parsing can be thought of as handle pruning

Bottom-up Parsing (cont'd)

Shift-Reduce Algorithms

- Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
- Shift is the action of moving the next token to the top of the parse stack

Bottom-up Parsing (cont'd)

Advantages of LR parsers:

- They work for nearly all grammars that describe programming languages.
- They work on a larger class of grammars than other bottom-up algorithms,
 - But are as efficient as any other bottom-up parser
- They can detect syntax errors as soon as possible
- The LR class of grammars is a superset of the class parseable by LL parsers.

Bottom-up Parsing (cont'd)

- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
 - There are only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

Bottom-up Parsing (continued)

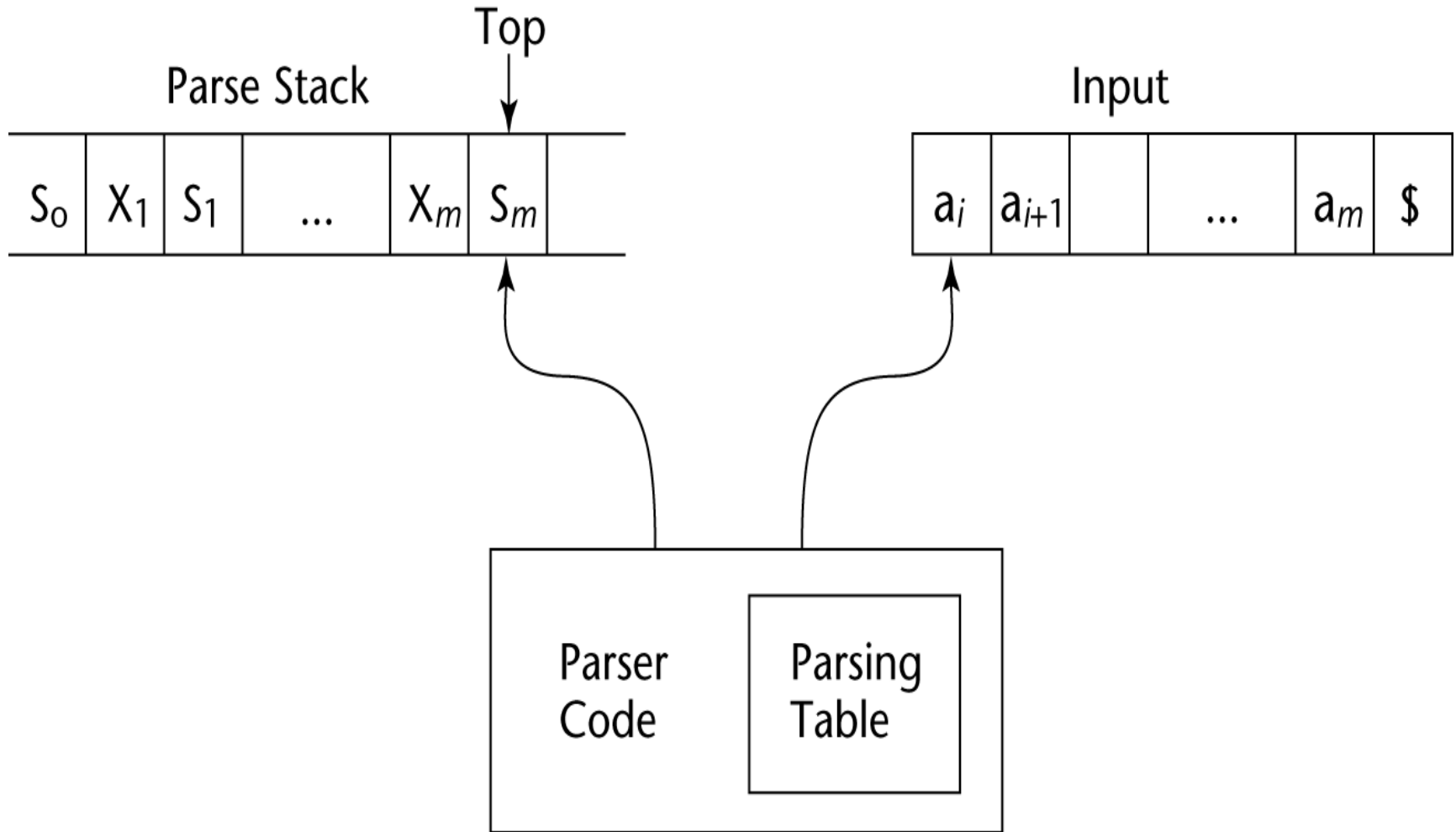
- An LR configuration stores the state of an LR parser

$$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m, a_i a_{i+1} \dots a_n \$)$$

Bottom-up Parsing (continued)

- LR parsers are table-driven, with 2 tables
 - ACTION table and a GOTO table
- ACTION table specifies the action of the parser, given the parser state and the next token
 - Rows are state names; columns are terminals
- GOTO table specifies which state to put on top of the parse stack after a reduction action is done
 - Rows are state names; columns are nonterminals

Structure of an LR Parser



Bottom-up Parsing (continued)

- Initial configuration: $(S_0, a_1 \dots a_n \$)$
- Parser actions:
 - For a Shift, the next symbol of input is pushed onto the stack, along with the state symbol that is part of the Shift specification in the ACTION table
 - For a Reduce:
 1. Remove the handle from the stack, along with its state symbols
 2. Push the LHS of the rule
 3. Push the state symbol from the GOTO table, using the state symbol just below the new LHS in the stack and the LHS of the new rule as the row and column into the GOTO table

Bottom-up Parsing (continued)

- Parser actions (continued):
 - For an Accept, the parse is complete and no errors were found.
 - For an Error, the parser calls an error-handling routine.

LR Parsing Table

	Action						Goto		
State	id	+	*	()	\$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Bottom-up Parsing (continued)

- A parser table can be generated from a given grammar with a tool, e.g., **yacc** or **bison**

Summary

- Syntax analysis: part of any language implementation
- Lexical analyzer is a pattern matcher that isolates small-scale parts of a program
 - Detects syntax errors
 - Produces a parse tree
- A recursive-descent parser is an LL parser (EBNF)
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- LR family of shift-reduce parsers is most common bottom-up parsing approach