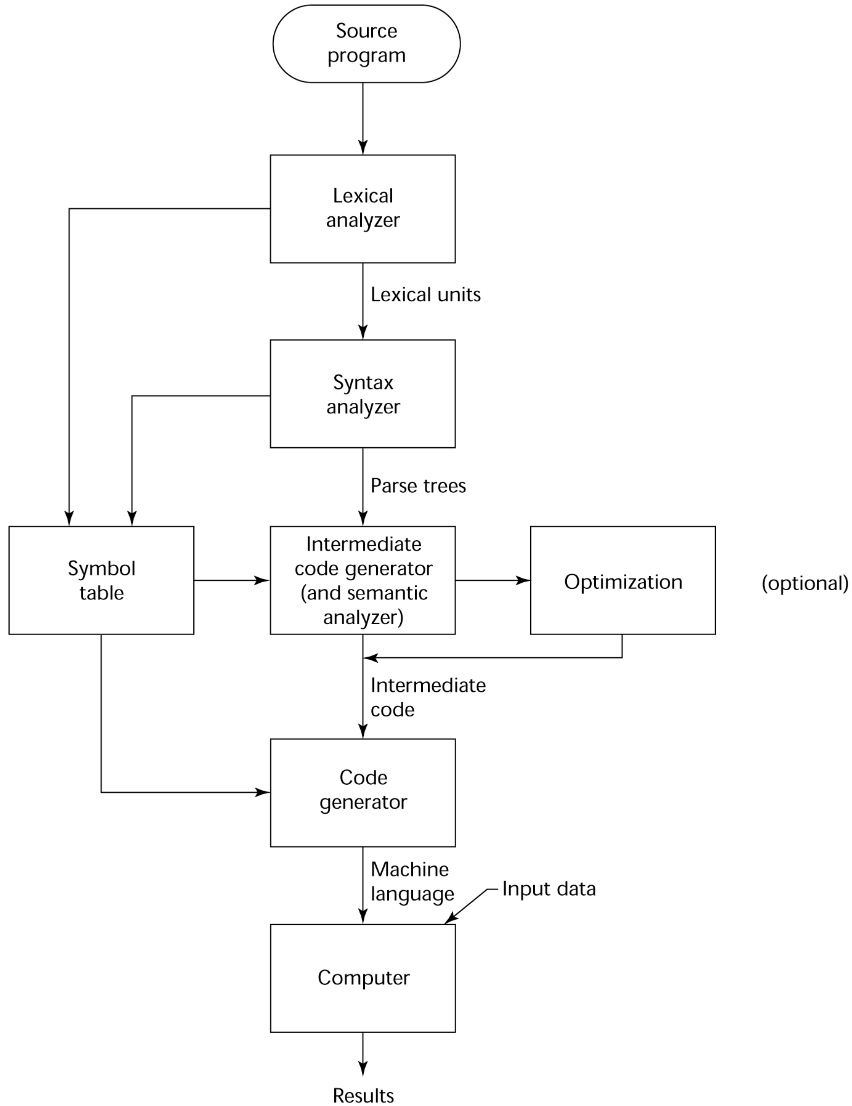
Topics

- Lexer (lexical analyzer)
- Parser (syntax analyzer)
  - Recursive-Descent Parsing
  - Bottom-Up Parsing

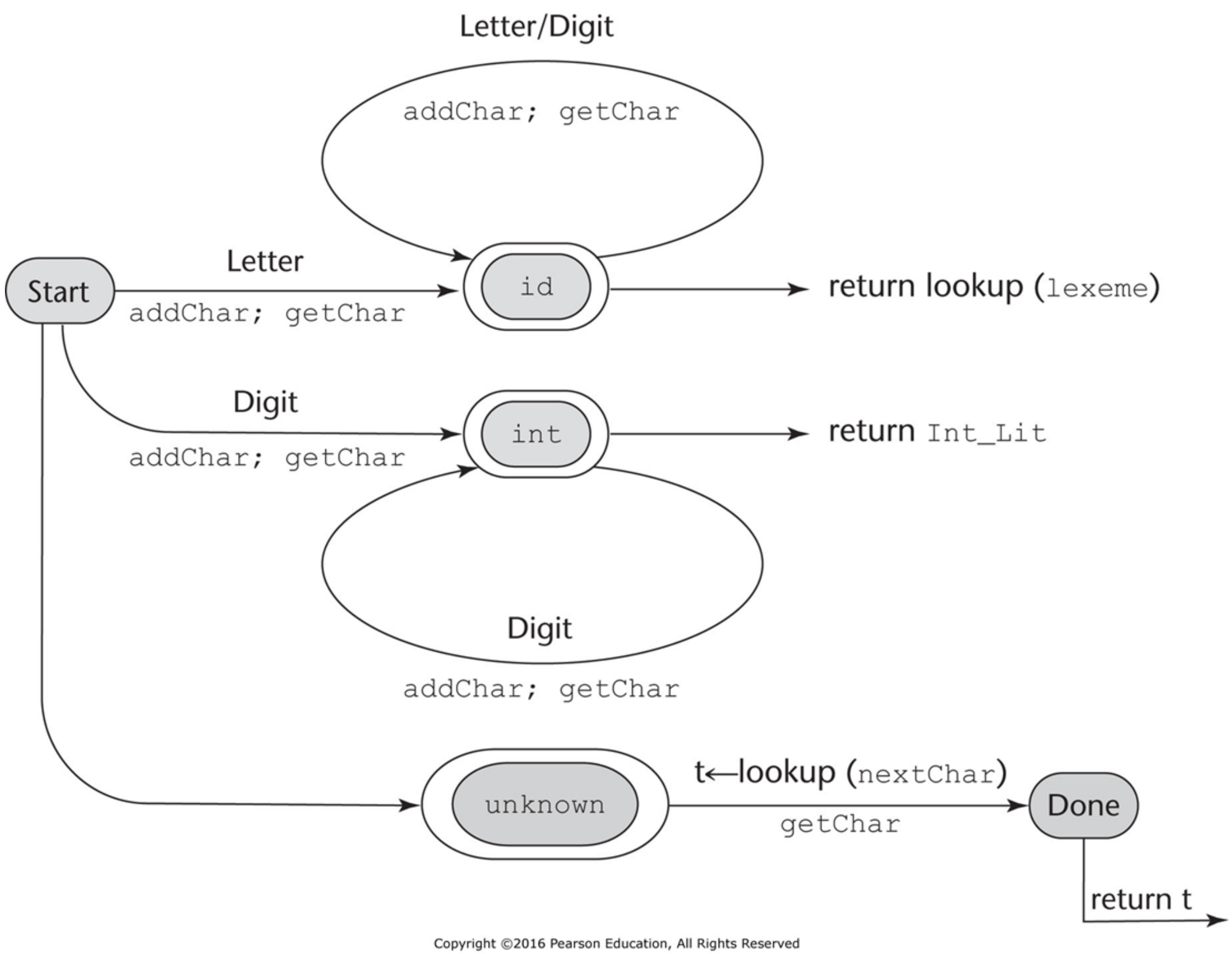
The Compilation Process



Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Lexer


Letter/Digit
addChar; getChar
Letter
Start
id
addChar; getChar
return lookup (lexeme)
Digit
int
return Int_Lit
addChar; getChar
Digit
addChar; getChar
t←lookup (nextChar)
unknown
Done
getChar
return t
Copyright ©2016 Pearson Education, All Rights Reserved

Parser

- Two categories of parsers
  - *Top down* – produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - *Bottom up* – produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation

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 (continued)

- A grammar for simple expressions:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | ( <expr> )
```

Recursive-Descent Parsing (continued)

```
/* 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 (continued)

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - 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 (continued)

```
/* 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 (continued)

- 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 $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

         where none of the β's begins with A

      2. Replace the original A-rules with

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

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

Recursive-Descent Parsing (continued)

- The inability to determine the correct RHS on the basis of one token of lookahead
  - 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

      $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$
  - Def: $FIRST(\alpha) = \{a \mid \alpha =>^* a\beta \}$

    (If $\alpha =>^* \varepsilon$, ε is in $FIRST(\alpha)$)

Recursive-Descent Parsing (continued)

- Left factoring can resolve the problem

  Replace

<variable> → identifier | identifier [<expression>]

with

<variable> → identifier <new>

<new> → ε | [<expression>]

or

<variable> → identifier [[<expression>]]

(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 rightmost derivation

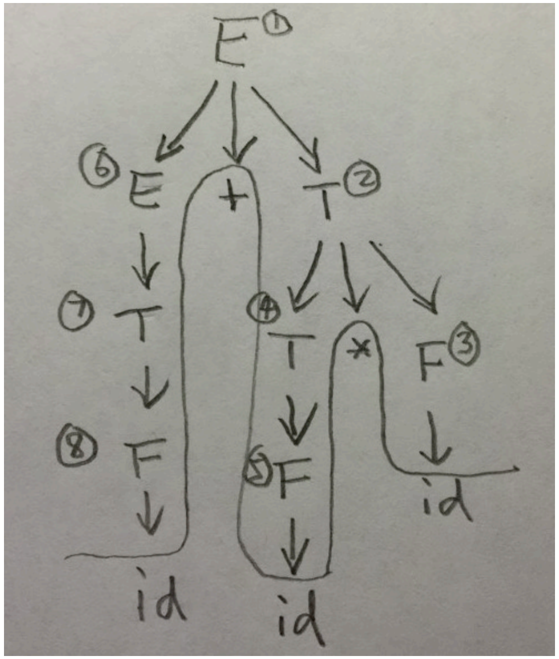
Rightmost derivation example

- Grammar:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

- Example:

$$\begin{aligned} E &\Rightarrow \underline{E + T} \\ &\Rightarrow E + \underline{T * F} \\ &\Rightarrow E + T * \underline{id} \\ &\Rightarrow E + \underline{F} * id \\ &\Rightarrow E + \underline{id} * id \\ &\Rightarrow \underline{T} + id * id \\ &\Rightarrow \underline{F} + id * id \\ &\Rightarrow \underline{id} + id * id \end{aligned}$$



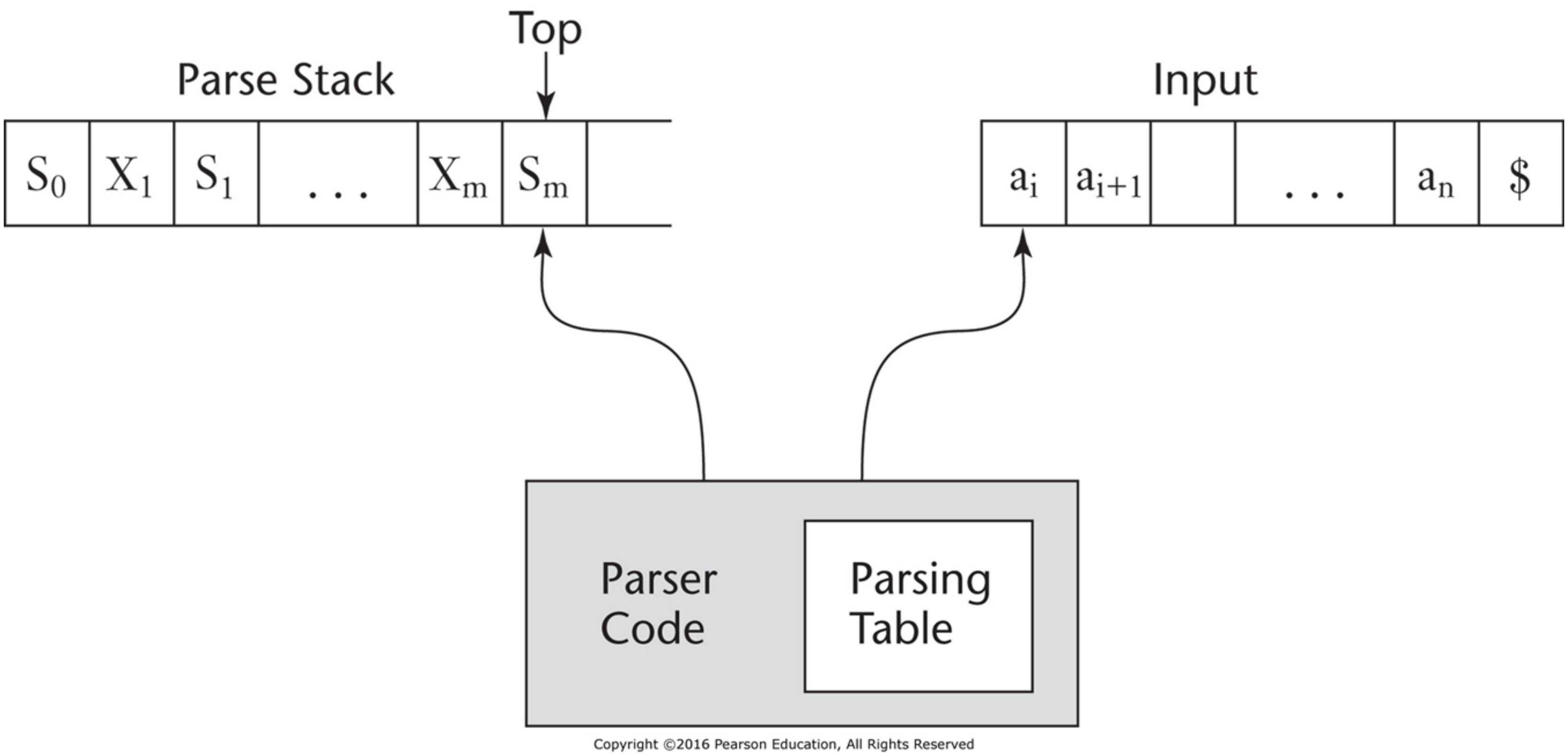
Handle and (simple) phrase

- 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 $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$

Handle and (simple) phrase (continued)

- 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

LR Parser


Top
Parse Stack
S_0
X_1
S_1
. . .
X_m
S_m
Input
a_i
a_{i+1}
. . .
a_n
$
Parser Code
Parsing Table
Copyright ©2016 Pearson Education, All Rights Reserved

Shift-Reduce Algorithms

- Initial configuration: $(S_0, a_1 \ldots a_n\$)$
- 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, remove the handle from the stack, along with its state symbols. Push the LHS of the rule. 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

LR Parsing Table

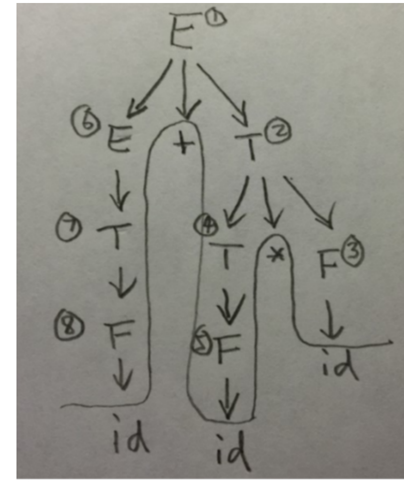
	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

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

Example

$$
\begin{aligned}
E &\Rightarrow \underline{E + T} \\
&\Rightarrow E + \underline{T * F} \\
&\Rightarrow E + T * \underline{id} \\
&\Rightarrow E + \underline{F} * id \\
&\Rightarrow E + \underline{id} * id \\
&\Rightarrow \underline{T} + id * id \\
&\Rightarrow \underline{F} + id * id \\
&\Rightarrow \underline{id} + id * id
\end{aligned}
$$



Stack	*Input*	*Action*
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Assignments

- Read assignment: Chapter 4
- Written assignment: assignment one (4%), due on February 1