LEXICAL AND SYNTAX ANALYSIS

PART II: PARSING

Chapter 4

Topics

- Introduction
- Lexical Analysis
- The Parsing Problem
- Review of BNF and EBNF Notations
- Recursive-Descent Parsing

The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly.
    - Recovery means getting back to a normal state and continues the analysis.
  - Produce the parse tree, or at least a trace of the parse tree, for the program.
    - Parse tree or its trace is used as the basis for translation.

Copyright © 2018 Pearson. All rights reserved.

The Parsing Problem (continued)

- 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:
      - Each node is visited before its branches.
      - Branches are visited in left-to-right order: corresponds to left-most derivation.
  - *Bottom up* - produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation
- Useful parsers look only one token ahead in the input

Copyright © 2018 Pearson. All rights reserved.

Advantages of Using 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

- Reasons to Separate Lexical and Syntax Analysis
  - *Simplicity* - less complex approaches can be used for lexical analysis; separating them simplifies the parser
  - *Efficiency* - separation allows optimization of the lexical analyzer
  - *Portability* - parts of the lexical analyzer may not be portable, but the parser always is portable

Summary-BNF

- BNF uses following notations:
  - Non-terminals enclosed in < and >.
  - Symbols without angle brackets are terminals.
  - ”→” means “is defined as” (some variants use “::=” or “:=” instead)
  - Rules written as $X \rightarrow Y$
    - X is LHS of rule and can only be a nonterminal.
    - Y is RHS of rule: Y can be
      a. a terminal, nonterminal, or concatenation of terminals and nonterminals, or
      b. a set of strings separated by alternation symbol |.
- Notation ε: Used to represent an empty string (a string of length 0).

An Example Derivation: Using Leftmost Derivation

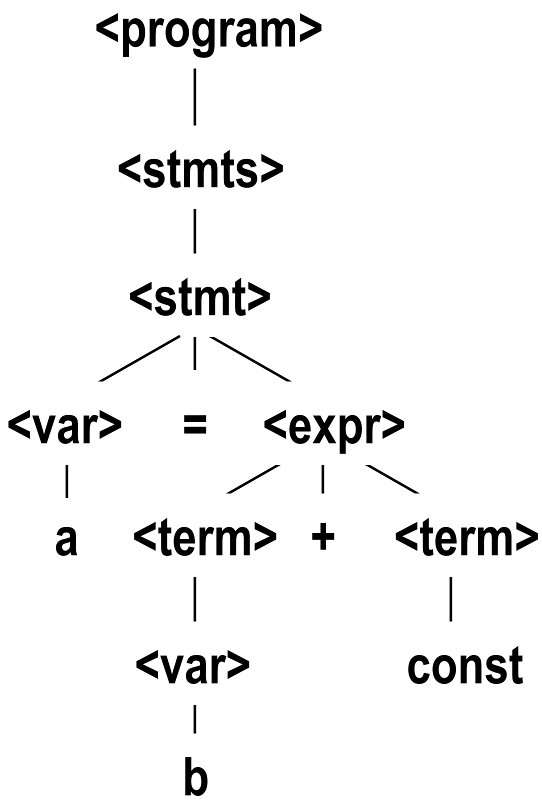
1. **<program> → <stmt_list>**
2. **< stmt_list > → <stmt>; | <stmt> ; < stmt_list >**
3. **<stmt> → <var> = <expr>**
4. **<var> → a | b | c | d**
5. **<expr> → <term> + <term> | <term> - <term>**
6. **<term> → <var> | const**

- Derivation for this sentence: a = b + const;

<program> => <stmt_list> => <stmt>;
=> **<var>** = <expr>;
=> a = **<expr>**;
=> a = **<term>** + <term>;
=> a = **<var>** + <term>;
=> a = b + **<term>**;
=> ***a = b + const;***

Parse Tree

- Grammar rules represent the hierarchical syntactic structure of the language sentences.
- Parse Trees represent the derivation steps for the hierarchical structures. Each internal node of the tree is a nonterminal and each leaf is a terminal.
- Each step in the derivation is a level in the tree
- A traversal of the tree (in inorder) is a representation of the expression that you parsed.
- Traversal of the tree for execution is in postorder.



`a = b + const`

The Parsing Problem (continued)

- Notational conventions for grammar symbols and strings
  - Terminal symbols: lower case letters at the beginning of the alphabet (a,b, c, …): small-scale syntactic constructs, called lexemes.
  - Nonterminal symbols: uppercase letter at the beginning of the alphabet (A, B, C, …): connotative names or abbreviations of language constructs, such as <while_stmt>
  - Terminals or nonterminals: uppercase letters at the end of alphabet (W, X, Y, Z)
  - Strings of terminals: lowercase letters at the end of alphabet (w, x, y, z): Sentences of a language.
  - Mixed string (terminals and/or nonterminals): lowercase Greek letters (α, β, δ, ϒ) representing RHSs of grammar rules.

Copyright © 2018 Pearson. All rights reserved.

The Parsing Problem for Top-down Parsers

- Top-down Parsers
  - Given a sentential form, $xA\alpha$,
    - x is a string of terminals
    - A is a leftmost nonterminal
    - α is a mixed string
  - 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
  - For example, A-rules are A->bB, A->cBb, and A->a
    - A top-down parser must select among these 3 rules to get the next sentential form: $xbB\alpha$, $xcBb\alpha$, or $xa\alpha$
  - **The decision to select a rule of A-rules to replace A in $xA\alpha$ is called the parsing decision problem for top-down parsers.**

The Parsing Problem for Top-down Parsers

- Most common top-down parsers choose the correct RHS for the leftmost nonterminal based on the next token of the input that matches the first symbol of RHSs.
  - For our example: the first symbols of the 3 A-rules RHSs are a, b, or c.
  - **In general, the selection is not straightforward because some RHSs may begin with a nonterminal.**
- The most common top-down parsing algorithms:
  - **Recursive descent** - a coded implementation based on BNF description.
  - **Table driven implementation** of the BNF rules.
- Both are called **LL algorithms**: First L for left-to-right scanning, and the second L is for leftmost 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

Conversion from EBNF to BNF and Vice Versa

- BNF to EBNF:

  (i) Look for recursion in grammar:

  A ::= a A | B ⇒ EBNF rule: A ::= { a } B

  (ii) Look for common string that can be factored out with grouping and options.

  A ::= a B | a ⇒ EBNF rule: A := a [B]

- EBNF to BNF:

  (i) Options: [ ]

  A ::= a [B] C ⇒ BNF rules: A' ::= a N C, N ::= B | ε

  (ii) Repetition: { }

  A ::= a { B1 B2 ... Bn } C ⇒ BNF rules: A' ::= a N C, N ::= B1 B2 ... Bn N | ε

Recursive-Descent Parsing (continued)

- A grammar for simple expressions:

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

Recursive-Descent Parsing (continued)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`
- The coding 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

Copyright © 2018 Pearson. All rights reserved.

Recursive-Descent Parsing (continued)

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

void expr() {

/* Parse the first term */
/* tracing output statements are removed */

  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();
  }
}
```

Copyright © 2018 Pearson. All rights reserved.

Recursive-Descent Parsing (continued)

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>)}
*/
void term() {
/* tracing output statements are removed */
/* Parse the first factor */
  factor();

/* As long as the next token is * or /, call
   lex to get the next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
} /* End of function term */
```

Copyright © 2018 Pearson. All rights reserved.

Recursive-Descent Parsing (continued)

- These particular routines do not have detectable errors associated with the grammar rule.
- Convention: Every parsing routine leaves the next token in **`nextToken`.**

- 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

Copyright © 2018 Pearson. All rights reserved.

Recursive-Descent Parsing (continued)

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

 void factor() {

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

 /* For the RHS id or INT_CODE, just call lex to get the next token*/
     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

- Trace of `(sum + 47) / total`

```
Call expr()
  Call term()
     Call factor()
                  Token is LP_CODE:(
                  Call expr()
                            Call term()
                                      Call factor()
                                                 Token is ID_CODE:sum
                                      Return from factor
                            Return from term
                            Token is ADD_OP:+
                            Call term()
                                      Call factor()
                                                 Token is INT_CODE:47
                                      Return from factor
                                  Return from term
                  Return from expr
                  Token is RP_CODE:)
     Return from factor
     Token is DIV_OP:/
     Call factor()
          Token is ID_CODE:total
     Return from factor
  Return from term
Return from expr
```

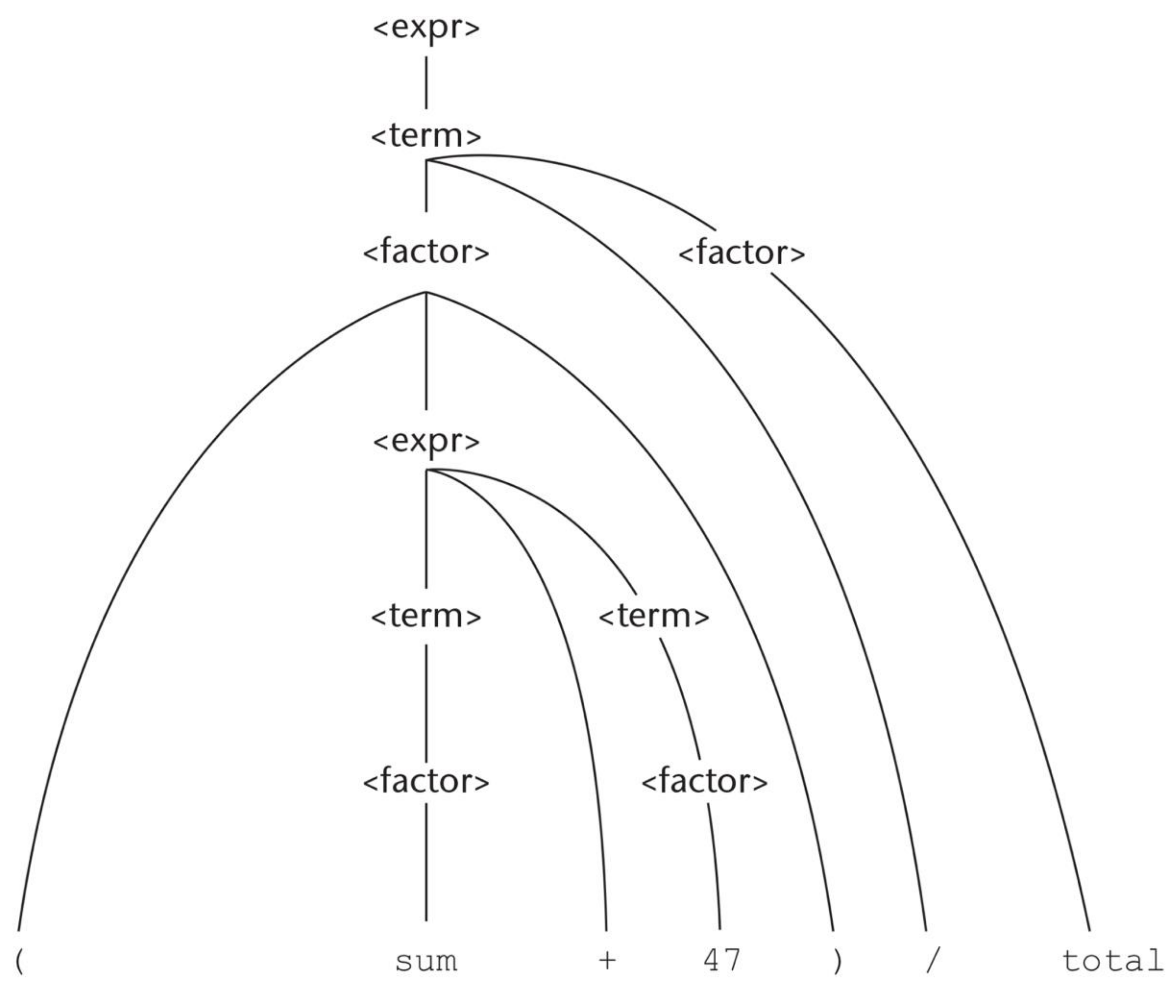
Recursive-Descent Parsing (continued)

- **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>
```

Copyright © 2018 Pearson. All rights reserved.

Figure 4-2 Parse tree for (sum + 47)/ total



Recursive-Descent Parsing (continued)
Example of Java if Statement

<ifstmt> → **if** (<boolexpr>) <statement> [**else** <statement>]

The recursive-descent subprogram for this rule follows:

```
/* Function ifstmt Parses strings in the language generated by the rule:
<ifstmt> -> if (<boolexpr>) <statement> [else <statement>]*/

void ifstmt() {
/* Be sure the first token is 'if' */
if (nextToken != IF_CODE)
   error();
else {
/* Call lex to get to the next token */
    lex();
    /* Check for the left parenthesis */
    if (nextToken != LEFT_PAREN)
        error();
```

Copyright © 2018 Pearson. All rights reserved.

Example of Java if statement (continued)

```
else {
    /* Parse the Boolean expression */
    boolexpr();
    /* Check for the right parenthesis */
    if (nextToken != RIGHT_PAREN)
        error();
    else {
    /* Parse the then clause */
        statement();
      /* If an else is next, parse the else clause */
      if (nextToken == ELSE_CODE) {
          /* Call lex to get over the else */
          lex();
          statement();
      } /* end of if (nextToken == ELSE_CODE ... */
    } /* end of else of if (nextToken != RIGHT ... */
  } /* end of else of if (nextToken != LEFT ... */
 } /* end of else of if (nextToken != IF_CODE ... */
} /* end of ifstmt */
```

Recursive-Descent Parsing (continued)

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

   where none of the β's begins with A

2. Replace the original A-rules with

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

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

Where ε specifies the empty string, called an erasure rule, because it erases its LHS from the sentential form.

Recursive-Descent Parsing (continued)

- Example: Left recursion problem

  E ::= E + T | T

  T ::= T * F | F

  F ::= (E) | id
- For the E-rules: α_1 = + T and β = T
  - E ::= TE'
  - E' ::= + T E' | ε
- For T-rules: α_1 = * F and β = F
  - T ::= F T'
  - T' ::= * F T' | ε
- Since there is no left recursion in the F-rules, they remain the same.

Recursive-Descent Parsing (continued)

- Indirect left recursion causes the same problem.
  - For example, consider the following grammar:

    A -> B a A

    B -> A b
  - Algorithm to modify a given grammar to remove indirect left recursion is not covered here.

Copyright © 2018 Pearson. All rights reserved.

Recursive-Descent Parsing (continued)

- The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness.
  - The inability to determine the correct RHS on the basis of one token of lookahead.
  - There is a simple test of non-left recursive grammar that indicates if this can be done, called pairwise disjointness test.
  - Def: $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$

    (If $\alpha \Rightarrow^* \varepsilon$, ε is in $\text{FIRST}(\alpha)$)

    Where =>* indicates 0 or more derivation steps.

  - An algorithm to compute the FIRST set is not covered here. We can rely on computing it by inspection in our examples.

Copyright © 2018 Pearson. All rights reserved.

Recursive-Descent Parsing (continued)

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

    $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$

    (The intersection of the two sets, $FIRST(\alpha_i)$ and $FIRST(\alpha_j)$ must be empty.)
  - Example:

    $A \to a \mid bB \mid Bb$

    B -> cB | d

    FIRST sets for the A-rules are clearly disjoint.
  - Example of a grammar failing the disjointness test

    A –> a B | B A b

    B -> a A | b

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 meta-symbols of EBNF)

Copyright © 2018 Pearson. All rights reserved.

Ch. 4 Example

- Prob. 1 (p.193): Perform the pairwise disjointness test for the following grammar rules
  a. `A -> aB | b | cBB`
  b. `B -> aB | bA | | aBb`
  c. `C -> aaA | b | caB`
  - Solutions

  a. FIRST(aB) = {a}, FIRST(b) = {b}, FIRST(cBB) = {c}, Passes the test

  b. FIRST(aB) = {a}, FIRST(bA) = {b}, FIRST(aBb) = {a}, Fails the test

  c. FIRST(aaA) = {a}, FIRST(b) = {b}, FIRST(caB) = {c}, Passes the test

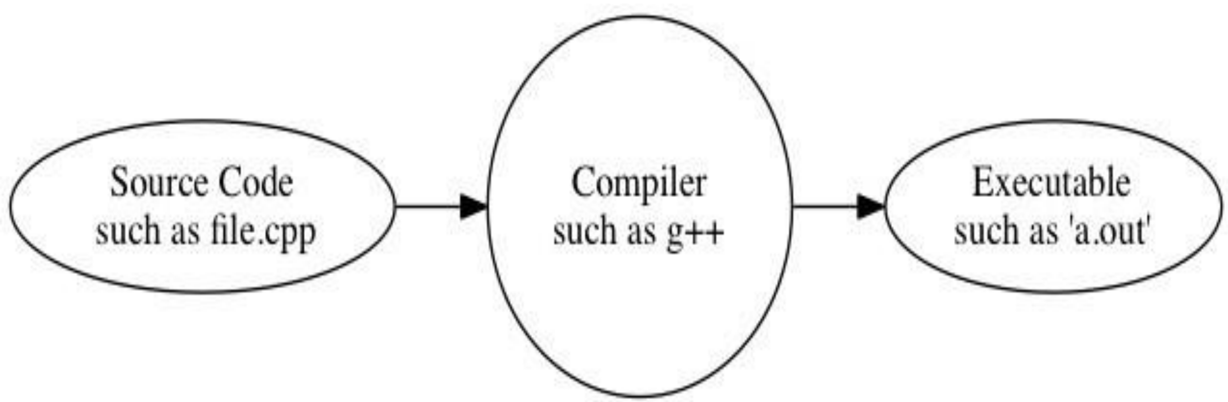
The Complexity of Parsing

- Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
  - Refers to the order of time they take to parse a string of length n.
  - Alg. Require backing up and reparsing which requires that part of the parse tree to be removed and rebuilt.
  - Reparsing is required when the parser has made a mistake in the parsing process.
- 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).
  - Generality is traded for efficiency
  - Commercial compilers are very efficient ($O(n)$).

Copyright © 2018 Pearson. All rights reserved.

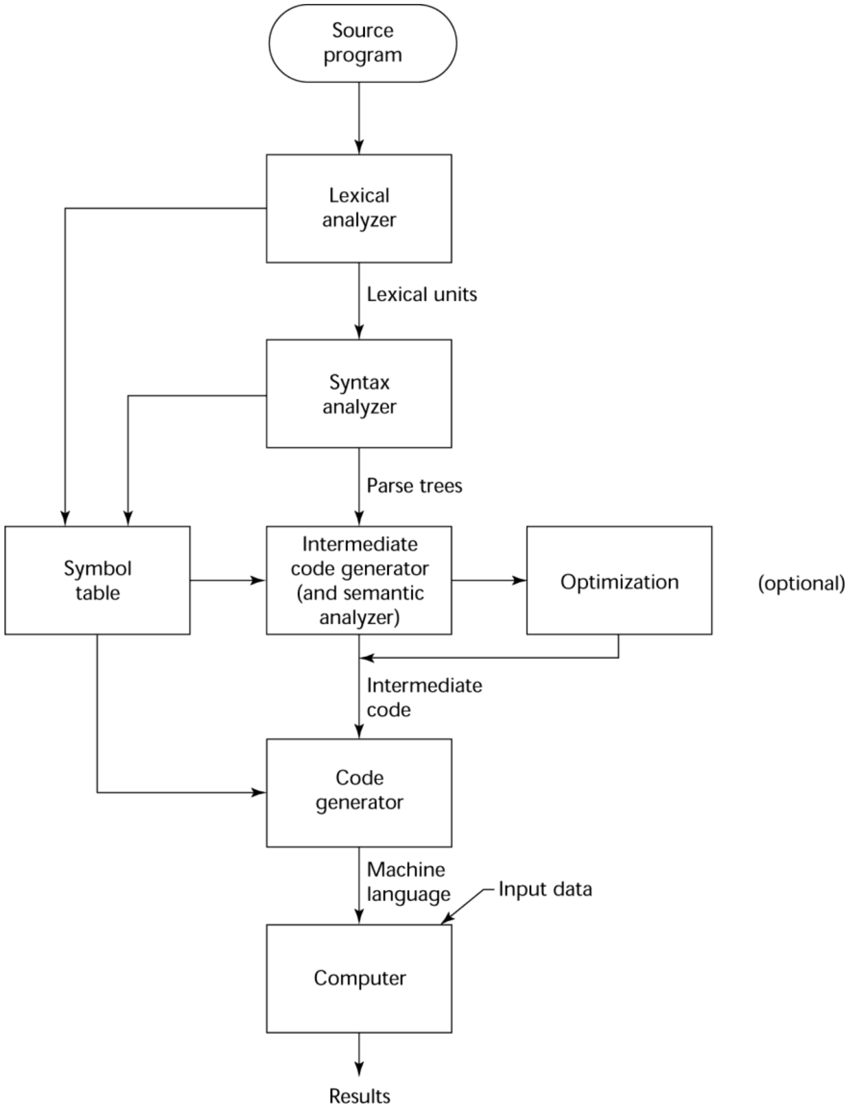
Compilation

- Translate high-level program (source language) into machine code (machine language)
  - An executable file, executable code, executable program, or simply an executable or a binary is a data file that can be executed directly by the hardware over and over again.
  - This is specific to the machine you are compiling for
    - Compiling on one machine to run on another is called “cross-compilation”
  - Slow translation, fast execution



The Compilation Process

- Compilation process has several phases:
  - Lexical analyzer converts characters in the source program into lexical units
  - Syntax analyzer transforms lexical units into parse trees which represent the syntactic structure of program
  - Semantics analyzer enforces the semantic rules of the language
  - Optimizer improves the code



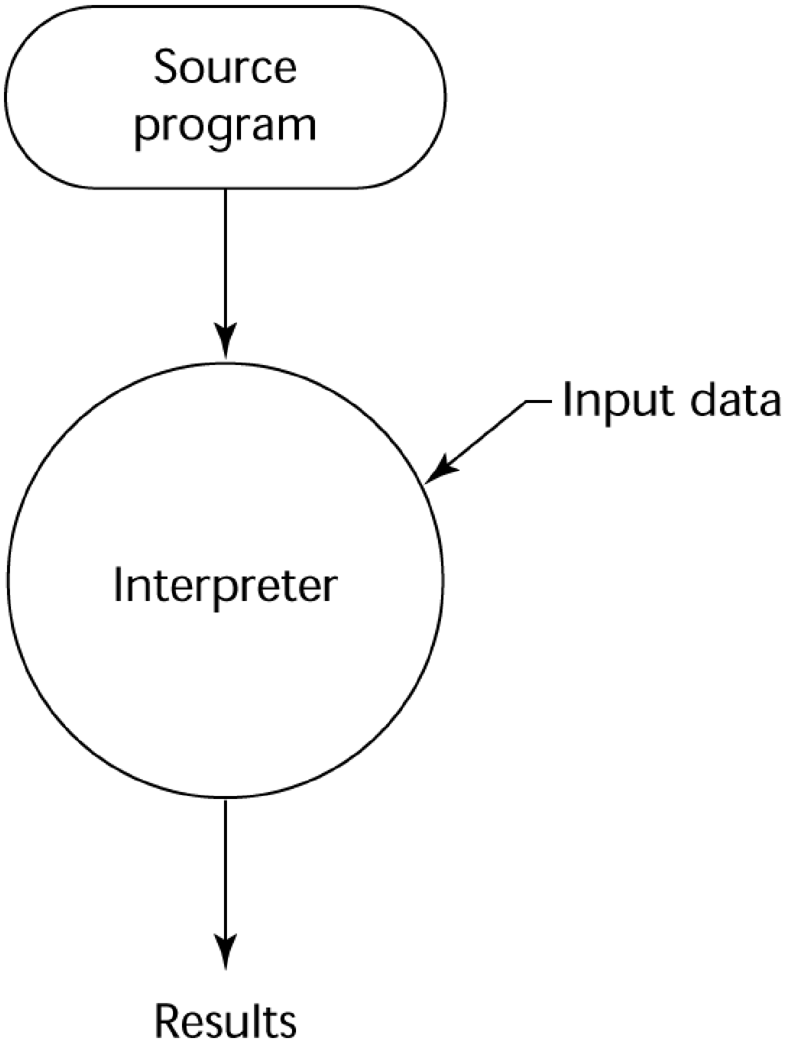
Additional Compilation Terminologies

- **Load module** (executable image): An integrated part of the operating system (which makes it essentially invisible) which loads the content of an executable code file (the user and system code together) into memory.
- **Linking and loading**: the process of collecting system program units and combining them to a user program and probably some library objects files into one executable file.
- **Run-time system**
  - Modern, high-level languages require that a program have additional support during execution. This is sometimes called the run-time system. The run-time system contains lots of code that is not written by the programmer, but was written by others and used when a program in the language is run.

Pure Interpretation

- An interpreter is a program that takes another program as input and executes it, possibly line-by-line, without translating it to an intermediate form.
  - Easier implementation of programs (run-time errors can easily and immediately be displayed)
  - Slower execution (10 to 100 times slower than compiled programs)
  - Often requires more space
  - Now rare for traditional high-level languages
  - Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

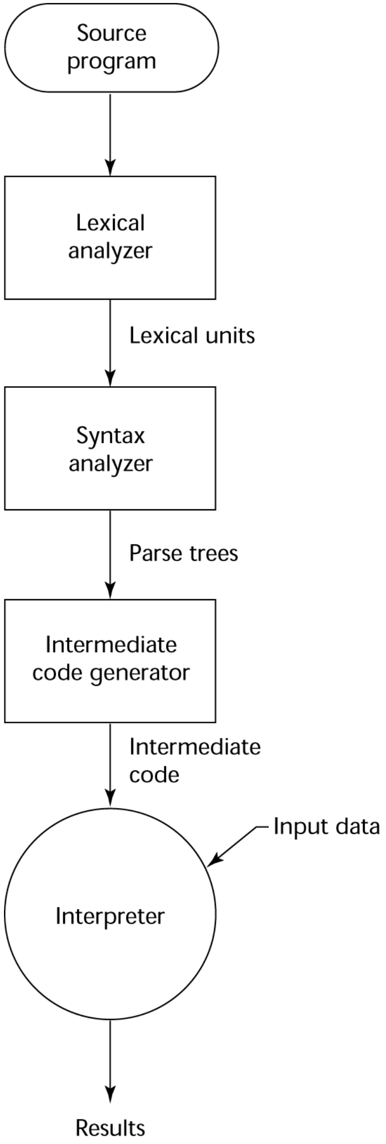
Pure Interpretation Process



Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process



Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms into machine code when they are called
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
- In essence, JIT systems are delayed compilers

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
- A recursive-descent parser is an LL parser
  - EBNF
  - Detects syntax errors
  - Produces a parse tree

Copyright © 2018 Pearson. All rights reserved.