Chapter 2

Scanning and Parsing

Syntax Analysis – Part 1

Scanner

Low-level, small scale language constructs

- Uses pattern matching to group input characters into tokens
- Removes comments
- Saves text of identifiers, numbers, strings
- Tags source locations (file, line, column) for error messages
- A *finite automaton* based on regular expressions

The scanner is usually a function that is called by the parser when it needs the next token

Syntax Analysis – Part 2

Parser

Large scale language constructs

- Expressions, statements, program units, **etc.**
- A *push-down automaton* based on a context-free grammar, or BNF

Scanning

Front–End to parser

- Pattern matcher

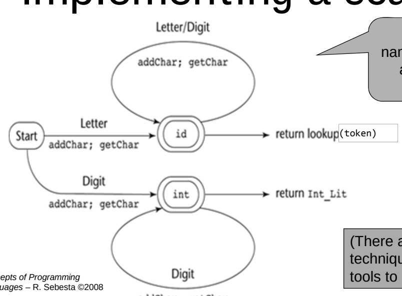
  Char strings → Tokens

Example:

```
sum = oldsum + val/100;
```

Token	Category
sum	ID
=	ASSIGN_OP
oldsum	ID
+	ADD_OP
val	ID
/	DIV_OP
100	INT_CONST
;	SEMICOLON

Implementing a Scanner



State diagram for names, reserved words, and integer literals

(There are other techniques and automated tools to do this)

Concepts of Programming Languages – R. Sebesta ©2008

Design a state-transition diagram that describes the tokens and write a subprogram to implement it

Scanning

Implementation:

```
int scan() {
  getChar();
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT) {
        addChar();
        getChar();
      }
      return lookup(token);
      break;
      …
```

Concepts of Programming Languages – R. Sebesta ©2008

Scanning

```
…
case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return INT_LIT;
      break;
  }
}
```


Parsing

Goals

- Find syntax errors, produce messages and continue
- Build Parse Tree
  - Top-Down: From the root to the leaves (left-most derivation)
  - Bottom-Up: From the leaves to the root (Reverse of right-most derivation)

Top – Down Parsers

Does a leftmost derivation

Uses preorder traversal of parse tree

Using one-token look ahead must decide which replacement rule to use

- Can't have left-recursive rules
- Rule alternatives must have unique leftmost terminal

Algorithms:

- Recursive Descent Parser using the BNF description
- LL (Left–to–right scan, Leftmost derivation) (table-driven solution)

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing

```
/* Parses strings generated by the rule:
   <expr> → <term> {(+ | -) <term>}         */
void expr() {
  term();      // Parse the first term
  /* As long as the next token is + or -, call scan to
  get the next token, and parse the next term */
  while (nextToken == ADD_OP ||
         nextToken == SUBTRACT_CODE){
    scan();
    term();
  }
}
```

Convention: Every parsing routine leaves the next token in `nextToken`

Bottom – Up Parsers

Produces the reverse of a rightmost derivation, thus avoiding the Left Recursion Problem

- LR Grammars
  (Left-to-right, Rightmost Derivation)

Advantages of LR Parsers

1. Majority of current programming languages have a grammar that can use LR Parsers
2. More efficient (now) and more grammars than other bottom–up parsers
3. Quickly detect syntax errors
4. Grammars that can be compiled by an LR Parser is a superset of LL Parsers