CS 230
Programming Languages

09 / 20 / 2022

Instructor: Michael Eckmann

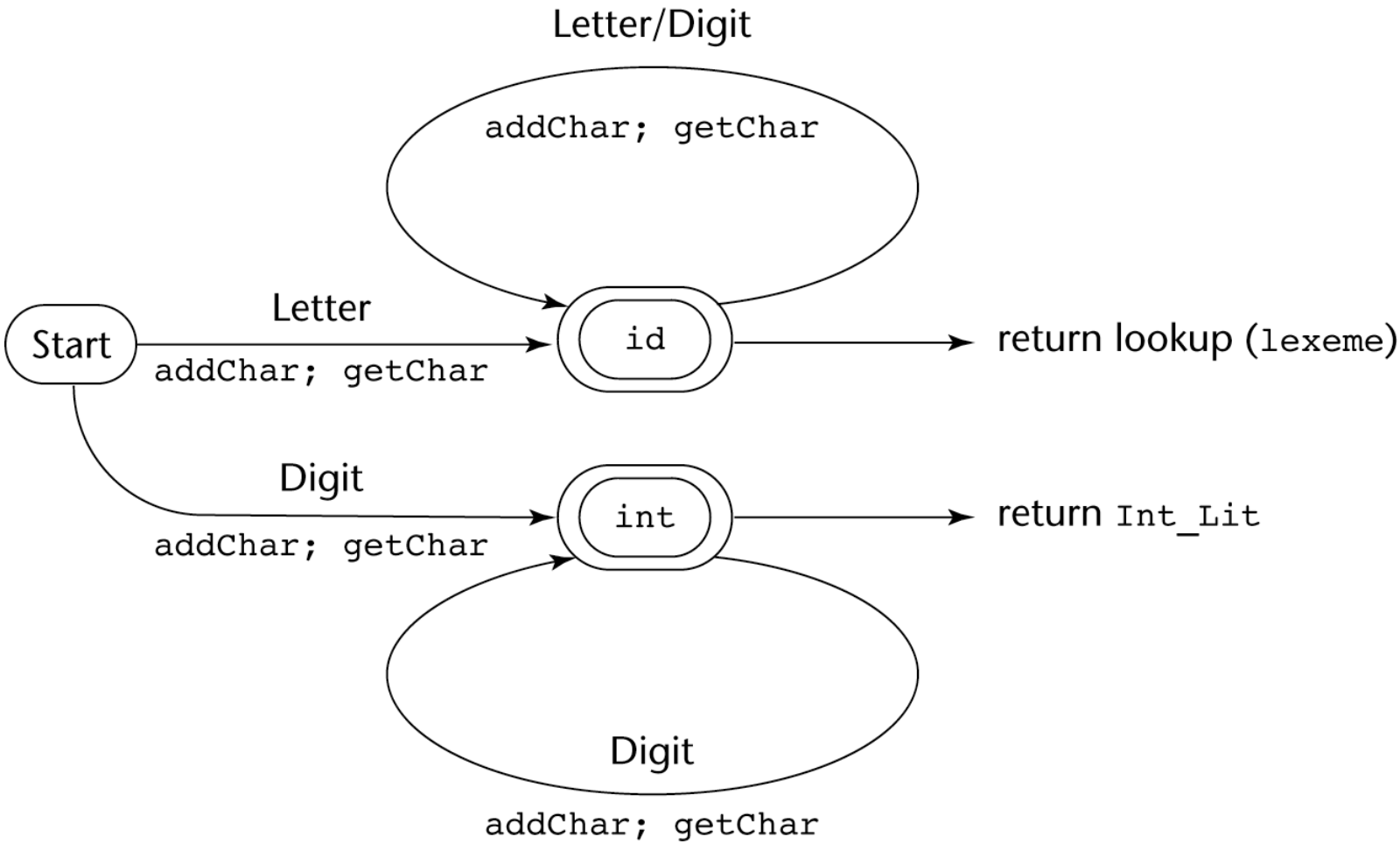
Today's Topics

- Questions? / Comments?
- Lexical Analyzer
  - Finite automaton
- Parsers
  - Top-down, recursive descent parsers

Lexical Analysis

- Lexical analyzers are finite automatons
- A finite automaton is a 5-tuple (Q, Sigma, q0, delta, A) where
  - Q is a finite set of states
  - Sigma is a finite set of input symbols
  - q0 (the start state) is an element of Q
  - delta (the transition function) is a function from QxSigma to Q
  - A (the accepting states) is a subset of Q

Lexical Analysis



Lexical Analysis

- The preceding diagram is a state diagram for a lexical analyzer for only the program names and reserved words and literals in some language.
- The states are represented by ovals
  - Q = { Start, id, int }
- The input symbols are specified along the arrows
  - Sigma = { Letter, Digit }
- The arrows are the transitions among states
  - delta can be easily drawn into a chart with the headings:

    From-state, input symbol, To-state
- The start state is Start (that is, q0 = Start)
- The accepting states are drawn with two ovals
  - A = {id, int}

Lexical Analysis

- We start at the Start state and depending on what input symbol we get (a Letter or a Digit) we move to the appropriate state. And so on until we hit an accepting state (valid.)
- If we get an input symbol while in a state that isn't on an arrow leaving that state, then we end (return).
  - However, if we're in the Start state and get something other than a Letter or a Digit as the input symbol, we do not have a program name, reserved word nor a literal which is what that finite automaton was designed to detect.

Lexical Analysis

- 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)

Lexical Analysis

- Implementation:

```
int lex() {
  getChar();
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT)
      {
        addChar();
        getChar();
      }
      return lookup(lexeme);
      break;
...
```

Lexical Analysis

```
...
case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return INT_LIT;
      break;
  }  /* End of switch */
}  /* End of function lex */
```

Parsers

- Before we get into parsing, we need to remember what leftmost derivations and rightmost derivations are.
- Anyone care to explain?

- What is a sentential form?

Parsers

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message, and recover quickly --- that is be able to find as many syntax errors in the same pass through the code (is this easy?)
  - Produce the parse tree for the program
- Two categories of parsers
  - Top down parsers – build the tree from the root down to the leaves
  - Bottom up parsers – build the tree from the leaves upwards to the root.

Parsers

- Top down - produce the parse tree, beginning at the root
  - Order is that of a leftmost derivation
- Bottom up - produce the parse tree, beginning at the leaves
  - Order is that of the *reverse* of a rightmost derivation
- The parsers we'll be dealing with look only one token ahead in the input
  - What do you think that means?

Top Down Parsers

- As input we have a possible sentence (program) in the language that may or may not be syntactically correct.
- The lexical analyzer is called by the parser to get the next lexeme / token code from the input sentence (program.)

Top Down Parsers

- A step in the life of a top down parser...
- Think of this now as during some arbitrary step in the top down parser we have a sentential form that is part of a leftmost derivation (aka *left sentential form*), the goal is to find the next sentential form in that leftmost derivation.
- Given a left sentential form xAα
  - where x is a string of terminals
  - A is a non-terminal
  - α is a string of terminals and non-terminals
- Since we want a leftmost derivation, what would we do next?

Top Down Parsers

- We have a left sentential form xAα (some string in the derivation)
- Expand A (the leftmost non-terminal) using a production that has A as its LHS.
- How might we do this?

Top Down Parsers

- We have a left sentential form xAα
- Expand A (the leftmost non-terminal) using a production that has A as its LHS.
- How might we do this?
  - We would call the lexical analyzer to get the next lexeme.
  - Then determine which production to expand based on the lexeme/token returned.
  - For grammars that are "one lookahead", exactly one RHS of A will have as its first symbol the lexeme/token returned. If there is no RHS of A with its first symbol being the lexeme/token returned --- what do we think occurred?

Top Down Parsers

- Top down parsers can be implemented as recursive descent parsers --- that is, a program based directly on the BNF description of the language.
- Or they can be represented by a parsing table that contains all the information in the BNF rules in table form.
- These top down parsers are LL algorithms --- L for left-to-right scan of the input and L for leftmost derivation.

Top Down Parsers

- LL parsers are further designated with a number *k*, as LL(*k*)
  - *k* is the number of look ahead tokens it uses to determine which production to use to expand the leftmost non-terminal
- So, how would you designate the parsers we're dealing with?

Top Down Parsers

- Left recursion is problematic for top-down parsers.
- e.g. <A> -> <A> b
- Each non-terminal is written as a function in the parser. So, a recursive descent parser function for this production would continually call itself first and never get out of the recursion.

- Left recursion is problematic for top-down parsers even if it is indirect.
- e.g. <A> -> <B> b

  <B> -> <A> a
- Bottom up parsers do not have this problem with left recursion.

Top Down Parsers

- When a nonterminal has more than one RHS, the first terminal symbol that can be generated in a derivation for each of them must be unique to that RHS. If it is not unique, then top-down parsing (with one lookahead) is impossible for that grammar.
- Because there is only one lookahead symbol, which lex would return to the parser, that one symbol has to uniquely determine which RHS to choose.

Top Down Parsers

- The pairwise disjointness test is used on grammars to determine if they have productions with more than 1 RHS starting with the same lexeme/token.
- What it does is, it determines the first symbol for every RHS of a production and for those with the same LHS, the first symbols must be different for each.
- <A> -> a <B> | b<A>b | c <C> | d
- The 4 RHSs of A here each have as their first symbol {a}, {b}, {c}, {d} respectively. These sets are all disjoint so these productions would pass the test.
- If there were another RHS say,

c <B>

- not pairwise disjoint.

Top Down Parsers

- Note: if the first symbol on the RHS is a nonterminal, we would need to expand it to determine the first symbol.
- e.g.
- <A> -> <B> a | b<A>b | c <C> | d
- <B> -> <C> f
- <C> -> s <F>
- Here, to determine the first symbol of the first RHS of A, we need to follow <B>, and then <C> to find out that it is s. So, the 4 RHSs of A here each have as their first symbol {s}, {b}, {c}, {d} respectively. These sets are all disjoint so the productions for <A> would pass the test. How about for the <B> and <C> productions?

Recursive descent example

- So called because subroutines/functions (one for each non-terminal) may be recursive and descent because it is a top-down parser.
- EBNF is well suited for these parsers because EBNF reduces the number of non-terminals (as compared to plain BNF) which reduces the number of subroutines/functions.
- A grammar for simple expressions:

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

- Anyone remember what the curly braces mean in EBNF?

Recursive descent example

- Please note: the parens in the first two productions are EBNF symbols
- The parens in the <factor> production are actually in the language being described by the grammar.

Recursive descent example

- 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 subroutine/function

Recursive descent example

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

void expr() {

   /* Parse the first term */

   term();  /* term() is it's own function
               representing the non-terminal <term>*/

/* As long as the next token is + or -, call lex to
   get the next token, and parse the next term */

    while (nextToken == PLUS_OP || nextToken == MINUS_OP)
    {
        lex();
        term();
    }
 }
```

Recursive descent example

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

void term() {

   /* Parse the first factor */

   factor();  /* factor() is it's own function
                 representing the non-terminal <factor>*/

/* As long as the next token is * or /, call lex to
   get the next token, and parse the next term */

    while (nextToken == MULT_OP || nextToken == DIV_OP)
    {
        lex();
        factor();
    }
 }
```

Recursive descent example

- ***Note: by convention each subroutine leaves the next token to be processed 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
- Let's now look at an example of how the parser would handle multiple RHS's.

Recursive descent example

```
/* Function factor
   Parses strings in the language generated by the
   rule:    <factor> -> id  |  (<expr>)  */
 void factor()
 {
   if (nextToken == IDENT)
   /* For the RHS id, call lex to get the next token*/

      lex();

   /* If the RHS is ( <expr> ) - call lex to pass over
      the leftparenthesis, call expr, and check for
      the right parenthesis */

   else if (nextToken == LEFT_PAREN)
   {
     lex();
     expr();
     if (nextToken == RIGHT_PAREN)
        lex();
     else
        error();
   }

  else error(); /* Neither RHS matches */

 }
```