Lexical and Syntax Analysis (2)

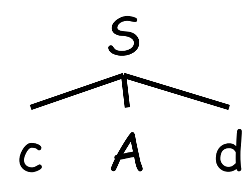
In Text: Chapter 4

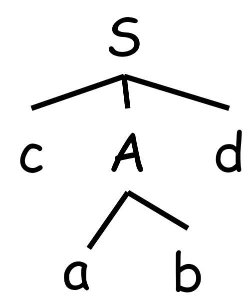
Motivating Example

- Consider the grammar
  S -> cAd
  A -> ab | a
- Input string: cad
- How to build a parse tree top-down?

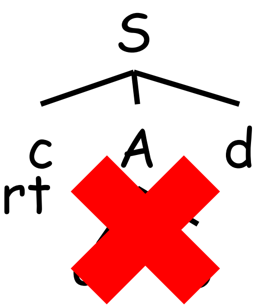
Recursive-Descent Parsing

- Initially create a tree containing a single node S (the start symbol)
- Apply the S-rule to see whether the first token matches
  - If matches, expand the tree
    - Apply the A-rule to the leftmost nonterminal A
      - Since the first token matches both alternatives (A1 and A2), randomly pick one (e.g., A1) to apply

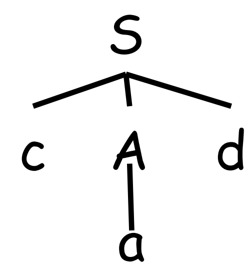




Recursive-Descent Parsing



- Since the third token d does not match b, report failure and go back to A to try another alternative
- Rollback to the state before applying A1 rule, and then apply the alternative rule
- The third token matches, so parsing is successfully done



Recursive-Descent Parsing Algorithm

Suppose we have a scanner which generates the next token as needed.
Given a string, the parsing process starts with the start symbol rule:

1. **if** there is only one RHS then
2. for each terminal in the RHS
3. compare it with the next input token
4. if they match, then continue
5. else report an error
6. for each nonterminal in the RHS
7. call its corresponding subprogram and try match
8. **else** // there is more than one RHS
9. choose the RHS based on the next input token (the lookahead)
10. for each chosen RHS
11. try match with 2-7 mentioned above
12. if no match is found, then report an error

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

- A grammar for simple expressions:

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

An Example

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

void expr() {
  printf("Enter <expr>\n");
/* 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();
  }
  printf("Exit <expr>\n");
}
```

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

An Example (cont'd)

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>) */
void term() {
  printf("Enter <term>\n");
/* 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();
  }
  printf("Exit <term>\n");
} /* End of function term */
```

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

 void factor() {
  printf("Enter <factor>\n");
 /* 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 */

  printf("Exit <factor>\n");
}
```

Token codes

```
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

Recursive-Descent Parsing (continued)

pp. 176-179

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>
…
Next token is: -1 Next lexeme is EOF
```

Key points about recursive-descent parsing

- Recursive-descent parsing may require backtracking
- LL(1) does not allow backtracking
  - By only looking at the next input token, we can always precisely decide which rule to apply
- By carefully designing a grammar, i.e., LL(1) grammar, we can avoid backtracking

Two Obstacles to LL(1)-ness

- Left recursion
  - E.g., id_list -> id_list_prefix ;
    id_list_prefix -> id_list_prefix, id | id
  - When the next token is id, which rule should we apply?
- Common prefixes
  - E.g., A -> ab | a
  - When the next token is a, which rule should we apply?

Common prefixes

- Unable to decide which RHS should use by simply checking one token of lookahead
- Pairwise Disjointness Test
  - For each nonterminal *A* with more than one RHS, for each pair of rules, the possible first characters of the strings (FIRST set) should be disjoint
    - If *A* -> $\alpha_1|\alpha_2$, then FIRST(α_1) ∩ FIRST(α_2) = ϕ

LL(1) Grammar

- Grammar which can be processed with LL(1) parser
- Non-LL grammar can be converted to LL(1) grammar via:
  - Left-recursion elimination
  - Left factoring by extracting common prefixes

Left-Recursion Elimination

- Replace left-recursion with right-recursion

id_list -> id_list_prefix ;
id_list_prefix -> id_list_prefix, id | id

=>

id_list -> id id_list_tail

id_list_tail -> ; | , id id_list_tail

Left Factoring

- Extract the common prefixes, and introduce new nonterminals as needed

  A -> ab | a

  =>

  A -> aB

  B -> b | ε

Non-LL Languages

- Simply eliminating left recursion and common prefixes does not garantee to make LL(1)
- An example in Pascal:

  stmt -> if condition then_clause else_clause

  | other_stmt

  then_clause -> then stmt

  else_clause -> else stmt | ε
- How to parse "if C1 then if C2 then S1 else S2" ?

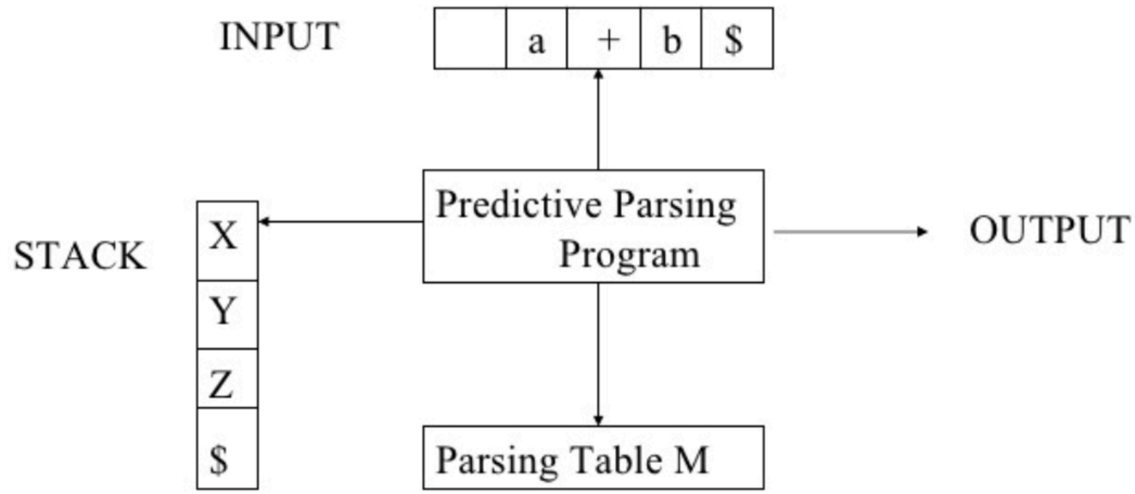
Non-LL Languages

- Define "disambiguating rule", use it together with ambiguous grammar to parse top-down
  - E.g., in the case of a conflict between two possible productions, the one to use is the one that occurs first, textually in the grammar
  - to pair the else with the nearest then
- "Disambiguating rule" can be also defined for bottom-up parsing

Table-Driven Parsing

- It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls
- The non-recursive parser looks up the production to be applied in a parsing table.
- The table can be constructed directly from LL(1) grammars

Table-Driven Parsing



- An input buffer
  - Contains the input string
  - The string can be followed by $, an end marker to indicate the end of the string
- A stack
  - Contains symbols with $ on the bottom, with the start symbol initially on the top
- A parsing table (2-dimensional array M[A, a])
- An output stream (production rules applied for derivation)

Input: a string w, a parsing table M for grammar G

Output: if w is in L(G), a leftmost derivation of w; otherwise, an error indication

Method:

```
set ip to point to the first symbol of w$
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $, then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else                          /* X is a non-terminal */
        if M[X, a] = X->Y1Y2…Yk, then
            pop X from the stack
            push Yk, …, Y2, Y1 onto the stack
            output the production X->Y1Y2…Yk
        end
        else error()
until X = $
```

An Example

- Input String: id + id * id
- Input parsing table for the following grammar

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | id

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$			$F \to (E)$		

LL Parsing

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \textbf{id}$			$F \to (E)$		

Stack	Input	Output
$E	id + id * id$	E -> TE'
$E'T	id + id * id$	T -> FT'
$E'T'F	id + id * id$	F -> id
$E'T'id	id + id * id$	
$E'T'	+ id * id$	
...		
$	$	E' -> ε