Compilers

Spring 2009

Homework 2

Solution

Problem 1 [10 points]: Predictive Top-Down Parsing
Explain why a left-recursive grammar cannot be parsed using the predictive top-down parsing algorithms.

Answer: The predictive parsing method, either based on a set of recursive functions or on its non-recursive variant using a table, rest on the premise that whenever a given terminal is seen the algorithm must choose the correct production. This means that for each terminal symbol there must be a unique production that allow the algorithm to "see" or "guess" the production correctly. Furthermore, whenever a terminal is see the corresponding production is expanded as it corresponds to the parse tree being expanded. On a left-recursive grammar there is a possibility that the algorithm will try to expand a production without consuming any of its input. In this case, guessing right does not help as it due to the recursion the tree is continuously being expanded without any advanced in the input symbols. Thus the parsing process never terminates.

Problem 2 [50 points]: Table-based LL(1) Predictive Top-Down Parsing
Consider the following CFG G = (N={S, A, B, C, D}, T={a,b,c,d}, P, S) where the set of productions P is given below:

S → A
A → BC | DBC
B → Bb | ε
C → c | ε
D → a | d

a) Is this grammar suitable to be parsed using the recursive descendent parsing method? Justify and modify the grammar if needed.
b) Compute the FIRST and FOLLOW set of non-terminal symbols of the grammar resulting from your answer in a)
c) Construct the corresponding parsing table using the predictive parsing LL method.
d) Show the stack contents, the input and the rules used during parsing for the input w = dbb

Answers:

a) No because it is left-recursive. You can expand B using a production with B as the left-most symbol without consuming any of the input terminal symbols. To eliminate this left recursion we add another non-terminal symbol, B' and productions as follows:

S → A
A → BC | DBC
B → bB' | ε
B' → bB' | ε
C → c | ε
D → a | d

b) FIRST(S) = { a, b, c, d, ε } FOLLOW(S) = { $ }
FIRST(A) = { a, b, c, d, ε } FOLLOW(A) = { $ }
FIRST(B) = { b, ε } FOLLOW(B) = { c, $ }
FIRST(B') = { b, ε } FOLLOW(B') ={ c, $ }
FIRST(C) = { c, ε } FOLLOW(C) = { $ }
FIRST(D) = { a, d} FOLLOW(D) = { b, c, $ }

Non-terminals A, B, B', C and S are all nullable.

c) The parsing table is as shown below:

	a	b	c	d	$
S	S→A	S→A	S → A	S→A	S→A
A	A→DBC	A→BC	A→BC	A→DBC	A→BC
B	B→ε	B→b B'	B→ε	B→ε	B→ε
B'	B'→ε	B'→b B'	B'→ε	B'→ε	B'→ε
C	C → ε	C → ε	C → c	C → ε	C → ε
D	D → a			D → d	

d) The stack and input are as shown below using the predictive, table-driven parsing algorithm:

STACK	INPUT	RULE/OUTPUT
$S	dbb$	
$ A	dbb$	S → A
$ CBD	dbb$	A → DBC
$ CBd	dbb$	D → d
$ CB	bb$	
$ CB'b	bb$	B → bB'
$ CB'	b$	
$ CB'b	b$	B → bB'
$ CB'	$	
$ C	$	B' → ε
$	$	C → ε
$	$	*halt*

Problem 3 [40 points]: LL parsing using Mutually Recursive Functions

Consider the following context-free grammar. (It corresponds roughly to the syntax of lists in the programming language LISP.) S is the start symbol, and the terminals are a , ( ).

S → ( )
S → a
S → (A)
A → S
A → A , S

(a) Show precisely why this grammar is not LL(1). (Hint: This will require computing some, but not all, of the FIRST and FOLLOW sets.)
(b) Rewrite this grammar to make it suitable for recursive descent parsing.
(c) On the basis of your revised grammar from part (b), write a recursive procedure S that parses this grammar by recursive descent. Your procedure may be written in C, C++, Java or pseudo-code. You may assume the existence of a global variable token that holds the next input token, a function advance() that reads the next token into token, and a function error that may be called if the input is not in the language generated by the grammar.

Solution:

(a). A grammar is LL(1) if and only if its predictive parsing table has no multiply-defined entries. Consider the right-hand sides of the first and third productions for S. The terminal (is in FIRST(()) and also in FIRST((A)). Therefore the table entry for the row labeled S and the column labeled (will have (at least) two entries for these two productions. So the grammar cannot be LL(1). (Note that there was no need to calculate any FOLLOW() sets after all!)

(b) This requires removing left-recursion and left-factoring:

S → (S'
S → a
S' →)
S' → A)
A → SA'
A' → ,SA'
A' → ε

(c) Here's C/Java-like code:

```
void s() {
  if (token == '(') {
    advance();
    s1();
  } else if (token == 'a')
    advance();
  else error();
  }
```

```
void s1() {
  if (token == ')')
    advance();
  else {
    a();
  if (token == ')')
    advance();
  else
    error();
  }
}

void a() {
  s();
  a1();
}

void a1() {
  if (token == ',') {
    advance();
    s();
    a1();
  }
}
```