COMS W4115
Programming Languages and Translators
Lecture 9: Predictive Top-Down Parsers
February 20, 2013

Lecture Outline

1. Review
2. FIRST
3. FOLLOW
4. How to construct a predictive parsing table
5. LL(1) grammars
6. Transformations on grammars

1. Review

- Top-down parsing consists of constructing or tracing a parse tree for an input string starting from the root and creating the nodes of the parse tree in preorder.
- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input string with a procedure associated with each nonterminal of the grammar. See Fig. 4.13, p. 219.
- A nonrecursive predictive parser uses an explicit stack and a parsing table to do deterministic top-down parsing.
- In this class we will develop an algorithm to construct a predictive parsing table for a large class of useful grammars called LL(1) grammars.
- For this algorithm we need two functions on grammars, FIRST and FOLLOW.

2. FIRST

- FIRST(Î±) is the set of terminal symbols that begin the strings derivable from a string of terminal and nonterminal symbols Î± in a grammar.
  If Î± can derive Îµ, then Îµ is also in FIRST(Î±).
- Algorithm to compute FIRST(*X*):

1. If *X* is a terminal, then FIRST(*X*) = { *X* }.
2. If *X* â†' Îµ is a production, then add Îµ to FIRST(*X*).
3. If *X* â†' Y_1Y_2 ... Y_k is a production for *k* â‰¥ 1, and
   for some *i* â‰¤ *k*, Y_1Y_2 ... Y_{i-1} derives the empty string, and a is in FIRST(Y_i), then add a to FIRST(*X*).
   If Y_1Y_2 ... Y_k derives the empty string, then add Îµ to FIRST(*X*).

- **Example.** Consider the grammar *G*:

```
S â†’ ( S ) S | Îµ
```

For *G*, FIRST(S) = {`(, Îµ`}.

3. FOLLOW

- FOLLOW(*A*) is the set of terminals that can appear immediately to the right of *A* in some sentential form in a grammar.
  Let us assume the string to be parsed is terminated by an end-of-string endmarker $. Then if *A* can be the rightmost symbol in some sentential form, the right endmarker $ is also in FOLLOW(*A*).
- Algorithm to compute FOLLOW(*A*) for all nonterminals *A* of a grammar:

1. Place $ in FOLLOW(*S*) where *S* is the start symbol of the grammar.
2. If *A* â†' Î±*B*Î² is a production, then add every terminal symbol a in FIRST(Î²) to FOLLOW(*B*).
3. If there is a production *A* â†' Î±*B*, or a production *A* â†' Î±*B*Î², where FIRST(Î²) contains Îµ,
   then add every symbol in FOLLOW(*A*) to FOLLOW(*B*).

- **Example.** For *G* above, FOLLOW(S) = {) , $}.

4. How to Construct a Predictive Parsing Table

- Input: Grammar *G*.
- Output: Predictive parsing table *M*.
- Method:

```
for (each production A â†’ Î± in G) {
   for (each terminal a in FIRST(Î±))
      add A â†’ Î± to M[A, a];
   if (Îµ is in FIRST(Î±))
      for (each symbol b in FOLLOW(A))
```

```
        add A â†’ Î± to M[A, b];
}
make each undefined entry of M be error;
```

- **Example 1.** Predictive parsing table for the grammar:

```
S â†’ +SS | *SS | a;
```

FIRST(S) = {+, *, a}
FOLLOW(S) = {+, *, a, $}

	Input Symbol			
Nonterminal	a	+	*	$
S	S â†’ a	S â†’ +SS	S â†’ *SS	**error**

- **Example 2.** Predictive parsing table for the grammar:

```
S â†’ ( S ) S | Îµ
```

FIRST(S) = {(, Îµ}
FOLLOW(S) = {), $}

	Input Symbol		
Nonterminal	(	)	$
S	S â†’ (S)S	S â†’ Îµ	S â†’ Îµ

- **Example 3.** Predictive parsing table for the grammar:

```
S â†’ S ( S ) | Îµ
```

FIRST(S) = {(, Îµ}
FOLLOW(S) = {(,), $}

	Input Symbol		
Nonterminal	(	)	$
S	S â†’ S(S) S â†’ Îµ	S â†’ Îµ	S â†’ Îµ

5. LL(1) Grammars

- A grammar is LL(1) iff whenever A â†’ Î± | Î² are two distinct productions, the following three conditions hold:

1. For no terminal a do both Î± and Î² derive strings beginning with a.
2. At most one of Î± and Î² can derive the empty string.
3. If Î² derives the empty string, then Î± does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if Î± derives the empty string, then Î² does not derive any string beginning with a terminal in FOLLOW(A).

- We can use the algorithm above to construct a predictive parsing table with uniquely defined entries for any LL(1) grammar.
- The first "L" in LL(1) means scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one symbol of lookahead to make each parsing action decision.

6. Transformations on Grammars

- Two common language-preserving transformations are often applied to grammars to try to make them parsable by top-down methods. These are eliminating left recursion and left factoring.
- Eliminating left recursion:
  - Replace

```
expr â†’  expr + term
     |  term
```

by

```
expr  â†’  term expr'

expr' â†’  + term expr'
      |  Îµ
```

- Left factoring:
- Replace

```
stmt â†’ if ( expr ) stmt else stmt
     | if (expr) stmt
     | other
```

by

```
stmt  â†’ if ( expr ) stmt stmt'
      | other

stmt' â†’ else stmt
      | Îµ
```

7. Practice Problems

Consider the following grammar G for Boolean expressions:

```
B â†’ B or T | T
T â†’ T and F | F
F â†’ not B | ( B ) | true | false
```

- What precedence and associativity does this grammar give to the operators `and, or, not`?
- Compute FIRST and FOLLOW for each nonterminal in G.
- Transform G into an equivalent LL(1) grammar G'.
- Construct a predictive parsing table for G'.
- Show how your predictive parser processes the input string

  `true and not false or true`

  Draw the parse tree traced out by your parser.

8. Reading

- ALSU, Section 4.4.

---

aho@cs.columbia.edu