Lecture 13

Top-Down Parser

A top-down parser starts with the root of the parse tree. The root node is labeled with the goal (start) symbol of the grammar. The top-down parsing algorithm proceeds as follows:

1. Construct the root node of the parse tree
2. Repeat until the fringe of the parse tree matches input string
    a. At a node labeled *A*, select a production with *A* on its lhs
    b. for each symbol on its rhs, construct the appropriate child
    c. When a terminal symbol is added to the fringe and it does not match the fringe, backtrack

The key is picking right production in step a. That choice should be guided by the input string. Let's try parsing using this algorithm using the expression grammar.

x – 2 * y

P	*Sentential Form*	*input*
-	*Goal*	- x – 2 * y
1	*expr*	- x – 2 * y
2	*expr* + *term*	- x – 2 * y
4	*term* + *term*	- x – 2 * y
7	*factor* + *term*	- x – 2 * y
9	<id,x> + *term*	- x – 2 * y
9	<id,x> + *term*	x - – 2 * y

This worked well except that "–" does not match "+". The parser made the wrong choice of production to use at step 2. The parser must backtrack and use a different production.

P	*Sentential Form*	*input*
-	*Goal*	- x – 2 * y
1	*expr*	- x – 2 * y
3	*expr* – *term*	- x – 2 * y
4	*term* – *term*	- x – 2 * y
7	*factor* – *term*	- x – 2 * y
9	<id,x> – *term*	- x – 2 * y
9	<id,x> – *term*	x - – 2 * y

This time the "–" and "–" matched. We can advance past "–" to look at "2". Now, we need to expand "*term*"

P	*Sentential Form*	*input*
-	<id,x> – *term*	x – - 2 * y
7	<id,x> – *factor*	x – - 2 * y
9	<id,x> – <num,2>	x – - 2 * y
-	<id,x> – <num,2>	x – 2 - * y

The 2's match but the expansion terminated too soon because there is still unconsumed input and there are no non-terminals to expand in the sentential form ⇒ Need to backtrack.

P	*Sentential Form*	*input*
-	<id,x> – *term*	x – - 2 * y
5	<id,x> – *term* * *factor*	x – - 2 * y
7	<id,x> – *factor* * *factor*	x – - 2 * y
8	<id,x> – <num,2> * *factor*	x – - 2 * y
-	<id,x> – <num,2> * *factor*	x – 2 - * y
-	<id,x> – <num,2> * *factor*	x – 2 * - y
9	<id,x> – <num,2> * <id,y>	x – 2 * - y
-	<id,x> – <num,2> * <id,y>	x – 2 * y -

This time the parser met with success. All of the input matched.

Left Recursion

Consider another possible parse:

P	*Sentential Form*	*input*
-	*Goal*	- x – 2 * y
1	*expr*	- x – 2 * y
2	*expr* + *term*	- x – 2 * y
2	*expr* + *term* + *term*	- x – 2 * y
2	*expr* + *term* + *term* + *term*	- x – 2 * y
2	*expr* + *term* + *term* + *term* + ….	- x – 2 * y

Parser is using productions but no input is being consumed.

Top-down parsers cannot handle left-recursive grammars. Formally, a grammar is left recursive if $ A Î NT such that $ a derivation $A \Rightarrow^* A\,a$, for some string a Î (NT ∪ T)*.

Our expression grammar is left recursive. This can lead to non-termination in a top-down parser. Non-termination is bad in any part of a compiler! For a top-down parser, any recursion must be a *right recursion.* We would like to convert left recursion to right To remove left recursion, we can transform the grammar. Consider a grammar fragment:

$$A \;?\; A\,a \mid b$$

where neither a nor b starts with A. We can rewrite this as:

$$A \;?\; b\,A'$$

$$A' \;?\; a\,A' \mid e$$

where A' is a new non-terminal. This grammar accepts the same language but uses only right recursion. The expression grammar we have been using contains two cases of left-recursion. Applying the transformation yields

expr	**?**	*term expr'*
expr'	**?**	*+ term expr'*
	|	*– term expr'*
term	**?**	*factor term'*
term'	**?**	** factor term'*
	|	*/ factor term'*
	|	*e*

These fragments use only right recursion. They retain the original left associativity. A top-down parser will terminate using them.

Predictive Parsing

If a top down parser picks the wrong production, it may need to backtrack. Alternative is to *look-ahead* in input and use context to pick the production to use correctly. How much look-ahead is needed? In general, an arbitrarily large amount of look-ahead symbols are required.. Fortunately, large classes of CFGs can be parsed with limited lookahead. Most programming languages constructs fall in those subclasses

The basic idea in predictive parsing is: given $\boldsymbol{A} \rightarrow \alpha \mid \beta$, the parser should be able to choose between α and β. To accomplish this, the parser needs FIRST and FOLLOW sets.

Definition: FIRST sets: for some rhs $\alpha \in \boldsymbol{G}$, define FIRST($\alpha$) as the set of tokens that appear as the first symbol in some string that derives from α. That is, $\underline{\mathbf{x}} \in$ FIRST(α) *iff* $\alpha \Rightarrow^{*} \underline{\mathbf{x}}\gamma$, for some γ.