Construction of the LL(1) Table

Agata Gruza

// How the LL(1) table is constructed, showing a few lines of the LL(1) table, and discussing what the entries in those lines imply for parsing.

Overview

LL(1) table is an essential part of the recursive descent parsing algorithm*. This table need to be created if we want to finish implementation of the parser. Construction of LL(1) table is based on two significant algorithms: First and Follow. LL(1) table consists of rows and columns and the size of this table is m x n where m is the number of rows and n is the number of columns.

Row headings are nonterminals of the grammar. If we have correctly constructed LL(1) table and if grammar is unambiguous then the number of distinct nonterminals (no duplicates) is the number of rows in LL(1) table. One row can have many different rules and one rule number can repeat many times in one row. In each row has to be at least one rule number. The maximum number of rows is a number of rules (upperbound).

Columns headings on the other hand are terminals (tokens) of the grammar. If grammar is unambiguous then the number of distinct terminals (no duplicates) is the number of columns in LL(1) table (plus eof if grammar does not contain one). Columns headings are “potential” lookahead tokens. In each column has to be at least one rule number.

We want to construct LL(1) that is unambiguous, which mean there is only one way to do each of the parses.

For what do we need LL(1) Table?

LL(1) table is an important part of the building the compiler. One of the steps to get there is parsing. We assume we have context free grammar (CFG) for the programming language. The approach here is to expend a nonterminal in a top-down and left most manner. We can establish the lookahead token by making use of LL(1) table. To do that we need to build LL(1) table and use the reference while constructing recursive descent parser.

Example

The following is an example of grammar where capital letters are nonterminals and lower case letters and eof are terminals (tokens):

rule-1. <S> → <A> <B> **eof**
rule-2 <B> → **b** <B>
rule-3 <B> → **d**
rule-4. <B> → **e** <A> **c**
rule-5. <B> → ε
rule-6. <A> → **b** <C> <A>
rule-7. <A> → ε
rule-8. <C> → **c**

Then the LL(1) table would look like:

	eof	**b**	**d**	**e**	**c**
S		1			
B	5	2	3	4	
A		6,7	7	7	7
C					8

Numbers in a table cells are rule numbers For example if I want to expand A during parsing and my lookahead is c, then I should apply rule 7. On the other hand, if I want to expand A during parsing and my lookahead token is eof then I would report a syntax error because there is no rule that applies. If lookahead token is b and I want to expand A then there are two rules that will apply here: rule 6 and 7. We have conflict and we do not know which rule to apply, They are ways to “go around” this problem but right now this grammar is not LL(1) and therefore can not be used for a recursive descent parser based on single .lookahead token.

Some things we can do to clean up a grammar are to*:
● Remove ambiguity.

● Remove left recursion.
● Remove common prefixes.
● Use an English description to provide a solution to an inherent problem.
● Rewrite rules.
● Use more than one symbol of lookahead.

To avoid ambiguity I will choose one of the rules (lets say 6), discard rule 7 and explain why I did choose rule 6.

We can observe that when we expend rule-5. <B> → ε and r rule-7. <A> → ε we get ε. What does it mean? It means that we must evaluate First(ε). Nevertheless, the empty string is not a token and the scanner will never return the empty string as a lookahead token. To “go around” this problem we need to use Follow() algorithm. As a result in row A and B all entries with column headers that are tokes in Follow(B) and Follow(A) would contain 5 and 7 respectively.

Summary

LL(1) table is a crucial part of the recursive descent parsing algorithm. Construction of this table is based on two significant algorithms: First and Follow. LL(1) table needs to be created if we want to finish implementation of the parser.