One-State Pushdown Recognizer of LL(1) Grammars
Ramon A. Mata-Toledo Ph.D.
Rollins College
Saturday, October 23, 2021

To avoid backtracking when designing a parser for an LL(1) grammar (a term yet to be defined), it may be necessary to impose constraints on the languages. Some of these constraints are:

1) Given production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \ldots \mid \alpha_n$

   FIRST $(\alpha_i) \cap$ FIRST $(\alpha_j) = \varnothing$ for all $i \neq j$ where FIRST(α_i) is the set of all terminals that can appear in the first position of sentences derived from nonterminal A. That is, two productions with the same left corner cannot have common elements. This is the main reason for eliminating self-left recursive productions.

2) For every nonterminal A which is nullable, that is, $A \xrightarrow{*} \varepsilon$ (the empty string/sentence), the set of its initial symbols must be disjoint from the set of symbols that may follow any sequence generated from this nonterminal. That is,

$$\textbf{FIRST (A)} \cap \textbf{FOLLOW(A)} = \varnothing$$

   Calculate FIRST(α_i) and FOLLOW(A) following the rules already studied in class. Mark all the nonterminal symbols that are nullable.

   To implement an LL(1) grammar which recognizer can be simulated by one-state stack or pushdown it is necessary to define new set that will help us to determine which production to use when processing the current token (symbol). The new sets are called the SELECT sets. There is a SELECT set for each production. *Notice that it is to the entire production to which we are calculating the set*. These sets are constructed using the FIRST and FOLLOW sets of the LL(1) grammar we are working on. If there are more than one production with the same nonterminal symbol, there may be more than one SELECT set for that nonterminal symbol.

LL(1) Grammars

Given a production $A \rightarrow \alpha$ where α is a string of terminal and nonterminal, we define the **SELECT** set of the production rule as follows:

- If the production *is not nullable* then **SELECT**$(A \rightarrow \alpha)$ = FIRST(α)
- If the production *is nullable* then **SELECT**$(A \rightarrow \alpha)$ = FIRST$(\alpha) \bigcup$ FOLLOW(A) - $\{ \varepsilon \}$

A context-free grammar is called an LL(1) grammar if the left-hand side of each production of the grammar consists of single nonterminal and *if all productions with the same left-hand side have disjoint SELECT sets*. That is, if the following condition is satisfied for all productions of the grammar:

$$\textbf{SELECT}(A_i) \cap \textbf{SELECT}(A_j) = \varnothing \textbf{ for } i \neq j \textbf{ where } \varnothing \textbf{ is the empty set}$$

The expression "disjoint sets' means that the sets have no common elements. The implementation of the one-state pushdown (stack) recognizer requires *the use of at most one production for each combination of stack and input symbol when implemented as shown below*. As indicated in previous lectures, the first "L" in LL(1) stands for scanning the input from *left to right*, *the second* "L" for producing a leftmost derivation (equivalent to a top-down derivation), and the "1" for using the current symbol only (the *lookahead*) at each step to make parsing action decisions.

Implementation of an LL(1) grammar recognizer

Given a context-free grammar, a *one-state pushdown* recognizer can be specified as follows:

- **The input set** is the terminal set of the grammar augmented with an endmarker (┤). The endmarker is used to indicate the end of the input string. It can be considered the equivalent of EOF.

- Valid symbols that can be pushed into the **stack** are:
  - the bottom marker (▼).

- the nonterminal symbols of the grammar, and
- those terminal symbols that appear in the righthand side of productions in positions other than at the extreme left.

- The **starting stack** consists of:
  - the bottom marker (▼) . This symbol indicates that the stack is "empty."
  - the starting nonterminal.

- The control mechanism is described by a one-state control table that has:
  - rows labeled with stack symbols
  - columns with input symbols and the end marker ┤ which indicates the end of the string; it is the equivalent of EOL. This symbol is sometimes called by its French name (le truc or the "thing". Some of you will call it the "thingy')
  - entries as indicated below.

Action Rules or When do I Apply a Production Rule?

***A production is applied whenever the symbol on top of the stack is the left-hand side of a production and the current input symbol is in the selection set of this production*. The rules to apply are the following**:

1) To apply a production of the form A → *b*α where b is a terminal and α a string of terminals and nonterminals, use the transition:

   REPLACE(α^r), ADVANCE

   where (α^r) *indicates that the string α is pushed onto the stack on reverse so that the leftmost symbol of the production is on top of the stack*. Advance indicates to process the next input symbol.

2) To apply a production of the form A → α where α a string of terminal and nonterminal *not beginning* with a terminal use the transition:

   REPLACE(α^r), RETAIN (keep the current input symbol)

   or

   POP instead of REPLACE(ε) when α is ε (the empty string)

3) If terminal *b* is a stack symbol, then the table entry for row *b* and column *b* is POP, ADVANCE.4) The table entry for the bottomarker row and the endmarker column is ACCEPT or ☺.

5) *If there is a nullable production for nonterminal A* and no entry for stack symbol A and input *b* is implied by rule 2, then either the nullable production can be applied or the sequence can be REJECTed or ☹.

Extended Follow Rule. - Add the endmarker symbol to the FOLLOW of the starting symbol. This will allow the compiler to recognize an empty file.

Example No. 1

Given the grammar shown below fill in the action entries in the table and their corresponding rules. Verify the FIRST and FOLLOW sets. Do not forget to find the nullable sets first.

1	S → AbB	FIRST(S) =	FIRST(AbB) = {a,b,c,e}
2	S → d	FIRST(A) =	FIRST(d) = {d}
3	A → CAb	FIRST(B) =	FIRST(Cab) = {a,e}
4	A → B	FIRST(C) =	FIRST(B) = {c}
5	B → cSd	FOLLOW(S) =	FIRST(cSd) = {c}
6	B → ε	FOLLOW(A) =	FIRST(ε) = {} = ∅
7	C → a	FOLLOW(B) =	FIRST(a) = {a}
8	C → ed	FOLLOW(C) =	FIRST(ed) = {e}

The action table for the grammar is shown on the next page.

Action Table

	a	b	c	d	e	┤
S						
A						
B						
C						
a						
b						
d						
▼						☺

Action Rules (for a given input string)

1	**4**	**7**
2	**5**	**8**
3	**6**	**9**

Activity: Verify whether or not the following string are well-formed strings (equivalent to syntax error free). Find first the top-down derivation for these strings. Apply the rules to recognize these strings using the corresponding action rules.

1) Eddbcdd
2) d
3) b

References

Although all references mentioned below have these rules in one form or another. However, the main source, with some modifications by this professor, is reference No. 1.

1) Compiler Design by Lewis II, Stearns, Rosenkratz, The IBM System Programming Series. Addison-Wesley. 1972.

2) The Theory of Parsing, Translation, and Compiling by A. Aho and J. Ulmann. Prentice-Hall. 1972.