Syntactic Analyzers: LL(1) and SLR(1) Grammars

Team

* Juan José Escobar Saldarriaga
* Samuel Llano Madrigal
* Class Code: 7308

Development environment

* Operating System: Windows 11
* Programming Language: Python 3.12.2
* Tools: IDE's and editors such as Visual Studio Code

Algorithm Description

* This program allows the user to enter a grammar from its derivation rules and then verify if it is type LL(1) and/or SLR(1), in addition to all the processes such as tables or sets necessary for its parser, since it can then verify if a string belongs to the grammar or not.

Explanation and Structure

First and Follow

From chapter 4.4.2 of Compilers: Principles, Techniques, & Tools. 2nd ed. Boston: Pearson/Addison Wesley, 2007.

* To compute FIRST (X) for all grammar symbols X, apply the following rules until no more terminals or can be added to any FIRST set.

    1. If X is a terminal, then FIRST (X) = {X}
    2. If X is a nonterminal and X Y1Y2...Yk is a production for some k 1, then place a in FIRST (X) if for some i a is in FIRST (Yi), and is in all of FIRST Y1 ;::: ; FIRST (Yi 1); that is, Y1 Yi 1 If is in FIRST Yj for all j = 1 2;::: ;k, then add to FIRST (X).
       For example, everything in FIRST (Y1) is surely in FIRST (X). If Y1 does not derive , then we add nothing more to FIRST (X), but if Y1 , then we add FIRST (Y2), and so on.
    3. If X is a production, then add to FIRST X

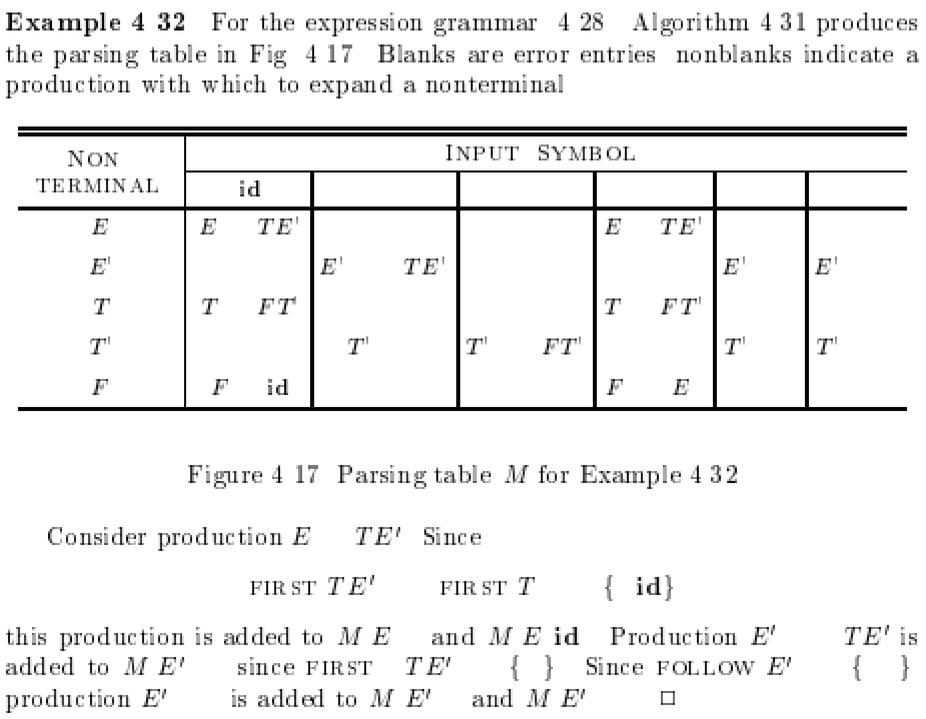
* To compute FOLLOW (A) for all nonterminals A apply the following rules until nothing can be added to any FOLLOW set.

    4. Place $ in FOLLOW (S), where S is the start symbol, and $ is the input right endmarker.
    5. If there is a production A alphaBbeta, then everything in FIRST (beta) except epsilon is in FOLLOW (B).

6. If there is a production A -> alphaB or a production A -> alphaBbeta where FIRST (beta) contains epsilon, then everything in FOLLOW (A) is in FOLLOW (B)

LL(1) Table and Parser

An LL(1) parser constructs a parse table using FIRST and FOLLOW sets to drive top-down parsing. Each table entry maps a nonterminal and terminal to a production. Parsing decisions are made with one lookahead token, hence "1" in LL(1). This implementation computes FIRST and FOLLOW sets (Ch. 4, §4.4), builds the LL(1) table (§4.4.1), and parses input by simulating a predictive parser. Ambiguity or multiple entries for a table cell indicate the grammar is not LL(1). This approach is suitable for simple, unambiguous grammars.



Example 4 32 For the expression grammar 4 28 Algorithm 4 31 produces the parsing table in Fig 4 17 Blanks are error entries nonblanks indicate a production with which to expand a nonterminal

NON TERMINAL	INPUT SYMBOL					
	id					
E	$E \quad TE'$			$E \quad TE'$		
E'		$E' \quad TE'$			E'	E'
T	$T \quad FT'$			$T \quad FT'$		
T'		T'	$T' \quad FT'$		T'	T'
F	$F \quad$ **id**			$F \quad E$		

Figure 4 17 Parsing table M for Example 4 32

Consider production $E \quad TE'$ Since

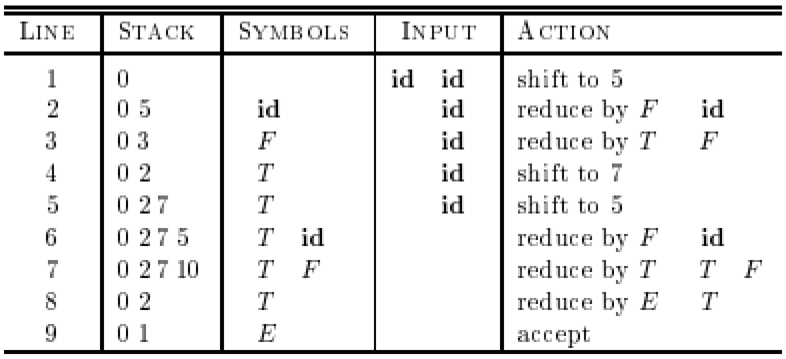
$$\text{FIRST } TE' \qquad \text{FIRST } T \qquad \{ \ \mathbf{id}\}$$

this production is added to $M \ E$ and $M \ E$ **id** Production $E' \qquad TE'$ is added to $M \ E'$ since FIRST TE' $\{ \ \}$ Since FOLLOW E' $\{ \quad \}$ production E' is added to $M \ E'$ and $M \ E'$ □

Reference: Aho et al., Ch. 4, especially §4.4 (Predictive Parsing)

LR(0) Automaton

LR(0) parsers build a deterministic finite automaton (DFA) of item sets representing parsing states. Each item tracks a production and a dot that indicates parsing progress. The closure and goto functions compute how parsing states transition based on grammar symbols. The automaton serves as the foundation for more powerful bottom-up parsers like SLR(1). This corresponds to LR(0) item construction (Ch. 4, §4.6), capturing the shift-reduce behavior without lookahead.



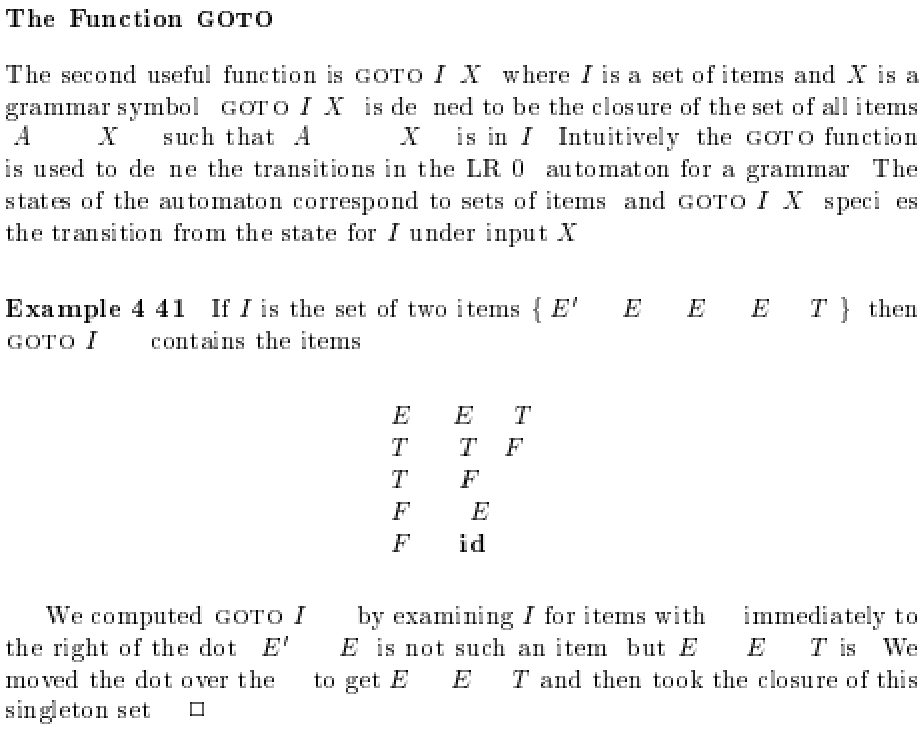
LINE	STACK	SYMBOLS	INPUT	ACTION
1	0		id id	shift to 5
2	0 5	id	id	reduce by F id
3	0 3	F	id	reduce by T F
4	0 2	T	id	shift to 7
5	0 2 7	T	id	shift to 5
6	0 2 7 5	T id		reduce by F id
7	0 2 7 10	T F		reduce by T T F
8	0 2	T		reduce by E T
9	0 1	E		accept

Figure 4 34 The parse of **id** **id**

Reference: Aho et al., Ch. 4, §4.6 (LR(0) Items and DFA Construction)

Tables: Action and Goto

The ACTION and GOTO tables guide an LR parser's behavior. ACTION specifies shift, reduce, or accept operations based on terminal input and current state. GOTO defines transitions for nonterminals after reductions. These tables are constructed from the LR(0) automaton and the FOLLOW sets for SLR(1) parsing (§4.8.3). The code builds these tables by inspecting each item's dot position, and using FOLLOW sets to assign reductions only where valid.



The Function GOTO

The second useful function is GOTO I X where I is a set of items and X is a grammar symbol GOTO I X is de ned to be the closure of the set of all items A X such that A X is in I Intuitively the GOTO function is used to de ne the transitions in the LR 0 automaton for a grammar The states of the automaton correspond to sets of items and GOTO I X speci es the transition from the state for I under input X

Example 4 41 If I is the set of two items { E' E E E T } then GOTO I contains the items

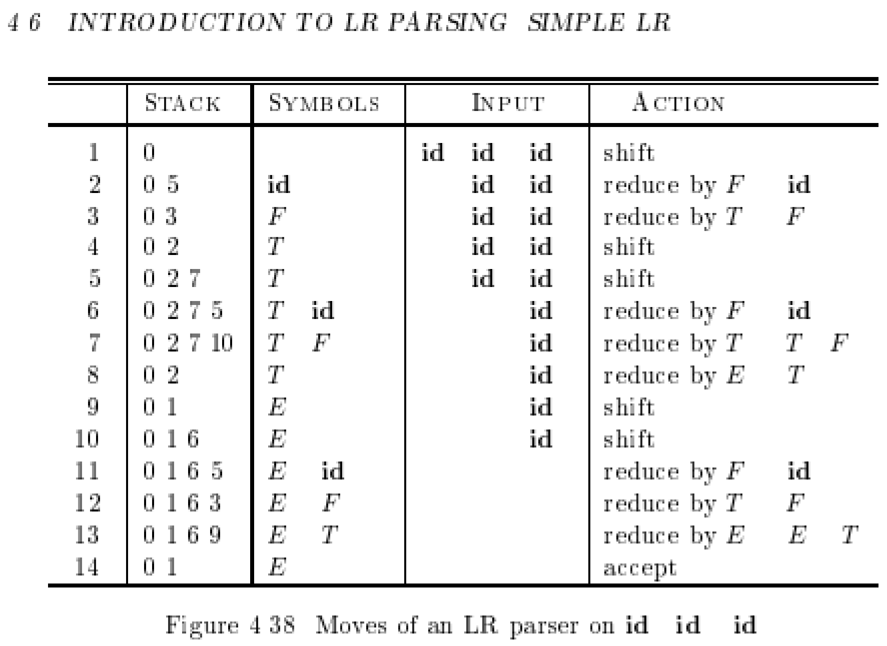
$$\begin{array}{l} E \quad E \quad T \\ T \quad T \quad F \\ T \quad F \\ F \quad E \\ F \quad \mathbf{id} \end{array}$$

We computed GOTO I by examining I for items with immediately to the right of the dot E' E is not such an item but E E T is We moved the dot over the to get E E T and then took the closure of this singleton set □

Reference: Aho et al., Ch. 4, §4.6 (Constructing SLR Parsing Tables)

SLR(1) Parser

An SLR(1) parser is a bottom-up parser that uses LR(0) item states and FOLLOW sets to make parsing decisions. It resolves shift/reduce conflicts by consulting FOLLOW sets when the dot reaches the end of a production. The parser uses a stack to track state transitions, processing input based on the ACTION and GOTO tables. This approach supports more complex grammars than LL(1) but is simpler than canonical LR(1). The code implements SLR(1) parsing as detailed in Aho et al. (Ch. 4, §4.8.3), including grammar augmentation and conflict resolution.

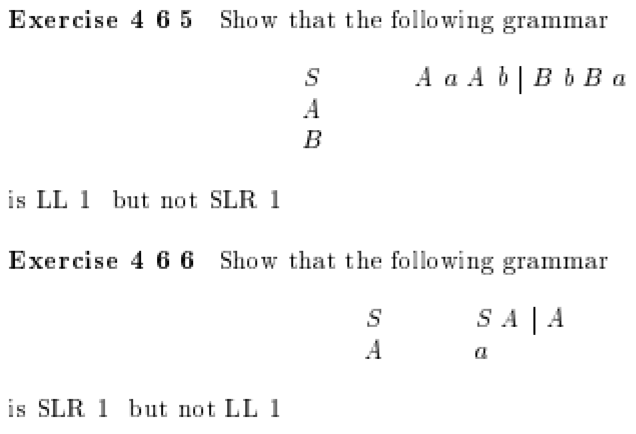


4 6 *INTRODUCTION TO LR PARSING SIMPLE LR*

	STACK	SYMBOLS	INPUT	ACTION
1	0		**id** **id** **id**	shift
2	0 5	**id**	**id** **id**	reduce by *F* **id**
3	0 3	*F*	**id** **id**	reduce by *T* *F*
4	0 2	*T*	**id** **id**	shift
5	0 2 7	*T*	**id** **id**	shift
6	0 2 7 5	*T* **id**	**id**	reduce by *F* **id**
7	0 2 7 10	*T* *F*	**id**	reduce by *T* *T* *F*
8	0 2	*T*	**id**	reduce by *E* *T*
9	0 1	*E*	**id**	shift
10	0 1 6	*E*	**id**	shift
11	0 1 6 5	*E* **id**		reduce by *F* **id**
12	0 1 6 3	*E* *F*		reduce by *T* *F*
13	0 1 6 9	*E* *T*		reduce by *E* *E* *T*
14	0 1	*E*		accept

Figure 4 38 Moves of an LR parser on **id** **id** **id**

Reference: Aho et al., Ch. 4, §4.6 (SLR Parsing)



Exercise 4 6 5 Show that the following grammar

$$\begin{array}{ll} S & A\ a\ A\ b \mid B\ b\ B\ a \\ A & \\ B & \end{array}$$

is LL 1 but not SLR 1

Exercise 4 6 6 Show that the following grammar

$$\begin{array}{ll} S & S\ A \mid A \\ A & a \end{array}$$

is SLR 1 but not LL 1