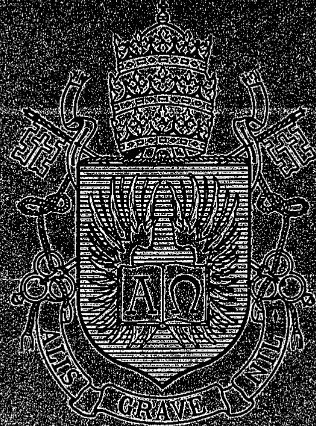


PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 12/71

A SET OF PROGRAMS TO TEST AND PARSE LL(k) TYPE LANGUAGES

by

Lucas Tofolo de Macedo

Computer Science Department - Rio Datacenter

Pontificia Universidade Catolica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

B C — PUC

UC 31507-5

A SET OF PROGRAMS TO TEST AND PARSE LL(k) TYPE LANGUAGES

Lucas Tofolo de Macedo
Member of The Applications
Division of Rio Datacenter
PUC/RJ

This paper will be published in the Proceding of The Fifth Asilomar Conference on Circuits and Systems (San Francisco - November - 1971)

Series Editor: Prof. A. L. Furtado December/ 1971.

ABSTRACT

This paper describes a system which tests the LL(k) condition for a given grammar and provides a recognizer for strings of the language generated by this grammar. During the recognition phase the system generates code so that we have, in a semantic stack, information about the code generated so far.

1. INTRODUCTION

There is a class of context-free grammars (cfg) which can be parsed in a top down fashion deterministically with a look ahead of k symbols. The languages generated by these grammars (LL(k)) belong to a class of languages which can be parsed in a length of time proportional to n, where n is the length of the input string, This class is reasonably wide and is of great utility in construction of sintax - directed compilers. This paper describes a system which is an implementation of two algorithms which are due to [D. J. Rosenkrantz and R. E. Stearns, 1970] and [P. M. Lewis II and R. E. Stearns, 1968] respectivelly. The first one tests the LL(k) condition for a given grammar, and the second provides a recognizer for strings of the language generated by this grammar.

The system, during the recognition phase, generates code, consequently being a sintax - directed compiler for the languages with LL(k) characteristics. We are considering the grammar being studied to be ε - free and reduced.

2. TEST OF THE LL(k) CONDITION

The results which follow allow us to formulate, in computational terms, the test for the LL(k) condition for a given grammar.

Definition 1 - Let be a cfg $G = (V_N, V_T, P, S)$ and k in $\mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.

We define

(i) If α is in V^*, then $\alpha/k = \begin{cases} \alpha, \text{ if } |\alpha| \leq k \\ \alpha', \text{ if } \alpha = \alpha'\beta, \ |\alpha'| = k \\ [\text{and } \alpha', \beta \text{ are in } V^+ \end{cases}$

(ii) If $A \subseteq V^*$, then $A/k = \{w/k \mid w \text{ is in } A\}$

(iii) If $A \to \alpha$ is the p-th production in P, then $L_p(A) = L(\alpha) = \{w \text{ in } V_T^* \mid \alpha \overset{*}{\Longrightarrow} w, \text{ where } \alpha \text{ is in } V^+\}$

(iv) If A in V_N is such that $A \Longrightarrow \alpha \overset{*}{\Longrightarrow} w$, and $A \to \alpha$ is the p-th production in P, then one can write $A \overset{*}{\Longrightarrow} w \ (p)$.

Definition 2 - A cfg $G = V_N, V_T, P, S)$ is LL(k) if and only if, for all w in V_T^*/k, A in V_N and w_1 in V_T^*, there exist at most one production in P, for instance the p-th production, and w_2, w_3 in V_T^* such that the following three conditions hold:

(i) $S \stackrel{*}{\Longrightarrow} w_1 \; A \; w_3$

(ii) $A \stackrel{*}{\Longrightarrow} w_2 \; (p)$

(iii) $(w_2 \; w_3)/k = w$

<u>Definition 3</u> - Let $G = (V_N, V_T, P, S)$ be a cfg and k in $\mathbb{N}$. It is possible to construct another cfg $G' = (V'_N, V'_T, P', S')$ as follows:

(i) $S' = (S, \{\varepsilon\})$

(ii) If $A \to A_1 \; A_2 \; \ldots \; A_n$ is the p-th production in P, then $(A, R) \to (A_1, R_1) \; (A_2, R_2) \; \ldots \; (A_n, R_n)$ is the (p, R) production in P', where $R \subseteq V_T^* /k$, $R_n = R$ and for all i, $1 \leq i < n$, we have $R_i = \left[L(A_{i+1} \; A_{i+2} \; \ldots \; A_n) R \right] /k$

(iii) $V'_T = V_T$

(iv) V'_N is the set of ordered pairs (A, R), where A is in V_N, that actually occur in the productions obtained by (ii).

It is easy to see that the grammar G' is structurally equi - valent to the grammar G, and so, we have the following

<u>Lemma 1</u> Given G and G', as defined above, then for all (A, R) in V'_N and α, β in $(V'_N \cup V_T)^*$, if $S' \stackrel{\ell *}{\Longrightarrow} \alpha \; (A, R) \; \beta$, then $R = L(\beta)/k$.

With the results above, the procedure to test if a given cfg is a LL(k) one, can be described as follows:
Given a cfg $G = (V_N, V_T, P, S)$ and one integer k, we construct another cfg G', as in definition 3 and then, for each w in V_T^*/k and (A, R) in V'_N, we test if there exist at most one p, in P, such that w belongs to $(L_p(A)R)/k$
If every w belonging to (A, R) verifies this condition, the grammar is LL(k). To prove that this test works, we need a lemma which characterizes LL(k) grammars in terms of the entities computed in the test.

Lemma 2 - A cfg $G = (V_N, V_T, P, S)$ is LL(k) if and only if, for all A in V_N, w in V_T^*/k, and $R \subseteq V_T^*/k$, there exist at most one p in P such that, for some w_1 in V_T^* and w_2 in V^*, the three following conditions hold.

(i) $S \overset{*\ell}{\Longrightarrow} w_1 A w_2$

(ii) $R = L(w_2)/k$

(iii) W is in $(L_p(A)L(w_2))/k$

The meaning of Lemma 2 is that the choice of p can be obtained from a finite amount of information, namely A and $L(w_2)/k$ - (or R).

In the system, the construction of the grammar G' and the sets $(L_p(A)\ R)/k$ can be explained as follows.

The variables of G' are ordered pairs (A, R) where A is in V_N and $R \subseteq V_T^* /k$. Such pairs are obtained as follows:

a). <u>Construction of the sets R</u> - initially we construct the sets $L^K(A)$, defined as follows:

$L^k(A) = \{ w \text{ in } V_T^* /k \mid A \overset{\ell *}{\Longrightarrow} wz$, where A is in V_N and z is in $V_T^*\}$.

Such sets are necessary by the following.

<u>Lemma 3</u> - $R_i = \left[L(A_{i+1})R_{i+1} \right] /k$, for $1 \leq i < n$

So, for the maximum value of i ($i = n - 1$), $R_{i+1} = R_n = R$ (by definition 3, (ii)) and as the set R is initially ε (by definition 3, (i)), we have an effective way of obtai ning the R_i's, that is by concatenation of the $L(A_{i+1})$ with R_{i+1}. This concatenation is processed by one subroutine - subroutine CAT - in which, given two one-dimensional arrays, we obtain a third one-dimensional array whose elements are the results of the concatenation of each ele ment of the first array with all the elements of the second.

b). <u>Construction of the sets $(L_p(A)\ R)\ /k$</u> – the sets $(L_p(A)\ R)\ /k$ are obtained by the concatenation of the sets $L_p^k(A)$ with the sets R, where the sets $L_p^k(A)$ are defined as follows:

if $A \to A_1 A_2 \ldots A_n$ is the p-th production in P ,
then $L_p^k(A) = (\ldots (L^k(A_1) L^k(A_2))/k \ldots L^k(A_n))/k)/k$.

Thus, since that we just have the sets $L^k(A)$ (see to pic a), we can obtain the sets $L_p^k(A)$ by using the subroutine CAT.

All sets mentioned above are stored in lists so that we have a sequential search which is proportional to the number of elements in them.

The test is processed during the construction of the sets $(L_{p_i}(A)R)/k$, where p_i $(i = \overline{1, \ell})$ is one of the ℓ's alternatives A-productions in P, and R is fixed . It is made a comparison of each element of $(L_{p_i}(A)R)/k$ with the elements of the $(L_{p_s}(A)R)/k$ that are just constructed $(s = \overline{1, i-1})$. If some element of the first set is equal to some element of the list already obtained, the grammar is not LL(k).

The system answers if the grammar is LL(k), for k varying in the interval [1, 5] , in this order. If the condition doesn't holds for a value of k up to five, the system give the corresponding message.

3. LL(k) RECOGNITION AND CODE GENERATION

The implementation of the LL(k) recognizer follows the algorithm described in [1], and becomes straightforward due the fact that we have the information from the $(L_p(A)R)/k$ sets physically stored in the list.

This recognizer is similar to those of the deterministic pushdown automaton type [3] , and also includes the utilization of a bi-dimensional array where the information about the alternative productions for each variable of the grammar is stored.

Everytime a terminal is accepted, the semantic information associated with the element,necessary to the code generation is stacked in a second stack-semantic stack. While each production is recognized in its entirety, the control is transfered to a semantic subroutine (in PL/I) for that production, which uses the elements on the top of the semantic stack to generate code and information that will next be placed on the new top of the stack. In order to know which subroutine we have to call in each case, we can number it according to the production of the grammar.

Example: Let a grammar, supposedely LL(k)*, have the following production:

$\underline{AE} \rightarrow T^{(1)} \mid + \; T^{(2)} \mid \underline{AE} - T^{(3)} \mid \underline{AE} + T^{(4)} \mid T^{(5)}$

$T \rightarrow F^{(6)} \mid T * F^{(7)} \mid T / F^{(8)}$

$F \rightarrow i^{(9)}$, where $i = a_1, a_2, a_3, \ldots$

Suppose we want recognize the string $a_1 * a_2 / a_3$ —|. The illustration below shows the process of code generation

READ HEADER	PUSHDOWN STACK (PD)	SEMANTICS STACK (S)
↓ $a_1 * a_2 / a_3$ —|	AE	
↓ $a_1 * a_2 / a_3$ —|	T - | 5	
↓ $a_1 * a_2 / a_3$ —|	T / F - | 8 - | 5	

* At the end of this paper we present an example of a grammar for arithmetic expressions that is LL(2). The grammar above is used to provide us with a sample example.

READ HEADER	PUSHDOWN STACK (PD)	SEMANTICS STACK (S)

↓
$a_1 * a_2 / a_3$ —|

PD	
T	
*	
F	
–	7
/	
F	
–	8
–	5

↓
$a_1 * a_2 / a_3$ —|

PD	
F	
–	6
*	
F	
–	7
/	
F	
–	8
–	5

↓
$a_1 * a_2 / a_3$ —|

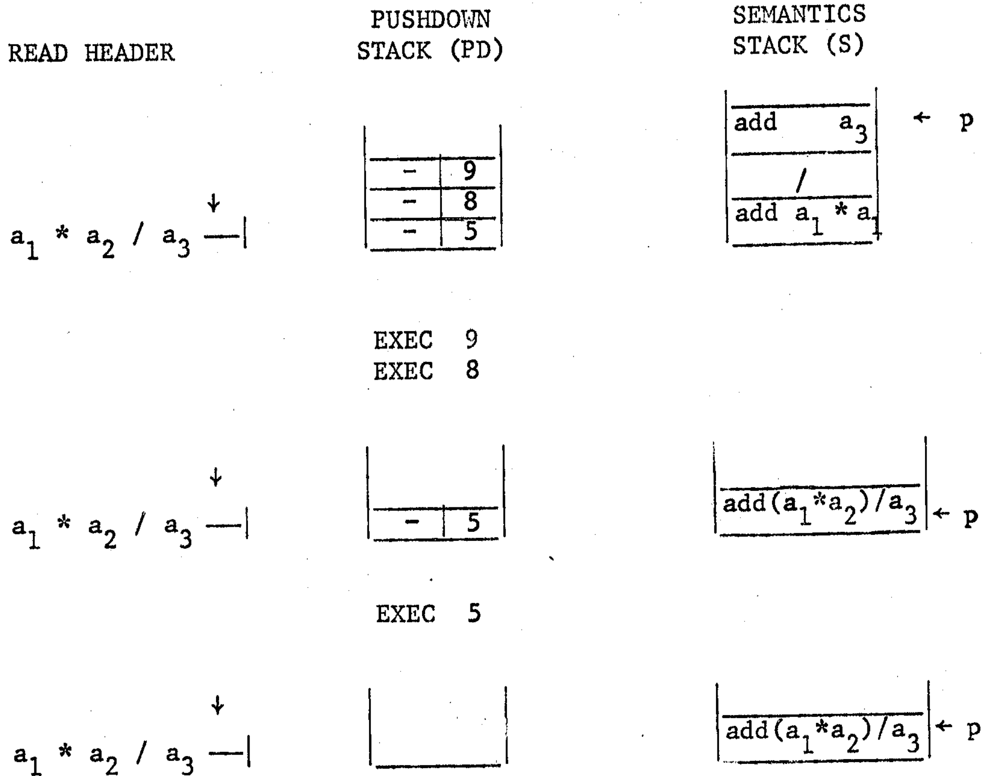
PD	
I	
–	9
–	6
*	
F	
–	7
/	
F	
–	8
–	5

READ HEADER	PUSHDOWN STACK (PD)	SEMANTICS STACK (S)
$a_1 * a_2 / a_3$ —| (↓ at *)	- | 9 - | 6 * ⋮	add a_1 ← p
	EXEC 9 EXEC 6	
$a_1 * a_2 / a_3$ —| (↓ at *)	* F | 7 ⋮	add a_1 ← p
$a_1 * a_2 / a_3$ —| (↓ at a_2)	F | 7 / ⋮	* add a_1 ← p
$a_1 * a_2 / a_3$ —| (↓ at a_2)	I - | 9 - | 7 ⋮	* aad a_1 ← p

READ HEADER	PUSHDOWN STACK (PD)	SEMANTICS STACK (S)
↓ $a_1 * a_2 / a_3$ —|	- | 9 - | 7 / ⋮	add a_2 * add a_1 ← p

EXEC 9
EXEC 7

READ HEADER	PUSHDOWN STACK (PD)	SEMANTICS STACK (S)
↓ $a_1 * a_2 / a_3$ —|	/ F - | 8 - | 5	add $a_1 * a_2$ ← p
↓ $a_1 * a_2 / a_3$ —|	F - | 8 - | 5	/ add $a_1 * a_2$ ← p
↓ $a_1 * a_2 / a_3$ —|	I - 9 8 - 5	 / add $a_1 * a_2$ ← p



The semantic subroutine **executes** the following types of opera-

tions:

9). It would be a CONTINUE, for instance

7). It generates a code to multiply the content of the address pointed by S_p and S_{p-2}, and store the result in an third address (temporary address) which is stored in S_{p-2} After this, it sets $p \leftarrow p - 2$.

1). It generates a code to turn negative the content of the address pointed by S_p , and stores the result in another address (temporary address) which is stored in S_{p-2}. After this, it sets $p \leftarrow p - 2$.

We assume here, to simplify the example, that the lexical analyser which preprocess the variables, gives the storage location of the variable and not the position in the symbol table where theirs properties are stored, as it normally occurs.

One could also think of subroutines for detecting error that would take over the control everytime a test for a terminal in the top of the pushdown fails. In this case, the first stack would also stores the number of the production being looked for.

APPENDIX

LL(2) grammar for arithmetic expressions.

A	→	QS
S	→	+ QS | - QS | —⊣
Q	→	(B | i * Q | i / Q | i
B	→	QP
P	→	+ QP | - QP |) * Q |) / Q |)

REFERENCES

(1) P. M. Lewis II and R. E. Stearns — "Syntax Directed Transduction" Appendix I - Recognition of LL(k) Languages. JACM - vol. 15 - Nº 3 - July 1968.

(2) D. J. Rosenkrantz and R. E. Stearns — "Properties of Deterministic Top-Down Grammars" Information and Control 17, 226 - 256 - 1970

(3) J. E. Hopcroft and J. D. Ullman — "Formal Languages and Their Relation to Automata" Addison - Wesley - 1969.

(4) L. T. Macedo — "Manipulação de Gramáticas Livres do Contexto e Reconhecimento de LL(k)" Pontifícia Universidade Católica do Rio de Janeiro - Tese de Mestrado - 1971.