A Practical Method for the Implementation of Syntactic Parsers

PIER MARCO RICCHETTI [1, 2], JOÃO JOSÉ NETO [1]
[1]Departamento de Engenharia de Computação e Sistemas Digitais
Universidade de São Paulo - Escola Politécnica
Av Prof. Luciano Gualberto, trav 3, 158 – Cidade Universitária
05508-900 São Paulo
BRASIL
[2]Núcleo de Pesquisa em Engenharia e Computação
Universidade São Judas Tadeu
Av Taquari, 546
03166-000 São Paulo
BRASIL
pier.ricchetti, joao.jose@poli.usp.br http://www.pcs.usp.br/~lta/

Abstract: - The implementation of syntactic parsers is a very important task in compiler construction. There are several classic methods and algorithms used for syntactic parser construction [1][2][3][4][5]. In [3] there is a proposal of a method for the automatic construction of syntax acceptors from context-free grammars, and also a first extension of this method in order to allow the generation of syntax trees while accepting input sentences. In this paper we present a further extension of that method by including the addition of semantic actions to the generated device. This is a very practical method which can be used for educational purposes, by means of a step-by-step construction and understanding of each algorithm applied to the original grammar. Moreover, the transducer representing the desired parser activates semantic actions while a syntactic parsing tree is automatically generated for the given input sentence.

Key-Words: - context-free languages, automata, grammars, syntax, parser, syntax tree, automatic generation

1 Introduction

A language is a set of valid sentences that follows given production rules over an alphabet. Syntactic parsers may be implemented to check the correctness of such sentences by applying algorithms and special techniques to the language's rules.

This paper shows techniques regarding the construction of a syntactic parser that includes semantic actions and the generation of the corresponding syntactic tree, in a step-by-step way that can be used for educational purposes too.

2 Previous Works

Conway [6] has proposed a structured pushdown automaton that could be divided by its submachines. This automaton was further formalized by Lomet [7]. Neto [3] proposed a context free parsing algorithm based in such formalization. This algorithm was further modified by Iwai [8] including adaptivity.

3 Our Proposal for Parser Generation

The automatic generation of a parser from a given grammar (set of production rules) usually leads to non-optimized formulations and to undesired non-deterministic operation of the generated device. In this paper, a method is proposed that starts from a grammatical formulation of the desired context-free language and both generates an equivalent automaton and removes most non-deterministic transitions by reworking and simplifying the given grammar, and then constructing a corresponding structured pushdown transducer [3]. This grammar includes identifying labels that are used in the construction of the parser. Such a parser is based on a structured pushdown transducer that checks the syntax, accepts the sentence, promotes the execution of semantic actions associated to the grammatical rules needed in the derivation of the sentence, and explicitly generates the corresponding syntax tree.

4 Problem Solution

In this section, we describe the method proposed for the automatic construction of a parser starting from a context-free grammar given as a set of simple rules denoted in Wirth's notation.

4.1 Grammar Format

The initial grammar should have all its rules following one of the three forms below only. In the chosen notation, P is the non-terminal being defined by the production rule; Y and Z are semantic actions to be performed, and μ is a symbol sequence, $\mu \in (V_N \cup V_T)^*$, where V_N is the set of non-terminals in the grammar and V_T is the alphabet of the language.

(1) i : P → P{Y} μ {Z} (left-recursive rules).
(2) i : P → {Y} μ {Z}P (right-recursive rules).
(3) i : P → {Y} μ {Z} (other rules).

(in the templates above, i represents a sequence number identifying the corresponding rule).

In the case of a non-terminal for which there are both left- and a right-recursive rules, the composition of such rules is classified in group (1). In the cases in which non-terminal P occurs inside the term μ, the corresponding rule is classified in group (3).

4.2 Labeling Production Rules

Labels are used for keeping relevant information on the original grammar, for further use in the construction of the desired parser. A label is a sequence of symbols that carries information on the kind of the original rule, on the terminals and non-terminals involved, and so on.

Considering μ as a sequence of symbols β_k and the corresponding labels β_k', with β_k , $\beta_k' \in (V_N \cup V_T)^*$, $1 \le k \le n$, the labeling process is performed by applying the rules below:

- each rule is identified uniquely by its sequence number i = 0, 1, 2, ...;
- non-terminals are not labeled;
- terminal x receives a corresponding label x;
- semantic actions W are named Ā and labeled W;
- the start of any production rule is labeled with [
- the end of a production rule is labeled as], preceded by L, R or G according to the recursivity class of the non-terminal defined by the current rule. (L = left, R = right, G = general).

Rules (4), (5), (6) below show the labeling of the rule patterns (1), (2), (3) respectively.

$$[\quad \{Y\} \quad \beta_1, \quad \beta_2, \quad \beta_n, \quad \{Z\}P_i^L]$$
$$(4)\ P \rightarrow \uparrow P \uparrow \bar{A} \uparrow \quad \beta_1 \uparrow \beta_2 \uparrow \ldots \beta_n \uparrow \bar{A} \uparrow$$

$$[\quad \{Y\} \quad \beta_1, \quad \beta_2, \quad \beta_n, \quad \{Z\}P_i^R]$$
$$(5)\ P \rightarrow \uparrow \bar{A} \uparrow \quad \beta_1 \uparrow \beta_2 \uparrow \ldots \beta_n \uparrow \bar{A} \uparrow \quad P \uparrow$$

$$[\quad \{Y\} \quad \beta_1, \quad \beta_2, \quad \beta_n, \quad \{Z\}P_i^G]$$
$$(6)\ P \rightarrow \uparrow \bar{A} \uparrow \quad \beta_1 \uparrow \beta_2 \uparrow \ldots \beta_n \uparrow \bar{A} \uparrow$$

4.3 Grouping Production Rules

Production rules may be grouped according to the three classes defined in (3.1), so that we obtain up to three expressions for each non-terminal, one for class (1) productions, one for class (2) productions, and one for class (3) productions. For example, in production rules of class (1), as the expression (4) above, defining μ_z as a β labeled sequence, and $1 \le z \le m$, we obtain the expression (7) below:

$$[\quad \{Y_1\} \quad \{Z_1\}P_1^L$$
$$(7)\ P \rightarrow \uparrow P \uparrow (\uparrow \bar{A} \uparrow \quad \mu_1 \uparrow \bar{A} \uparrow \quad | \uparrow \ldots$$

$$\{Y_2\} \quad \{Z_2\}P_2^L$$
$$\ldots \uparrow | \uparrow \bar{A} \uparrow \quad \mu_2 \uparrow \bar{A} \uparrow \quad | \quad \ldots$$

$$\{Y_m\} \quad \{Z_m\}P_m^L \quad]$$
$$\ldots | \uparrow \bar{A} \uparrow \quad \mu_m \uparrow \bar{A} \uparrow \quad) \uparrow$$

In a similar way, we obtain expressions for the remaining classes of productions.

4.4 Removing Self-recursions

The three kind of presented rules for a given non-terminal (L, R, G) are now grouped in a single rule. Considering that for the set of three productions:

(8) $X \rightarrow Xa$
$X \rightarrow bX$
$X \rightarrow c$

the general solution for non-terminal X is b*ca*. We may easily extend this result for our (more general) case, giving the following expression:

$$[\quad \{Y_1\}$$
$$(9)P \rightarrow \uparrow (\uparrow (\uparrow \varepsilon \uparrow \backslash \uparrow \bar{A} \uparrow$$

$$\{Z_1\}P_1^R \quad \{Y_2\} \quad \{Z_2\}P_2^R$$
$$\mu_1 \uparrow \bar{A} \uparrow \quad | \uparrow \bar{A} \uparrow \quad \mu_2 \uparrow \bar{A} \uparrow \quad | \ldots$$

$$\{Y_m\} \qquad \{Z_m\}P_n^{R}$$
$$\ldots \mid \uparrow \bar{A} \uparrow \quad \mu_m \uparrow \bar{A} \uparrow \quad) \uparrow$$

$$\{Y_1\} \qquad \{Z_1\}P_1^{G} \qquad \{Y_2\}$$
$$(\uparrow \bar{A} \uparrow \quad \mu_1 \uparrow \bar{A} \uparrow \quad \mid \uparrow \bar{A} \uparrow$$

$$\{Z_2\}P_2^{G} \qquad \{Y_m\}$$
$$\mu_2 \uparrow \bar{A} \uparrow \quad \mid \ldots \mid \uparrow \bar{A} \uparrow \quad \mu_m \uparrow$$

$$\{Z_m\}P_m^{G} \qquad \{Y_1\} \qquad \{Z_1\}P_1^{L}$$
$$\bar{A} \uparrow \quad) \uparrow (\uparrow \varepsilon \uparrow \backslash \uparrow \bar{A} \uparrow \quad \mu_1 \uparrow \bar{A} \uparrow$$

$$\{Y_2\} \qquad \{Z_2\}P_2^{L} \qquad \{Y_m\}$$
$$\mid \uparrow \bar{A} \uparrow \quad \mu_2 \uparrow \bar{A} \uparrow \quad \mid \ldots \mid \uparrow \bar{A} \uparrow \quad \mu_m \uparrow$$

$$\{Z_m\}P_m^{L} \qquad]$$
$$\bar{A} \uparrow \quad) \uparrow) \uparrow$$

4.5 Simplifying the Rules

At this point, some additional classical techniques are applied to the rules obtained in 3.4 in order to reduce non-determinism and rule complexity, e.g.

- reworking options that have the same prefixes by putting common prefixes in evidence;
- replacing left-recursive non-terminals by the corresponding grammatical expressions;
- eliminating cyclic recursions;

In this paper, for space reasons, such procedures have been omitted in order to avoid extra details in the exposition of the proposed solution. In [3], interested readers may find further information on this subject.

4.6 State Assignment

The assignment of states in correspondence to each position in the expressions defining the simplified grammar helps its conversion into a corresponding structured pushdown transducer. In order to ease explaining the assignment method, we assume that:

- E is an expression.
- a, b, c, d are all µ-like sequences which include references to semantic actions;
- r, s are the state numbers assigned to the left and to the right extremities of a parenthesized sequence within an expression;
- x, y are variables whose numeric values are used for identifying the states assigned;

State assignment is made according to the following directions:

- the parenthesized groups are first detected;
- new states are assigned to the boundaries of such groups;

If the group corresponds to one of the expression templates:

$$E = \underset{r}{\uparrow} (\uparrow a_1 \uparrow \mid \ldots \mid \uparrow a_m \uparrow) \quad \underset{s}{\uparrow}$$

or

$$E = \underset{r}{\uparrow} (\uparrow a_1 \uparrow \mid \ldots \mid \uparrow a_m \uparrow \backslash \uparrow b_1 \uparrow \mid \ldots \mid \uparrow b_n \uparrow) \quad \underset{s}{\uparrow}$$

we assume that:

- if r has been previously assigned, then x := r, else x is assigned a new state number, and state x is assigned to the points at the leftmost extremity of all sub-expressions corresponding to each syntactical option in the group being considered;
- if s has been previously assigned then y := s, else y receives a new state number and the state y will be assigned to the rightmost extremity of all sub-expressions corresponding to each syntactical option in the group;

If the group corresponds to the following shape:

$$E = \underset{r}{\uparrow} (\uparrow \varepsilon \uparrow \backslash \uparrow b_1 \uparrow \mid \ldots \mid \uparrow b_n \uparrow) \quad \underset{s}{\uparrow}$$

state assignment is made as follows:

$$E = \underset{r}{\uparrow} \quad (\uparrow \varepsilon \uparrow \backslash \uparrow b_1 \uparrow \mid \ldots \mid \uparrow b_n \uparrow) \quad \underset{x \; y \; y \; x \; y \; x \; s}{\uparrow}$$

If the group matches the next expression template:

$$E = \underset{r}{\uparrow} (\uparrow a_1 \uparrow \backslash \uparrow b_1 \uparrow \mid \ldots \mid \uparrow b_n \uparrow) \uparrow$$

then state assignment is performed as follows:

$$E = \underset{r}{\uparrow} \quad (\uparrow a_1 \uparrow \backslash \uparrow b_1 \uparrow \mid \ldots \mid \uparrow b_n \uparrow) \uparrow$$

All unnumbered points remaining in the expression are then assigned new state numbers.

4.7 Building the Transducer

Let T be any sub-expression of E. The desired transducer is finally obtained by considering the following cases for T:

(a) $T = \uparrow_x \varepsilon \uparrow_y$

generates an empty transition from state x to state y.

(b) $T = \uparrow_x A \uparrow_y$

(where A is a semantic action) generates an empty transition from state x to state y.

(c) $T = \uparrow_x N \uparrow_y$

(where N is a non-terminal) generates an empty transition from x to the first state of the automaton/transducer that represents the non-terminal N. State y is pushed upon a stack and is later popped at the end of the execution of the sub-machine representing N.

(d) $T = \uparrow_x (\uparrow_y \varepsilon \uparrow_z \backslash \uparrow_t \mu \uparrow_u) \uparrow_v$

generates:

- an empty transition from x to y;
- an empty transition from z to v;
- an empty transition from z to t;
- an empty transition from t to v;
- an empty transition from u to t;

The first and final states of the transducer are the states associated to the start and to the end of the expression defining the rule of the grammar that describes the root non-terminal of the grammar.
All labels are included at the destination states of each transition in the transducer.

4.8 Format for the Output Parse Tree

Our transducer produces as its output the parse trees associated to its input sentences.
A parse tree is generated in a text format. Let σ be a terminal (tree leaf), γ_q, $1 \le q \le n$ a sub-tree or a leaf, and P_w, $1 \le w \le n$ a non-terminal, there are the following conventions for the representations of the generated tree [3]:

- ( ) denotes a leaf associated to the empty string;
- (σ) denotes a leaf corresponding to a terminal σ;
- (γ_1 γ_2 ... γ_n X_1), denotes X_1 as the root node of the sub-tree having leafs γ_1 γ_2 ... γ_n ;
- [... X_1) ... X_2) ... X_n) denotes the same tree as (((... X_1) ... X_2) ... X_n). The left bracket matches all n right parentheses.
- [...] matching brackets are used as delimiters and sometimes may be omitted.

Fig. 1 below shows the parsing tree represented by [(a) Q) (b) R) (c) (d) S)].

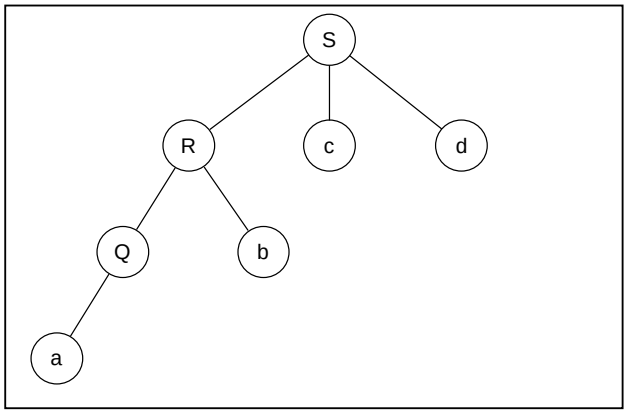

Fig. 1 – Parsing tree

4.9 Parsing a Sentence

The transducer generated in 3.7 checks the validity of a given sentence and, in addition, will produce a syntax tree while executing the corresponding semantic actions.
In order to achieve this goal, each label created in the automaton must have its symbols checked in a special table that determines the corresponding actions to be output and the actions to be performed onto a symbol stack.
Table 1 gives both the output to be generated and the stack actions for each possible label symbol.
It associates to each label element the corresponding output and stack move, in order to produce the parsing tree in the format defined in section 3.8.

Label	Output	Stack Move
ε	( )	(no move)
σ	(σ)	(no move)
P_i^L	P_i^L)	(no move)
P_i^G	P_i^G) π	$\uparrow \pi$
P_i^R	(	$\downarrow$) P_i^R
[	[(	$\downarrow$]
]	π]	$\uparrow \pi$]
{W}	(execute action W)	

Table 1: output and stack moves in label analysis.

In this table, the symbol ↑ represents a "pop" action, and ↓ represents a "push" action onto the stack. The symbol π represents the sequence of symbols in the stack comprised between its top and the first occurrence of the meta-symbol "]".

5 Illustrating Example

In the following example, the derivation rules in a context-free grammar G = ({S, X}, {0, 1}, R, S) are extended with the inclusion of calls to semantic actions A, B, C, D, E in order to enable it to perform binary-to-decimal conversion while deriving sentences.

The set of extended derivation rules defining G are:

R' = { S → { } X{A}
X → { } 0 {B}
X → { } 1 {C}
X → X{ } 0 {D}
X → X{ } 1 {E} }

See that each rule has its right hand side surrounded by a pair of calls to semantic actions, denoted in braces. Let *value* be a variable. The meaning of the called semantic actions are the following:

{A} : print *value*
{B} : *value* ← 0
{C} : *value* ← 1
{D} : *value* ← 2**value*
{E} : *value* ← 2**value*+1
{ } : no action

5.1 Building the Parser

By applying the method exposed in section 3, we obtain the following simplified rule shown in (10).

$$
\begin{aligned}
(10)\ S \rightarrow\ & [\ [\ \{\}\ 0\ \{B\}X_1^G \\
& \uparrow\ (\ \uparrow(\ \uparrow A\uparrow\ (\ \uparrow 0\uparrow \bar{A}\uparrow \\
& 1\ \{C\}X_2^G\ \{\} \\
& |\ \uparrow 1\uparrow\bar{A}\uparrow\ )\uparrow)\uparrow(\ \uparrow\varepsilon\uparrow\backslash\uparrow\bar{A}\ \uparrow \\
& 0\ \{D\}X_3^L\ \{\}\ 1\ \{E\}X_4^L\ ] \\
& 0\uparrow\ \bar{A}\ \uparrow\ |\uparrow\bar{A}\uparrow\ 1\uparrow\bar{A}\uparrow\ )\uparrow \\
& \{A\}S_0^G\ ] \\
&)\uparrow\bar{A}\uparrow\ )\uparrow
\end{aligned}
$$

This single rule generates the following automaton / transducer:

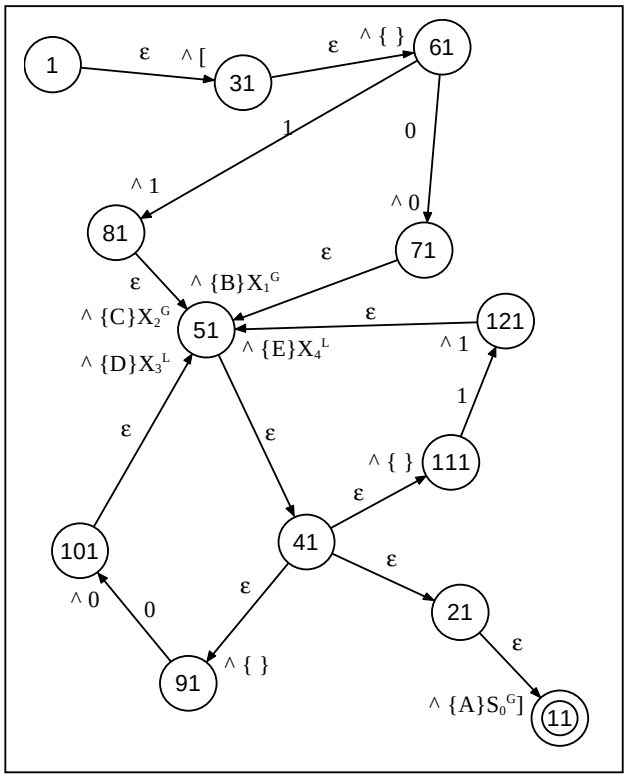

Fig.2 - Automaton/transducer generated from G

All labels are attached to the destination side of the corresponding transitions. For a better view, the ^ symbol is used before each label in fig.2.

5.2 Processing a Sample Input String

Table 2 shows the parsing of the string "101" based on processing the labels according to Table 1, for each transition of the transducer in fig.2.

Symbol	Transition	Label	Output	Semantic Action
[1] 0 1	1 → 31	[	[(	
	31 → 61	{ }	[(	
	61 → 81	1	[((1)	
	81 → 51	$\{C\}X_2^G$	[((1)X_2^G)	{C} (value = 1)
	51 → 41		[((1)X_2^G)	
	41 → 91	{ }	[((1)X_2^G)	
1 [0] 1	91 → 101	0	[((1)X_2^G)(0)	
	101→ 51	$\{D\}X_3^L$	[((1)X_2^G)(0)X_3^e)	{D} (value = 2)
	51 → 41		[((1)X_2^G)(0)X_3^L)	
	41 → 111	{ }	[((1)X_2^G)(0)X_3^L)	

1 0 [1]	111 → 121	1	[((1)X_2^G)(0)X_3^L)(1)	
	121 → 51	{E}X_4^L	[((1)X_2^G)(0)X_3^L)(1)X_4^L)	{E} (value = 5)
	51 → 41		[((1)X_2^G)(0)X_3^L)(1)X_4^L)	
	41 → 21		[((1)X_2^G)(0)X_3^L)(1)X_4^L)	
	21 → 11	{A}S_0^e]	[((1)X_2^G)(0)X_3^L)(1)X_4^L)]	{A} (print 5)

Table 2: parsing of the string "101", with output generated and semantic actions performed.

In this case, the execution of the semantic actions during the output analysis is possible because all the actions occurs after each transition.

In grammars where actions are specified to be applied before processing the non-terminals, other order for action implementation would be chosen.

The printed result is 5, corresponding to the binary numeral 101 given as input.

The parsing of string 101 generates the output [((1)X_2^G)(0)X_3^L)(1)X_4^L)]. Here, X_4^L corresponds to the root of the tree. The number 4 shows the number of the original grammar's rule applied (4^{th} rule, starting at 0), the letter L shows that this rule refers to a left-recursive non-terminal.

This node has as descendants two sub-trees: the terminal (1) and ((1)X_2^G)(0)X_3^L). X_3^L is a node that represents the 3^{rd} rule, is left-recursive and has as descendants the terminal (0) and ((1)X_2^G). X_2^G is a node that represents the 2^{nd} rule, and corresponds to a general (non-recursive) rule and has the terminal (1) as a descendant. The tree can then be constructed as follows:

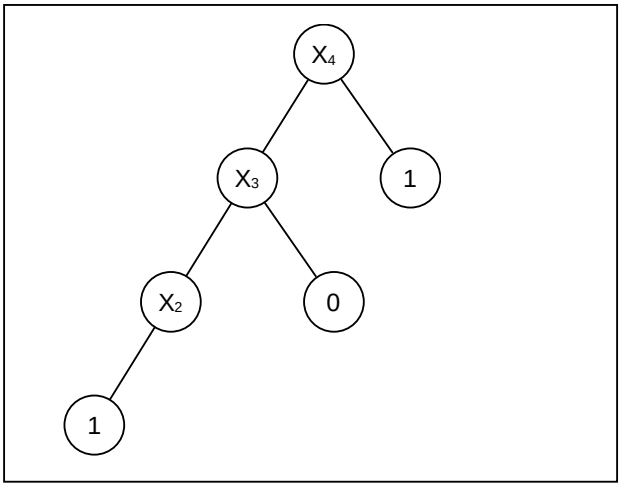

Fig. 3 – Parse tree generated for the sentence 101

6 Conclusion

Although there are many good and well-known classical methods for the construction of parsers from context-free grammars, we have shown, in this paper, an alternative method.

This is a very practical method that allow extending the automatically generated structured pushdown automata-based acceptors described in [3],[4] by including calls to semantic actions specified along the rules of the grammatical description of the language to be parsed, given in Wirth's notation.

The resulting transducer is fully compatible with the original one, and is able to simultaneously accept the input sentence, generate the corresponding parse tree and perform the specified semantic actions.

The application of this method demands almost no practice and theoretical background from its users. It also shows to be very inexpensive, attractive and easy to understand and learn, so it is advisable for being used not only in real applications but especially for educational purposes too.

References:

[1] APPEL, ANDREW, *Modern Compiler Implementation in C.* 1.ed, The Press Syndicate of The University of Cambridge, 1997.

[2] PRICE, A. M.A.; TOSCANI, S.S., *Implementação de Linguagens de Programação: Compiladores* 2.ed. Porto Alegre: Editora Sagra Luzzatto, 2004.

[3] NETO, J. J., *Contribuições à Metodologia de Construção de Compiladores:* Tese para a obtenção do título de Professor Livre-Docente junto ao Departamento de Engenharia de Computação e Sistemas Digitais PCS-EPUSP, 1993

[4] NETO, J.J.; PARIENTE, C.B.; LEONARDI, F., Compiler Construction – A Pedagogical Approach, *ICIE*, 1999

[5] TREMBLAY, J. P.; SORENSON, P.G., *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985.

[6] CONWAY, M. E., *Design of a Separable Transition Diagram Compiler Communications of the ACM*, 6, 7, 1963, pp. 396-408.

[7] LOMET, *D. B. A Formalization of Transition Diagram Systems Journal of The ACM*, 20, 2, , 1973, pp. 235-257.

[8] IWAI, M.K. *Um Formalismo Gramatical Adaptativo Para Linguagens Dependentes de Contexto:* Tese para a obtenção do título de Doutor em Engenharia PCS-EPUSP, 2000