Global Index Languages

A Dissertation

Presented to

The Faculty of the Graduate School of Arts and Sciences

Brandeis University

Department of Computer Science

James Pustejovsky, Advisor

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

by

José M. Castaño

February, 2004

The signed version of this dissertation page is on file at the Graduate School of Arts and Sciences at Brandeis University.
This dissertation, directed and approved by José M. Castaño's committee, has been accepted and approved by the Graduate Faculty of Brandeis University in partial fulfillment of the requirements for the degree of:

DOCTOR OF PHILOSOPHY

Adam Jaffe, Dean of Arts and Sciences

Dissertation Committee:
James Pustejovsky, Dept. of Computer Science, Chair.
Martin Cohn, Dept. of Computer Science
Timothy J. Hickey, Dept. of Computer Science
Aravind K. Joshi, University of Pennsylvania

Al Campris y su eterna perserverancia.

A la Pipi y su gran amor.

Debes amar
la arcilla que va en tus manos
debes amar
su arena hasta la locura
y si no,
no lo emprendas, que será en vano

S. Rodríguez.

Acknowledgments

Many people contributed to this work, in many different ways. Also many circumstances led me to this unexpected path, far away from my initial interests when I arrived at Brandeis.

First and foremost, I would like to express my gratitude to James Pustejovsky who supported and encouraged me during these years. I would like to thank Martin Cohn for introducing me to Formal Languages and having always time for me. My sincere thanks to Tim Hickey and Ray Jackendoff. I am extremely grateful to Aravind Joshi for being part of my committee and for his precise comments and suggestions. Edward Stabler and Stuart Shieber helped me to find a way at the initial stages of this endeavour.

Many thanks to my Argentinian friends and colleagues, Aldo Blanco, José Alvarez, Ricardo Rodríguez, Ana Arregui, Alejandro Renato and the extraordinary friends I met at Brandeis, Pablo Funes, Francesco Rizzo, María Piñango, Domingo Medina, Marc Verhagen, Piroska Csúri, Anna Rumshisky, Roser Saurí and Bob Knippen, my American friend.

I cannot express how grateful I am to my *compañera* Daniela in this and every route. My family has a very special dedication.

It is my belief that every intellectual work is fruit of a community. This work was influenced by many people, some I don't know. My highest appreciation to those anonymous reviewers who made an effort to understand and comment on my submissions during these years.

ABSTRACT

Global Index Languages

A dissertation presented to the Faculty of
the Graduate School of Arts and Sciences of
Brandeis University, Waltham, Massachusetts

by José M. Castaño

Context-free grammars (CFGs) is perhaps the best understood and most applicable part of formal language theory. However there are many problems that cannot be described by context-free languages: "the world is not context-free" [25]. One of those problems concerns natural language phenomena. There is increasing consensus that modeling certain natural language problems would require more descriptive power than that allowed by context-free languages. One of the *hard* problems in natural language is *coordination*. In order to model coordination, it might be necessary to deal with the multiple copies language: $\{ww^+ \mid w \in \Sigma^*\}$ [29].

Innumerable formalisms have been proposed to extend the power of context-free grammars. *Mildly context-sensitive* languages and grammars emerged as a paradigm capable of modeling the requirements of natural languages. At the same time there is a special concern to preserve the 'nice' properties of context-free languages: for instance, polynomial parsability and semi-linearity. This work takes these ideas as its goal.

This thesis presents an abstract family of languages, named Global Index Languages (GILs). This family of languages preserves the desirable properties mentioned above: bounded polynomial parsability, semi-linearity, and at the same time has an "extra" context-sensitive descriptive power (e.g the "multiple-copies" language is a GI language). GIls descriptive power is shown both in terms of the set of string languages included in GILs, as well as the *structural descriptions* generated by the corresponding grammars. We present a two-stack automaton model for GILs and a grammar model (Global Index Grammars) in Chapter 2 and 3. Then characterization and representation theorems as well as closure properties are also presented. An Earley type algorithm to solve the recognition problem is discussed in Chapter 5, as well as an LR-parsing algorithm for the deterministic version of GILs. Finally, we discuss in Chapter 6 the relevance of Global Index Languages for natural language phenomena.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Context-free grammars (CFGs) constitute perhaps the most developed, well understood and most widely applicable part of formal language theory. However there are many problems that cannot be described by context-free languages. In [25] seven situations which cannot be described with a context-free framework are given.

The purpose of this thesis is to explore an alternative formalism with restricted context-sensitivity and computational tractability. We try to obtain a model with efficient computation properties. Such a model is relevant, at least, for the following areas:

- Natural Language modeling and processing.
- Computational Biology
- Possibly also relevant to programming languages which might use context sensitive features (like Algol 60).

Many formalisms have been proposed to extend the power of context-free grammars using *control* devices, where the control device is a context-free grammar (see [87] and [26] regarding control languages). In an automaton model those extensions either use constrained embedded or additional stacks (cf. [7, 81, 59]). The appeal of this approach is that many of the attractive properties of context-free languages are preserved (e.g. polynomial parsability, semilinearity property, closure properties).

Such models can be generalized and additional control levels, additional levels of embeddedness, or additional stacks can be added. They form hierarchies of levels of languages, where a language of level k properly includes a language of level $k-1$. Examples of those related hierarchies are: Khabbaz's [49], Weir's [87], multiple context free grammars [71], and multi-push-down grammars [21]. As a consequence those generalizations provide more expressive power but at a computational cost. The complexity of the recognition problem is dependent on the language level. In general, for a level k language the complexity of the recognition problem is a polynomial where the degree of the polynomial is a function of k. We will exemplify this issue with some properties of natural language and we will consider some biology phenomena, that can be modeled with GILs.

We will focus in natural language problems from a language theoretic point of view, in particular related to the so called *Mildly-Context Sensitive formalisms*. Within Mildly-Context Sensitive formalisms we focus on Linear Indexed Grammars (LIGs) and Tree Adjoining Grammars (TAGs) which are weakly equivalent.

1.1.1 Natural Language

Regarding natural languages (one of the seven cases mentioned above), there is increasing consensus[1] on the necessity of formalisms more powerful than CFGs to account certain typical natural languages constructions. These constructions are exemplified as [26]:[2]

- *reduplication*, leading to languages of the form $\{ww \mid w \in \Sigma^*\}$
- *multiple agreements*, corresponding to languages of the form $\{a^n b^n c^n \mid n \geq 1\}$, $\{a^n b^n c^n d^n \mid n \geq 1\}$, etc.
- *crossed agreements*, as modeled by $\{a^n b^m c^n d^m \mid n\ m \geq 1\}$

Mildly context-sensitive grammars [43] have been proposed as capable of modeling the above mentioned phenomena.[3] It has been claimed that a NL model must have the following properties:[4]

[1] See for example [73, 23, 29], among others.

[2] The following type of languages are also relevant to model molecular biology phenomena [70]

[3] However there are other phenomena (e.g. *scrambling*, Georgian Case and Chinese numbers) might be considered to be beyond certain *mildly context-sensitive* formalisms.

[4] See for example, [46, 86] and [55, 47] on *semi-linearity*.

- *Constant growth* property (or *semi-linearity*, a stronger version of this requirement)
- Polynomial parsability
- *Limited cross-serial* dependencies
- Proper inclusion of context-free languages

The most restricted *mildly context-sensitive* grammar formalisms are linear indexed Grammars (LIGs), head grammars (HGs), tree adjoining grammars (TAGs), and combinatory categorial grammars (CCGs) which are weakly equivalent. These formalisms have long been used to account for natural language phenomena. Many different proposals for extending the power of LIGs/TAGs have been presented in the recent past. However, an increase in descriptive power is correlated with an increase in time and space parsing complexity. One of the characteristics of *mildly context sensitive* formalisms is that they can be described by a geometric hierarchy of language classes. In such hierarchies there is a correlation between the complexity of the formalism (grammar or automata), its descriptive power, and the time and space complexity of the recognition problem. Tree Adjoining languages belong to the class $\mathcal{L}_2$ in Weir's hierarchy. A TAG/LIG is a level-2 control grammar, where a CFG *controls* another CFG. Their expressive power regarding *multiple agreements* is limited to 4 counting dependencies (e.g., $L = \{a^n b^n c^n d^n | n \geq 1\}$). The equivalent machine model is a nested or embedded push down automata (NPDA or EPDA). They have been proven to have recognition algorithms with time complexity $\mathsf{O}(\mathsf{n}^6)$ considering the size of the grammar a constant factor ([67, 62]).

More expressive power can be gained in a *mildly context-sensitive* model by increasing the progression of this mechanism: in the grammar model adding another level of grammar control, and in the machine model allowing another level of embeddedness in the stack.

This is reflected in the pumping lemma. A lemma for level k of the hierarchy involves "pumping" 2^k positions in the string. Therefore the following relations follow:

The family of level-k languages contains the languages

$\{a_1^n ... a_{2^k}^n \mid n \geq 0\}$ and $\{w^{2^{k-1}} | w \in \Sigma^*\}$

but does not contain the languages

$\{a_1^n ... a_{2^k+1}^n \mid n \geq 0\}$ and $\{w^{2^{k-1}+1} | w \in \Sigma^*\}$

An increase in the language level increases the time (and space) complexity but it still is in P: for a level-k control grammar (or a level-k of stack embeddedness, or ordering) the recognition problem is in $\mathsf{O}(\mathsf{n}^{3\cdot 2^{k-1}})$ [86, 87].

There are other formalisms related to *mildly context-sensitive* languages that are not characterized by a hierarchy of grammar levels, but their complexity and expressive power is bound to some parameter in the grammar (in other words the expressive power is bound to a parameter of the grammar itself). Some of those formalisms are: Minimalist grammars [78, 38, 54], Range Concatenative Grammars [34, 10, 12], and the related Literal Movement Grammars (LMGs). For LMGs the complexity of the recognition problem is $O(|G|m(+p)n^{1+m+2p})$ where p is the arity of the predicates, and m the number of terms in the productions.

In Coupled Context-Free Grammars the recognition problem is in $O(|P|\cdot n^{3l})$ where l is the rank of the grammar [28] and Multiple Context Free Grammars is $O(n^e)$ where e is the degree of the grammar. Multivalued Linear Indexed Grammars [63], Tree Description Grammars [47] are other such examples.

Contextual Grammars [52, 66] is another formalism with some *mildly context sensitive* characteristics. However, Contextual Languages do not properly include context-free languages. In this case the complexity of the recognition problem is a bound polynomial: $O(n^6)$ space and $O(n^9)$ time [36].

1.1.2 Biology and the Language of DNA

Recently, generative grammars have been proposed as models of biological phenomena and the connections between linguistics and biology have been exploited in different ways (see [40, 69]).

Gene regulation, gene structure and expression, recombination, mutation and rearrangements, conformation of macromolecules and the computational analysis of sequence data are problems which have been approached from such a perspective. Searls [70, 68] proposes a grammar model to capture the *language of DNA*. He shows that the language of nucleic acids is not regular, nor deterministic nor linear, that it is ambiguous and that it is not context-free. Those problems which are beyond context free are: *Tandem Repeats* (multiple copies), *Direct Repeats* (the copy language), *Pseudoknots* (crossing dependencies) represented as: $L_k = \{uv\bar{u}^R\bar{v}^R\}$.

Unbounded number of repeats, inverted repeats, combinations, or *interleaved* repeats, are characteristic of the language of DNA. Probably the work by Searls is the most salient in this field. He proposed String Variable Grammars to model the language of DNA. String Variable Languages are a subset of Indexed Languages, and they are able to describe those phenomena. However there is no known algorithm that decides the recognition problem for String Variable Languages. Recently Rivas and Eddy [64] proposed an algorithm which is $O(n^6)$ to predict RNA structure including pseudoknots, and proposed a formal grammar [65] corresponding to the language that can be parsed using such an algorithm.

1.1.3 Our proposal

Unlike *mildly context-sensitive* grammars and related formalisms, in the model we present here, which we call Global Index Languages, (GILs), there is no correlation between the descriptive power (regarding the three phenomena we mentioned) and the time complexity of the recognition problem, which is in time $O(n^6)$. We will show that the automaton and grammar model proposed here extends in a natural way the properties of Push Down Automata and context-free languages. We will also show that it is able to describe phenomena relevant both to natural language and biology.

The following languages can be generated by a Global Index Grammar:

- $L_m = \{{a_1}^n {a_2}^n ... {a_k}^n \mid n \geq 1, k \geq 2\}$ any number of agreements or dependencies.
- $L(G_{wwn}) = \{ww^+ \mid w \in \{a,b\}^*\}$ unbounded number of copies or repeats (which allows also an unbounded number of crossing dependencies).
- $L(G_{mix}) = \{w | w \in \{a,b,c\}^* \ and \ |a|_w = |b|_w = |c|_w \geq 1\}$ The mix language, which is conjectured not to be an Indexed Language.
- $L(G_{genabc}) = \{a^n (b^n c^n)^+ | \ n \geq 1\}$ unbounded number of agreements.
- $L(G_{sum}) = \{ \ a^n b^m c^m d^l e^l f^n \mid n = m + l \geq 1\}$, dependent branches

There are many similarities between Linear Indexed Grammars and Global Index Grammars, however there are many GI languages which do not belong to the set of Linear Indexed Languages

(LILs). This suggests that Linear Indexed Languages might be properly included in the family of Global Index Languages. The recognition problem for Global Index Languages (GILs) has a bounded polynomial complexity, and there is no correlation between the recognition problem complexity and the expressive power. GILs include multiple copies languages and languages with any finite number of dependencies as well as the MIX language. We also discuss the structural descriptions generated by GIGs along the lines of Gazdar [29] and Joshi [82, 44]. We show in Chapter 6 that the relevant structural descriptions that can be generated by LIGs/TAGs can also be generated by a GIG. However GIGs can generate structural descriptions where the dependencies are expressed in dependent paths of the tree. GILs are also semi-linear, a property which can be proved following the proof presented in [37] for counter automata. Therefore GILs have at least three of the four properties required for mild context sensitivity: a) semi-linearity b) polynomial parsability and c) proper inclusion of context free languages. The fourth property, *limited cross-serial dependencies* does not hold for GILs, given they contain the MIX (or Bach) language [11].

In this work we will address three aspects of the problem: language theoretic issues, parsing and modeling of natural language phenomena. We detail the properties of GILs in the following section.

1.2 Outline of the Thesis and Description of the Results

Language Theoretic Aspects (Chapters 2-4)

We define the corresponding Automaton model, which we call LR-2PDAs in Chapter 2 (a significant part of this chapter was published in [18]). We present the corresponding grammar model : Global Index Grammars (GIGs), and the family of Global Index Languages (GILs) in Chapter 3. The initial presentation of Global Index Languages was published in [17]. A subset of GIGs (trGIGs), is also presented in Chapter 3, although no further implications are pursued. In addition, a proof of equivalence between the grammar model and the automaton model (the characterization Theorem) and a Chomsky-Schützenberger representation Theorem are presented in Chapter 3.

In Chapter 4 we discuss some of the closure properties of Global Index Languages and their relationship with other languages. The set of Global Index Languages is an Abstract Family of Languages. An abstract family of languages (AFL) is a family of languages closed under the following

six operations: union, catenation, Kleene star, intersection with regular languages, free morphisms and inverse morphisms. We show by construction using GIGs that GILs are closed under union, concatenation and Kleene star. We also show that GILs are closed under intersection with regular languages and inverse homomorphism (proved by construction with LR-2PDAs). We also show that GILs are closed under substitution by e-free CFLs, e-free regular sets, e-free homomorphism. Most of the properties of GILs shown in these chapters are proved using straightforward extensions of CFL proofs. This is due to the fact that GILs inherit most of CFL properties. We also prove that GILs are semi-linear.

Parsing (Chapter 5)

We present a recognition algorithm that extends Earley's algorithm for CFGs using a graph-structured stack [80] to compute the operations corresponding to the *index* operations in a GIG. The graph-structured stack enables the computations of a Dyck language over the set of indices and their complements. The time complexity of the algorithm is $O(n^6)$. However it is $O(n)$ for LR or state bound grammars and is $O(n^3)$ for *constant indexing* grammars which include a large set of ambiguous indexing grammars. The space complexity is $O(n^4)$. We evaluated the impact of the size of the grammar and the indexing mechanism separately. The result, $O(|I|^3 \cdot |G|^2 \cdot n^6)$, shows that the size of the grammar has the same impact as in the CFG case, but the indexing mechanism could have a higher impact.

We also give an LR parsing algorithm for for deterministic GILs. Deterministic LR parsing techniques are relevant, for example, for higher level programming languages or deterministic approaches for shallow parsing in natural language. They are also the basis for non-deterministic approaches, such as a non-deterministic Generalized LR parsing. The construction of the *Goto* function shows an interesting (though not surprising) fact about GILs: the *Goto* function defines a PDA that recognizes the set of viable prefixes.

Natural Language Modeling (Chapter 6)

We present a comparison of LIGs and GIGs (some of these results were presented in [20]). LIGs, Combinatory Categorial Grammars, Tree Adjoining Grammars and Head Grammars are weakly

equivalent. There is a long and fruitful tradition that has used those formalisms for Natural Language modeling and processing. GILs include CFLs by definition, so the descriptive power of CFGs regarding by natural language phenomena is inherited by GIGs. Those natural language phenomena that require an extra power beyond CFLs can be dealt with GILs with much more flexibility than with LILs/TALs. We also show that GIGs can generate trees with dependent paths. The set inclusion relation between CFLs, LILs/TALs, and GILs can be represented as in figure 1.1. The question still to be answered is whether the area with lines in the LILs and LCFR languages set is empty or not. Our conjecture is that it is. We discuss the relationship of LIGs with GILs in depth in section 6.2.

Finally, we disscuss how some properties of HPSGs can be modeled with GIGs (presented in [19]. Other formalisms that might share structural similarities with GIGs could be Minimalist Grammars and Categorial Grammars.

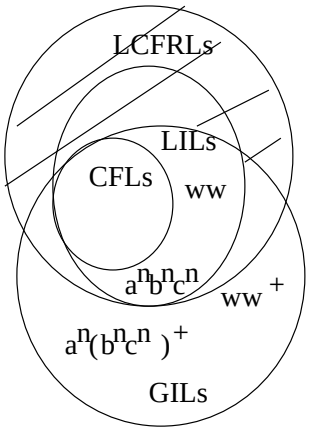


Figure 1.1: Language relations

Chapter 2

Two Stack Push Down Automata

2.1 Automata with two Stacks

Different models of 2-Stack Push Down Automata have been proposed, (e.g.,[31, 83, 9], also generalized models of multistacks (e.g.,[7, 21, 84]). Restricted classes of PDA with two stacks (2-PDA) were proposed as equivalent to a second order EPDA[1], and n-Stack PDA, or multi-store PDAs as equivalent to n-order EPDA.[2] It is well known that an unrestricted 2-PDA is equivalent to a Turing Machine (cf. [41]). Those models restrict the operation of additional stacks (be embedded or ordered) allowing to *read* or *pop* the top of an additional stack only if the preceding stacks are empty.

The model of 2-PDA that we want to explore here is a model which preserves the main characteristics of a PDA but has a restricted additional storage capacity: allows accessing some limited **left context** information, but behaves like a PDA, in the sense that it is forced to consume the input from left to right and has a limited capacity to go back from right to left.

We need to guarantee that the proposed automaton model does not have the power of a Turing Machine. The simulation of a TM by a 2-PDA is enabled by the possibility of representing the head position and the tape to the left side of the head in a TM with one stack and the tape to the right side with the second stack. Consequently it is possible to simulate the capacity that a TM

[1] See [7, 81, 59]

[2] E.g., [21, 84, 85].

has to move from left to right and from right to left, either reading or replacing symbols. In order to simulate these movements in a 2-PDA, the input must be consumed and it must be possible to transfer symbols from one stack to the other. Alternatively, the input might be initially stored in the second stack (e.g., [83, 13]).

We adopt the following restrictions to the transitions performed by the kind of 2-PDA we propose here: the Main stack behaves like an ordinary PDA stack (allows the same type of transitions that a PDA allows). However transitions that push a symbol into the second auxiliary stack do not allow ϵ moves.[3]

These constraints force a left to right operation mode on the input in the LR-2PDA, because once the input is consumed, it is not possible to write into the auxiliary stack. Consequently it is possible to "use" the information stored in the auxiliary stack only once. Moving back to the left, "erases" the symbols as they are removed from the stack. Given that it is not possible to re-store them again in the auxiliary stack, they cannot be recovered (unless some more input is consumed). A more detailed comparison of TMs, PDAs, and the LR-2-PDA is presented in section 2.3. Its expressive power is beyond PDA (context free grammars) and possibly beyond EPDA (e.g., TAGs), a 2nd order PDA in the progression hierarchy presented in [86]. In the next two subsections we review some models that proposed 2-stack or nested stack automata. In 2.3 we define the LR-2PDA.

2.1.1 EPDA and equivalent 2 Stack Machines

An EPDA, M', is similar to a PDA, except that the push-down store may be a sequence of stacks. It is initiated as a PDA but may create new stacks above and below the current stack. The number of stacks is specified in each move. Then a transition function looks like (from [45]):

δ' (input symbol, current state, stack symbol)=

(new state, $sb_1, sb_2, ..., sb_m$, push/pop on current stack,$st_1, st_2, ..., st_n$)

where $sb_1, sb_2, ..., sb_m$ are a finite number of stacks introduced below the current stack, and $st_1, st_2, ..., st_n$ are stacks introduced above the current stack.

[3]We might restrict even more the operation of the auxiliary stack disallowing ϵ moves that *pop* symbols from the auxiliary Stack. Consequently this would produce a more restricted set of languages.

The possibility of swapping symbols from the top stack to embedded stacks produces the context sensitive power of $\mathcal{L}_2$ languages in Weir's hierarchy. It allows the NPDA, for instance, to recognize *crossed dependencies* as in the example depicted in the next figure:

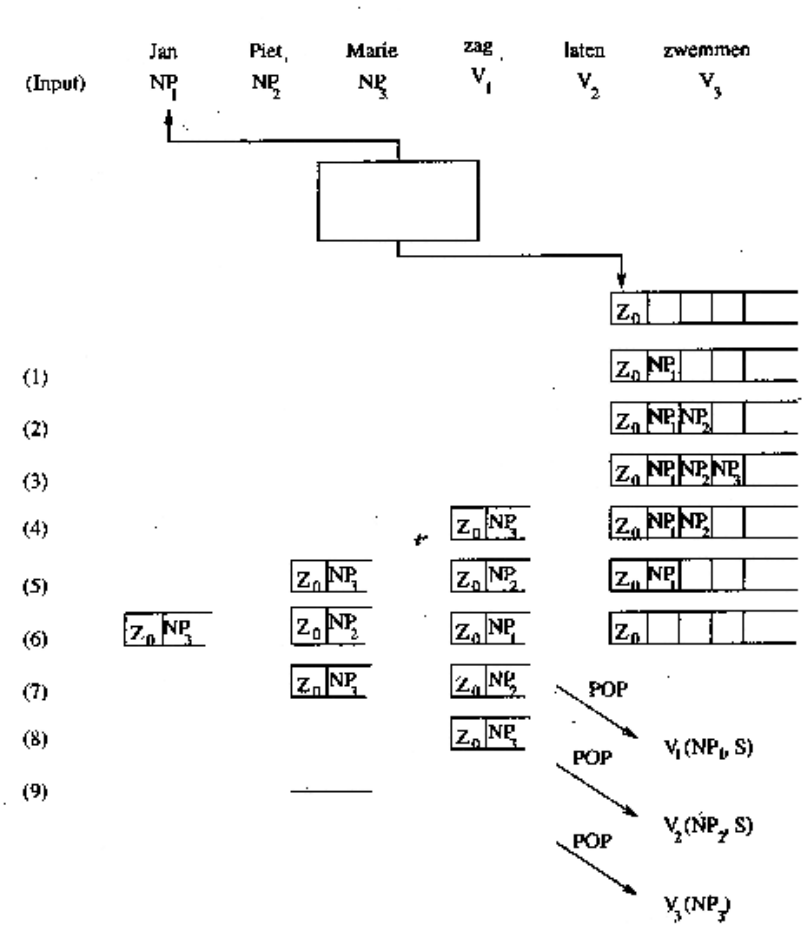


Figure 2.1: Dutch Crossed Dependencies (Joshi & Schabes 97)

2 Stack automata (2-SA) were shown to be equivalent to EPDA [7, 81, 59], they use the additional stack in the same way as the embedded stacks are used in the EPDA. Symbols written in the second stack can be accessed only after the preceding stack is empty. They also require the use of a symbol for the bottom of the stack, so as to separate different sub-stacks.

The possible moves for a 2-SA automaton are the following [7]:

- Read a symbol from the input tape

- push a string onto, or pop a symbol from, the main stack
- push a string onto the auxiliary stack.
- pop a symbol from the auxiliary stack provided the main stack contains the bottom of the stack symbol
- create a new set of empty stacks (add a string of bottom of the stack symbols).
- remove bottom of the stack symbols.

Wartena [85] shows the equivalence of the concatenation of one-turn multi-pushdown automata restricted with respect to reading (as in the 2-SA case), and the alternative restriction with respect to writing. The reading/writing restriction is related to the requirement that the preceding stacks be empty. The kind of automata we present implements a restriction on the writing conditions on the additional stack. But this restriction is not related to conditions on the the main stack but to conditions on the input.

2.1.2 Shrinking Two Pushdown Automata

Recently there has been interest on Church-Rosser Languages, Growing Context Sensitive Languages (defined by a syntactic restriction on context sensitive grammars) and the corresponding model of two pushdown automata (see [24, 16, 15, 53, 58]. In [16] Buntrock and Otto's Growing Context Sensitive Languages (GCSLs) are characterized by means of the Shrinking Two Pushdown Automata (sTPDA). GCSLs are a proper subclass of CSLs: The Gladkij language $\{w\#w^R\#w \mid w \in \{a,b\}^*\}$ belongs to CSL $\setminus$ GCSLs. GCSLs contain the non-context-free and non-semi-linear $\{a^{2^n} \mid n \geq 0\}$ [16]

The copy language is not a GCSL $\{ww|w \in \{a,b\}^*\}$, but $\{a^n b^n c^n | n \geq 1\}$ is a GCSL. The recognition problem for growing context sensitive grammars is NP-complete ([14]).

The TPDA initializes the input in one of the stacks. The shrinking-TPDA, is defined as follows:

Definition 1 *a) A TPDA is a 7-tuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$*, where*

- Q *is the finite set of states,*

- Σ *is the finite input alphabet,*
- Γ *is the finite tape alphabet with* $\Gamma \supsetneq \Sigma$ *and* $\Gamma \cap Q = \varnothing$
- $q_0 \in Q$ *is the initial state,*
- $\perp \in \Gamma \setminus \Sigma$ *is the bottom marker of pushdown store,*
- $F \subseteq Q$ *is the set of final (or accepting) states, and*
- $d : Q \times \Gamma \times \Gamma \to \mathcal{P}(Q \times \Gamma^* \times \Gamma^*)$

It can be seen that this kind of automaton needs some restriction, otherwise it is equivalent to a TM. The restriction is implemented using a weight function over the automaton configuration, and the requirement that every move must be weight reducing [57].

b) A TPDA is M is called shrinking if there exists a weight function $\phi : Q \cup \Gamma \to \mathbb{N}_+$ *such that for all* $q \in Q$ *and* $a, b \in \Gamma$ *if* $(p, u, v) \in (\delta, a, b)$ *then* $\phi(puv) < \phi(qab)$.

We can consider the automaton model we present in the next section as preserving a *shrinking* property that conditions the use of the additional stack: it has to consume input.

2.2 Left Restricted 2 Stack Push Down Automatas (LR-2PDAs)

We define here the model of automaton that we described and proposed at the beginning of this section. We use a standard notation similar to the one used in [77] or [41]. The current state, the next symbol read and the top symbols of the stacks determine the next move of a LR-2PDA. Either of these symbols may be ϵ, causing the machine to move without reading a symbol from the input or from the stacks. The definition of the transition function δ ensures that transitions with epsilon moves are allowed only if they do not push a symbol into the auxiliary stack, i.e., only if they are equivalent to a PDA transition function. Part a) in the definition of the transition function is equivalent to a PDA. Part b) enables transitions that affect the auxiliary stack if and only if an input symbol is consumed (i.e. if it is not ϵ). Part c) is a relaxation of the previous one and enables

ϵ-moves that pop elements from the second stack. If part c) is removed from the definition a different class of automata is defined: we will call it sLR-2PDA. For Natural Language purposes it might be desirable to keep stronger restrictions on the auxiliary stack. The auxiliary stack should be emptied only *consuming* input. Such a restriction could be accomplished in the sLR-2PDA case because the auxiliary stack cannot be emptied using ϵ-moves, but this would rule out reduplication (copy) languages (See the introduction) as we will see below. The set of languages that can be recognized by a sLR-2PDA is therefore a proper subset of the set of languages that can be recognized by a LR-2PDA.

Definition 2 *A LR-2PDA is a 7-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ *where* $Q, \Sigma, \Gamma,$ *and* F *are all finite sets and*

1. Q *is the set of states,*
2. Σ *is the input alphabet,*
3. Γ *is the stacks alphabet,*
4. δ *is the transition function (where* $\mathcal{P}_F$ *denotes a finite power set, and* Γ_ϵ *is* $\Gamma \cup \{\epsilon\}$ *) :*
   a. $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma_\epsilon \times \{\epsilon\} \rightarrow \mathcal{P}_F(Q \times \Gamma^* \times \{\epsilon\})$ *and*
   b. $Q \times \Sigma \times \Gamma_\epsilon \times \Gamma \rightarrow \mathcal{P}_F(Q \times \Gamma^* \times \Gamma^2)$
   c. $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma_\epsilon \times \Gamma \rightarrow \mathcal{P}_F(Q \times \Gamma^* \times \{\epsilon\})$[4]
5. $q_0 \in Q$ *is the start state*
6. $\bot \in \Gamma \setminus \Sigma$ *is the bottom marker of the stacks*
7. $F \subseteq Q$ *is the set of accept states.*

Definition 3 *A sLR-2PDA is a LR-2PDA where* δ *is:*

a. $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma_\epsilon \times \{\epsilon\} \rightarrow \mathcal{P}_F(Q \times \Gamma^* \times \{\epsilon\})$ *and*

[4]The two transition types a. and c. can be conflated in:

a'. $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}_F(Q \times \Gamma^* \times \{\epsilon\})$.

We keep them separate to show the parallelism with the type of productions in the grammar model.

b. $Q \times \Sigma \times \Gamma_\epsilon \times \Gamma \to \mathcal{P}_F(Q \times \Gamma^* \times \Gamma^*)$

The *instantaneous descriptions* (IDs) of a LR-2PDA describe its configuration at a given moment. They specify the current input, state and stack contents. An ID is a 4-tuple $(q, w, \gamma_1, \gamma_2)$ where q is a state w is a string of input symbols and γ_1, γ_2 are strings of stack symbols. If $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ is a LR-2PDA, then $(q, aw, Z\alpha, O\beta) \vdash_M (p, w, \zeta\alpha, \eta\beta)$ if $\delta(q, a, Z, O)$ contains (p, ζ, η). The reflexive and transitive closure of $\vdash_M$ is denoted by $\vdash_M^*$. The *language* accepted by a LR-2PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ is $\{w \mid (q_0, w, \perp, \perp) \vdash^* (p, \epsilon, \perp, \perp)\}$ for some p in F. (Acceptance by empty stacks)

2.3 Comparison with TM, PDA and LR-2PDA

A Turing Machine: the input tape is also a storage tape. The input can be transversed as many times as desired, plus it has an unlimited work-space, that can be also used and reused at any time.

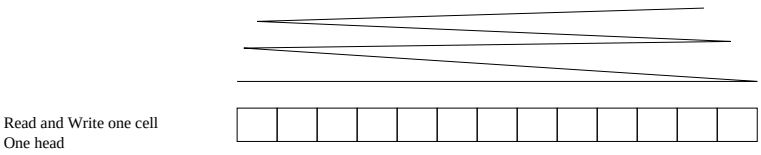


Figure 2.2: A Turing Machine

A PDA: The input can be transversed only once, from left to right. There is a work space (storage data structure) that also can be transversed only once (as soon there is a move to the left, whatever remains at the right is forgotten). A clear example of maximum transversal and input storage is the language $L_r = \{ww^R \mid w \in \Sigma^*\}$ which can be recognized by a PDA. It has the capacity of remembering a sequence of symbols and matching it with another equal length sequence, but it cannot recognize $L_c = \{ww \mid w \in \Sigma^*\}$ which would require additional use of the storage capacity.

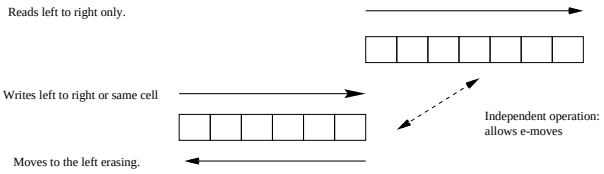


Figure 2.3: A PDA

A LR-2PDA: The input can be transversed only once, from left to right. There are two workspaces which also can be transversed only once (as soon there is a move to the left, whatever remains at the right is forgotten). Transfer from one stack to the other is restricted. In order to avoid the possibility of transferring symbols from one stack to the other indefinitely, the second storage can be written to only when consuming input; if information from the first ordered stack is transferred to the second auxiliary stack, then the possibility of storing information from the input seen at this move is lost. The following languages which can be recognized by an LR-2PDA are examples of a maximum use of the storage of the input: $L_{2ra} = \{ww^R w^R w \mid w \in \Sigma^*\}$ and $L_{2rb} = \{ww^R ww^R \mid w \in \Sigma^*\}$. This exemplifies how many times (and the directions in which) the same substring can be transversed, but unlike a TM, a LR-2PDA is not able to transverse a substring indefinitely. This capacity is reduced even further in the case of sLR-2PDA: both types of operations on the auxiliary stack are made dependent on consuming input. However, given the constraints imposed on LR-2PDAs, an LR-PDA can recognize the language $L_3 = \{www \mid w \in \Sigma^*\}$. While recognizing the second copy all the information stored about the first copy is lost, but it can store again in the second stack (the second copy). Consider the following transitions (where a is a symbol in Σ). The corresponding machine is detailed below in the exsection 2.4.

1. $(q_1, a, \epsilon, \epsilon) = (q_1, \epsilon, a)$ (store the symbols of the first copy)

2. $(q_2, \epsilon, \epsilon, a) = (q_2, a, \epsilon)$ (transfer symbols from auxiliary to main stack)

3. $(q_3, a, a, u) = (q_3, \epsilon, au)$ (recognize the second copy using the symbols stored in the main stack, and store the symbols of the second copy back

A loop back from state q_3 to state q_2 allows recognition of the language of any finite number of copies (example 2).

A sLR-2PDA cannot recognize the copy language: if the auxiliary stack is used to store all the symbols of the input tape then it would be of no more use than storing all the symbols in the main stack: the symbols can only be transferred from the auxiliary stack to the main stack only if the input tape is being read, consequently what is being read cannot be stored in the main or the auxiliary stack if there is a *transferral* occurring. The crucial fact is that transitions type c. (above)

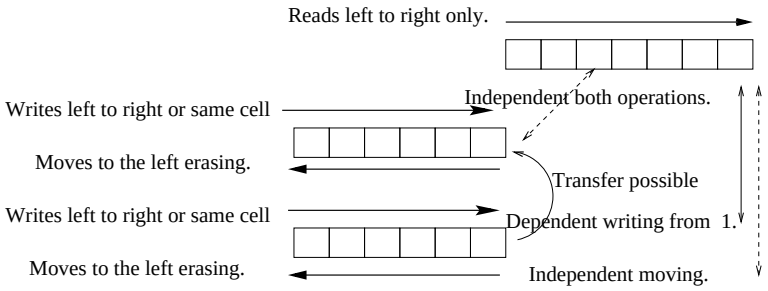


Figure 2.4: LR-2PDA

are not allowed. However the language $L = \{wuw \mid u, w \in \Sigma^* \; and \; |w| \leq |u|\}$ is recognized by a sLR-2PDA.

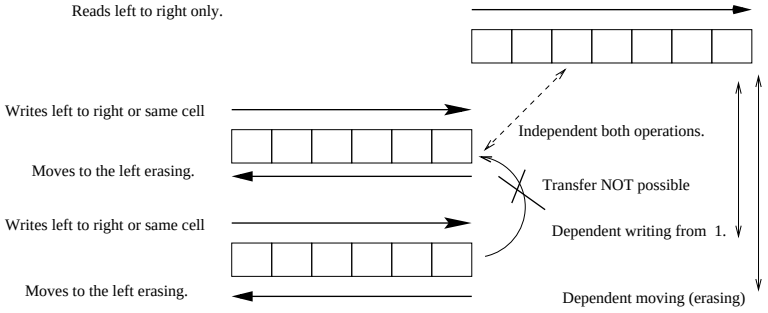


Figure 2.5: sLR-2PDA

2.4 Examples of LR-2PDAs

The first example is a sLR-2PDA. Examples 2 and 3 are LR-2PDAs.

Example 1 (Multiple Agreements) .

$M_5 = (\{q_0, q_1, q_2, q_3\}, \{a, b, c, d, e\}, \{x\}, \delta, q_0, \perp, q_3)$ *where* δ*:*

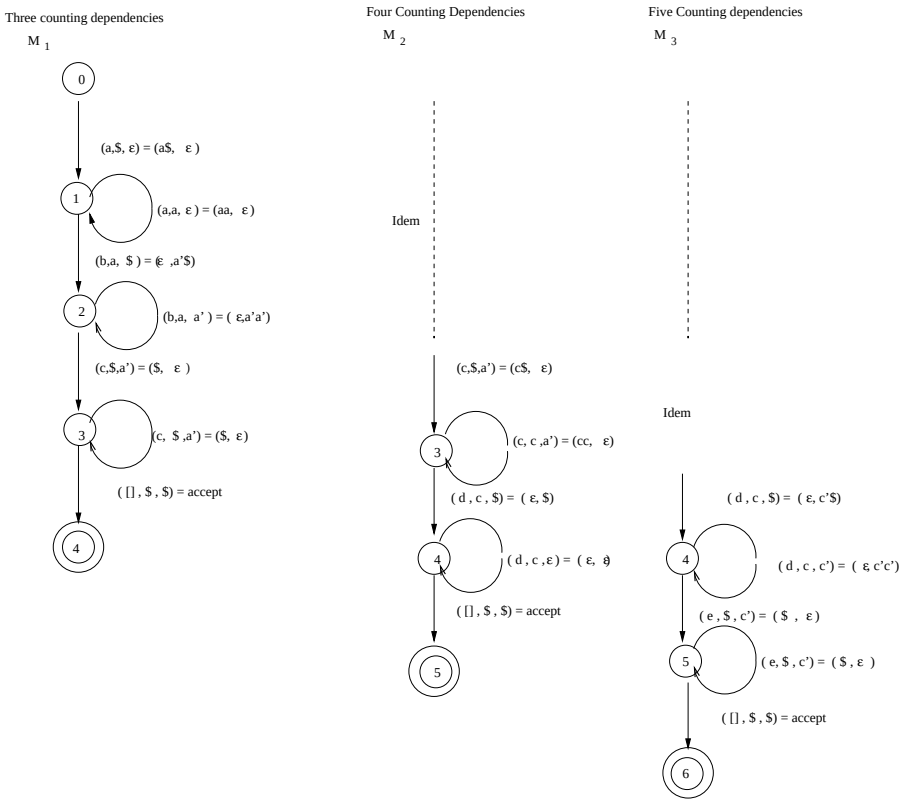


Figure 2.6: Multiple agreement in an sLR-2PDA

1. $\delta(q_0, a, \perp, \perp) = \{(q_1, x\perp, \perp)\}$
2. $\delta(q_1, a, x, \perp) = \{(q_1, xx, \perp)\}$
3. $\delta(q_1, b, x, \perp) = \{(q_1, \epsilon, x\perp)\}$
4. $\delta(q_1, b, x, x) = \{(q_1, \epsilon, xx\})$
5. $\delta(q_1, c, \perp, x) = \{(q_2, \perp, \epsilon), (q_1, x\perp, \epsilon)\}$
6. $\delta(q_1, c, x, x) = \{(q_1, xx, \epsilon)\}$
7. $\delta(q_1, d, x, \perp) = \{(q_1, \epsilon, x\perp), (q_2, \epsilon, \perp)\}$
8. $\delta(q_1, d, x, x) = \{(q_1, \epsilon, xx\}$
9. $\delta(q_1, e, \perp, x) = \{(q_2, \perp, \epsilon)\}$
10. $\delta(q_2, c, \perp, x) = \{(q_2, \perp, \epsilon)\}$
11. $\delta(q_2, d, x, \perp) = \{(q_2, \epsilon, \perp)\}$
12. $\delta(q_2, e, \perp, x) = \{(q_2, \perp, \epsilon)\}$

$$L(M_5) = \{a^n b^n c^n \mid n \geq 1\} \cup \{a^n b^n c^n d^n \mid n \geq 1\} \cup \{a^n b^n c^n d^n e^n \mid n \geq 1\}$$

It can be easily seen that the following is a proper generalization:

Claim 1 *For any alphabet $\Sigma = \{a_1, ..., a_k\}$, where $k > 3$, there is an sLR-2PDA that recognizes the language $L(M_k) = \{a_1^n a_2^n ... a_k^n \mid n \geq 1\}$*

Proof Construct the sLR-2PDA M_k as follows:

$M_k = (\{q_0, q_1, q_2\}, \Sigma, \Sigma, \delta, q_0, \perp, q_2)$ where δ is as follows, such that $o, e \geq 3 < k$ and o is odd and e is even:

1. $\delta(q_0, a_1, \perp, \perp) = \{(q_1, a_1\perp, \perp)\}$
2. $\delta(q_1, a_1, a_1, \perp) = \{(q_1, a_1a_1, \perp)\}$
3. $\delta(q_1, a_2, a_1, \perp) = \{(q_1, \epsilon, a_2\perp)\}$
4. $\delta(q_1, a_2, a_1, a_2) = \{(q_1, \epsilon, a_2a_2\})$
5. $\delta(q_1, a_o, \perp, a_{o-1}) = \{(q_1, a_o\perp, \epsilon)\}$
6. $\delta(q_1, a_o, a_o, a_{o-1}) = \{(q_1, a_oa_o, \epsilon)\}$
7. $\delta(q_1, a_e, a_{e-1}, \perp) = \{(q_1, \epsilon, a_e\perp)\}$
8. $\delta(q_1, a_e, a_{e-1}, a_e) = \{(q_1, \epsilon, a_ea_e\}$ and
9. $\delta(q_1, a_k, \perp, a_{k-1}) = \{(q_2, \perp, \epsilon)\}$ if k is odd or
10. $\delta(q_1, a_k, a_{k-1}, \perp) = \{(q_2, \epsilon, \perp)\}$ if k is even.

□

In the next example, transitions 1 to 4 recognize the first occurrence of w, transitions 5 to 8, reverse the order of the stored symbols, transitions 9 to 12 recognize either the following copy or the last copy. If an intermediate copy is recognized transitions 13 and 14 allow going back to state 2 and reverse the order of the stored symbols.

Example 2 (The language of multiple copies) $M_{wwn} = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{x, y\}, \delta, q_0, \perp, q_3)$ *where* δ *(s.t.* u *is any symbol in the vocabulary):*

1. $\delta(q_0, a, \perp, \perp) = \{(q_1, \perp, x\perp)\}$
2. $\delta(q_0, b, \perp, \perp) = \{(q_1, \perp, y\perp)\}$
3. $\delta(q_1, a, \perp, u) = \{(q_1, \perp, xu)\}$
4. $\delta(q_1, b, \perp, u) = \{(q_1, \perp, yu)\}$
5. $\delta(q_1, \epsilon, \perp, x) = \{(q_2, x\perp, \epsilon)\}$
6. $\delta(q_1, \epsilon, \perp, y) = \{(q_2, y\perp, \epsilon)\}$
7. $\delta(q_2, \epsilon, w, x) = \{(q_2, xw, \epsilon)\}$
8. $\delta(q_2, \epsilon, w, y) = \{(q_2, yw, \epsilon)\}$
9. $\delta(q_2, a, x, \perp) = \{(q_3, \epsilon, x\perp), (q_3, \epsilon, \perp)\}$
10. $\delta(q_2, b, y, \perp) = \{(q_3, \epsilon, y\perp), (q_3, \epsilon, \perp)\}$
11. $\delta(q_3, a, x, u) = \{(q_3, \epsilon, xu), (q_3, \epsilon, \perp)\}$
12. $\delta(q_3, b, y, u) = \{(q_3, \epsilon, yu), (q_3, \epsilon, \perp)\}$
13. $\delta(q_3, \epsilon, \perp, x) = \{(q_2, x\perp, \epsilon)\}$
14. $\delta(q_3, \epsilon, \perp, y) = \{(q_2, y\perp, \epsilon)\}$

$L(M_4) = \{ww^+ \mid w \in \{a, b\}^+\}$

Example 3 (Crossed dependencies) $M_{cr3} = (\{q_0, q_1, q_2, q_3\}, \{a, b, c, d, e, f\}, \{x, y, z\}, \delta, q_0, \perp, q_3)$ *where* δ*:*

1. $\delta(q_0, a, \perp, \perp) = \{(q_1, \perp, x\perp)\}$
2. $\delta(q_1, a, \perp, x) = \{(q_1, \perp, xx)\}$
3. $\delta(q_1, b, \perp, x) = \{(q_1, \perp, yx)\}$
4. $\delta(q_1, b, \perp, y) = \{(q_1, \perp, yy)\}$
5. $\delta(q_1, c, \perp, y) = \{(q_1, \perp, zy)\}$
6. $\delta(q_1, c, \perp, z) = \{(q_1, \perp, zz)\}$
7. $\delta(q_1, \epsilon, \perp, z) = \{(q_2, z\perp, \epsilon)\}$
8. $\delta(q_2, \epsilon, z, z) = \{(q_1, zz, \epsilon)\}$
9. $\delta(q_2, \epsilon, z, y) = \{(q_1, yz, \epsilon)\}$
10. $\delta(q_2, \epsilon, y, y) = \{(q_1, yy, \epsilon)\}$
11. $\delta(q_2, \epsilon, y, x) = \{(q_1, xy, \epsilon)\}$
12. $\delta(q_2, d, x, \perp) = \{(q_3, \epsilon, \perp)\}$
13. $\delta(q_3, d, x, \perp) = \{(q_3, \epsilon, \perp)\}$
14. $\delta(q_3, e, y, \perp) = \{(q_3, \epsilon, \perp)\}$
15. $\delta(q_3, f, z, \perp) = \{(q_3, \epsilon, \perp)\}$

$L(M_{cr3} = \{a^n b^m c^l d^n e^m f^l \mid n, m, l \geq 1\}$

2.5 Determinism in LR-2PDA

An LR parser for a CFL is essentially a compiler that converts an LR CFG into a DPDA automata [41, 2]. Therefore understanding the deterministic LR-2PDA is crucial for obtaining a deterministic LR parser for GILs defined in chapter 6. Deterministic GILs are recognized in linear time as we mentioned in the introduction. The corresponding definition of a deterministic LR-2PDA is as follows:

2.5.1 Deterministic LR-2PDA

Definition 4 *A LR-2PDA is deterministic iff:*

1. *for each* $q \in Q$ *and* $Z, I \in \Gamma$*, whenever* $\delta(q, \epsilon, Z, I)$ *is nonempty, then* $\delta(q, a, Z, I)$ *is empty for all* $a \in \Sigma$*.*
2. *for no* $q \in Q$*,* $Z, I \in \Gamma$*, and* $a \in \Sigma \cup \{\epsilon\}$ *does* $\delta(q, a, Z, I)$ *contain more than one element.*
3. *for each* $q \in Q$*,* $A \in \Sigma$ *and* $Z, I \in \Gamma$*, whenever* $\delta(q, A, Z, \epsilon)$ *is nonempty, then* $\delta(q, A, Z, I)$ *is empty for all* $I \in \Gamma$*.*

Condition 1 prevents the possibility of a choice between a move independent of the input symbol (ϵ-move) and one involving an input symbol.

Condition 2 prevents the choice of move for any equal 4-tuple.

Condition 3 prevents the possibility of a choice between a move independent of the second stack symbol (ϵ-move) and one involving a symbol from the stack.

We can regard ϵ to be an abbreviation for all the possible symbols in the top of the stack. Therefore, in the following examples of transitions, the first case is not compatible with the third, nor with the fourth, because they imply a non-deterministic choice.

1. $(q, a, \epsilon, \epsilon) = (q, \epsilon, i)$ 2. $(q, b, \epsilon, \epsilon) = (q, \epsilon, j)$
3. $(q, a, \epsilon, i) = (q, \epsilon, \epsilon)$ 4. $(q, a, \epsilon, j) = (q, \epsilon, j)$

2.5.2 LR-2PDA with deterministic auxiliary stack

Even if a GIG grammar is non-deterministic, it might be an advantage to be able to determine whether the non-determinism lies in the CFG back-bone only and the indexing mechanism does not introduce non-determinism. If the non-determinism is in the CFG backbone only we can guarantee that the recognition problem is in $O(n^3)$. The construction of the LR(k) parsing tables would show whether the indexing mechanism for a given GIG is non-deterministic.

Definition 5 *A LR-2PDA has a deterministic indexing, or is a deterministic indexing LR-2PDA if and only if:*

1. *for each $q \in Q$ and $Z, I \in \Gamma$, whenever $\delta(q, \epsilon, Z, I)$ is nonempty, then $\delta(q, a, Z, I)$ is empty for all $a \in \Sigma$.*

2. *for no $q \in Q$, $Z, I \in \Gamma$, and $a \in \Sigma \cup \{\epsilon\}$ does $\delta(q, a, Z, I)$ contain two elements $= (q_i, Z_i, I_i)$ and (q_j, Z_j, I_j) such that $I_i \neq I_j$.*

3. *for each $q \in Q$, $A \in \Sigma$ and $Z, I \in \Gamma$, whenever $\delta(q, A, Z, \epsilon)$ is nonempty, then $\delta(q, A, Z, I)$ is empty for all $I \in \Gamma$.*

Chapter 3

Global Index Grammars

3.1 Indexed Grammars and Linear Indexed Grammars

Indexed grammars, (IGs) [1], and Linear Index Grammars, (LIGs;LILs) [29], have the capability to associate stacks of indices with symbols in the grammar rules. IGs are not semi-linear (the class of ILs contains the language $\{a^{2^n} \mid n \geq 0\}$). LIGs are Indexed Grammars with an additional constraint on the form of the productions: the stack of indices can be "transmitted" only to one non-terminal. As a consequence they are semi-linear and belong to the class of MCSGs. The class of LILs contains L_4 but not L_5 (see Chapter 1).

An **Indexed Grammar** is a 5-tuple (N, T, I, P, S), where N is the set of non-terminal symbols, T the set of terminals, I the set of *indices*, S in V is the start symbol, and P is a finite set of productions of the form:

a. $A \rightarrow \alpha$ b. $A[..] \rightarrow B[i..]$ c. $A[i..] \rightarrow [..]\alpha$

where A and B are in V, i is in I , [..] represents a stack of indices, a string in I^*, and α is in $(V \cup T)^*$.

The relation *derives*, $\Rightarrow$, on *sentential forms*, i.e., strings in $(VI^* \cup T)^*$, is defined as follows. Let β and γ be in $(VI^* \cup T)^*$, δ be in I^*, and X_i in $V \cup T$.

1. If $A \rightarrow X_1 ... X_k$ is a production of type a. then

   $\beta A \delta \gamma \Rightarrow \beta X_1 \delta_1 ... X_k \delta_k \gamma$

   where $\delta_i = \delta$ if X_i is in V and $\delta_i = \epsilon$ if X_i is in T.

2. If $A[..] \rightarrow B[x..]$ is a production of type b. then

   $\beta A \delta \gamma \Rightarrow \beta B x \delta \gamma$

3. If $[x..]A \rightarrow [..]\ X_1 ... X_k$ is a is a production of type c. then

   $\beta A x \delta \gamma \Rightarrow \beta X_1 \delta_1 ... X_k \delta_k \gamma$where $\delta_i = \delta$ if X_i is in V and $\delta_i = \epsilon$ if X_i is in T.

Example 4 $L(G_1) = \{a^{2^n}\ n \geq 0\}, G_1 = (\{S, A\}, \{a\}, \{i\}, P, S)$ *where P:*

$S[..] \rightarrow S[i..]$, $S[..] \rightarrow A[..]\ A[..]$ $A[i..] \rightarrow A[..]\ A[..]$ $A[] \rightarrow a$

A **Linear Indexed Grammar** is an Indexed Grammar where the set of productions have the following form, where A and B are non terminals and α and γ are (possible empty) sequences of terminals and non terminals:

a. $A[..] \rightarrow \alpha\ B[..]\ \gamma$

b. $A[i..] \rightarrow \alpha\ B[..]\ \gamma$

c. $A[..] \rightarrow \alpha B[i..]\ \gamma$

Example 5 $L(G_1) = \{a^n b^n c^n d^n\ | n \geq 0\}, G_1 = (\{S, B\}, \{a, b, c, d\}, \{i\}, P, S)$, *where P is:*

$S[..] \rightarrow aS[i..]d$, $S[..] \rightarrow B[..]$, $B[i..] \rightarrow bB[..]c$, $B[\,] \rightarrow \epsilon$

3.2 Global Index Grammars

In the IG or LIG case, the stack of indices is associated with variables. It is a grammar that *controls* the derivation through the variables of a CFG. The proposal we present here uses the stack of indices as a unique designated global control structure. In this sense, these grammars provide a global but restricted context that can be updated at any local point in the derivation. GIGs are a kind of *regulated rewriting* mechanism [25] with global context and history of the derivation (or ordered derivation) as the main characteristics of its regulating device. The introduction of indices in the derivation is restricted to rules that have terminals in the right-hand side. This feature makes the use

of indices dependent on *lexical* information, in a linguistic sense, and allows the kind of recognition algorithm we propose in Chapter 5. An additional constraint imposed on GIGs is strict leftmost derivation whenever indices are introduced or removed by the derivation.

Definition 6 *A GIG is a 6-tuple* $G = (N, T, I, S, \#, P)$ *where* N, T, I *are finite pairwise disjoint sets and 1)* N *are non-terminals 2)* T *are terminals 3)* I *a set of stack indices 4)* $S \in N$ *is the start symbol 5)* $\#$ *is the start stack symbol (not in I,N,T) and 6)* P *is a finite set of productions, having the following form,*[1] *where* $x \in I$, $y \in \{I \cup \#\}$, $A \in N$, $\alpha, \beta \in (N \cup T)^*$ *and* $a \in T$.

a.1 $A \underset{\epsilon}{\rightarrow} \alpha$	*or*	$A \rightarrow \alpha$	*(epsilon rules or context free (CF) rules)*
a.2 $A \underset{[y]}{\rightarrow} \alpha$	*or*	$[..]A \rightarrow [..]\alpha$	*(epsilon rules or CF with constraints)*
b. $A \underset{x}{\rightarrow} a\ \beta$	*or*	$[..]A \rightarrow [x..]a\ \beta$	*(push rules or opening parenthesis)*
c. $A \underset{\bar{x}}{\rightarrow} \alpha$	*or*	$[x..]A \rightarrow [..]\alpha$	*(pop rules or closing parenthesis)*

Note the difference between *push* (type b) and *pop* rules (type c): *push* rules require the right-hand side of the rule to contain a terminal in the first position. *Pop* rules do not require a terminal at all. The constraint on *push* rules is a crucial property of GIGs, without it GIGs would be equivalent to a Turing Machine. In the next subsection we present an even more restricted type of GIG that requires both *push* and *pop* rules in Greibach Normal Form.

Derivations in a GIG are similar to those in a CFG except that it is possible to modify a string of indices. This string of indices is not associated with variables, so we can consider them *global*. We define the *derives* relation $\Rightarrow$ on *sentential forms*, which are strings in $I^*\#(N \cup T)^*$ as follows. Let β and γ be in $(N \cup T)^*$, δ be in I^*, x in I, w be in T^* and X_i in $(N \cup T)$.

1. If $A \underset{\mu}{\rightarrow} X_1 ... X_k$ is a production of type a. (i.e. $\mu = \epsilon$ or $\mu = [x]$, $x \in I$) then:

   $\delta\#\beta A\gamma \Rightarrow \delta\#\beta X_1 ... X_k\gamma$

   Using a different notation:

   $\delta\#\beta A\gamma \underset{\epsilon}{\Rightarrow} \delta\#\beta X_1 ... X_k\gamma$ (production type a.1 or context free)

   $x\delta\#\beta A\gamma \underset{[x]}{\Rightarrow} x\delta\#\beta X_1 ... X_k\gamma)$ (production type a.2)

[1] The notation at the left makes explicit that operation on the stack is associated to the production and not to terminals or non-terminals. It also makes explicit that the operations are associated to the computation of a Dyck language (using such notation as used in e.g. [39]). The notation at the right is intended to be more similar to the notation used in IGs and LIGs.

This is equivalent to a CFG *derives* relation in the sense that it does not affect the stack of indices (*push* and *pop* rules).

2. If $A \underset{\mu}{\rightarrow} aX_1...X_n$ is a production of type (b.) or *push*: $\mu = x, x \in I$, then:
   $\delta\#wA\gamma \Rightarrow x\delta\#waX_k...X_n\gamma$ (or $\delta\#wA\gamma \underset{x}{\Rightarrow} x\delta\#waX_1...X_n\gamma$)

3. If $A \underset{\mu}{\rightarrow} X_1...X_n$ is a production of type (c.) or *pop* : $\mu = \bar{x}, x \in I$, then:
   $x\delta\#wA\gamma \Rightarrow \delta\#wX_1.....X_n\gamma$ (or $x\delta\#wA\gamma \underset{\bar{x}}{\Rightarrow} \delta\#wX_1...X_n\gamma$)

The reflexive and transitive closure of $\Rightarrow$ is denoted as usual by $\overset{*}{\Rightarrow}$. We define the language of a GIG G, $L(G)$ to be: $\{w|\#S \overset{*}{\Rightarrow} \#w$ and w is in $T^*\}$

It can be observed that the main difference between, IGs, LIGs and GIGs, is the interpretation of the *derives* relation relative to the behavior of the stack of indices. In IGs the stacks of indices are distributed over the non-terminals on the right-hand side of the rule. In LIGs indices are associated with only one non-terminal at the right-hand side of the rule. This produces the effect that there is only one stack affected at each derivation step, with the consequence that LILs are semilinear. GIGs share this *uniqueness* of the stack with LIGs, but to an extreme: there is only one stack to be considered. Unlike LIGs and IGs, the stack of indices in a GIG is independent of non-terminals in the GIG case. GIGs can have rules where the right-hand side of the rule is composed only of terminals and affects the stack of indices. Indeed *push* rules (type b) are constrained to start the right-hand side with a terminal as specified in GIG's definition (6.b) (in other words *push* productions must be in Greibach Normal Form).

The *derives* definition requires a *leftmost* derivation for those rules (*push* and *pop* rules) that affect the stack of indices.

3.2.1 trGIGS

We now define trGIGs as GIGs where the *pop* rules (type c.) are constrained to start with a terminal in a similar way as *push* (type b.) rules in GIGs. We also introduce some examples of trGIGs modeling multiple agreements and crossed dependencies.

Definition 7 *A trGIG is a GIG where the set of production of types (c) (*pop*) rules, are as follows, where $x \in I$, $A \in N$, $\alpha, \beta \in (N \cup T)^*$, $x \in I$, and $a \in T$:*

c. $A \underset{\bar{x}}{\rightarrow} a\ \beta$ *or* $[x..]A \rightarrow [..]\ a\ \beta$

Example 6 (multiple agreements) $L(G_5) = \{a^n b^n c^n d^n e^n \mid n \geq 1\}$,
$G_5 = (\{S, A, C, D^1, D, E\}, \{a, b, c, d, e\}, \{a', g'\}, S, \#, P)$ *and* P *is:*

$S \rightarrow ACE \quad A \underset{i}{\rightarrow} aAb \quad A \underset{i}{\rightarrow} ab \quad C \underset{\bar{i}}{\rightarrow} cD^1 \quad D^1 \rightarrow CD$

$C \underset{\bar{i}}{\rightarrow} cD \quad D \underset{j}{\rightarrow} d \quad E \underset{\bar{j}}{\rightarrow} eE \quad E \underset{\bar{j}}{\rightarrow} e$

The derivation of $w = aabbccddee$*:*

$\#S \Rightarrow \#ACE \Rightarrow i\#aAbCE \Rightarrow ii\#aabbCE \Rightarrow i\#aabbcD^1E \Rightarrow i\#aabbcCDE \Rightarrow \#aabbccDDE \Rightarrow$
$j\#aabbccdDE \Rightarrow jj\#aabbccddE \Rightarrow j\#aabbccddeE \Rightarrow \#aabbccddee$

Example 7 (crossing dependencies) $L(G_{cr}) = \{a^n b^m c^n d^m \mid n, m \geq 1\}$, *where* $G_cr = (\{S, A, B, C\}, \{a, b, c, d\}, \{x$
and P *is:*

1. $S \rightarrow A\ D$ 2. $A \rightarrow a\ A\ c \mid a\ B\ c$ 3. $B \underset{x}{\rightarrow} b\ B \mid b$ 4. $D \underset{\bar{x}}{\rightarrow} d\ D \mid d$

Now we generalize example 6 and show how to construct a trGIG that recognizes any finite number of dependencies (similar to the LR-2PDA automaton presented in the previous chapter):

Claim 2 *The language* $L_m = \{a_1^n a_2^n ... a_k^n \mid n \geq 1, k \geq 4\}$ *is in trGIL*

Proof Construct the GIG grammar
$G_k = (\{S, E_2, O_3, ..., A_k\}, \{a_1, ..., a_k\}, \{a'_2, a'_4, ..., a'_j\}, S, \#, P)$ such that $k \geq 4$ and $j = k$ if k is even or $j = k - 1$ if k is odd, and P is composed by:

1. $S \rightarrow a_1\ S\ E_2$ 2. S$\rightarrow a_1\ E_2$

and for every E_i and O_i such that i is odd in O$_i$ and i is even in E_i add the following rules:

3a.$E_i \underset{a_i}{\rightarrow} a_i\ E_i$ 4a. $E_i \underset{a_i}{\rightarrow} a_i\ O_{i+1}$ 5a. $O_i \underset{\bar{a}_{i-1}}{\rightarrow} a_i\ O_i\ E_{i+1}$

6a. $O_i \underset{\bar{a}_{i-1}}{\rightarrow} a_i\ E_{i+1}$

If k is even add: 3b.$A_k \rightarrow a_k\ A_k$ 4b. $A_k \rightarrow a_k$

If k is odd add: 5b. $A_k \underset{\bar{a}_{k-1}}{\rightarrow} a_k\ A_k$ 6b. $O_k \underset{\bar{a}_{k-1}}{\rightarrow} a_k$

$L(G_n) = \{a_1^n a_2^n ... a_k^n \mid n \geq 1, 4 \leq k\}$

We can generalize the same mechanism even further as follows:

Example 8 (Multiple dependencies) $L(G_{gdp}) = \{\ a^n (b^n c^n)^+ \mid n \geq 1\}$,
$G_{gdp} = (\{S, A, R, E, O, L\}, \{a, b, c\}, \{i\}, S, \#, P)$ *and* P *is:*

$S \rightarrow AR \quad A \rightarrow aAE \quad A \rightarrow a \quad E \underset{i}{\rightarrow} b \quad R \underset{i}{\rightarrow} b\ L$

$L \rightarrow OR \mid C \quad C \underset{\bar{i}}{\rightarrow} c\ C \mid c \quad O \underset{\bar{i}}{\rightarrow} c\ OE \mid c$

The derivation of the string aabbccbbcc shows five dependencies.

$\#S \Rightarrow \#AR \Rightarrow \#aAER \Rightarrow \#aaER \Rightarrow i\#aabR \Rightarrow ii\#aabbL \Rightarrow ii\#aabbOR \Rightarrow$
$i\#aabbcOER \Rightarrow \#aabbccER \Rightarrow i\#aabbccbRii\#aabbccbbL \Rightarrow ii\#aabbccbbC \Rightarrow$
$i\#aabbccbbcC \Rightarrow \#aabbccbbcc$

3.2.2 GIGs examples

We conjecture that the following languages can be defined with a GIG and cannot be defined using a trGIG. Above, we mentioned the difference between *push* and *pop* rules in GIGs. This difference enables the use of left recursive *pop* rules so that the order of the derivation is a mirror image of the left-right order of the input string. This is not possible in trGIGs because *pop* rules cannot be left recursive.

Example 9 (Copy Language) $L(G_{ww}) = \{ww \mid w \in \{a,b\}^*\}$

$G_{ww} = (\{S,R\},\{a,b\},\{i,j\},S,\#,P)$ and $P =$

1. $S \underset{i}{\rightarrow} aS$ 2. $S \underset{j}{\rightarrow} bS$ 3. $S \rightarrow R$ 4. $R \underset{\bar{i}}{\rightarrow} Ra \mid a$ 5. $R \underset{\bar{j}}{\rightarrow} Rb \mid b$

The derivation of the string *abbabb*:

$\#S \Rightarrow i\#aS \Rightarrow ji\#abS \Rightarrow jji\#abbS \Rightarrow ji\#abbRb \Rightarrow i\#abbRbb \Rightarrow \#abbRabb \Rightarrow \#abbabb$

Example 10 (Multiple Copies) $L(G_{wwn}) = \{ww^+ \mid w \in \{a,b\}^*\}$

$G_{wwn} = (\{S,R,A,B,C\},\{a,b\},\{i,j\},S,\#,P)$ and $P =$

$S \rightarrow AS \mid BS \mid C \quad C \rightarrow RC \mid L \quad R \underset{\bar{i}}{\rightarrow} RA \quad R \underset{\bar{j}}{\rightarrow} RB \quad R \underset{[\#]}{\rightarrow} \epsilon \quad A \underset{i}{\rightarrow} a$

$B \underset{j}{\rightarrow} b \quad L \underset{\bar{i}}{\rightarrow} La \mid a \quad L \underset{\bar{j}}{\rightarrow} Lb \mid b$

The derivation of *ababab*:

$\#S \Rightarrow \#AS \Rightarrow i\#aS \Rightarrow i\#aBS \Rightarrow ji\#abS \Rightarrow ji\#abC \Rightarrow ji\#abRC \Rightarrow i\#abRBC \Rightarrow \#abRABC \Rightarrow$
$\#abABC \Rightarrow i\#abaBC \Rightarrow ji\#ababS \Rightarrow ji\#ababL \Rightarrow i\#ababLb \Rightarrow \#ababab$

In the next example, we can see that by using left recursive *pop* productions we can generate a language with a higher number of crossing dependencies than we can by using a trGIG. It is easy to see that by generalizing the same mechanism any finite number of crossing dependencies can be generated.

Example 11 (Crossing Dependencies) $L(G_{cr3}) = \{a^n b^m c^l d^n e^m f^l\}$

$G_{cr3} = (\{S,F,L,B,C,D,E\},\{a,b,c,d,e,f\},\{i,j,k\},S,\#,P)$ and $P =$

1. $S \rightarrow FL$ 2. $F \underset{i}{\rightarrow} aF \mid aB$ 3. $B \underset{j}{\rightarrow} bB \mid bC$ 4. $C \underset{k}{\rightarrow} cC \mid c$
5. $L \underset{\bar{k}}{\rightarrow} Lf \mid Ef$ 6. $E \underset{\bar{j}}{\rightarrow} Ee \mid De$ 7. $D \underset{\bar{i}}{\rightarrow} Dd \mid d$

This can be even further generalized, obtaining sets of crossing dependencies, as exemplified by $L_{mcr} = \{a^n b^m c^l d^n e^m f^l g^n h^m i^l\}$ or $\{a^n b^m c^l (d^n e^m f^l)^+\}$

The next example shows the MIX (or Bach) language (the language with equal number of a's b's and c's in any order, see e.g. [11]). It was conjectured in [29] that the MIX language is not an IL. We show that GILs are semilinear, therefore ILs and GILs could be incomparable under set inclusion. This language though conceptually simple, turns out to be a very challenging problem. It allows an incredible number of possible strings of any given length in the language, and at the same time it requires that three dependencies be observed. The number of permutations for a string of length n is $n!$. There are also factorial duplicates for the number of times a character occurs in the string. In the mix language each character occurs in one third of the string and there are three characters. The number of strings of length $3n$ is $(3n)!/(n!)^3$.

Example 12 (MIX language) $G_{mix} = (\{S\}, \{a, b, c\}, \{i, j, k\}, S, \#, P)$ and P is:

$S \underset{i}{\rightarrow} cS \quad S \underset{j}{\rightarrow} bS \quad S \underset{k}{\rightarrow} aS$
$S \underset{\bar{i}}{\rightarrow} SaSbS \mid SbSaS$
$S \underset{\bar{j}}{\rightarrow} SaScS \mid ScSaS$
$S \underset{\bar{k}}{\rightarrow} SbScS \mid ScSbS$
$S \rightarrow \epsilon$
$L(G_{mix}) = L_{mix} = \{w | w \in \{a, b, c\}^* \; and \; |a|_w = |b|_w = |c|_w \geq 1\}$

Example of a derivation for the string *abbcca*.

$S \Rightarrow k\#aS \Rightarrow \#aSbScS \Rightarrow \#abScS \Rightarrow j\#abbcS \Rightarrow \#abbcScSaS \Rightarrow \#abbccSaS \Rightarrow \#abbccaS \Rightarrow \#abbcca$

We present here the proof that $L(G_{mix}) = L_{mix}$. This is important because it is very difficult to build a proof for the claim that the MIX language belongs (or does not belong to) to a family of languages [2]. The difficulty is due to the number of strings in the language at the basic combination level (there are 90 strings of length 6, and 1680 strings of length 9), and the need to ensure that the three dependencies (between a's, b's and c's) are observed. It is easy to design GIG grammars where the respective languages are in L_{mix}, however it is not so easy to prove or assess whether the

[2] A. Joshi p.c.

language is indeed L_{mix}. It might just turn out to be a significant subset of L_{mix} (in other words it is not so easy to figure out if there are some set of strings in L_{mix} that might not be generated by a particular grammar).

Proof .

First direction: $L(G_{mix})$ is in $L_{mix} = \{w | w \in \{a,b,c\}^* \; and \; |a|_w = |b|_w = |c|_w \geq 0\}$.

It can be seen that all the derivations allow only equal number of $a's$, $b's$ and $c's$ due to the constraint imposed by the indexing mechanism.

Second direction: L_{mix} is in $L(G_{mix})$

Basis $n = 0$, ϵ is clearly in the language.

$n = 3$:

$$S \underset{k}{\Rightarrow} k\#a \underset{\bar{k}}{\Rightarrow} \#abc$$
$$S \underset{k}{\Rightarrow} k\#a \underset{\bar{k}}{\Rightarrow} \#acb$$
$$S \underset{j}{\Rightarrow} j\#b \underset{\bar{j}}{\Rightarrow} \#bac$$
$$S \underset{j}{\Rightarrow} j\#b \underset{\bar{j}}{\Rightarrow} \#bca$$
$$S \underset{i}{\Rightarrow} i\#b \underset{\bar{i}}{\Rightarrow} \#cba$$
$$S \underset{i}{\Rightarrow} i\#b \underset{\bar{i}}{\Rightarrow} \#cab$$

Inductive Hypothesis: Assume for any string $uwyz$ in L_{mix} then $uwyz$ is in $L(G_{MIX})$, where $|uwyz| = n, n \geq 0$. We need to prove that $ut_1 w\bar{t}_2 y\bar{t}_3 z$ is in $L(G_{MIX})$ ($|ut_i w\bar{t}_i y\bar{t}_i z| = n+3$). Where $t_1\bar{t}_2\bar{t}_3$ is any string of length three in L_{mix} and t designates the terminals that introduce the indices in the derivation, and $\bar{t}$ are members of the pairs that remove indices in the derivation. We show that G_{mix} generates any $ut_i w\bar{t}_i y\bar{t}_i z$

It is trivial to see that the following derivation is obtained (where a corresponds to t_k and b and c to the $\bar{t}_k$'s.

1. $S \overset{*}{\Rightarrow} \delta\#uS \Rightarrow k\delta\#uaS \Rightarrow \delta\#uaSbScS \overset{*}{\Rightarrow} \delta'\#uawbScS$

But at this point the strings b and c might block a possible derivation. This *blocking* effect can be illustrated by the following hypothetical situation: suppose $uwyz$ might be composed by $u = u'b_1$, $y = a_1 y' c_1 z$, $w = w' c_2$ and $y' = a_2 y'' b_2$ trigger the following derivation (perhaps the only one).

2. $S \overset{*}{\Rightarrow} j\delta'\#u'bS \Rightarrow \delta\#u'bSaScS \overset{*}{\Rightarrow} i\delta''\#u'bw'caScS \Rightarrow \delta''\#u'bw'caay''bcS$

If we try to combine the derivation proposed in 1. with the one proposed in 2. we reach the following point:

3. $S \stackrel{*}{\Rightarrow} j\delta'\#u'b_1S \Rightarrow kj\delta'\#u'b_1a_0S \Rightarrow j\delta'\#u'b_1a_0Sb_0Sc_0S$

At this point the next step should be the derivation of the non-terminal S at b_0Sc_0, removing the index j. This cannot be done because the derivation has to proceed with the preceding S (at a_0Sb_0) due to the leftmost derivation constraint for GIGs. Therefore, there is no guarantee that the necessary sequence of indices are removed.

We prove now that such a *blocking* effect in the indexing is not possible: there are always alternative derivations, because the grammar is highly ambiguous.

We need to account whether there is a possible configuration such that a conflict of indices arises. In other words, is there any possible combination of indices :

$t_i \cdots t_j \cdots t_k \cdots \bar{t}_j \cdots \bar{t}_k \cdots \bar{t}_i \cdots \bar{t}_j \cdots \bar{t}_k \cdots \bar{t}_i$

that cannot be derived by G_{MIX}? Remember t designates the terminals that introduce the indices in the derivation, and $\bar{t}$ are members of the pairs that remove indices in the derivation. We show now that this situation is impossible and that therefore L_{mix} is in $L(G_{MIX})$.

Claim 3 *For any sequence of possibly conflicting indices the conflicting indices can be reduced to two conflicting indices in a derivation of the grammar G_{mix}.*

Case 1. there are three different indices ($t_i \neq t_j \neq t_k$):

$t_iut_jvt_k...$ clearly $S \Rightarrow i\#t_iS \Rightarrow \#t_iSt_jSt_kS$ and the complements of t_i, t_k, t_j are reduced to the case of two conflicting indices.

Case 2. there is more than one element in one of the indices :

$t_it_j(t_i)^n[\bar{t}_j\bar{t}_j(\bar{t}_i\bar{t}_i)^{n+1}]$ for example: $ab(a)^n[cacb](cb)^n$ $ab(a)^n[accb](cb)^n$

$S \Rightarrow t_iS[t_it_i]$ reduces to the following problem which is derived by the embedded S

$t_j(t_i)^n[\bar{t}_j..\bar{t}_j...(\bar{t}_i...\bar{t}_i)^n]$

And this problem is reduced to:

$t_jt_iS\bar{t}_j\bar{t}_j\bar{t}_i\bar{t}_iS$ i.e.: $baScacbS$ where the two non terminal S expand the ${t_i}^{n-1}$ triples.

This in turn can be easily derived by:

$\delta\#S \Rightarrow j\delta\#bS \Rightarrow \delta\#baScS \Rightarrow i^{n-1}\delta\#baa^{n-1}cS \Rightarrow ki^{n-1}\delta\#baa^{n-1}caS \Rightarrow$

$i^{n-1}\delta\#baa^{n-1}cacbS \Rightarrow \delta\#baa^{n-1}cacb(cb)^{n-1}$

Claim 4 *For any sequence of two possible conflicting indices there is always at least a derivation in G_{mix}.*

Assume the possible empty strings u, w, x, z and y do not introduce possible blocking indices (i.e. if indices are introduced, they are also removed in the derivation of the substring). This assumption is made based on the previous two claims, given we are considering just two possible conflicting indices.

Formally:

$S \stackrel{*}{\Rightarrow} \#uS \stackrel{*}{\Rightarrow} \#uwS \stackrel{*}{\Rightarrow} \#uwxS \stackrel{*}{\Rightarrow} \#uwxzS \stackrel{*}{\Rightarrow} \#uwxzy$

Therefore for the following combination:

$t_i u t_j w \bar{t}_i x \bar{t}_i z \bar{t}_j y \bar{t}_j$:

the following derivation is possible.

$S \stackrel{*}{\Rightarrow} i\#ut_iS \Rightarrow \#ut_iS\bar{t}_iS\bar{t}_iS \stackrel{*}{\Rightarrow} j\#ut_ivt_j\bar{t}_iS\bar{t}_iS \stackrel{*}{\Rightarrow}$

$j\#ut_ivt_j\bar{t}_ix\bar{t}_iS \Rightarrow \#ut_ivt_j\bar{t}_ix\bar{t}_iS\bar{t}_jS\bar{t}_j \stackrel{*}{\Rightarrow} \#ut_ivt_j\bar{t}_ix\bar{t}_iz\bar{t}_jy\bar{t}_j$

Similar alternatives are possible for the remaining combinations:

$ut_iv\bar{t}_iwt_jx\bar{t}_iz\bar{t}_jy\bar{t}_j$ $\quad$ $ut_iv\bar{t}_iwt_jx\bar{t}_jz\bar{t}_iy\bar{t}_j$

$ut_ivt_jw\bar{t}_jx\bar{t}_iz\bar{t}_jy\bar{t}_i$ $\quad$ $ut_ivt_jw\bar{t}_jx\bar{t}_iz\bar{t}_iy\bar{t}_j$

$ut_ivt_jw\bar{t}_ix\bar{t}_jz\bar{t}_jy\bar{t}_i$ $\quad$ $ut_ivt_jw\bar{t}_ix\bar{t}_jz\bar{t}_iy\bar{t}_j$

However the last one requires some additional explanation. Let us assume the t_i triple corresponds to $a(bc)$ and t_j triple corresponds to $b(ca)$ therefore it is: $ua_ivb_jwb_ixc_jzc_iya_j$

(or $ua_ivb_jc_ixc_jzb_iya_j$, etc.)

It can be seen that there is an alternative derivation:

$S \stackrel{*}{\Rightarrow} k\#ua_ivS \Rightarrow \#ua_ivb_jSc_jS \stackrel{*}{\Rightarrow} j\#ua_ivb_j\mathbf{wb}_i\mathbf{x}c_jS \Rightarrow$

$\#ua_ivb_jwb_ixc_j\mathbf{Sc}_i\mathbf{Sa}_j \stackrel{*}{\Rightarrow} \#ua_ivb_jwb_ixc_j\mathbf{zc}_i\mathbf{ya}_j$.

Therefore we conclude that there is no possible blocking effect and there is always a derivation:

$S \stackrel{*}{\Rightarrow} \#uawbycz$

□

We also used the *generate-and-test* strategy to verify the correctness of the above grammar G_{mix} for L_{mix} and two other grammars for the same language. The tests were performed on the strings of length 6 and 9.

3.3 GILs and Dyck Languages

We argue in this section that GILs correspond to the result of the "combination" of a CFG with a Dyck language. The well-known Chomsky-Schützenberger theorem [22], shows that any CFG is

the result of the "combination" of a Regular Language with a Dyck Language. The analogy in the automaton model is the combination of a Finite State Automaton with a stack, which results in a PDA. *Push* and *pop* moves of the stack have the power to compute a Dyck language using the stack symbols. This "combination" is formally defined in the CFG case as follows: each context-free language L is of the form, $L = \phi(D_r \cap R)$, where D is a semi-Dyck set, R is a regular set and ϕ is a homomorphism [39].

Dyck languages can also provide a representation of GILs. A GIG is the "combination" of a CFG with a Dyck language: GIGs are CFG-like productions that may have an associated stack of indices. This stack of indices, as we said, has the power to compute the Dyck language of the vocabulary of indices relative to their derivation order. As we have seen above, GILs include languages that are not context free. GILs can be represented using a natural extension of Chomsky-Schützenberger theorem (we will follow the notation used in [39]).

Theorem 1 (Chomsky-Schützenberger for GILs) .

For each GIL L, there is an integer r, a CFL and a homomorphism ϕ such that $L = \phi(D_r \cap CFL)$.

Proof Let $L = L(G)$, where $G = (NT, T, I, S, \#, P)$.
T and I are pairwise disjoint and $|T \cup I| = r$
The alphabet for the semi-Dyck set will be $\Sigma_0 = (T \cup I) \cup (\bar{T} \cup \bar{I})$. Let D_r be the semi-Dyck set over Σ_0.
Define ϕ as the homomorphism from ${\Sigma_0}^*$ into T^* determined by

$\phi(a) = a$, $\phi(\bar{a}) = \epsilon$ if $a \in T$.

$\phi(i) = \epsilon$, $\phi(\bar{i}) = \epsilon$ if $i \in I$.

Define a CFG grammar $G1 = (N, \Sigma_0, S_1, P1)$ such that P1 is given according to the following conditions. For every production $p \in P$ where $a \in T$, $i \in I$, *and* α, $\beta \in (N \cup T)^*$ create a new production $p_1 \in P1$, such that α_1 *and* $\beta_1 \in (N \cup T\bar{T})^*$ as follows: [3]

1. **if** $p = A \rightarrow \alpha a \beta$ **then** $p_1 = A_1 \rightarrow \alpha_1 a \bar{a} \beta_1$

2. **if** $p = A \underset{i}{\rightarrow} \alpha$ **then** $p_1 = A_1 \rightarrow i\alpha_1$

[3] We use a subscript 1 to make clear that either productions and non-terminals with such subscript belong to the CFG $G1$ and not to the source GIG.

3. **if** $p = A \underset{\bar{i}}{\rightarrow} \alpha$ **then** $p_1 = A_1 \rightarrow \bar{i}\alpha_1$

4. **if** $p = A \underset{[i]}{\rightarrow} \alpha$ **then** $p_1 = A_1 \rightarrow \bar{i}i\alpha_1$

L(G1) is a CFL, so we have defined D_r, ϕ and the corresponding CFL. We have to show that $L = \phi(D_r \cap L(G1))$. The proof is an induction on the length of the computation sequence.

First direction $L \subseteq \phi(D_r \cap L(G1))$. Suppose $w \in L$.

Basis

a) $w = \epsilon$ then $\#S \Rightarrow \#\epsilon$ and $S_1 \Rightarrow \epsilon$

b) $|w| = 1$ then $\#S \Rightarrow \#a$ and $S_1 \Rightarrow a\bar{a}$

Both ϵ and $a\bar{a}$ are in $D_r \cap L(G1)$

The induction hypothesis exploits the fact that any computation starting at a given configuration of the index stack and returning to that initial configuration is computing a Dyck language.

Induction Hypothesis. Suppose that if there is a GIL derivation $\delta\#uA \overset{k}{\Rightarrow} \delta\#uyB$ (where $u, y \in T^*$) implies there is a L(G1) derivation $u'A \overset{k}{\Rightarrow} u'zB$ (where $u', z \in (T\bar{T} \cup I \cup \bar{I})^*$) such that $z \in D_r$ and $\phi(z) = y$ for every $k < n, n > 1$ then:

Case (A) If:

$\#S \overset{n-1}{\Rightarrow} \#yB \Rightarrow \#ya$ by Induction Hypothesis and (1)

$S_1 \overset{n-1}{\Rightarrow} zA_1 \Rightarrow za\bar{a}$

It is clear that if $z \in D_r$, so is $za\bar{a}$. And if $\phi(z) = y$ then $\phi(za\bar{a}) = ya$

Case (B) If:

$\#S \overset{k}{\Rightarrow} \#uA \underset{i}{\Rightarrow} i\#uaB \overset{n-1}{\Rightarrow} i\#uayC \underset{i}{\overset{n}{\Rightarrow}} \#uaya$ by Induction Hypothesis, (2) and (3)

$S \overset{k}{\Rightarrow} u'A_1 \Rightarrow u'ia\bar{a}B_1 \overset{n-1}{\Rightarrow} u'ia\bar{a}zC_1 \overset{n}{\Rightarrow} u'ia\bar{a}z\bar{i}a\bar{a}$

If $u', z \in D_r$ so is $u'ia\bar{a}z\bar{i}a\bar{a}$. And if $\phi(z) = y$ and $\phi(u') = u$ then $\phi(u'ia\bar{a}z\bar{i}a\bar{a}) = uaza$

The **reverse direction** $\phi(D_r \cap L(G1)) \subseteq L$. Suppose $w \in \phi(D_r \cap L(G1))$.

Basis:

a) $w = \epsilon$ then $S_1 \Rightarrow \epsilon$ and $\#S \Rightarrow \#\epsilon$

b) $|w| = 1$ then $S_1 \Rightarrow a\bar{a}$ and $\#S \Rightarrow \#a$ and $\phi(a\bar{a}) = a$

Induction Hypothesis Suppose that if there is a CFL derivation $uA \overset{k}{\Rightarrow} uzB$ and $z \in D_r$ then there is a GIL derivation $\delta\#u' \overset{k}{\Rightarrow} \delta\#u'yA$ such that $\phi(z) = y$

Case (A) If:

$S_1 \overset{n-1}{\Rightarrow} zA_1 \Rightarrow za\bar{a}$ then

$\#S \overset{n-1}{\Rightarrow} \#yA \Rightarrow \#ya$

If $z \in D_r$ so is $za\bar{a}a\bar{a}$ and if $\phi(z) = y$ then $\phi(za\bar{a}a\bar{a}) = yaa$

Case (B) If:

$S_1 \overset{k}{\Rightarrow} u'A_1 \Rightarrow u'ia\bar{a}B_1 \overset{n-1}{\Rightarrow} u'ia\bar{a}zC_1 \Rightarrow u'ia\bar{a}z\bar{i}a\bar{a}$ then

$\#S \overset{k}{\Rightarrow} \#yA \underset{i}{\Rightarrow} i\#yaB \overset{n-1}{\Rightarrow} i\#uayC \underset{\bar{i}}{\Rightarrow} \#uaya$

If $u', z \in D_r$ so is $u'ia\bar{a}z\bar{i}a\bar{a}$ and if $\phi(z) = y$ and $\phi(u') = u$ then $\phi(u'ia\bar{a}z\bar{i}a\bar{a}) = uaza$

If other (possible) derivations were applied to $u'ia\bar{a}zC$ then the corresponding string would not be in D_r.

□

We exemplify how the Chomsky-Schützenberger equivalence of GIGs can be obtained constructing the grammar for the language $L1$:

Example 13 () $L(G_{ww}) = \{ww \mid w \in \{a,b\}^*\}$

Given $G_{ww} = (\{S, S'\}, \{a,b\}, \{i,j\}, S, \#, P)$ and $P =$

1. $S \underset{i}{\rightarrow} aS$ 2. $S \underset{j}{\rightarrow} bS$ 3. $S' \rightarrow \epsilon$ 4. $S \underset{\bar{i}}{\rightarrow} S'a$ 5. $S \underset{\bar{j}}{\rightarrow} S'b$
6. $S' \underset{\bar{i}}{\rightarrow} S'a$ 7. $S' \underset{\bar{j}}{\rightarrow} S'b$

Let D_4 be the semi-Dyck set defined over $\Sigma_0 = \{a, \bar{a}, b, \bar{b}, i, \bar{i}, j, \bar{j}\} = \Sigma \cup \bar{\Sigma}$, $\Sigma = T = \{a,b\} \cup I = \{i,j\}$ $CFL_{ww} = L(G_{wwD})$, and we construct G_{wwD} from G_{ww} as follows:

$G_{wwD} = (\{S, S'\}, \{a, \bar{a}, b, \bar{b}, i, \bar{i}, j, \bar{j}\}, S, \#, P)$ and $P =$

1. $S \rightarrow a\bar{a}iS$ 2. $S \rightarrow b\bar{b}jS$ 3. $S' \rightarrow \epsilon$ 4. $S \rightarrow S'\bar{i}a\bar{a}$ 5. $S \rightarrow S'b\bar{b}\bar{j}$
6. $S' \rightarrow S'a\bar{a}\bar{i}$ 7. $S' \rightarrow S'b\bar{b}\bar{j}$

3.4 Equivalence of GIGs and LR-2PDA

In this section we show the equivalence of GIGs and LR-2PDA. The proof is an extension of the equivalence proof between CFGs and PDAs. This is on our view a consequence of the fact that GIGs are a natural extension of CFGs.

Theorem 2 *If L is a GIL, then there exists a LR-2PDA M such that L= L(M).*

Proof Assume a GIG G that generates L, use G to construct an equivalent LR-2PDA. We use as a basis the algorithm used in [77] to convert a CFG into an equivalent PDA, the only change that we require is to take into account the operations on the indices stated in the grammar rules. The general description of the algorithm is as follows. Given a GIG G:

1. Place the marker symbol $\perp$ and the start variable on the stack.
2. Repeat the following steps forever.

   a. If the top of the stack is a variable A, select one of the rules for A and substitute A by the string on the right-hand side of the rule, and affect the second stack accordingly.

   b. If the top of the stack is a terminal a, compare it to the next symbol in the input. If they match repeat. Otherwise reject this branch of the non-determinism.

   c. If the top of the stack is $\perp$ accept.

The corresponding formalization is: Let $G = (V, T, I, S, \#, P)$ be a GIG. Construct the LR-2PDA $M = (\{q_0, q_1, q_2\}, T, \Gamma, \delta, q_0, \perp, q_2)$ where $\Gamma = V \cup T \cup I$ and δ:

1. $\delta(q_0, \epsilon, \epsilon, \epsilon) = \{(q_1, S\perp, \perp)\}$
2. $\delta(q_1, \epsilon, A, \epsilon) = \{(q_1, w, \epsilon) |\ where\ A \rightarrow w\ is\ a\ rule\ in\ P\}$
3. $\delta(q_1, \epsilon, A, j) = \{(q_1, w, j) |\ where\ A \underset{[j]}{\rightarrow} w\ is\ a\ rule\ in\ P\}$
4. $\delta(q_1, a, A, u) = \{(q_1, w, ju) |\ where\ A \underset{j}{\rightarrow} aw\ is\ a\ rule\ in\ P\}$
5. $\delta(q_1, \epsilon, A, j) = \{(q_1, w, \epsilon) |\ where\ A \underset{\bar{j}}{\rightarrow} w\ is\ a\ rule\ in\ P\}$
6. $\delta(q_1, a, a, \epsilon) = \{(q_1, \epsilon, \epsilon)\}$

Now we have to prove that L(G) = L(M). Remember GIGs require *leftmost* derivations for the rules that affect the stack to apply. Therefore each sentential must be of the form $x\alpha$ where $x \in T$ and $alpha \in V \cup T$

Claim 5 $\#S \overset{*}{\Rightarrow} \#x\alpha$ *iff* $(q1, x, S, \perp) \overset{*}{\vdash} (q1, \epsilon, \alpha, \perp)$

Notice that the proof is trivial if all the derivations are context free equivalent, i.e. if $\#S \underset{\delta_1}{\overset{*}{\Rightarrow}} \#x\alpha$ if and only if $(q1, x, S, \perp) \underset{\delta_2}{\overset{*}{\vdash}} (q1, \epsilon, \alpha, \perp)$

where $\delta_1, \delta_2 = \epsilon$ or $\delta = [\#], \delta_2 = \perp$

This holds true by the equivalence of PDA and CFG. Therefore we need to prove that whenever the stacks are used, the corresponding equivalence holds, i.e. that the claim above is a particular case of: $\#S \overset{*}{\Rightarrow} \delta\#x\alpha$ iff $(q1, x, S, \bot) \overset{*}{\vdash} (q1, \epsilon, \alpha, \delta)$

First direction (if) Basis:

$\#S \Rightarrow \#x$ if $(q1, x, S, \bot) \vdash (q1, \bot, \bot)$

where $x \in T \cup \{\epsilon\}$ and $\delta = \epsilon$ or $\delta = [\#]$. By (1.) or (2.) respectively.

Induction Hypothesis. Assume $(q1, x, S, \bot) \overset{k}{\vdash} (q1, \epsilon, \alpha, \delta)$ implies $\#S \overset{*}{\Rightarrow} \delta\#y\beta$ for any $k \geq 1$ and show by induction on k that $\#S \overset{*}{\Rightarrow} \delta\#x\alpha$ Let $x = ya$

Case A. This case is a standard CFL derivation.

$(q_1, ya, S, \bot) \overset{k-1}{\vdash} (q_1, a, \beta, \delta) \vdash (q_1, \epsilon, \alpha, \delta)$.

Given the last step implies $\beta = A\gamma$ for some $A \in V$, $A \underset{\iota}{\rightarrow} a\eta \in P$ where $\iota = \epsilon$ and $\alpha = \eta\gamma$ Hence:

$\#S \overset{*}{\Rightarrow} \delta\#y\beta \Rightarrow ya\eta\gamma = x\alpha$

Case B. Push moves in the auxiliary stack.

$(q_1, ya, S, \bot) \overset{k-1}{\vdash} (q_1, a, \beta, \gamma) \underset{j}{\vdash} (q_1, \epsilon, \alpha, j\gamma)$ where $\delta = j\gamma$.

The last step implies $A \underset{j}{\rightarrow} a\eta \in P$

Hence: $\#S \overset{*}{\Rightarrow} \gamma\#y\beta \underset{j}{\Rightarrow} j\gamma\#ya\eta\gamma = x\alpha$

Case C. Pop moves in the auxiliary stack.

$(q_1, ya, S, \bot) \overset{k-1}{\vdash} (q_1, a, \beta, j\delta) \underset{\bar{j}}{\vdash} (q_1, \epsilon, \alpha, \delta)$.

The last step implies $A \underset{\bar{j}}{\rightarrow} a\eta \in P$ Hence: $\#S \overset{*}{\Rightarrow} j\delta\#y\beta \underset{\bar{j}}{\Rightarrow} \delta\#ya\eta\gamma = x\alpha$

The only if direction: Suppose that $\#S \overset{k}{\rightarrow} \delta\#w\alpha$ implies that $(q_1, w, S, \bot) \overset{*}{\vdash} (q_1, \epsilon, \alpha, \delta)$ for any $k \geq 1$

Case A context free

If $\#S \overset{k}{\Rightarrow} \delta\#yA\gamma \Rightarrow \delta\#ya\eta\gamma = \delta\#x\alpha$

where $x = ya$ and $\alpha = \eta\gamma$ then by IH and 2 or 3:

$(q_1, ya, S, \bot) \overset{*}{\vdash} (q_1, a, A\gamma, \delta) \vdash (q_1, \epsilon, \eta\gamma, \delta)$

Case B Push

$\#S \overset{*}{\Rightarrow} \mu\#yA\gamma \underset{j}{\Rightarrow} j\mu\#ya\eta\gamma = j\mu\#x\alpha$ then by IH and 4.

$(q_1, ya, S, \bot) \overset{k-1}{\vdash} (q_1, a, A\gamma, \mu) \underset{j}{\vdash} (q_1, \epsilon, \alpha, j\mu)$ where $\delta = j\mu$.

Case C Pop

$\#S \overset{*}{\Rightarrow} j\delta\#yA\gamma \underset{\bar{j}}{\Rightarrow} \delta\#ya\eta\gamma = \delta\#x\alpha$ then by IH and 5.

$(q_1, ya, S, \bot) \overset{k-1}{\vdash} (q_1, a, A\gamma, j\delta) \underset{\bar{j}}{\vdash} (q_1, \epsilon, \alpha, \delta)$

Given the values $\delta, \alpha = \epsilon$ (which excludes Case B Push to be the last step or move but not as a step $\leq k-1$) we obtain $\#S \overset{*}{\Rightarrow} \delta\#x\alpha = \#x$ iff $(q_1, x, S, \bot) \vdash (q_1, \epsilon, \alpha, \delta) = (q_1, \epsilon, \bot, \bot)$ $\square$

Theorem 3 *If L is recognized by a LR-2PDA M, then L is a GIL.*

Proof Given a LR-2PDA construct an equivalent GIG as follows. This conversion algorithm to construct a CFG from a PDA follows the one in [41] (including the notation), the only modification is to mimic the operation of the auxiliary stack, with the stack of indices in the grammar, i.e. items 3. and 4. This construction is defined so that a leftmost derivation in G of a string w is a simulation of the LR-2PDA M when processing the input w. Let M be the LR-2PDA $(Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$. Assume Γ is composed of two disjoint subsets Γ_1, Γ_2, such that Γ_1 is used in the main stack and Γ_2 is used in the auxiliary stack. Let G be a GIG $= (V, \Sigma, I, S, \#, P)$ where V is the set of variables of the form qAp, q and p in Q, A in Γ, plus the new symbol S. $I = \Gamma_2$, and P is the set of productions.

1. $S \underset{\epsilon}{\rightarrow} q_0 \bot q$ for each q in Q

   And for each $q, q_1, q_2, ..., q_n$ in Q, each a in $\Sigma \cup \{\epsilon\}$, A, B_1, B_2, B_{n-1} in Γ and i, j in I.

2. $[qAq_n] \underset{\epsilon}{\rightarrow} a\ [q_1 B_1 q_2]\ [q_2 B_2 q_3]\ ...[q_{n-1} B_{n-1} q_n]$ such that
   $\delta(q, a, A, \epsilon)$ contains $(q_1, B_1 B_2 ... B_{n-1}, \epsilon)$.

3. $[qAq_n] \underset{[j]}{\rightarrow} a\ [q_1 B_1 q_2]\ [q_2 B_2 q_3]\ ...[q_{n-1} B_{n-1} q_n]$ such that
   $\delta(q, a, A, \mathbf{j})$ contains $(q_1, B_1 B_2 ... B_{n-1}, \mathbf{j})$.

4. $[qAq_n] \underset{j}{\rightarrow} a\ [q_1 B_1 q_2]\ [q_2 B_2 q_3]\ ...[q_{n-1} B_{n-1} q_n]$ such that
   $\delta(q, a, A, \mathbf{i})$ contains $(q_1, B_1 B_2 ... B_{n-1}, \mathbf{ji})$. (push)

5. $[qAq_n] \underset{\bar{j}}{\rightarrow} a\ [q_1 B_1 q_2]\ [q_2 B_2 q_3]\ ...[q_{n-1} B_{n-1} q_n]$ (pop) such that
   $\delta(q, a, A, \mathbf{j})$ contains $(q_1, B_1 B_2 ... B_{n-1}, \epsilon)$.

In all the cases if $n = 1$, then the production is $[qAq_1] \underset{\delta}{\rightarrow} a$. Notice that in case 4, a must be in Σ, and cannot be ϵ given the constraints imposed on the possible transitions in a LR-2PDA.

We need to show that $L(G) = L(M)$. We follow the proof from [41], proving by induction on the steps in a derivation of G or moves of M that:

$\#[qAp] \overset{*}{\Rightarrow} \#w$ if and only if $(q, w, A, \bot) \overset{*}{\vdash} (p, \epsilon, \bot, \bot)$

If only rules created with step 1 or 2 and the corresponding transitions are used then, these are a CFG and a PDA and therefore follows from the standard CFG, PDA equivalence.

The basis is also proven for both directions, given the only type of rules/moves applicable are 1 and 2 (in this case where $\delta = [\perp]$).

First (if) direction:

Induction Hypothesis: If $(q, w, A, \gamma_1) \overset{k}{\vdash} (p, \epsilon, \perp, \gamma_k)$ then $\gamma_1 \# u[qAp] \overset{*}{\Rightarrow} \gamma_k \# uw$, where $\gamma_i \in I^*$ and $u, w \in \Sigma^*$. In other words, the induction hypothesis states that whenever the LR-2PDA M recognizes a string emptying the main stack and changing the auxiliary stack from a string of indices γ_1 to a string of indices γ_k, in k moves (for any $k \geq 1$) then the GIG G will derive the same string w and string of indices γ_k

I. $(q, ay, A, \perp) \vdash (q_1, y, B_1 B_2 ... B_m, \gamma_1) \overset{k-1}{\vdash} (p, \epsilon, \perp, \gamma_k)$

Decomposing y in m substrings, ie. $y = y_1 ... y_m$ where each y_i has the effect of popping B_i from the stack (see the details corresponding to the PDA-CFG case in [41]) then there exist states $q_2, ..., q_m+1$, $q_m + 1 = p$, such that the inductive hypothesis holds for each:

II. $(q_i, y_i, B_i, \gamma_i) \overset{*}{\vdash} (q_{i+1}, \epsilon, \gamma_{i+1})$ then $\gamma_i \#[q_i B_i q_{i+1}] \overset{*}{\Rightarrow} \gamma_{i+1} \# y_i$

for $1 \leq j \leq m$

The first move in I.:

$(q, ay, A, \perp) \vdash (q_1, y, B_1 B_2 ... B_m, \gamma_1)$ implies

$\#[qAq_n] \underset{\delta}{\Rightarrow} \gamma_1 \# a\, [q_1 B_1 q_2]\, [q_2 B_2 q_3]\, ... [q_{n-1} B_{n-1} q_n]$ where δ can be only be some j in I then $\gamma_1 = j$ or δ is $[\perp]$, then $\gamma_1 = \epsilon$.

Therefore I. implies:

$\#[qAq_n] \Rightarrow \gamma_1 \# a [q_1 B_1 q_2]\, [q_2 B_2 q_3]\, ... [q_{n-1} B_{n-1} q_n] \overset{*}{\Rightarrow} \gamma_k \# a y_1 y_2 .... y_m = \#w$

The same holds for the last symbol that is extracted from the stack (from II again):

III. $(q, ay, A, \perp) \overset{k-1}{\vdash} (q_{n-1}, y_m, B_m, \gamma_{k-1}) \vdash (p, \epsilon, \perp, \gamma_k)$ implies:

$\#[qAq_n] \overset{*}{\Rightarrow} \gamma_{k-1} \# a y_1 y_2 ... y_{m-1} [q_{n-1} B_{n-1} q_n] \underset{\delta}{\Rightarrow} \gamma_k \# a y_1 y_2 .... y_m = \#w$

where $\gamma_k = \perp$, then either 1. $\gamma_{k-1} = \perp$ and $\delta = \epsilon$ *or* $[\#]$ or 2. $\gamma_{k-1} = j \in I$ and $\delta = \bar{j}$

Now the "only if" part (the basis was proven above), so the induction hypothesis is:

If $\gamma_1 \# u[qAp] \overset{k}{\Rightarrow} \gamma_k \# uw$ then $(q, w, A, \gamma_1) \overset{*}{\vdash} (p, \epsilon, \perp, \gamma_k)$ for any $k \geq 1$.

Suppose:

I. $\#u[qAq_n] \underset{\delta}{\Rightarrow} \gamma_1\#ua\,[q_1B_1q_2]\,[q_2B_2q_3]\,...[q_{n-1}B_{n-1}q_n] \overset{k-1}{\Rightarrow} \gamma_k\#uw$

where $q_n = p$.

Again, we can decompose w as $ax_1x_2...x_n$ such that

$\gamma_i\#u_i[q_iB_iq_{i+1}] \overset{*}{\Rightarrow} \gamma_{i+1}\#u_ix_i$

therefore by the induction hypothesis:

$(q_i, x_i, B_i, \gamma_i) \overset{*}{\vdash} (q_{i+1}, \epsilon, \bot, \gamma_{i+1})$ for $1 \leq i \leq n$

Now we can add $B_{i+1}...B_n$ to each stack in the sequence of ID's so that:

II. $(q_i, x_i, B_iB_{i+1}...B_n, \gamma_i) \overset{*}{\vdash} (q_{i+1}, \epsilon, B_{i+1}...B_n, \gamma_{i+1})$

From the first step of the derivation in I. we obtain (remember $w = ax_1x_2...x_n$):

$(q, w, A, \bot) \vdash (q_1, x_1x_2...x_n, B_1B_2...B_n, \gamma_1)$

where γ_1 is $\bot$ if δ in the first step of I. is ϵ or $[\bot]$.

and from this move and II., where $i = 1, 2, ...n$, then $(q, w, A, \bot) \overset{*}{\vdash} (p, \epsilon, \bot, \gamma_k)$

If we consider the last step from I.:

$\#u[qAq_n] \overset{k-1}{\Rightarrow} \gamma_{k-1}\#uax_1...x_m\,[q_{n-1}B_{n-1}q_n] \underset{\delta}{\Rightarrow} \gamma_k\#uw$ which in turn implies:

$(q, w, A, \bot) \overset{*}{\vdash} (q_{n-1}, x_m, B_n, \gamma_k - 1) \vdash (q_n, \epsilon, \bot, \gamma_k)$ where the possibilities are either $x_m = x_n$ or $x_m = \epsilon$ and given we require γ_k to be $\bot$ then the only possibilities are:

(A) either $\gamma_k = \gamma_{k-1} = \bot$ and $\delta = \epsilon$ or $\delta = [\bot]$ or

(B) $\delta = \bar{\gamma}_{k-1}$ then $\gamma_k = \bot$

The final observation is that we satisfy the first claim assigning the following values: $q = q_O$ and $A = Z_0$. So according to 1. in the construction of the GIG G, we obtain:

$\#S \overset{*}{\Rightarrow} \#w$ iff $(q_0, w, Z_0) \overset{*}{\vdash} (p, \epsilon, \bot, \bot)$. $\square$

Chapter 4

Global Index Languages Properties

4.1 Relations to other families of languages

There are two clear cut relations that place GILs between CFLs and CSLs: GILs are included in CSLs and GILs include CFLs. Most of other relations are more difficult to determine. GILs might include LILs and perhaps LCFRLs. GILs and ILs might be incomparable. GILs are incomparable to the Growing Context Sensitive Languages (GCSLs). However GCSLs might include trGILs. It is interesting to establish the relation of GILs to LILs and LCFRLs because there is an extensive tradition that relates these families of languages to natural language phenomena. Moreover, the relation to LILs is interesting because of their similarities both in the grammar and the automaton model. The relation to Growing Context Sensitive languages is interesting for a similar reason. The corresponding automaton is another two stack machine, and the constraints imposed on the use of the additional stack in the GIL automaton might be seen as a *shrinking* property. We do not address possible relationship with the counter languages [42] however, another family of languages that has some similarities with GILs.

- Proper inclusion of GILs in CSLs.

  The additional stack in GIGs/LR-2PDA, corresponds to an amount of tape equal to the input in a LBA. Due to the constraints on the *push* productions at most the same amount of symbols as the size of the input can be introduced in the additional stack. Therefore a LBA can simulate a LR-2PDA.

- Proper inclusion of GILs in ILs.

  This is an open question. ILs are not semi-linear and GILs are semi-linear. GILs contain some languages that might not be ILs, such as the MIX language.

- Proper inclusion of LILs in GILs.

  We consider this quite possible. We discuss this issue in Chapter 6. However it remains as an open question.

- GILs and Growing Context Sensitive Languages are incomparable.

  GCSLs include the language $\{ba^{2^n}|n \geq 1\}$ which is not semi-linear. The copy language $\{ww|w \in \Sigma^*\}$ is not a GCSL [53].

- CFLs are properly included in GILs.

  This result is obtained from the definition of GIGs.

4.2 Closure Properties of GILs

The family of GILs is an Abstract Family of Languages.

Proposition 1 *GILs are closed under: union, concatenation and Kleene closure.*

This can be proved using the same type of construction as in the CFL case (e.g. [41]), using two GIG grammars, instead of two CFG grammars. For concatenation, care should be taken that the two sets of stack indices be disjoint.

Proof Let L_1 and L_2 be GILs generated respectively by the following GIGs:

$G_1 = (N_1, T_1, I_1, P_1, \#, S_1)$ and $G_2 = (N_2, T_2, I_2, P_2, \#, S_2)$

Since we may rename variables at will without changing the language generated, we assume that N_1, N_2, I_1, I_2 are disjoint. Assume also that S_3, S_4, S_5 are not in N_1 or N_2.

Union $L_1 \cup L_2$:. Construct the grammar $G_3 = (N_1 \cup N_2, T_1 \cup T_2, I_1 \cup I_2, P_3, \#, S_3)$ where P_3 is $P_1 \cup P_2$ plus the productions $S_3 \rightarrow S_1 \mid S_2$. A string w is in L_1 if and only if the derivation $\#S_3 \Rightarrow \#S_1 \stackrel{*}{\Rightarrow} \#w$ is a derivation in G_3. The same argument holds for w in L_2 starting the derivation $\#S_3 \Rightarrow \#S_2$. There are no other derivations from S_3 than $\#S_3 \Rightarrow \#S_1$ and $\#S_3 \Rightarrow \#S_2$ Therefore $L_1 \cup L_2 = L_3$.

Concatenation:. Construct $G_4 = (N_1 \cup N_2, T_1 \cup T_2, I_1 \cup I_2, P_4, \#, S_4)$ such that P_4 is $P_1 \cup P_2$ plus the production $S_4 \rightarrow S_1 S_2$. The proof that $L(G_4) = L(G_1)L(G_2)$ is similar to the previous one. A string w is in L_1 if and only if the derivation $\#S_4 \Rightarrow \#S_1 \stackrel{*}{\Rightarrow} \#wS_2$ is a derivation in G_4, and a string u is in L_2 if and only if the derivation $\#wS_2 \stackrel{*}{\Rightarrow} \#wu$, is a derivation in G_4 for some w, given the only production that uses S_1 and S_2 is S_4 $L(G_4) = L_1 \cdot L_2$. Notice that $\#S_4 \Rightarrow \#S_1 \stackrel{*}{\Rightarrow} \delta\#wS_2$ for some δ in I_1 does not produce any further derivation because the I_1 and I_2 are disjoint sets, therefore also no $\delta \in I_2$ can be generated at this position.

Closure:.Let $G_5 = (N_1 \cup \{S_5\}, T_1, I_1, P_5, \#, S_1)$ where P_5 is P_1 plus the productions:

$S_5 \underset{[\#]}{\rightarrow} S_1\ S_5\ |\epsilon$.

The proof is similar to the previous one.

□

Proposition 2 *GILs are closed under substitution by ϵ-free CFLs, ϵ-free regular sets, ϵ-free homomorphisms.*

This proof adapts the substitution closure algorithm given in [41] to prove substitution closure by CFLs. The substituting CFG must be in Greibach Normal Form (so as to respect the constraints on productions that push indices in the stack).

Proof Let L be a GIL, $L \subseteq \Sigma^*$, and for each a in Σ let L_a be a CFL. Let L be $L(G)$ and for each a in Σ let L_a be $L(G_a)$, where each G_a is in GNF. Assume that the non-terminals of G an G_a are disjoint. Construct a GIG grammar $G1$ such that:

(a) the non terminals of $G1$ are all the non-terminals of G and of each G_a;

(b) the terminals of $G1$ are the terminals of the substituting grammars G_a;

(c) the set of indices I and # are the set of indices and # from G

(d) the start symbol of $G1$ is the start symbol of G.

(e) The productions of $G1$ are all the productions of each G_a together with all the productions from G modified as follows: replace each instance of a terminal a in each production by the start symbol of G_a

□

Note that the requirement that the substituting grammar be in GNF implies that it is an ϵ-free CFL. GILs do not tolerate an ϵ substitution due to the constraints on type b. (*push*) productions which require Greibach normal form.

GILs are also closed under intersection with regular languages and inverse homomorphism. Again, a similar proof to that used in the CFL case works for these closure properties: in both cases a LR-2PDA with a modified control state is used (again see [41] for the CFL case).

Proposition 3 *If L is a Global Index Language and R is a regular set, $L \cap R$ is a Global Index Language.*

Proof Let L be $L(M)$ for an LR-2PDA M $=(Q_G, \Sigma, \Gamma, \delta_G, q_0, \perp, F_G)$ and let R be $L(A)$ for a DFA A $= (Q_A, \Sigma, \delta_A, p_0, F_A)$. Construct M' $= (Q_A \times Q_G, \Sigma, \Gamma, \delta, [p_0, q_0], F_A \times F_G)$ such that δ is defined as follows:

$\delta([p, q], a, X, Y)$ contains $([p', q'], \gamma, \eta)$ if and only if:

$\delta_A(p, a) = p'$ and $\delta_G(q, a, X, Y)$ contains (q', γ, η).

Note that a may be ϵ, in which case $p' = p$ and $\eta = \epsilon$ or $\eta = Y$

It is easy to see that $([p_0, q_0], w, Z, \perp) \overset{i}{\vdash} ([p, q], \epsilon, \gamma, \eta)$ if and only if (an induction proof is given by [41] for the CFL case)

$(q_0, w, \perp, \perp) \overset{i}{\vdash} (q, \epsilon, \gamma, \eta)$ and $\delta(p_0, w) = p$

□

Proposition 4 *Global Index Languages are closed under inverse homomorphism.*

Proof Let $h : \Sigma \rightarrow \Delta$ be a homomorphism and let L be a GIL. Let $L = L(M)$ where M is the LR-2PDA $(Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$.

Construct an LR-2PDA $M1$ accepting $h^{-1}(L)$ in the following way. On input a $M1$ generates the string $h(a)$ stores it in a buffer and simulates M on $h(a)$. Therefore $L(M1) = \{w \mid h(w) \in L\}$.

□

4.3 Semilinearity of GILs and the Constant Growth property

The property of semilinearity in a language requires that the number of occurrences of each symbol in any string (also called the commutative or Parikh image of a language) to be a linear combination of the number of occurrences in some finite set of strings. Parikh [60] showed that CFLs are semilinear,

i.e. that CFLs are *letter-equivalent* to a regular set. This property has been extensively used as a tool to characterize formal languages (e.g. [33, 37]). Semilinearity has been assumed as a good formulation of the *constant growth* property of natural languages (e.g. [46, 86]). However, recently it has been claimed that semilinearity might be too strong a requirement for models of natural language (e.g. [55, 35]). Roughly speaking, the constant growth property requires that there should be no big gaps in the growth of the strings in the languages. We will not discuss here whether semilinearity is the correct approach to formalize constant growth. concept.

We assume a definition of semilinearity of languages such as that found in [39, 60].

Definition 8 *A set V in $\mathbb{N}^n$*

a) is linear if $V = \{v_0 + c_1 v_1 + ... + c_m v_m | c_i \in \mathbb{N},\ v_i \in \mathbb{N}^n$ for $1 \leq i \leq m\}$

b) is semilinear if V is the finite union of linear sets.

The Parikh mapping is a function that counts the number of occurrences in every word of each letter from an ordered alphabet. The Parikh image of a language is the set of all vectors obtained from this mapping to every string in the language.

Definition 9 *Parikh Mapping (see [39]) :*

Let Σ be $\{a_1, ..., a_n\}$, define a mapping from Σ^* into N^n as follows:

$\psi(w) = (|a_1|(w), ..., |a_n|(w))$

where $|a_i|(w)$ denotes the number of occurrences of a_i in w.

The commutative or Parikh image of a language $L \in \Sigma^*$, $\psi(L) = \{\psi(w) | w \in L\}$. If $\psi(L)$ is semi-linear, L is a semi-linear language. Two languages L_1, L_2 in Σ^* are letter-equivalent *if $\psi(L_1) = \psi(L_2)$.*

Intuitively, the property of semi-linearity of GILs follows from the properties of the *control* mechanism that GILs use.

We will use the following lemma in the proof of semilinearity for GIGs.

Lemma 4 *If X and Y are semilinear subsets of $\mathbb{N}^n$, then their intersection $X \cap Y$ is also semilinear.*

The proof of this lemma is in [30], as Theorem 5.6.1.

Theorem 5 *If L is a GIL then L is a semi-linear language.*

We follow the proof presented in [37] to prove that the class of languages $L(M_c)$, where M_c is obtained from an automata class $\mathcal{C}$ by augmenting it with finitely many reversal-bounded counters, is semilinear provided that the languages corresponding to C are semilinear.

The proof exploits the properties of Theorem 1: every GIL $L = \phi(D_r \cap L1)$ such that L1 is a CFL. D_r and $L1$ are semilinear by the Parikh theorem ($L1$ and D_r are CFLs). We modify the construction of $L1$ in the proof of Theorem 1.

Proof Given a GIG $G = (N, \Sigma, I, \#, S, P1)$ Define a CFG grammar $G1 = (N, \Sigma_1, S, P)$ such that Σ_1 is $(\Sigma \cup I \cup \bar{I})$ and $P1$ is given according to the following conditions. For every production $p \in P$ where $a \in T,\ i \in I,\ and\ \alpha, \in (N \cup T)^*$ create a new production $p_1 \in P1$, as follows:

1. **if** $p = A \rightarrow \alpha$ **then** $p_1 = A \rightarrow \alpha$
2. **if** $p = A \underset{i}{\rightarrow} \alpha$ **then** $p_1 = A \rightarrow i\alpha$
3. **if** $p = A \underset{\bar{i}}{\rightarrow} \alpha$ **then** $p_1 = A \rightarrow \bar{i}\alpha$
4. **if** $p = A \underset{[i]}{\rightarrow} \alpha$ **then** $p_1 = A \rightarrow \bar{i}i\alpha$

It can be seen that $L(G1) \subseteq {\Sigma_1}^*$ and that for each w that is in the GIL $L(G)$ there is a word $w1 \in L(G1)$ such that $|i|(w1) = |\bar{i}|(w1)$. This follows from Theorem 1. By the Parikh theorem [60] the Parikh map of the CFL $L(G1)$, $\psi(L(G1))$ is an effectively computable semi-linear set S_1. Let S_2 be the semilinear set $\{(a_1, ..., a_m, i_n, \bar{i}_n ..., i_p, \bar{i}_p) | a_j, i_k, \bar{i}_k \in \mathbb{N}\}$ where the pairs of i_k and $\bar{i}_k$ coordinates correspond to symbols in I and the corresponding element in $\bar{I}$. The intersection of S_1 and S_2 is a computable semilinear set S_3 according to [30]. Computing the intersection of S_1 and S_2 amounts to compute the properties that the *control* performed by the additional stack provides to GILs, in other words, removing the illegal derivations, note that S_2 is a subset of S_1.

From $S3$ we obtain S_0 the semilinear set corresponding to $\psi(L(G))$, removing the pairs of i_k and $\bar{i}_k$ coordinates.

□

Corollary 6 *The emptiness problem is decidable for GILs.*

4.4 Pumping Lemmas and GILs

This section and the remainder of this chapter will present some exploratory ideas rather than established results.

We can consider pumping lemmas for different families of languages as specifying the following facts about them:

- Regular Languages: Inserting any number of copies of some substring in a string always gives a regular set.
- Context Free Languages: every string contains 2 substrings that can be repeated the same number of times as much as you like.
- GCFLs, LCFRLs, others and GILs, there is always a number of substrings that can be repeated as many times as you like.

Following upon this last observation, the following definition is given in [35] as a more adequate candidate to express properties of natural language. This is a candidate for a pumping lemma for GILs.

Definition 10 (Finite Pumpability) .

Let L be a language. Then L is finitely pumpable if there is a constant c_0 such that for any $w \in L$ with $|w| > c_0$, there are a finite number k and strings $u_0, ..., u_k$ and $v_1, ..., v_k$ such that $w = u_0 v_1 u_1 v_2 u_2 ... u_{k-1} v_k uk$ and for each $i, 1 \leq |v_i| < c_0$ and for any $p \geq 0$, $u_0 {v_1}^p u_1 {v_2}^p u_2 ... u_{k-1} {v_k}^p uk \in L$.

Compare the previous definition with the following k-pumpability property of *LCFRS* ([71], and also in [35]) which applies to any level-k language.

Definition 11 (k-pumpability) *Let L be a language. Then L is (universally) $k-$pumpable if there are constants c_0,k such that for any $w \in L$ with $|w| > c_0$, there are strings$u_0, \cdots, u_k$ and $v_1, \cdots, v_k$ such that $u_0 v_1 u_1 v_2 u_2 \cdots u_{k-1} v_k u_k$, for each $i : 1 \leq |v_i| < c_0$, and for any $u_0 {v_1}^p u_1 {v_2}^p u_2 ... u_{k-1} {v_k}^p uk \in L$.*

In this case the k value is a constant determined by the k-level of the language.

4.4.1 GILs, finite turn CFGs and Ultralinear GILs

We showed that there is no boundary in the number of dependencies that can be described by a GIL (see the language L_m in the previous chapter). This fact and the candidate pumping lemma for GILs are related to the following observation. For any CFG that recognizes a number of pairs of dependencies, (e.g., $L_a = \{a^n b^n, c^m d^m | n \geq 1, m \geq 1\}$ has 2 pairs of dependencies or *turns* [32, 39, 88, 5]) there is a GIG that generates the total number of dependencies included in the

pairs. This is done by "connecting" the CFG pairs of dependencies (*turns*) with "dependencies" (*turns*) introduced by the GIG extra stack. This "connection" is encoded in the productions of the corresponding terminals. For example, in L_a it is possible to "connect" the b's and c's, in a *turn* of the GIG stack: any production that has a b in the right-hand side also adds an index symbol to the GIG stack (our notation for this will be b_x), and every production that has a c in the right-hand side removes the same index symbol from the production stack (we will denote this by $c_{\bar{x}}$).

$L_a = \{a^n b^n c^m d^m | n \geq 1, m \geq 1\} \rightarrow L_b = \{a^n {b_x}^n {c_{\bar{x}}}^m d^m | n \geq 1, m \geq 1\} = \{a^n b^n c^n d^n | n \geq 1, \}$

Conjecture 1 *For any k-turn CFG, where $k > 1$ is finite, there is a GIG that generates a language with $(2 \cdot k)$ dependencies.*

These observations are also related to the fact that LIGs (and the equivalent automata), can connect one-turn for each stack [85]. Therefore the boundary of the number of dependencies is four.

4.5 Derivation order and Normal Form

We introduced GIGs as a type of control grammar, with leftmost derivation order as one of its characteristics. GIG derivations required leftmost derivation order. We did not address the issue of whether this is a normal form for unconstrained order in the derivation, or if an extension of GIGs with unconstrained order in the derivation would be more powerful than GIGs. In this section we will speculate about the possibility of alternative orders in the derivation, e.g. (rightmost) or free derivation order.

The following grammar does not generate any language in either leftmost or rightmost derivation order:

Example 14 $L(G_6) = \{a^n b^m c^n d^m\}$, *where*
$G_6 = (\{S, A, B, C, D\}, \{a, b, c, d\}, \{i, j\}, S, \#, P)$ *and* $P=$

1. $S \rightarrow A\ B\ C\ D$
2. $A \underset{i}{\rightarrow} aA$
3. $A \underset{i}{\rightarrow} a$
4. $B \underset{j}{\rightarrow} bB$
5. $B \underset{j}{\rightarrow} b$
6. $C \underset{\bar{i}}{\rightarrow} cC$
7. $C \underset{\bar{i}}{\rightarrow} c$
8. $D \underset{\bar{j}}{\rightarrow} dD$
9. $D \underset{\bar{j}}{\rightarrow} d$

The only derivation ordering is an interleaved one, for instance all the a's are derived first, then all the c's, followed by the b's and finally the d's. However we know that such a language is a GIL. It is our belief that the following conjecture holds, though it might be difficult to prove. Unrestricted derivation order would introduce only unnecessary additional ambiguity.

Conjecture 2 *Unrestricted order of the derivation in a GIG is not more powerful than leftmost derivation order.*

Following the previous example, the next example might seem impossible with leftmost/rightmost derivation. It looks like an example that might support the possibility that unrestricted order of the derivation might yield more powerful grammars.

Example 15 $L(G_{6b}) = \{a^n b^m c^l d^m e^n f^l \mid n, m, n \geq 1\}$, *where*
$G_{6b} = (\{S, A, B, C, D, E, F\}, \{a, b, c, d, e, f\}, \{i, j, k\}, S, \#, P)$ *and* P *is:*

1. $S \rightarrow ABCDEF$
2. $A \underset{i}{\rightarrow} aA$
3. $A \underset{i}{\rightarrow} a$
4. $B \underset{j}{\rightarrow} bB$
5. $B \underset{j}{\rightarrow} b$
6. $C \underset{k}{\rightarrow} cC \mid c$
7. $D \underset{\bar{j}}{\rightarrow} dD$
8. $D \underset{\bar{j}}{\rightarrow} d$
9. $E \underset{\bar{i}}{\rightarrow} eE$
10. $E \underset{\bar{i}}{\rightarrow} e$
11. $F \underset{\bar{k}}{\rightarrow} fF$
12. $F \underset{\bar{k}}{\rightarrow} f$

A derivation with unrestricted order would proceed as follows:
$S \Rightarrow ABCDEF \Rightarrow i\#aABCDEF \Rightarrow ji\#abCDEF \Rightarrow kji\#abcDEF \Rightarrow$
$ji\#abcDEf \Rightarrow i\#abcdEf \Rightarrow \#abcdef$

However the following grammar generates the same language in leftmost derivation:

Example 16 $L(G_{6c} = \{a^n b^m c^l d^m e^n f^l \mid n, m, n \geq 1\}$, *where*
$G_{6b} = (\{S, A, B, C, D, E, F\}, \{a, b, c, d, e, f\}, \{i, j, k\}, S, \#, P)$ *and* P *is:*

1. $S \rightarrow A$
2. $A \underset{i}{\rightarrow} aA$
3. $A \underset{i}{\rightarrow} aB$
4. $B \underset{j}{\rightarrow} bB$
5. $B \underset{j}{\rightarrow} bC$
6. $C \underset{k}{\rightarrow} cCf \mid cMf$
7. $M \rightarrow DE$
8. $D \underset{\bar{j}}{\rightarrow} dD$
9. $D \underset{\bar{j}}{\rightarrow} d$
10. $E \underset{\bar{i}}{\rightarrow} Ee$
11. $E \underset{\bar{i}}{\rightarrow} De$

Similarly the following language which contain *unordered* crossing dependencies is a GIL:
$\{a^n b^m c^l d^p b^m_2 a^n_2 d^p_2 c^l_2 | n, m, l, p \geq 1\}$

The next example also shows that leftmost and rightmost derivation ordering might produce two different languages with the same grammar for each ordering, introducing therefore another dimension of ambiguity.

Example 17 $L(G_7) = \{ww | w \in \{a, b\}^*\} \cup \{ww^R | w \in \{a, b\}^*\}$ *(in leftmost and rightmost derivation respectively)*

$G_7 = (\{S, S', A, B\}, \{a, b\}, \{i, j\}, S, \#, P)$ *and* $P =$

1. $S \underset{i}{\rightarrow} aS$ 2. $S \underset{j}{\rightarrow} bS$ 3. $S \rightarrow S'$ 4. $S' \rightarrow S'A$ 5. $S' \rightarrow S'B$
6. $S' \rightarrow A$ 7. $S' \rightarrow B$ 8. $A \underset{\bar{i}}{\rightarrow} a$ 9. $B \underset{\bar{j}}{\rightarrow} b$

Rightmost derivation produces:

$$\#S \underset{i}{\rightarrow} i\#aS \underset{j}{\rightarrow} ji\#abS \rightarrow ji\#abS'B \rightarrow ji\#abAB \underset{\bar{j}}{\rightarrow} i\#abAb \underset{\bar{i}}{\rightarrow} \#abab$$

Leftmost derivation produces:

$$\#S \underset{i}{\rightarrow} i\#aS \underset{j}{\rightarrow} ji\#abS \rightarrow ji\#abS'A \rightarrow ji\#abBA \underset{\bar{j}}{\rightarrow} i\#abbA \underset{\bar{i}}{\rightarrow} \#abba$$

Chapter 5

Recognition and Parsing of GILs

5.1 Graph-structured Stacks

In this section we introduce the notion of a graph-structured stack [80] to compute the operations corresponding to the *index* operations in a GIG. It is a device for efficiently handling of non-determinism in stack operations.[1]

If all the possible stack configurations in a GIG derivation were to be represented, the number of possible configurations would grow exponentially with the length of the input. As an initial approach, each node of the graph-structured stack will represent a unique index and a unique length of the stack. Each node at length n can have an edge only to a node at length $n-1$. This is a departure from Tomita's graph-structured stack. While the number of possible nodes increases, the number of edges connecting a node to others is limited. The set of nodes that represents the top of the stack will be called *active nodes* (following Tomita's terminology).

For instance, in figure 5.1, the *active nodes* are represented by circles and the *inactive* are represented by squares. The numbers indicate the length.

The graph-structured stack at the left represents the following possible stack configurations: iii#, iji#, ijj#, jjj#, jij#, jii#. The one at the right is equivalent to the maximal combination of stack configurations for indices i, j with maximum length of stack 3. (i.e.) #, i#, j#, ii#, ij#, ji#, jj#, iii#, etc.

[1] See [51] for a similar approach to Tomita's, and [50] for an approach to statistical parsing using graph algorithms.

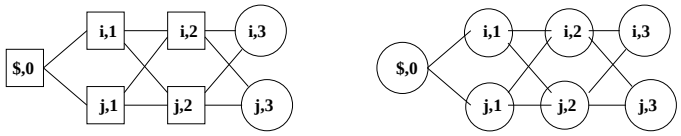


Figure 5.1: Two graph-structured stacks

The following operations are possible on the graph-structured stack.

Push(*newnode,oldnode*). Creates a *newnode* if it is not in the graph and creates an edge from *newnode* to *oldnode* if necessary.

Pop(*curretnode*). Retrieves all the nodes that are connected by an edge from the *currentnode*. Because at this point we use nodes of unambiguous length, *currentnode* connects only to nodes of current length-1. Therefore the number of edges connecting a node to a preceding one is bound by the size of the indexing vocabulary.

5.2 GILs Recognition using Earley Algorithm

5.2.1 Earley Algorithm.

Earley's parsing algorithm [27] computes a leftmost derivation using a combination of top-down prediction and bottom-up recognition. Earley's main data structures are states or "items", which are composed of a "dotted" rule and a starting position: $[A \rightarrow \alpha \bullet \beta, p_j]$. These items are used to represent intermediate steps in the recognition process. Earley items are inserted in sets of states. There are as many sets as positions in the input string. Therefore, given a string $w = a_1 a_2 ... a_n$ with $n \geq 0$ any integer i such that $0 \leq i \leq n$ is a position in w. An item $[A \rightarrow \alpha \bullet \beta, p_j]$ is inserted in the set of states S_i if α corresponds to the recognition of the substring $a_j ... a_i$.

Algorithm 1 (Earley Recognition for GFGs) .

Let $G = (N, T, S, P)$ be a CFG. Let $w = a_1 a_2 \cdots a_n$ be an input string, $n \geq 0$, and $a_i \in T$ for $1 \leq i \leq n$.

Create the Sets S_i:

1 $S_0 = [S' \rightarrow \bullet S\$, 0]$

2 For $0 \leq i \leq n$ do:

Process each item $s \in S1_i$ in order performing one of the following:

a) **Predictor**: (top-down prediction closure)

If $[B \rightarrow \alpha \bullet A\beta, j] \in S_i$ and $(A \rightarrow \gamma) \in P$:

add $[A \rightarrow \bullet\gamma, i]$ to S_i

b) **Completer**: (bottom-up recognition)

If $[B \rightarrow \gamma \bullet, j] \in S_i$:

for $[A \rightarrow \alpha \bullet B\beta, k] \in S_j$:

add $[A \rightarrow \alpha B \bullet \beta, k]$ *to* $S1_i$

c) **Scanner**: equivalent to shift in a shift-reduce parser.

If $[A \rightarrow \alpha \bullet a\beta, j] \in S_i$ and $w_i + 1 = a$:

add $[A \rightarrow \alpha a \bullet \beta, j]$ to $S_i + 1$

3 **If** S_{i+1} is empty, **Reject**.

4 **If** $i = n$ and $\mathrm{S}_{n+1} = \{[S' \rightarrow S\$ \bullet, 0]\}$ then **accept**

5.2.2 First approach to an Earley Algorithm for GILs.

We use the graph-structured stack and we represent the stack nodes as pairs of indices and counters (the purpose of the counters is to keep track of the length of the stack for expository purposes). We modify Earley items adding two parameters: Δ, a pointer to an active node in the graph-structured stack, and O, used to record the ordering of the rules affecting the stack; such that $O \leq n$ where n is the length of the input.[2] Therefore Earley items for GILs might be as follows: $[\Delta, \mathbf{O}, A \rightarrow \alpha \bullet A\beta, p_j]$

Algorithm 2 (Earley Recognition for GIGs) .

Let $G = (N, T, I, \#, S, P)$ be a GIG.

Let $w = a_1 a_2 \cdots a_n$ be an input string, $n \geq 0$, and $a_i \in T$ for $1 \leq i \leq n$.

Initialize the graph-structured stack with the node $(\#, 0)$.

Create the Sets S_i:

1 $\mathrm{S}_0 = [(\#, \mathbf{0}), 0, S' \rightarrow \bullet S\$, 0]$

2 **For** $0 \leq i \leq n$ do:

Process each item $s \in S_i$ in order performing one of the following:

a) Predictor

b) Completer

c) Scanner

[2] Actually $O \leq 2n$, if *pop* rules "erase" symbols from the stack. An example of such case would be the following grammar: G_d = ({S},{a,b},{i},{#},{S},{P}) with P: $S \underset{i}{\rightarrow} aS \mid bS \qquad S \underset{\bar{i}}{\rightarrow} S \mid \epsilon$

3 **If** S_{i+1} is empty, **Reject**.

4 **If** $i = n$ and $\mathrm{S}_{n+1} = \{[(\#, 0), 0, S' \rightarrow S\$\bullet, 0]\}$ then **accept**

The structure of the main loop of Earley's algorithm remains unchanged except for the requirement that the initial item at S_0 (line 1) and the *accepting* item at $S_n + 1$ (line 4) point to the *empty stack* node $(\#, 0)$, and have the corresponding initial order in the derivation.

The operations **predictor**, **scanner**, **completer** used in the **For** loop in **2** are modified as follows, to perform the corresponding operations on the graph-structured stack. As we said, we represent nodes in the graph-structured stack with pairs (δ, C) such that $C \leq n$.

1. Predictor

If $[(\delta_1, C_1), O_1, B \underset{\mu}{\rightarrow} \alpha \bullet A\beta, j] \in S_i$ and $(A \underset{\delta}{\rightarrow} \gamma) \in P$:

1.2 **add every** $[(\delta_2, C_2), O_2, A \underset{\delta}{\rightarrow} \bullet\gamma, i]$ to S_i

such that:

if $\delta \in I$ **then** $\delta_2 = \delta$, $O_2 = O_1 + 1$ and

push($(\delta_2, C_2), (\delta_1, C_1)$ **)** s.t. $C_2 = C_1 + 1$

if $\delta = \bar{i}$ **and** $i = \delta_1$ **then** $O_2 = O_1 + 1$ and

$(\delta_2, C_2) \in$ **pop(**(δ_1, C_1)**)** s.t. $C_2 = C_1 - 1$

if $\delta = \epsilon$ **then** $(\delta_1, C_1) = (\delta_2, C_2)$ and $O_1 = O_2$ **(ϵ move)**

if $\delta = [\delta_1]$ **then** $(\delta_1, C_1) = (\delta_2, C_2)$ and $O_1 = O_2$

2. Scanner

If $[\Delta, O, A \underset{\mu}{\rightarrow} \alpha \bullet a\beta, j] \in S_i$ and $w_i + 1 = a$:

add $[\Delta, O, A \underset{\mu}{\rightarrow} \alpha a \bullet \beta, j]$ to $S_i + 1$

3. Completer A

If $[\Delta_1, O, B \underset{\mu}{\rightarrow} \gamma \bullet, j] \in S_i$:

for $[(\delta_2, C_2), O - 1, A \underset{\delta}{\rightarrow} \alpha \bullet B\beta, k] \in S_j$ where $\delta_2 = i$ if $\mu = \bar{i}$:

add $[\Delta_1, O - 1, A \underset{\delta}{\rightarrow} \alpha B \bullet \beta, k]$ *to* S_i

3. Completer B

If $[\Delta_1, O, B \underset{\mu}{\rightarrow} \gamma \bullet, j] \in S_i$ where $\mu = \epsilon$ or $\mu = [i]$:

for $[(\delta_2, C_2), O, A \underset{\delta}{\rightarrow} \alpha \bullet B\beta, k] \in S_j$ where $\delta_2 = i$ if $\mu = [i]$:

add $[\Delta_1, O, A \underset{\delta}{\rightarrow} \alpha B \bullet \beta, k]$ *to* S_i

The following example shows the trace of the items added to a chart parser implementing the above algorithm. This implementation used Shieber's deduction engine backbone [72]. The example

shows the recognition of the string *aabbccdd* with the grammar G_2 transformed into a trGIG. Each item is represented as follows:

item(LeftSymbol, Predotlist, Postdotlist,

[Stacksymbol, StackCounter, OrderCounter, StackOperation],

initialposition, finalposition)

The stack operation is represented as follows: "il" is a *push* rule of the stack index "i" and "ir" is a *pop* rule.

Example 18 (Trace of *aabbccdd***)** $L(G_2) = \{a^n b^n c^n d^n \mid n \geq 0\}, G_2 = (\{S, B\}, \{a, b, c, d\}, \{i\}, \#, P, S)$, *where* P *is:*

$S \underset{i}{\rightarrow} aSd, \quad S \rightarrow B \quad B \underset{\bar{i}}{\rightarrow} bBc \quad B \underset{\bar{i}}{\rightarrow} bc$

```
1 Adding to chart: <10898> item(<start>,[],[S],[#,0,0,e],0,0)
2 Adding to chart: <27413> item(S,[],[B],[#,0,0,e],0,0)
3 Adding to chart: <23470> item(S,[],[a,S,d],[i,1,1,il],0,0)
Adding to stack: <22605> item((i,1),(#,0))
4 Adding to chart: <10771> item(S,[a],[S,d],[i,1,1,il],0,1)
5.Adding to chart: <27540> item(S,[],[B],[i,1,1,e],1,1)
6.Adding to chart: <23599> item(S,[],[a,S,d],[i,2,2,il],1,1)
Adding to stack: <22732> item((i,2),(i,1))
7 Adding to chart: <23598> item(B,[],[b,B,c],[#,0,2,ir],1,1)
8 Adding to chart: <23598> item(B,[],[b,c],[#,0,2,ir],1,1)
9 Adding to chart: <10640> item(S,[a],[S,d],[i,2,2,il],1,2)
10.Adding to chart: <27671> item(S,[],[B],[i,2,2,e],2,2)
11.Adding to chart: <23724> item(S,[],[a,S,d],[i,3,3,il],2,2)
Adding to stack: <22863> item((i,3),(i,2)])
12 Adding to chart: <23725> item(B,[],[b,B,c],[i,1,3,ir],2,2)
13.Adding to chart: <23725> item(B,[],[b,c],[i,1,3,ir],2,2)
14. Adding to chart: <27798> item(B,[b],[B,c],[i,1,3,ir],2,3)
15. Adding to chart: <23851> item(B,[b],[c],[i,1,3,ir],2,3)
16  Adding to chart: <23852> item(B,[],[b,B,c],[#,0,4,ir],3,3)
17. Adding to chart: <23852> item(B,[],[b,c],[#,0,4,ir],3,3)
18. Adding to chart: <26897> item(B,[b],[B,c],[#,0,4,ir],3,4)
```

```
19  Adding to chart: <22956> item(B,[b],[c],[#,0,4,ir],3,4)
20. Adding to chart: <27783> item(B,[c,b],[],[#,0,4,ir],3,5)
21. Adding to chart: <23085> item(B,[B,b],[c],[#,0,3,ir],2,5)
22. Adding to chart: <27654> item(B,[c,B,b],[],[#,0,3,ir],2,6)
23. Adding to chart: <10625> item(S,[B],[],[#,0,2,e],2,6)
24. Adding to chart: <23215> item(S,[S,a],[d],[#,0,2,il],1,6)
25. Adding to chart: <10754> item(S,[d,S,a],[],[#,0,2,il],1,7)
26. Adding to chart: <23342> item(S,[S,a],[d],[#,0,1,il],0,7)
27. Adding to chart: <10883> item(S,[d,S,a],[],[#,0,1,il],0,8)
28. Adding to chart: <10982> item(<start>,[S],[],[#,0,0,e],0,8)
yes
```

The following example shows why some bookkeeping of the order of the derivation is required in Earley items. This bookkeeping is performed here by the order parameter. Consider the language $L_{ww} = \{ww \,|w \in \{a,b\}^*\}$, the corresponding productions of the grammar G_{ww} repeated below and the string *aaba*

1. $S \underset{i}{\rightarrow} aS$ 2. $S \underset{j}{\rightarrow} bS$ 3. $S \rightarrow R$ 4. $R \underset{\bar{i}}{\rightarrow} Ra \mid a$ 5. $R \underset{\bar{j}}{\rightarrow} Rb \mid b$

The following derivation is not possible (in particular step **4**). The pairs in parenthesis represent the nodes of the graph-structured stack:

(#,0) S $\overset{1}{\underset{i}{\Rightarrow}}$ (x,1) aS $\overset{2}{\underset{i}{\Rightarrow}}$ (i,2) aaS $\Rightarrow$(i,2) aaR$\overset{3}{\underset{\bar{i}}{\Rightarrow}}$ (i,1), aaRa $\overset{4}{\underset{\bar{j}}{\Rightarrow}}$ (?,?) aaba

However, the following sequences can be generated using Earley's Algorithm if no ordering constraint is enforced at the Completer Operation. In the following example the square brackets represent substrings to be recognized.

(#,0) S $\overset{1}{\underset{i}{\Rightarrow}}$ (i,1) aS $\overset{2}{\underset{i}{\Rightarrow}}$ (i,2) aaS $\overset{3}{\underset{j}{\Rightarrow}}$ (j ,3) aabS $\Rightarrow$(j ,3) aabR$\overset{4}{\underset{\bar{j}}{\Rightarrow}}$(i,2)aabR[b]$\overset{5}{\underset{\bar{i}}{\Rightarrow}}$

(i,1)aabR[a][b]$\overset{6}{\underset{\bar{i}}{\Rightarrow}}$(#,0)aaba[a][b]

After recognized this substring, the completer operation may *jump up* two steps and complete the "R" introduced after step 3, instead of the one introduced at step 5. In other words, the following complete operation would be performed :

Given $[(\#,0),6,R \underset{\bar{j}}{\rightarrow} a\bullet,3] \in S_4$ **and** $[(j,3),3,S \rightarrow \bullet R,3] \in S_3$:

add $[(\#,0),3,S \rightarrow R\bullet,3]$ *to* S_4

Then the following are sequentially added to S_4:

a) $[(\#,0),2,S \underset{j}{\rightarrow} bS\bullet,2]$,

b) $[(\#,0),1,S \underset{i}{\rightarrow} aS\bullet,1]$,

c) $[(\#,0),0,S \underset{i}{\rightarrow} aS\bullet,0]$

5.2.3 Refining the Earley Algorithm for GIGs

The Algorithm presented in the preceding section does not specify all the necessary constraints, at some predict steps, and at completer cases. It cannot handle cases of what we call *predict-complete* ambiguity, *completed-stack* ambiguity and *stack-position* ambiguity.

The *predict-complete* ambiguity

In this case, the complete operation of the previous algorithm does not specify the correct constraints. The exact prediction path that led to the current configuration has to be transversed back in the complete steps.

In other words, complete steps have to check that there is a corresponding *pop*-move item that has been completed for every item produced at a reduced *push* move. This is depicted in the following figure 5.2.

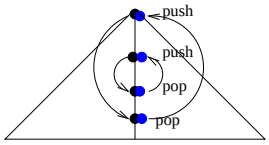


Figure 5.2: Push and Pop moves on the same spine

The following grammar produces alternative derivations which illustrate this issue.

Example 19 $G_{amb4} = (\{S,R\},\{a,b,c,d\},\{i,j\},S,\#,P)$ *where* P *is:*

1. $S \underset{i}{\rightarrow} aSd$ 2. $S \underset{j}{\rightarrow} aSc$ 3. $S \rightarrow R$ 4. $R \underset{\bar{i}}{\rightarrow} bRc$
5. $R \underset{\bar{i}}{\rightarrow} bc$ 6. $R \underset{\bar{j}}{\rightarrow} bRd$ 7. $R \underset{\bar{j}}{\rightarrow} bd$

Grammar G_{amb4} produces the following derivation:

$S \Rightarrow i\#aSd \Rightarrow ji\#aaScd \Rightarrow ji\#aaRcd \Rightarrow i\#aabRdcd \Rightarrow \#aabbcdcd$

But it does not generate the string *aabbcddc* nor *aabbdccd*. The ambiguity arises in the following configuration $S \overset{*}{\Rightarrow} aabR$ which would lead to the following items: $[(\#,2),4,R \rightarrow \bullet b\ c,2]$ and $[(\#,2),4,R \rightarrow \bullet b\ d,2]$. At this point, there is no way to control the correct derivation given no information on the *pop* moves has been stored.

A solution is to extend the graph-structured stack to include $\bar{I}$ nodes. Whenever a *push* item is being completed it has to be matched against the complement node on the stack. The existence of a corresponding $\bar{I}$ node stored in the stack has to be checked whenever an item with a reduced push node is completed. The following figure 5.11 depicts a graph that includes $\bar{I}$ nodes (*pop* nodes). It also shows some additional edges (the dashed edges) that depict the corresponding active nodes in the I domain. We will discuss this issue again in the next subsection.

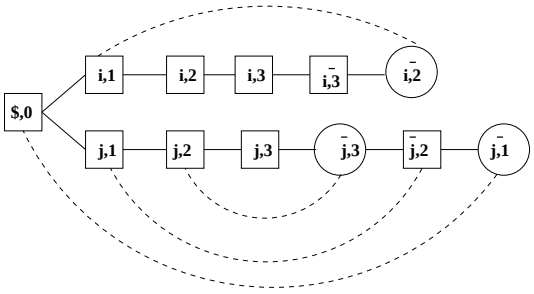


Figure 5.3: A graph structured stack with i and $\bar{i}$ nodes

***Completed stack* ambiguity**

The graph-structured stack we presented so far is built at prediction steps and no additional computation is performed on the graph at completion steps. Therefore there is no way to represent the difference between a configuration like the one depicted in figure 5.2 and the one we present in figure 5.4.

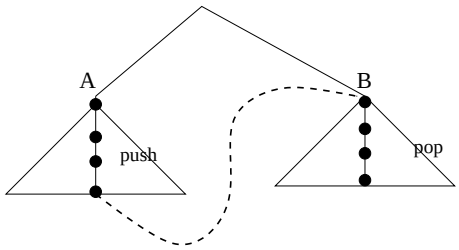


Figure 5.4: Push and Pop moves on different spines

Such a difference is relevant for a grammar like G_{amb5}

Example 20 $G_{amb5} = (\{S, A, R, B\}, \{a, b, c, d, e\}, \{i, j\}, S, \#, P)$ *where* P *is:*

$S \rightarrow AB \quad A \rightarrow R \quad A \underset{i}{\rightarrow} aAb \mid ab \quad R \underset{j}{\rightarrow} aRd \mid ad \quad B \underset{\bar{i}}{\rightarrow} cBd \mid cd \quad B \underset{\bar{j}}{\rightarrow} cBe \mid ce$

Grammar G_{amb5} generates the string *aadbccde* but it does not generate the string *aadbcced*. If we consider the recognition of the string *aaaddbccdde*, it will generate the following stack nodes for the derivation $A \stackrel{*}{\Rightarrow} aaaddb$: $a(i,j,1)a(i,j,2)a(i,j,3)d(\rightarrow j,3)d(\rightarrow j,2)b(\rightarrow i,1)$. The notation $(i,j,1)$ indicates that two nodes $(i,1)$ and $(j,1)$ where created at position a_1, and $(\rightarrow j,3)$ indicates that the node $(j,3)$ was completed at the position d_4. Consequently only the nodes $(j,3),(j,2)$ and $(i,1)$ are completed at the derivation point $S \stackrel{*}{\Rightarrow} aaaddbB$.

The edges depicted in figure 5.5 are all valid at position 3 (i.e accessible from the item $[\Delta,3,R \rightarrow aaa \bullet d,3]$ in S_3).

However they would be still valid at position 6: $[\Delta,3,B \rightarrow \alpha \bullet d,6]$ in S_6

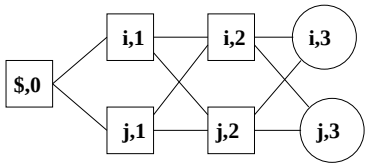


Figure 5.5: A predict time graph

Therefore, it is necessary to distinguish between prediction and complete stages. We need to introduce a new edge, a completion edge. The arrows in figure 5.6 are intended to designate this *completion* edge. They also indicate that there is a different kind of path between nodes. So, the simple lines indicate that there was a potential edge at the prediction step. Only those paths with additional arrows were *confirmed* at a complete step.

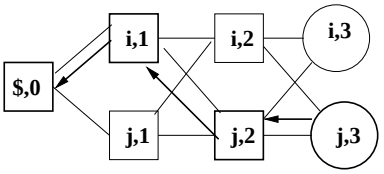


Figure 5.6: A complete steps in the graph

Stack position Ambiguity

Two (or more) different stacks might have the same active nodes available at the same position. However those alternative stacks might correspond to alternative derivations. This issue is illustrated by the two stacks represented in figure 5.7.

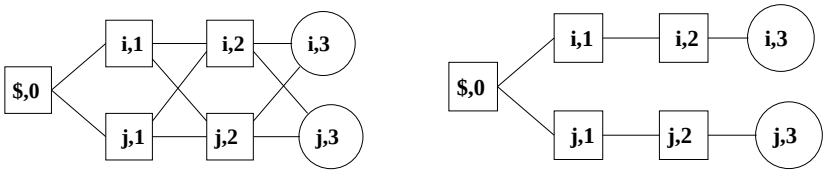


Figure 5.7: Two alternative graph-structured stacks

The alternative graph-structured stacks depicted in figure 5.7 might be generated at different positions, e.g. in the following alternative spines as illustrated in the following figure 5.8.

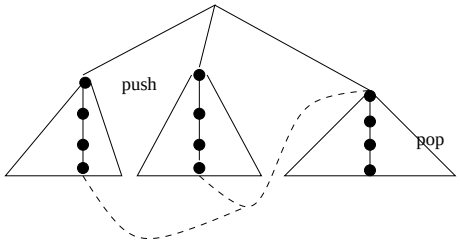


Figure 5.8: Ambiguity on different spines

Notice that although these stacks have the same active nodes, they do differ in the nodes that can be reached from those active nodes. They even might be continued by the same *pop* complement part of the graph. It must be noticed that the active nodes from the graph structures in figure 5.8 are indistinguishable from each other. Some information must be added to make sure that they point to active nodes in different graphs. The information that will enable to distinguish the corresponding nodes in the graph will be the position in which the *push* nodes are generated. This information replaces the order parameter that was used in the preliminary version of the algorithm. In this case the graph structured stack will be more like the one that is proposed in Tomita's algorithm, in the sense that there is not an unambiguous length of the stack at each node.

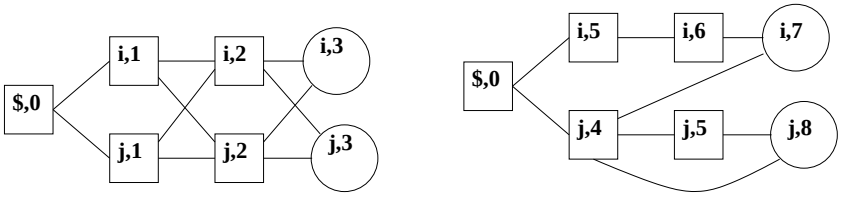


Figure 5.9: Two alternative graph-structured stacks distinguished by position

5.2.4 Earley Algorithm for GIGs: final version

Revised Operations on the stack

We will use the following notation. A node Δ_+ designates a node corresponding to a *push* move on the stack. It is a pair (δ, p) such that δ is an index and $p > 0$ ($p's$ are integers). We say $\Delta_+ \in I$. A node Δ_- designates a node corresponding to a *pop* move in the stack. It is a pair (δ, p) such that $\delta \in \bar{I}, p < 0$. We say $\Delta- \in \bar{I}$. We will use also the complement notation to designate the corresponding complements. Therefore $\bar{\Delta}_+$ will designate a node corresponding to a *pop* move which "removes" Δ_+. On the other hand $\bar{\Delta}_-$ will designate a *push* node which has been removed by Δ_-. We say $\Delta_i > \Delta_j$ if $\Delta_i = (\delta_i, p_i)$, $\Delta_j = (\delta_j, p_j)$, and $p_i > p_j$.

The following operations are possible on the graph-structured stack. The modifications are necessary due to the fact that the graph-structured stack contains both nodes in I and $\bar{I}$. These operations enable the computation of the Dyck language over the graph-structured stack.

Push(*newnode,oldactivnode,initialnode*). Creates a *newnode* if it is not in the graph and creates an edge from *newnode* to *oldnode* if necessary. It also creates an *i-edge* from *initialnode* to *newnode*. These edges will be represented then by triples: (*newnode*, *oldnode*, *initialnode*). We will discuss it again below.

PushC(*currentnode, precednode*). Creates a **complete** edge from *currentnode* to *precednode* if necessary.

Path(*currentnode*). This operation is the equivalent of the previous **Pop** function with a stack that keeps record of the nodes in $\bar{I}$. It finds the preceding unreduced push node after one or more reductions have been performed. It retrieves all the nodes in I that are connected by an edge to the push node that *currentnode* reduces. A node can connect to any node at a preceding position. Therefore the number of edges connecting a node to a preceding one is bound by the position of the current node, i.e. for a node at position p_i there might be p_{i-1} edges to other nodes. Note that the position is determined by the corresponding position of the terminal introduced by the push production.

Therefore, the path function obtains the nodes in I which can be active, after a *reduce* operation has been done, as is specified in algorithm 5 and depicted in figure 5.10.

We will use a slightly more complicated version that takes into account completed nodes and the initial nodes introduced by the **Push** operation.

Algorithm 3 . **Path**$(\Delta - _1)$:

where $\Delta - _1,\ \Delta - _3 \in \bar{I}$, *and* $\bar{\Delta} - _1,\ \Delta + _2 \in I$

1 *valueset* $= \{\}$

2 *For edge*$((\bar{\Delta} - 1, \Delta + _2)$

3 *add* $\Delta + _2$ *to valueset*

4 *For edge*$(\bar{\Delta} - _1, \Delta - _3)$

5 *add Path2*$(\Delta - _3)$ *to valueset*

6 *return valueset*

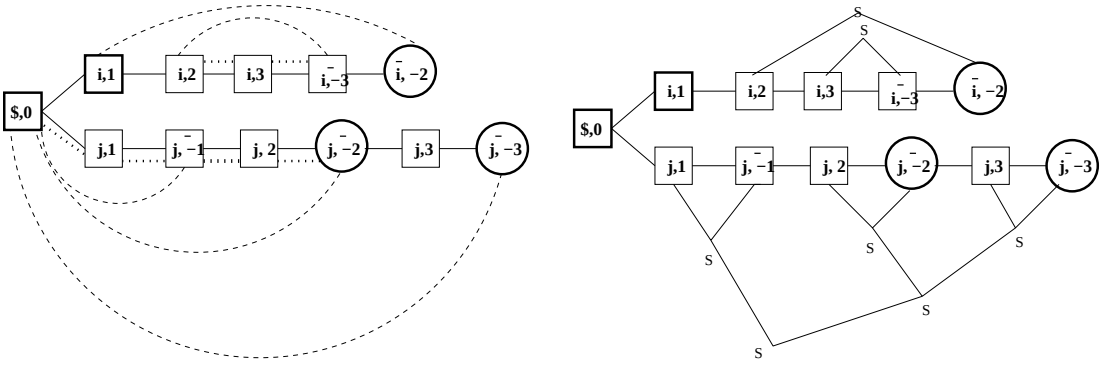


Figure 5.10: Paths and reductions in a graph structured stack with i and $\bar{i}$ nodes

The following operation operates on the reverse direction of the **Path** function. Both **Path** and **Path2** perform Dyck language reductions on the graph structured stack. While **Path** performs them at *predict* time, **Path2** performs them at *complete* time. **Path** obtains the closest non reduced nodes in I. **Path2** returns a set of nodes in $\bar{I}$ accessible from another node in $\bar{I}$. The returned set of nodes represent points in the derivation at which to continue performing reductions at complete time. The portion of the graph that is already reduced corresponds to completed nodeswhich already matched their respective complements. Completed nodes are designated by a particular kind of edge, named **edgeC**.

Algorithm 4 . *Path2*($\Delta - _1$):

where $\Delta - _1,\ \Delta - _3 \in \bar{I},\ \bar{\Delta} - _1 \in I$) *and :* $\Delta_2 \in \bar{I} \cup \{\#\}$

1 *valueset* $= \{\}$

2 *For edgeC*($(\bar{\Delta} - 1, \Delta_2)$

3 *add* Δ_2 *to valueset*

4 *For edgeC*($\bar{\Delta} - _1, \Delta - _3$)

5 *add Path2*($\Delta - _3$) *to valueset*

6 *return valueset*

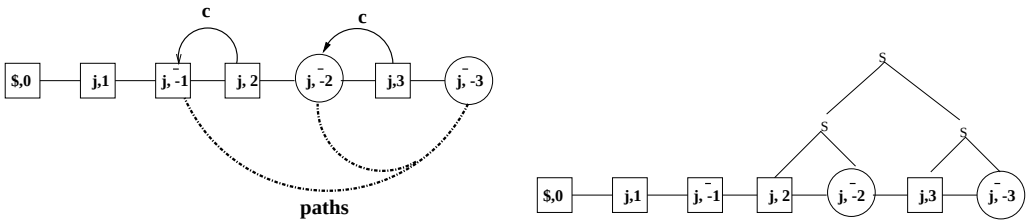


Figure 5.11: A graph structured stack with i and $\bar{i}$ nodes

New Earley items for GIGs

Earley items have to be modified to represent the portion of the non reduced graph-stack corresponding to a recognized constituent. Two parameters represent the portion of the stack: the current active node Δ_{Ac} and the initial node Δ_{In}.

$[\Delta_{Ac}, \Delta_{In},\ A \rightarrow \alpha \bullet a\beta, j] \in S_i$

Nodes are represented by pairs of elements in $I \cup \bar{I} \cup \{\#\}$, and positive, negative integers and zero respectively. We say a node Δ_+ is in I if it designates a pair (δ, p) such that $\delta \in I, p > 0$. We say a node Δ_- is in $\bar{I}$ if it designates a pair $(delta, p)$ such that $\delta \in \bar{I}, p < 0$. The empty stack is then represented by the node $(\#, 0)$. The vocabulary corresponding to the set of indices (I), their complements ($\bar{I}$) and the empty stack symbol ($\#$), will be denoted by DI (i.e. $DI = I \cup \bar{I} \cup \{\#\}$).

We need also to represent those constituents that did not generate a node of their own (a *push* or *pop* derivation), but inherit the last generated node. We refer to these as *transmitted* nodes. We use a substitution mapping cl from DI into $DI \cup \{c\}^*$, defined as follows:

$cl(\delta) = c\delta$ for any $\delta \in DI$

Δ_{cl} represents a node $(cl(\delta), p)$ such that $\delta \in DI$ and p is an integer. A node Δ_{cl} (or $cl(\Delta)$) is a pointer $(cl(\delta), p)$ to the node (δ, p) (Δ) in the stack.

As we mentioned when we introduced the *Push* operation **Push**(*newnode,oldactivnode,initialnode*), we will make use of an additional type of edge. This kind of edge relates Δ_{In} nodes, therefore we will call them *I-edges*. These edges will be introduced in the predict operation, by the following **Push** function. The interpretation of these edge is that there is a path connecting the two Δ_{In} nodes. Notice that an I-edge can be identical to a *push* or *pop* edge. I-edges are depicted in figure 5.12.

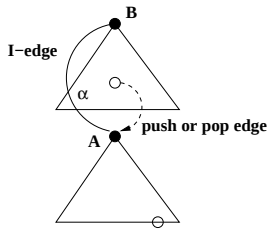


Figure 5.12: I-edges

Now that we have introduced $I-edges$ we present a second version of the path computation that takes into account $I-edges$ (here Δ_R) as boundary points to define a path. It also takes into account the difference between nodes corresponding to completed constituents (lines 7-12) and nodes corresponding to incomplete constituents (lines 1-6).

Algorithm 5 .

StorePath$(\Delta-_1, \Delta_R)$:

where $\Delta-_1$, $\Delta-_3 \in \bar{I}$, *and* $\bar{\Delta}-_1$, $\Delta+_2 \in I$, Δ_R, $\Delta_{R2} \in I \cup \bar{I}$, I *the set of indices.*

1 **If** $\bar{\Delta}-_1 = \Delta_R$:

2 **For** *edge*$(\bar{\Delta}-_1, \Delta+_2, \Delta_{R2})$

3 *add Path*$(\Delta-_1, \Delta+_2, \Delta_{R2})$ *to Pathset*

4 **For** *edge*$(\bar{\Delta}-_1, \Delta-_3, \Delta-_3)$ *and*

5 **For** *Path*$(\Delta-_3, \Delta+_2, \Delta_R 2)$

6 *add Path*$(\Delta-_1, \Delta+_2, \Delta_{R2})$ *to Pathset*

7 **else if** $\bar{\Delta}-_1 > \Delta_R$

8 **For** *edgeC*$(\bar{\Delta}-_1, \Delta+_2)$ *such that* $\Delta+_2 \geq \Delta_R$

9 *add Path*$(\Delta-_1, \Delta+_2, \Delta_R)$ *to Pathset*

10 **For** *edgeC* $(\bar{\Delta}-_1, \Delta-_3)$

11 **For** *Path* $(\Delta-_3, \Delta+_2, \Delta_{R2})$

12 *add Path*$(\Delta-_1 \Delta-_3, \Delta_R)$ *to Pathset*

Algorithm 6 (Earley Recognition for GIGs) .

Let $G = (N, T, I, \#, S, P)$ be a GIG.

Let $w = a_1 a_2 \cdots a_n$ be an input string, $n \geq 0$, and $a_i \in T$ for $1 \leq i \leq n$.

Initialize the graph-structured stack with the node $(\#, 0)$.

Create the Sets S_i:

1 $S_0 = [(\#, \mathbf{0}), (\#, \mathbf{0}),\ S' \rightarrow \bullet S\$, 0]$

2 For $0 \leq i \leq n$ do:

Process each item $s \in S_i$ in order

performing one of the following:

a) Predictor

b) Completer

c) Scanner

3 **If** S_{i+1} is empty, **Reject**.

4 **If** $i = n$ and $S_{n+1} = \{[(\#, \mathbf{0}), (\#, \mathbf{0}), S' \rightarrow S\$\bullet, 0]\}$ then **accept**

The structure of the main loop of Earley's algorithm remains unchanged except for the requirement that the initial item at S_0 (line 1) and the *accepting* item at $S_n + 1$ (line 4) have an *empty stack* Active and Initial node: $(\#, 0)$.

The operations **predictor**, **scanner**, **completer** used in the **For** loop in **2** are modified as follows to perform the corresponding operations on the graph-structured stack: as we said, we represent nodes in the graph-structured stack with pairs $(\delta_{\pm}, p)$ such that $-n \leq p \leq n$, $\delta \in DI$ and the absolute value of p ($abs(p)$) refers to the position in $w = a_1 ... a_n$ in which the index δ_+ in I was introduced. $Ac(\delta_1, p_1)$ designates an item active node (Δ_{Ac}) and $I(\delta_0, p_0)$ designates an item initial node.

Prediction and Scanner Operations

Prediction Operations will be divided in different subcases. These subcases are constrained by the type of production applied (*push*, *pop* or *e-move*). In the pop case there are two subcases (1.2.a): the active node in the triggering item is in I or (1.2.b) where the active node is in $\bar{I}$. In the second case, the function *path* is used, to retrieve the active nodes in I. This is also required for productions that check the top of the stack: $(A \underset{[\delta]}{\rightarrow} \gamma)$ when the triggering item Active node is in $\bar{I}$.

1.1 Predictor Push

If $[Ac(\delta_1, p_1), I(\delta_0, p_0),\ B \rightarrow \alpha \bullet A\beta, j] \in S_i, \quad (A \underset{\delta_2}{\rightarrow} \gamma) \in P \quad$ and $\quad \delta_2 \in I$:

add every $[Ac(\delta_2, p_2), I(\delta_2, p_2),\ A \rightarrow \bullet\gamma, i]$ to S_i

such that: $p_2 = i + 1, \quad$ and

push($(\delta_2, p_2), (\delta_1, p_1), (\delta_0, p_0)$ **)**

1.2.a Predictor Pop: Active Node is in I

If $[Ac(\delta_1, p_1), I(\delta_0, p_0),\ B \rightarrow \alpha \bullet A\beta, j] \in S_i, \quad (A \underset{\delta_2}{\rightarrow} \gamma) \in P \quad$ and $\quad \delta_2 \in \bar{I} = \bar{\delta_1}$:

add every $[Ac(\delta_2, p_2), I(\delta_2, p_2),\ A \rightarrow \bullet\gamma, i]$ to S_i

such that: $\delta_2 = \bar{\delta}_1$ and $p_2 = -p_1 \quad$ and

push($(\delta_2, p_2), (\delta_1, p_1), (\delta_0, p_0)$ **)**

StorePath$((\delta_2, p_2), (\delta_0, p_0))$

1.2.b Predictor Pop: Active Node is in $\bar{I}$

If $[Ac(\delta_1, p_1), I(\delta_0, p_0),\ B \rightarrow \alpha \bullet A\beta, j] \in S_i$ and $(A \underset{\delta_2}{\rightarrow} \gamma) \in P$ and $\delta_1,\ \delta_2 \in \bar{I}$:

add every $[Ac(\delta_2, p_2), I(\delta_2, p_2),\ A \rightarrow \bullet\gamma, i]$ to S_i

such that: $(\delta_3, p_3) \in$ **path(**$(\delta_1, p_1), (\delta_3, p_3), \delta_r, p_r$**)** and $\delta_2 = \bar{\delta}_3$ and $p_3 = -p_2$ and

push($(\delta_2, p_2), (\delta_1, p_1), (\delta_0, p_0)$ **)**

StorePath$((\delta_2, p_2), (\delta_r, p_r))$

1.3 Predictor e-move

If $[Ac(\delta_1, p_1), I(\delta_0, p_0),\ B \rightarrow \alpha \bullet A\beta, j] \in S_i \quad$ and $(A \underset{\delta}{\rightarrow} \gamma) \in P$ such that

$\delta = \epsilon$ or $\delta = [\delta_1]$ or $(\delta_2, p_2) \in$ **path**(δ_1, p_1), $\delta_1 \in \bar{I}$ and $\delta_2 = \delta$

add every $[Ac(\delta_1, p_1), I(cl(\delta_0), p_0),\ A \rightarrow \bullet\gamma, i]$ to S_i

2. Scanner

If $[\Delta_{Ac}, \Delta_{In},\ A \rightarrow \alpha \bullet a\beta, j] \in S_i$ and $w_i + 1 = a$:

add $[\Delta_A, \Delta_{In},\ A \rightarrow \alpha a \bullet \beta, j]$ to $S_i + 1$

Completer Operations

The different subcases of Completer operations are motivated by the contents of the Active Node Δ_{Ac} and the Initial Node Δ_{In} in the triggering completed item (item in S_i) and also by the contents of the Initial Node Δ_{I0} of the combination target item (item in S_j). All the completer operations

check properties of the combination target item in S_j against edges that relate its the initial node Δ_{I0} to the initial node of the completed triggering item Δ_{In}. These edges are of the form $(\Delta_{In}, \_, \Delta_{I0})$ (where the omitted element is the active node Δ_{Ac} of the combination target). We use the symbol '_' hereafter to indicate that the information stored in the corresponding field is not required in the computation. Therefore if the information is stored in a three dimension table, it means that only two dimensions are being used.

The idea is to be able to compute whether a tree has a valid stack (i.e. a stack that is in the Dyck[3] language of the indices and its complements ($I \cup \bar{I}$)). For a subtree to be in the Dyck language over DI is as follows: the initial node has to contain the node for the first index introduced in the derivation, and the active node has to contain its inverse. This is the equivalent of a production $S \rightarrow aS\bar{a}$ for a Dyck language over the alphabet $\{a, \bar{a}\}$. If a subtree is in the Dyck language over DI, a reduction can be performed. However, we need to check also whether the stack associated with each subtree is a proper substring of the Dyck language over $I \cup \bar{I}$.

The first completer subcase involves a combination item in S_j that had been generated by an *e-production*, (derived by Predictor 1.3 e-move) and therefore its initial node is a *transmitted node*, ie. $\Delta_{I0} = (cl(\delta), p)$. As a result of the combination, the initial node Δ_{In} is inherited by the result item. This is the reverse of predictor 1.3. At this step, the *transmitted* nodes are *consumed*.

3.a Completer A. Combination with an e-move item: $\delta_0 \in cl(I \cup \bar{I})$

If $[Ac(\delta_2, p_2), I(\delta_3, p_3),\ B \rightarrow \gamma \bullet, j] \in S_i$:

for $edge((\delta_3, p_3), (\_, \_,), (\delta_0, p_0))$ and

for $[Ac1(\delta_0, p_0)\ I(\delta_0, p_0),\ A \rightarrow \alpha \bullet B\beta, k] \in S_j$ such that:

$\delta_0 \in cl(I \cup \bar{I})$

add $[Ac(\delta_2, p_2), I(\delta_3, p_3),\ A \rightarrow \alpha B \bullet \beta, k]$ *to* S_i

Subcase 3.b combines a completed triggering item generated by a *push production* (using the prediction operation 1.1); therefore, the initial node Δ_{In} is in I. The active node Δ_{Ac} is also in I or it is the empty stack (i.e $\Delta_{Ac} = (\delta_2, p_2)$ such that $\delta \in I \cup \{\#\}$ and $0 \leq p_2 \leq n$). The constraint that there is an I-edge between Δ_{In} and Δ_{I0} is checked. No further constraints are checked. The result item preserves the active node from the triggering item and the initial node from the combination target item. A completion edge is introduced recording the fact that this *push* node Δ_{In} has been completed.

[3]Use the standard terminology concerning Dyck languages as found for instance in [39].

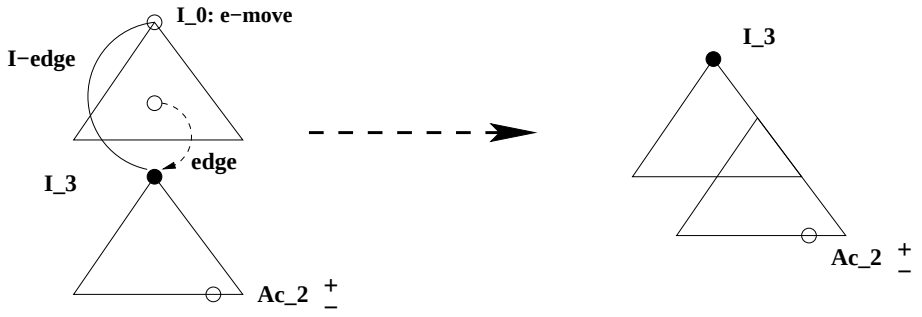


Figure 5.13: Completer A

3.b Completer B: Push with Active Node in $I \cup \{\#\}$, $(p_2 > -1)$

If $[Ac(\delta_2, p_2), I(\delta_3, p_3),\ B \rightarrow \gamma\bullet, j] \in S_i$ such that

$\delta_3 \in I$, $\delta_2 \in I \cup \{\#\}$, and $(p_2 > -1)$:

for $edge((\delta_3, p_3), (\_, \_), \delta_0, p_0))$ and

for $[\_\,, I(\delta_0, p_0),\ A \rightarrow \alpha\bullet B\beta, k] \in S_j$ such that: $\delta_0 \in I \cup \bar{I}$,

add $[Ac(\delta_2, p_2), I(\delta_0, p_0),\ A \rightarrow \alpha B\bullet\beta, k]$ *to* S_i and

pushC($(\delta_3, p_3), (\delta_0, p_0)$)

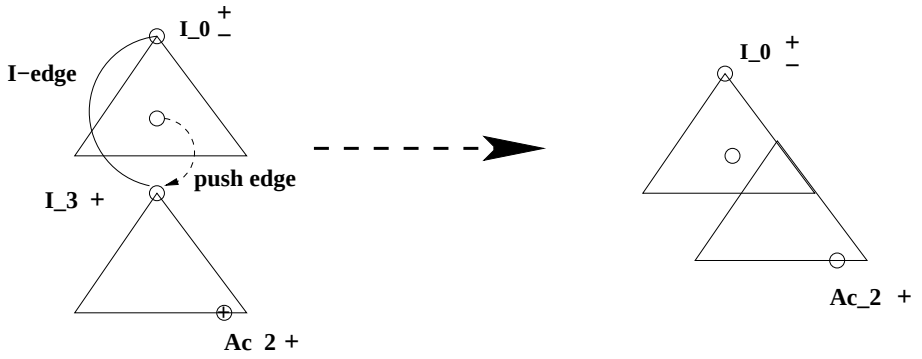


Figure 5.14: Completer B

Subcase 3.c also combines a triggering item generated by a *push production*, $\Delta_{In} \in I$. In this case the active node Δ_{Ac} is in $\bar{I}$ (i.e $\Delta_{Ac} = (\delta_2, p_2)$ such that $\delta_2 \in \bar{I}$ and $-n \leq p_2 < 0$). The existence of an edge between Δ_{In} and Δ_{I0} is checked as in the previous cases. There are two additional constraints. The first is $-p_2 < p_I$. In other words the active node Δ_{Ac} is a *closing parenthesis* of the Dyck language that has not yet exceeded its *opening parenthesis*. It should be noted that the string position in which an index in I is introduced constrains the possible orderings of the control

language elements. For instance, two nodes $\Delta_{In} = (i, 2)$ and $\Delta_{Ac} = (\bar{j} - 3)$, do not constitute a valid stack configuration for a subtree, because Δ_{Ac} should have met the corresponding matching Dyck element and a reduction should have been performed.

The substring of the Dyck language that corresponds to the completed subtree is indicated by the square brackets. This constraint then expresses the fact that $(_{\bar{Ac}}$ is outside the square brackets.

$(_{\bar{Ac}}\cdots[(_{I0}\cdots(_{I}\ \ldots\ldots)_{Ac}]$

The second constraint requires that Δ_{In} has to have a matching *closing parenthesis*, so the node $\bar{\Delta}_{In}$ has to have been completed. Following the preceding example $)_{\bar{I}}$ has to be inside the square brackets:

$(_{\bar{Ac}}\cdots[(_{I0}\cdots(_{In}\ \cdots\ )_{\bar{In}}\cdots)_{Ac}]$

The result item preserves the active node from the triggering item and the initial node from the combination item.

3.c Completer C: Push with Active Node in $\bar{I}$, $(p_2 < 0)$, and $-p_2 < p_3$

If $[Ac(\delta_2, p_2), I(\delta_3, p_3),\ B \rightarrow \gamma\bullet, j] \in S_i$ such that $\delta_3 \in I$, $\delta_2 \in \bar{I}$, $(p_2 < 0)$ and $-p_2 < p_3$ there is a completed edge$((\bar{\delta}_3, -p_3), (_))$

for $edge((\delta_3, p_3), (\_, \_), \delta_0, p_0))$ and

for $[\_, I(\delta_0, p_0),\ A \rightarrow \alpha\bullet B\beta, k] \in S_j$ such that: $\delta_0 \in I \cup \bar{I}$,

add $[Ac(\delta_2, p_2), I(\delta_0, p_0),\ A \rightarrow \alpha B\bullet\beta, k]$ *to* S_i

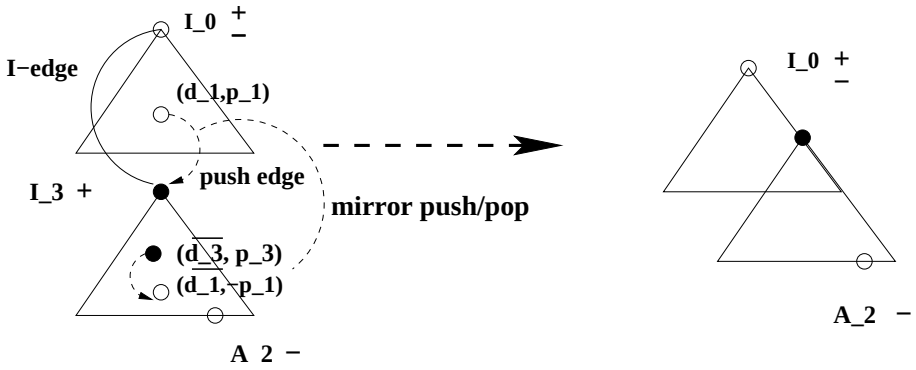


Figure 5.15: Completer C

Subcase 3.d is the last case that combines a completed item generated by a *push production*, $\Delta_{In} \in I$. In this case the active node Δ_{Ac} is also in $\bar{I}$ (like in subcase 3.c). But there is a different constraint: $\Delta_{Ac} = \bar{\Delta}_{In}$, i.e. Δ_{Ac} is the matching parenthesis of Δ_{In}. In other words:

$(\delta_2, p_2) = (\bar{\delta}_3, -p_3)$ such that $\delta_I \in I$ and $0 < p_3 \leq n$. Therefore the whole stack of the completed triggering item can be reduced and the active node of the result is updated to a node that precedes Δ_{In} in the stack, which is Δ_{Ac1}, i.e. (δ_1, p_1).

3.d Completer D: Push with Active Node in $\bar{I}$,($p_2 < 0$), and $\Delta_{Ac} = \bar{\Delta}_I$

If $[Ac(\delta_2, p_2), I(\delta_3, p_3),\ B \rightarrow \gamma \bullet, j] \in S_i$ such that $\delta_3 \in I$, $\delta_2 \in \bar{I}$, $(p_2 < 0)$,
$-p_2 = p_3$ and $\delta_2 = \bar{\delta}_3$
for $edge((\delta_3, p_3), (\delta_1, p_1), (\delta_0, p_0))$ and
for $[Ac(\delta_1, p_1), I(\delta_0, p_0),\ A \rightarrow \alpha \bullet B\beta, k] \in S_j$ such that: $\delta_0 \in I \cup \bar{I}$,
add $[Ac(\delta_1, p_1), I(\delta_0, p_0),\ A \rightarrow \alpha B \bullet \beta, k]$ *to* S_i

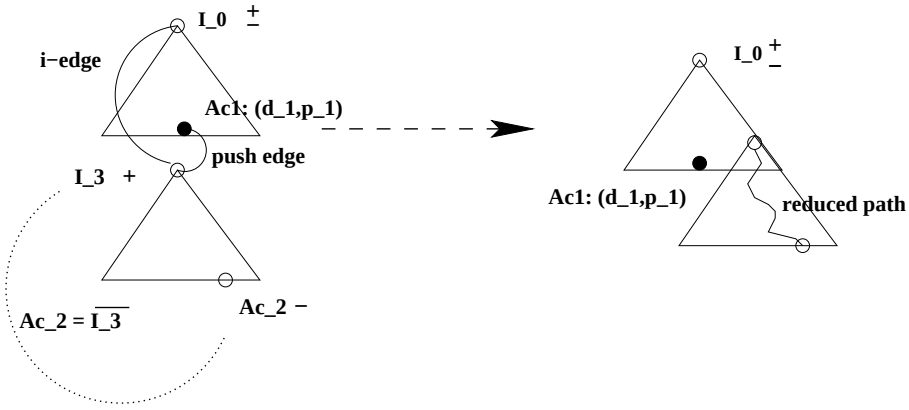


Figure 5.16: Completer D

The following Completer cases involve completed triggering items generated by *pop* productions.

Subcase 3.e corresponds to edges $(\Delta_{In}, \Delta_{Ac1}, \Delta_{I0})$ where $\Delta_{In} = \bar{\Delta}_{Ac1}$. This is the base configuration to perform a Dyck reduction. The Active Node of the result item are updated, product of the reduction, and checked to verify that they satisfy the control properties required by the case 3.d. The alternatives for performing a Dyck reduction on the graph are three:

a) The reduction is performed by the *path2* function: choose $(\delta_5, p_5) \in path2(\delta_2, p_2)$

b) The reduction is performed by a single edge from the complement of the Active Node (Δ_{Ac}) to a matching node of the initial node in the target combination item Δ_{I0}. In other words, there is an edge $((\bar{\delta}_2, -p_2), (\bar{\delta}_0, -p_0, \_))$. This option is similar to the previous one; the only difference is that Δ_{Ac} was not completed yet at this stage (the *path2* function requires completed nodes).

c) The completed item initial node and active node are equal $(\Delta_{In} = \Delta_{Ac})$. A node preceding the active node (Δ_{Ac1}) of the target item $(\delta_1, p_1)(\delta_5, p_5, \_)$, becomes the updated Active node of

the result. The Active node Δ_{Ac} is updated according to the performed reduction.

3.e Completer E. Pop items with Push items and graph reduction

If $[Ac(\delta_2, p_2), I(\delta_3, p_3),\ B \rightarrow \gamma \bullet, j] \in S_i$ such that $\delta_2\delta_3 \in \bar{I}$, $p_3, p_2 < 0$:

Compute **path2** (δ_2, p_2)

for edge$((\delta_3, p_3), (\delta_1, p_1), (\delta_0, p_0))$

where $\delta_3 = \bar{\delta}_1$, $\delta_1 \in I$, $p_3 = -p_1$, $\delta_0 \in I$ and $p_0 > 0$,

for $[Ac(\delta_1, p_1), I(\delta_0, p_0),\ A \rightarrow \alpha \bullet B\beta, k] \in S_j$ such that

If $p_1 > p_0$, and $-p_2 > p_0$ and $(\delta_5, p_5) \in path2(\delta_2, p_2)$, and $(\delta_5, p_5) = (\bar{\delta}_0, -p_0)$[4]

else if $-p_2 > p_0$ and there is an edge $(\bar{\delta_2}, -p_2), (\bar{\delta}_0, -p_0, \_)$, and $(\delta_5, p_5) \geq (\bar{\delta}_0, -p_0)$ [5]

else if $p_2 = p_3$,$\delta_2 = \delta_3$ for edge$(\delta_1, p_1)(\delta_5, p_5), _)$

add $[Ac(\delta_5, p_5), I(\delta_0, p_0),\ A \rightarrow \alpha B \bullet \beta, k]$ *to* S_i

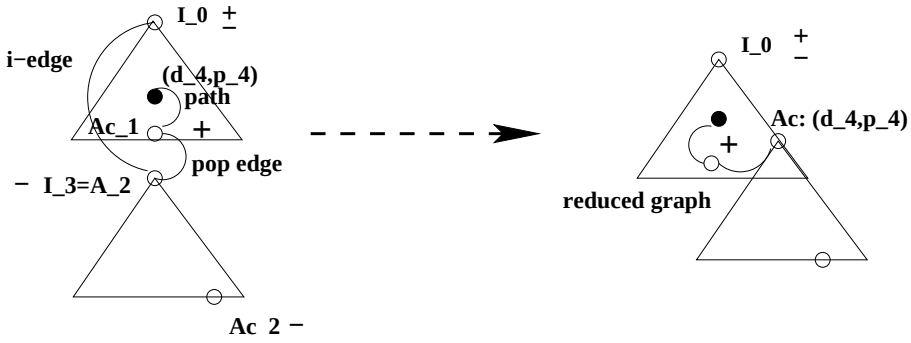


Figure 5.17: Completer E, last case

Subcase 3.f considers the combination of a *pop* item with a *push* item and checks the properties of the Active Node (Δ_{Ac}) to verify that it satisfies the control properties required in 3.c. Either the Active Node is a closing parenthesis, (Δ_{Ac-}), which has not yet exceeded the position of its opening parenthesis, alternatively, the Active Node is positive (Δ_{Ac+}). These two alternatives are allowed by $p_2 > p_3$, in other words $\Delta_{Ac} > \Delta_{In}$. The information that Δ_{In} was completed is stored by the **completed** edge $(\Delta_{In}, \Delta_{In})$.[6]

3.f Completer F. Pop items with Push items: $p_0 > 0, \delta_0 \in I$ **and** $p_2 > p_3$

[4]There is a complete path of $\bar{I}$ elements that reaches the one that complements δ_0, or exceeds it. Part of this constraint might be specified in the Completer Case G.

[5]$\bar{\delta}_2$ was introduced after δ_0 was reduced. This case is relevant for the language L_{genabc}

[6]An edge to itself.

If $[Ac(\delta_2, p_2), I(\delta_3, p_3),\ B \rightarrow \gamma\bullet, j] \in S_i$: such that $p_2 > p_3$

for edge$((\delta_2, p_2), (_, _), (\delta_0, p_0))$

for $[\_, I(\delta_0, p_0),\ A \rightarrow \alpha\bullet B\beta, k] \in S_j$

add $[Ac(\delta_2, p_2), I(\delta_0, p_0),\ A \rightarrow \alpha B\bullet\beta, k]$ *to* S_i

pushC edge $(\delta_3, p_3)(\_)$

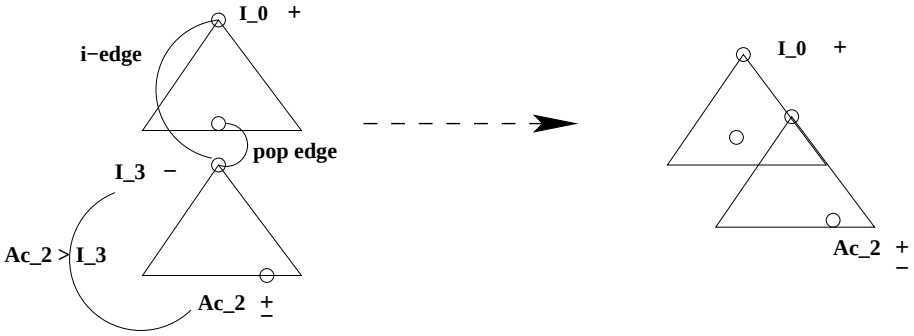


Figure 5.18: Completer F

The following subcases complete the remaining alternatives. Subcase 3.g combines completed items generated by *pop* productions with target items also generated by *pop* productions. Subcase 3.h combine completed items generated by the 1.3 predictor. Therefore the target item has to have the same contents in the Initial and Active node as the completed item.

3.g Completer G Pop items combine with Pop items[7]

If $[Ac(\delta_2, p_2), I(\delta_3, p_3),\ B \rightarrow \gamma\bullet, j] \in S_i$ such that $\delta_3 \in \bar{I}$,

for edge$((\delta_3, p_3), (_, _), (\delta_0, p_0))$, $\delta_0 \in \bar{I}$

for $[\_, I(\delta_0, p_0),\ A \rightarrow \alpha\bullet B\beta, k] \in S_j$

add $[Ac(\delta_2, p_2), I(\delta_0, p_0),\ A \rightarrow \alpha B\bullet\beta, k]$ *to* S_i and

pushC edge $(\delta_3, p_3)(\_)$

3. Completer H e-move

If $[Ac(\delta_2, p_2), I(cl(\delta_3), p_3),\ B \rightarrow \gamma\bullet, j] \in S_i$:

for $[Ac(\delta_2, p_2), (\delta_3, p_3),\ A \rightarrow \alpha\bullet B\beta, k] \in S_j$:

add $[Ac(\delta_2, p_2), (\delta_3, p_3),\ A \rightarrow \alpha B\bullet\beta, k]$ *to* S_i

The following table summarizes the different Completer cases.

[7]Here we could state the constraint $p_2 > p_0$, or compute a graph reduction if the condition is not met.

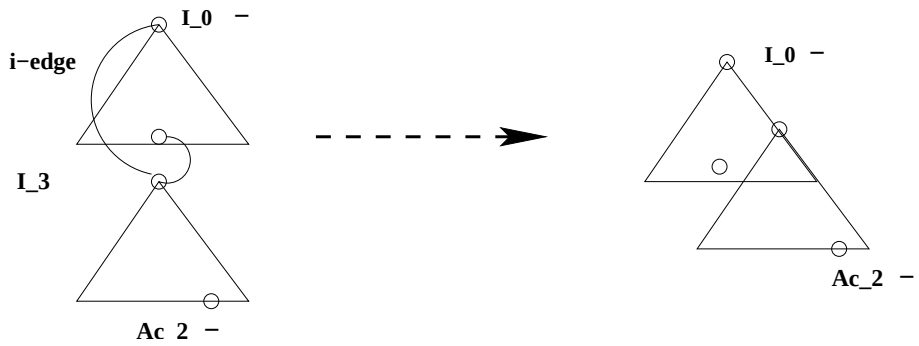


Figure 5.19: Completer G

Summary of Completer Cases

CASE	COMPLETED ITEM	ACTIVE NODE	TARGET ITEM	Specific Actions
A	+/-	+/-	transmitted	$\Delta_{I_0} \leftarrow \Delta_{In}$
B	+	+	+/-	**pushC**$(\Delta_{In}, \Delta_{I0})$
C	+	-	+/-	**check for** $\bar{\Delta}_{In}$
D	+	- $(\Delta_A = \bar{\Delta}_I)$	+/-	**reduce:** $\Delta_A \leftarrow \Delta_{In} - 1$
E	-	-	+/- ,Δ_{Ac}+	**reduce with path2**
F	-	+/- $(\Delta_A > \Delta_{In})$	+	**pushC**$(\Delta_{In}, -)$
G	-	+/ -	-	**pushC**$(\Delta_{In}, -)$
H	transmitted	+/-	+/-, transmitted	

5.3 Complexity Analysis

5.3.1 Overview

The algorithm presented above has one sequential iteration (a *for* loop). It has a maximum of n iterations where n is the length of the input. Each item in S_i has the form:

$[Ac(\delta_1, p_1), I(\delta_2, p_2), A \rightarrow \alpha \bullet B\beta, j]$ where: $0 \leq j, p_1, p_2, \leq i$

Thus there might be at most $O(i^3)$ items in each set S_i.

Scanner, Predictor e-move and Predictor Push operations on an item each require constant time. Predictor pop, might require $O(i^2)$. For each active node there might be at most i edges, therefore the *pop* procedure might take time $O(i^2)$ for each set S_i, which may have $O(i)$ active nodes.

The computation of the *Path* function might require $O(n^3)$ also.

The Completer Operations (**3.a to 3.h**) can combine an item from S_i:

$[Ac(\delta_2, p_2), I(\delta_3, p_3), B \rightarrow \alpha \bullet \gamma, j]$ where: $0 \leq j, p_2, p_3, \leq i$

with at most all $[\_, I0(\delta_0, p_0), A \rightarrow \alpha \bullet B\beta, k] \in S_j$ $(i \neq j)$, where: $0 \leq j, k, p_0, \leq j$

such that there is an $i-edge((\delta_3, p_3), (\delta_0, p_0))$. There are at most $O(j^2)$ such possible combinations, provided the information in the node $\mathrm{Ac1}(\delta_1, p_1)$ is not taken into account in the combination. In other words, only two dimensions are taken into account from the three dimensional table where i-edges and edges are stored. This is in general the case except for some restricted cases in which the node Δ_{Ac1} is bound to the node Δ_{In} and therefore does not increase the combinatorial possibilities. Given that there are at most $O(i^3)$ items in each set, also in S_j, then the complexity of this operation for each state set is $O(i^5)$. This is a general analysis. We consider in detail each subcase of the Completer in the next subsection.

The required time for each iteration (S_i) is thus $O(i^5)$. There are S_n iterations then the time bound on the entire 1-4 steps of the algorithm is $O(n^6)$.

It can be observed that this algorithm keeps a CFG $(O(n^3))$ complexity if the values of p_1 and p_2 are dependent on i, i.e., if for each item there is only one value for p_1 and p_2 for each i.

$[\mathbf{Ac}(\delta, \mathbf{p}_1), \mathbf{I}(\delta_2, \mathbf{p}_2), A \rightarrow \alpha \bullet B\beta, p_j]$, where $0 \leq j \leq i$

Languages that have this property will be called *constant indexing* languages. In such cases the number of items in each set S_i is proportional to i, i.e., $O(i)$ and the complexity of the algorithm is $O(n^3)$. This is true even for some ambiguous grammars such as the following:

$G_{amb} = (\{S, B, C, D, E\}, \{a, b, c, d\}, \{i, \}, \#, S, P)$ where P is

$$S \underset{i}{\rightarrow} aSd \qquad S \rightarrow BC \qquad B \underset{\bar{i}}{\rightarrow} bB \qquad B \underset{\bar{i}}{\rightarrow} b \qquad C \rightarrow cC \qquad C \rightarrow c$$
$$S \rightarrow DE \qquad D \rightarrow bD \qquad D \rightarrow b \qquad E \underset{\bar{i}}{\rightarrow} cE \qquad E \underset{\bar{i}}{\rightarrow} c$$

$L(G_{amb}) = \{a^n b^m c^n d^n | n, m \geq 1\} \cup \{a^n b^n c^m d^n | n, m \geq 1\}$

In this case, the ambiguity resides in the CFG backbone, while the indexing is so to speak, deterministic. Once the CF backbone chooses either the "BC" path or the "DE" path of the derivation, only one indexing alternative can apply at each step in the derivation.

A different case of ambiguity is $G_{amb2} = (\{S, B, C\}, \{a, b, c, d\}, \{i, \}, \#, S, P)$ where P is :

$$1. S \underset{i}{\rightarrow} aSd \qquad 2. S \rightarrow BC \qquad 3. B \underset{\bar{i}}{\rightarrow} bB \qquad 4. B \underset{\bar{i}}{\rightarrow} b \qquad 5.\ C \rightarrow cC \qquad 6. C \rightarrow c$$
$$7. B \rightarrow bB \qquad 8. B \rightarrow b \qquad 9. C \underset{\bar{i}}{\rightarrow} cC \qquad 10.\ C \underset{\bar{i}}{\rightarrow} c$$

$L(G_{am2}) = \{a^n b^k c^m d^n | n, m, k \geq 1, m + k \geq n\}$

In this case (where the ambiguity is between a *pop* and an *epsilon-move* the number of items corresponding to the rules 3, 4, 7 and 8 would be bounded by the square of the number of b's scanned so far. However, the similar grammar $G_{amb3} = (\{S, B, C\}, \{a, b, c, d\}, \{i, \}, \#, S, P)$ would produce items proportional to i in each set S_i:

$S \underset{i}{\rightarrow} aSd \qquad S \rightarrow BC \qquad B \underset{\bar{i}}{\rightarrow} bB \qquad B \rightarrow \epsilon \qquad C \underset{\bar{i}}{\rightarrow} cC \qquad C \rightarrow \epsilon$

$L(G_{am3}) = \{a^n b^k c^m d^n | n \geq 1,\ m, k \geq 0, m + k = n\}$

Therefore, the sources of increase in the time complexity are a) ambiguity between mode of operation in the stack (ϵ, push, or pop) through the same derivation CF path and b) a different index value used in the same type of stack operation. The expensive one is the first case.

The properties of Earley's algorithm for CFGs remain unchanged, so the following results hold: $O(n^6)$ is the worst case; $O(n^3)$ holds for grammars with constant or unambiguous indexing. Unambiguous indexing should be understood as those grammars that produce for each string in the language a unique indexing derivation. $O(n^2)$ for unambiguous context free back-bone grammars with unambiguous indexing and $O(n)$ for bounded-state[8] context free back-bone grammars with unambiguous indexing.

5.3.2 Time and Space Complexity

In this subsection we compute the time and space complexities in terms of the length of the input sentence (n) and the size of the grammar. There is an important multiplicative factor on the complexity computed in terms of the input size which depends on the size of the grammar (see [6]). The size of the grammar designated by $|G|$ corresponds to the number of rules and the number of non-terminals. Complexity results concerning formalisms related to Natural Language (including CFLs) mostly are worst case results (upper bounds) and they do establish polynomial parsing of these grammars. There are no mathematical average case results. All average case results reported are empirical. In practice, most algorithms for CFGs, TAGs, and other grammar formalisms run much better than the worst case. Usually, the real limiting factor in practice is the size of the grammar.

In the analysis we present here, we keep separate the size of the context-free backbone and the indexing mechanism. This will allow to compare what is the impact of the indexing mechanism as an independent multiplicative factor.

[8] Context Free grammars where the set of items in each state set is bounded by a constant.

Analysis of Scan, Predict, Complete

The scanner operation requires constant time. It checks the current word (input a_j) against the terminal a expected by the scanner.

Scanner

If $[\Delta_A, \Delta_{In},\ A \rightarrow \alpha \bullet a\beta, j] \in S_i$ and $w_i + 1 = a$:

add $[\Delta_A, C, \Delta_{In},\ A \rightarrow \alpha a \bullet \beta, j]$ to $S_i + 1$

Predictor operations have in general the following shape.

Predictor Operations

If $[Ac(\delta_1, p_1), \Delta_0,\ B \rightarrow \alpha \bullet A\beta, j] \in S_i$ and $(A \underset{\delta}{\rightarrow} \gamma) \in P$:

add every $[(\delta_2, p_2), \Delta_3,\ A \rightarrow \bullet\gamma, i]$ to S_i

A grammar rule is added to the state set. There is still the need to check for duplicates of the grammar rules. The same grammar rule may have different values for δ. The same grammar rules may apply at most n^2 times (if it is a *pop* rule): there might be $O(I \cdot n)$ possible values for $Ac(\delta_1, p_1)$, and for each value there might be $O(I \cdot n)$ path values.

The computation of the *path* in Predictor takes at most $O(n^3)$ for each vertex, there are at most $O(|I| \cdot n)$ edges in a path and we consider edges to two vertices: the active node and the initial node. The *push* (1.2) and predictor 1.3 operations take constant time regarding the input size.

Given there are two stack possible values (Δ_{Ac} and Δ_{In}), the complexity of the predictor operation is: $O(|I|^2 \cdot |G| \cdot n^4)$.

We consider now the more simple cases of the Completer operation which have the following shape.

Completer Case 1, no recalculation of Δ_{Ac} (No reduction involved)

If $[\Delta_{Ac}, \Delta_{In},\ B \rightarrow \gamma \bullet, j] \in S_i$:

for every i-edge $(\Delta_{In}, \Delta_{I0})$ and item:

for $[\_, \Delta_{I0},\ A \rightarrow \alpha \bullet B\beta, k] \in S_j$

add $[\Delta_{Ac}, \Delta_{I0},\ A \rightarrow \alpha B \bullet \beta, k]$ *to* S_i

The complexity is bound by the number of elements in S_j that satisfy the edge condition. It depends on k and p_0: $O(|I| \cdot |G| \cdot n^2)$. There might be $O(I \cdot \Delta_{In} - 1)$ possible values for each edge and $\Delta_{In},\ k$ are bounded by n.

In each set in S_i there might be at most $O(|I|^2 \cdot |G| \cdot n^3)$ items. There are $O(n)$ possible values for each Δ_{Ac},Δ_{In} and $j : 0 \leq j, abs(p_{Ac}), abs(p_{In}), n$.

Total complexity is then $n \times O(|I|^2 \cdot |G| \cdot n^3) \times O(I \cdot G \cdot n^2) = O(|I|^3 \cdot |G|^2 \cdot n^6)$

The remaining cases of the Completer involve more complex operations because the value of Δ_{Ac} has to be recalculated, performing reductions. These additional operations are performed provided certain constraints are met. We will analyze each of them in turn.

Completer Case 2, recalculation of Δ_{Ac}: Completer D and Completer E

3.d Completer D: Push with Active Node in $\bar{I}$

If $[\Delta_{Ac}, \Delta_{In},\ B \rightarrow \gamma \bullet, j] \in S_i$ such that $\Delta_{Ac} = \bar{\Delta}_I$

for $edge(\Delta_{In}, \Delta_{Ac1}, \Delta_0)$ and

for $[\Delta_{Ac1}, \Delta_{I0},\ A \rightarrow \alpha \bullet B\beta, k] \in S_j$ such that: $\Delta_0 \in I \cup \bar{I}$,

add $[\Delta_{Ac1}, \Delta_{I0},\ A \rightarrow \alpha B \bullet \beta, k]$ *to* S_i

In each set in S_i there might be at most $O(|I| \cdot |G| \cdot n^2)$ items that satisfy the condition $\Delta_{Ac} = \bar{\Delta}_I$. This requires the p_A value of Δ_{Ac} to be equal to $-p_I$ in Δ_{In} (their integer values are opposites).

Each such item can be combined with at most $O(|I|^2 \cdot |G| \cdot n^3)$ items in each set in S_j, given the node Δ_{Ac1} has to be considered to compute the result item added to the S_i set (the values of Δ_{Ac}, Δ_{IO} and k produce give $O(n^3)$). So the total is $n \times O(|I| \cdot n) \times O(|I| \cdot |G| \cdot n^2) \times O(|I| \cdot |G| \cdot n^2) = O(|I|^3 \cdot |G|^2 \cdot n^6)$

3.e Completer E. Pop items with Push items that re-compute the value Active node. Subcase 1

If $[\Delta_{Ac}, \Delta_{In},\ B \rightarrow \gamma \bullet, j] \in S_i$ such that $\Delta_{Ac}\ \Delta_{In} \in \bar{I}$, $p_{Ac},\ p_{In} < 0$:

Compute *path2*(Δ_{Ac})

for edge$(\Delta_{In}, \Delta_{Ac1}, \Delta_{I0})$ where $\Delta_{In} = \bar{\Delta}_{Ac1},\ \Delta_{Ac1},\ \Delta_{IO} \in I$

for $[\Delta_{Ac1}, \Delta_{I0},\ A \rightarrow \alpha \bullet B\beta, k] \in S_j$

such that the following conditions imposed on the edges are satisfied:

if $p_{Ac1} > p_{I0}$, and $-p_{Ac} > p_{I0}$ and $\bar{\Delta}_{I0} \in path2(\Delta_{Ac})$

else if $-p_{Ac} > p_{I0}$ and there is an edge $(\bar{\Delta}_{Ac},\ \bar{\Delta}_{I0}, \_)$

add $[\bar{\Delta}_{I0}, \Delta_{I0},\ A \rightarrow \alpha B \bullet \beta, k]$ *to* S_i

Computing the set *path2* for each Δ_{Ac} takes $O(|I| \cdot n^2)$ time, so its bound is $O(|I| \cdot i^3)$ for each set S_i (Δ_{Ac}) is bound by i).

There are at most $O(|I| \cdot |G| \cdot n^2)$ items in S_j that satisfy this condition. Checking if $\bar{\Delta}_{I0} \in path2(\Delta_{Ac})$ or there is an edge $(\bar{\Delta}_{Ac}, \bar{\Delta}_{I0}, \_)$ is done in constant time.

Total complexity is then $n \times O(|I|^2 \cdot |G| \cdot n^3) \times O(|I| \cdot |G| \cdot n^2) = O(|I|^3 \cdot |G|^2 \cdot n^6)$

3.e Completer E. Pop items with Push items that re-compute the value Active node. Subcase 2

If $[\Delta_{Ac}, \Delta_{In},\ B \rightarrow \gamma \bullet, j] \in S_i$ such that $\Delta_{Ac} = \Delta_{In} \in \bar{I}$:

for edge$(\Delta_{In}, \Delta_{Ac1}, \Delta_{I0})$ where $\Delta_{In} = \bar{\Delta}_{Ac1}$ and items

for $[\Delta_{Ac1}, \Delta_{I0},\ A \rightarrow \alpha \bullet B\beta, k] \in S_j$

for edge$(\Delta_{Ac1}, \Delta_5, _)$

add $[\Delta_5, \Delta_{In},\ A \rightarrow \alpha B \bullet \beta, k]$ *to* S_i

There are at most $O(|I| \cdot |G| \cdot n^2)$ items in each set S_i that satisfy the condition $\Delta_{Ac} = \Delta_{In}$

There are at most $O(|I| \cdot |G| \cdot n^2)$ items in each set S_j that satisfy the condition $\Delta_{In} = \bar{\Delta}_{Ac1}$

There are at most $O(|I| \cdot n)$ edges for each Δ_{Ac1}.

Total complexity is then $n \times O(|I| \cdot n) \times O(|I| \cdot |G| \cdot n^2) \times O(|I| \cdot |G| \cdot n^2) = O(|I|^3 \cdot |G|^2 \cdot n^6)$

Finally we have not considered the case of the *path2* with different values than $\bar{\Delta}_{I0}$. These can be done as a separate filtering on the completed items in S_i, within the same time bounds adding an operation to update the active node reducing all possible paths.

We have seen then that $O(|I|^3 \cdot |G|^2 \cdot n^6)$ is the asymptotic bound for this algorithm.
If we compare the influence of the grammar size on the time complexity of the algorithm for GIGs and the one for CFGs, the only change corresponds to the added size of I. Therefore if the set of indices is relatively small there is not a significant change in the influence of the grammar size.

If we compare the algorithm with LIG algorithms, the impact of the grammar size might be higher for the LIG algorithms and the impact of the indexing mechanism smaller.

The higher impact of the grammar size would be motivated in the following kind of completer required by LIGs (from [4]), that involves the combination of three constituents:

$$\frac{\begin{array}{l}[A[..] \rightarrow \Gamma_1 \bullet B[..\gamma]\Gamma_2, -, i, k, |-, -, -] \\ [B \rightarrow \Gamma_3 \bullet, \gamma, k, j, |C, p, q] \\ [C \rightarrow \Gamma 4 \bullet, \eta, p, q, |D, r, s]\end{array}}{[A[..] \rightarrow \Gamma_1 \bullet B[..\gamma]\Gamma_2, \eta, i, j, |D, rs]} \quad A[..] \rightarrow \Gamma_1 B[..\gamma]\Gamma_2$$

However the work load can be set either in the indexing mechanisms as is shown in the equivalent completor from [74], used to parse TAGs, where the set of nonterminals is {t,b,}.

Type 4 Completor:

$$\frac{\begin{array}{r}\langle t[\eta] \rightarrow \bullet\, t[\eta_r], i, -, -, i\rangle \\ \langle t[\eta_r] \rightarrow \Theta \bullet, i, j, k, l\rangle \\ \langle b[\eta] \rightarrow \Delta \bullet, j, p, q, k\rangle\end{array}}{\langle t[\eta] \rightarrow t[\eta_r] \bullet, i, p, q, l\rangle} \quad t[..\eta] \rightarrow t[..\eta\eta_r]$$

A note on Space Complexity

The space complexity is determined by the storage of Earley items. Given there are four parameter bound by n in each item, its space complexity is $O(n^4)$. Storage of the graph-structured stack takes $O(n^3)$. Storing the combination of i-edges and edges involves three nodes, and the number of possible nodes is bound by n, therefore it can be done in a $n \times n \times n$ table.

5.4 Proof of the Correctness of the Algorithm

The correctness of Earley's algorithm for CFGs follows from the following two invariants (top down prediction and bottom-up recognition) respectively:

Proposition 5 *An item $[A \rightarrow \alpha \bullet \beta, i]$ is inserted in the set S_j if and only if the following holds:*

1. $S \stackrel{*}{\Rightarrow} a_1 ... a_i A \gamma$
2. $\alpha \stackrel{*}{\Rightarrow} a_{i+1} ... a_j$

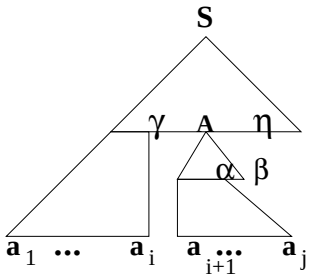


Figure 5.20: Earley CFG invariant

The corresponding invariants of Earley's algorithm for GIGs can be stated in a similar way, taking into account the operations on the stack. This is specified in terms of the Dyck language over $I \cup \bar{I}$

5.4.1 Control Languages

We defined the language of a GIG G, $L(G)$ to be: $\{w|\#S \stackrel{*}{\Rightarrow} \#w$ and w is in $T^*\}$. We can obtain an explicit control language[9] modifying the definition of derivation for any GIG $G = (N, T, I, S, \#, P)$ as follows. Let β and γ be in $(N \cup T)^*$, δ be in I^*, x in I, w be in T^* and X_i in $(N \cup T)$. Define C to be the Dyck language over the alphabet $I \cup \bar{I}$ (the set of stack indices and their complements).

1. If $A \underset{\mu}{\rightarrow} X_1 ... X_k$ is a production of type (a.) (i.e. $\mu = \epsilon$ or $\mu = [x]$, $x \in I$) then:
   $\langle \beta A \gamma, \#\delta \rangle \underset{\epsilon}{\Rightarrow} \langle \beta X_1 ... X_k \gamma, \#\delta \rangle$ or $\langle \beta A \gamma, \#\delta x \rangle \underset{[x]}{\Rightarrow} \langle \beta X_1 ... X_k \gamma, \#\delta x \rangle$

2. If $A \underset{\mu}{\rightarrow} a X_1 ... X_n$ is a production of type (b.) or *push*: $\mu = x, x \in I$, then:
   $\langle wA\gamma, \#\delta \rangle \Rightarrow \langle waX_k ... X_n \gamma, \#\delta x \rangle$

3. If $A \underset{\mu}{\rightarrow} X_1 ... X_n$ is a production of type (c.) or *pop* : $\mu = \bar{x}, x \in I$, then:
   $\langle wA\gamma, \#\delta x \rangle \Rightarrow \langle wX_1 ..... X_n \gamma, \#\delta x \bar{x} \rangle$

4. If $A \underset{\mu}{\rightarrow} X_1 ... X_n$ is a production of type (c.) or *pop* : $\mu = \bar{x}, x \in I, y \in \bar{I}$, and $\delta' y \in C$ then:
   $\langle wA\gamma, \#\delta x \delta' y \rangle \Rightarrow \langle wX_1 ..... X_n \gamma, \#\delta x \delta' y \bar{x} \rangle$

Then, define the language of a GIG G to be the control language $L(G, C)$: $\{w|\langle S, \# \rangle \stackrel{*}{\Rightarrow} \langle w, \#\delta \rangle$ and w is in T^*, δ is in C $\}$. It is apparent that both definitions of a GIG language are equivalent. Assuming such an alternative definition of a GIG language, we address the correctness of the algorithm as follows:

Proposition 6 (Invariants) .

An item $[Ac(y, p_1), I(z, p_0),\ A \rightarrow \alpha \bullet \beta, i]$ *is a valid item in the Set* S_j *if and only if the following holds:*

1. $\langle S, \# \rangle \stackrel{*}{\Rightarrow} \langle a_1 ... a_i A \gamma, \#\nu \rangle$ *for some* $\gamma \in V^*$ *and some* $\nu \in (I \cup \bar{I})^*$.

[9] See [25] for control languages pp. 173-185 and [87] for a control language representation for LIGs.

2. $(A \underset{\delta}{\rightarrow} \alpha\beta) \in P$

3. $\langle \alpha, \#\nu\delta \rangle \overset{*}{\Rightarrow} \langle a_{i+1}...a_j, \#\nu z'\mu y'\pi \rangle$

Such that:

a. $p_0 \leq i+1$ *and* $p_1 \leq j$.

b. $\delta = z'$ *iff* $\delta \in I \cup \bar{I}$, *else* $\delta = \epsilon$

c. $z, y \in I \cup \bar{I}$ *iff* $z = z', y = y'$, $\nu, \mu\pi \in (I \cup \bar{I})^*$, $\pi \in C$ *and* $\nu z'\mu y'\pi\rho \in C$ *for some* $\rho \in (I \cup \bar{I})^*$.

d. $z, \in cl(\delta'), \delta' \in DI$ *iff* $z'\mu y'\pi = epsilon$,

Figure 5.21 depicts the properties of the invariant, which naturally embed the invariant properties of the Earley algorithm for CFGs. The remaining conditions specify the properties of the string $\nu z'\mu y'\pi$ relative to the control language C. Condition a. specifies the constraints related to the position correlation with the strings in the generated language. Condition b. specifies the properties for the different types of productions that generated the current item. Condition c. specifies the properties of information of the graph-structured stack encoded in the item. Condition d. specifies that the item has a *transmitted* node from the graph if the empty string was generated in the control language.

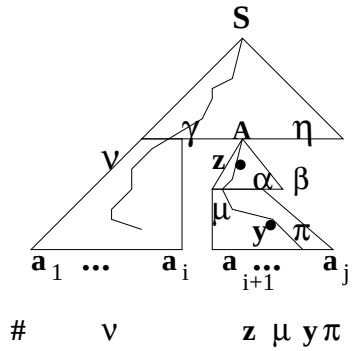


Figure 5.21: GIG invariant

We will use *Parsing Schemata* to prove this proposition. It will also help to understand the implementation of the algorithm we detail in the appendix.

5.4.2 Parsing Schemata

We use *Parsing Schemata*, a framework that allows high-level specification and analysis of parsing algorithms [75, 76]. This framework enables the analysis of relations between different parsing

algorithms by means of the relations between their respective parsing schemata. Parsing schemata are closely related to the *deductive parsing* approach (e.g.[72]).

We present parsing schemata rather informally here. Parsing Schemata are generalizations (or uninstantiations) of a parsing system $\mathbb{P}$. A parsing system $\mathbb{P}$ for a grammar G and a string $a_1...a_n$ is composed of three parameters a) $\mathcal{I}$ as set of items that represent partial specifications of parse results b) $\mathcal{H}$ a finite set of items (hypothesis) encodes the sentence to be parsed and $\mathcal{D}$ a set of deduction steps that allow to derive new items from already known items. Deduction steps are of the form $\eta_1, ..., \eta_k \vdash \xi$ which means that if all the required antecedents η_i are present, then the consequent ξ is generated by the parser. A set of final items, $\mathcal{F} \in \mathcal{I}$, represent the recognition of a sentence.

A Parsing schema extends parsing systems for arbitrary grammars and strings. As an example we introduce the Earley Schema for CFGs. Earley items, as we already saw are stored in sets according to each, say j, position in the string. The corresponding set of items in the Schema is encoded by an additional parameter in the items Schema items, defined by $\mathcal{I}_{Earley}$ below.

The input string is encoded by the set of hypothesis $\mathcal{H}_{Earley}$. Earley operations are defined by the allowed deductions: $\mathcal{D}^{Scan}$, $\mathcal{D}^{Pred}$, $\mathcal{D}^{Comp}$.

Parsing Schema 1 (Earley Schema for CFGs) .

This parsing schema is defined by a parsing system $\mathbb{P}_{Earley}$ for any CFG G and any $a_1...a_n \in \Sigma^*$ by:

$\mathcal{I}_{Earley} = \{[A \to \alpha \bullet \beta, i, j]\}$ such that $A \to \alpha\beta \in P$ *and* $0 \leq i \leq j$

$\mathcal{H}_{Earley} = \{[a, i-1, i] \mid a = a_i \wedge 1 \leq i \leq n\}$;

$\mathcal{D}^{Init} = \{\vdash [S' \to \bullet S, 0, 0]\}$,

$\mathcal{D}^{Scan} = \{[A \to \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \to \alpha a \bullet \beta, i, j+1]\}$

$\mathcal{D}^{Pred} = \{[A \to \alpha \bullet B\beta, i, j] \vdash [B \to \bullet\gamma, j, j]\}$,

$\mathcal{D}^{Comp} = \{[A \to \alpha \bullet B\beta, j, k], \quad [B \to \gamma \bullet \delta, k, j] \vdash [A \to \alpha B \bullet \beta, i, j]\}$,

$\mathcal{D}_{Earley} = \mathcal{D}^{Init} \cup \mathcal{D}^{Scan} \cup \mathcal{D}^{Pred} \cup \mathcal{D}^{Comp}$

$\mathcal{F}_{Earley} = \{[S' \to S \bullet, 0, n]\}$,

A final item $\mathcal{F}(\mathbb{P}_{Earley}) \in \mathcal{I}$ is correct if there is a parse tree for any string $a_1...a_n$ that conforms to the item in question.

$\mathcal{C}_{Earley} = \{[S' \to S \bullet, 0, n] \mid S \stackrel{*}{\Rightarrow} a_1...a_n\}$

The set of valid items is defined by the invariant of the algorithm.

$\mathcal{V}(\mathbb{P}_{Earley}) = \{[A \to \alpha \bullet \beta, i, j] | S \stackrel{*}{\Rightarrow} a_1...a_i A\gamma \wedge \quad \alpha \stackrel{*}{\Rightarrow} a_{i+1}...a_j\}$

Definition 12 *A parsing system* $(\mathbb{P})$ *is sound if* $\mathcal{F}(\mathbb{P}) \cap \mathcal{V}(\mathbb{P}) \subseteq \mathcal{C}(\mathbb{P})$ *(all valid final items are correct). A parsing system* $(\mathbb{P})$ *is complete if* $\mathcal{F}(\mathbb{P}) \cap \mathcal{V}(\mathbb{P}) \supseteq \mathcal{C}(\mathbb{P})$ *(all correct final items are valid). A Parsing system is correct if* $\mathcal{F}(\mathbb{P}) \cap \mathcal{V}(\mathbb{P}) = \mathcal{C}(\mathbb{P})$

5.4.3 Parsing Schemata for GIGs

The parsing schema for GIGs presented here is an *extension* of the parsing schema for CFGs given in the previous subsection. The parsing schema for GIGs corresponds to the algorithm 6 presented in section 5.2.

It can be observed that the major differences compared to the CFGs schema are the addition of different types of Prediction and Completer deduction steps. However there is an additional type of deduction step performed in this parsing schema, and this corresponds to the control language.

Therefore $\mathcal{IC}_{EGIG}$, a different set of items, will be used to perform the corresponding deduction steps. Those items in $\mathcal{IC}_{EGIG}$, will represent edges and paths in the graph. Items in $\mathcal{IC}_{EGIG}$ will participate in some restricted deduction steps.

Prediction Push and Pop deduction Steps, $\mathcal{D}^{PrPush}$, $\mathcal{D}^{PrPop1}$, and $\mathcal{D}^{PrPop2}$ have the following shape, and they introduce *prediction* items η_p in $\mathcal{IC}_{EarleyGIG}$.

$\eta_{ij} \vdash \eta_{jj} \ \wedge \eta_p$

Completers $\mathcal{D}^{CompB}$, $\mathcal{D}^{CompF}$, and $\mathcal{D}^{CompG}$ have the following shape, and they introduce *completion* items η_p in $\mathcal{IC}_{EarleyGIG}$.

$\eta_{ik}, \eta_{kj} \vdash \eta_{ij} \ \wedge \eta_p$

This kind of deduction step can be understood as shortform for two deduction steps.

Predictor steps can be decomposed as:

$\eta_{ij} \vdash \eta_{jj}$

$\eta_{ij} \vdash \eta_p$

Completer Steps can be decomposed as:

$\eta_{ik}, \eta_{kj} \vdash \eta_{ij}$

$\eta_{ik}, \eta_{kj} \vdash \eta_p$

Those deduction steps that compute the the Control language are $\mathcal{D}^{Path_a}$, $\mathcal{D}^{Path_b}$, $\mathcal{D}^{Path2_a}$, $\mathcal{D}^{Path2_b}$

Parsing Schema 2 (Earley Schema for GIGs) .

This parsing schema is defined by a parsing system $\mathbb{P}_{EGIG}$ for any GIG $G = (N, T, I, S, \#, P)$ and any $a_1 ... a_n \in \Sigma^*$ by:

$$\mathcal{I}_{EGIG} = \{[\Delta_{Ac}, \Delta_{In}, A \rightarrow \alpha \bullet \beta, i, j]\}$$
$$\text{such that } \Delta_{Ac}, \Delta_{In} \in \{(\delta, p) | \delta \in DI; or\ \delta = cl(\delta')\}\ ,$$
$$A \rightarrow \alpha\beta \in P \quad \text{and}$$
$$0 \leq i, p \leq j$$

$$\mathcal{IC}_{EGIG} = \{[P, \Delta_{Ac}, \Delta_{Ac1}, \Delta_{I0})]\} \cup \{[P, \Delta_1, \Delta_2]\}$$
$$\text{such that } \Delta_i \in \{(\delta, p) | \delta \in DI\ \wedge\ 0 \leq p \leq j\} \text{ and}$$
$$P \text{ is in } \{pop, push, popC, pushC, path, path2\}$$

$$\mathcal{H}_{EGIG} = \{[a, i-1, i] \mid a = a_i \wedge 1 \leq i \leq n\};$$

$$\mathcal{D}^{Init} = \{\vdash [(\#, 0), (\#, 0), S' \rightarrow \bullet S\$, 0, 0]\},$$

$$\mathcal{D}^{Scan} = \{[\Delta_{Ac}, \Delta_{In}, A \rightarrow \alpha \bullet a\beta, i, j], [a, j, j+1]\ \vdash$$
$$[\Delta_{Ac}, \Delta_{In}, A \rightarrow \alpha a \bullet \beta, i, j+1]\}$$

$$\mathcal{D}^{PrE} = \{[\Delta_{Ac}, \Delta_{I0}, A \rightarrow \alpha \bullet B\beta, i, j]\ \vdash$$
$$[\Delta_{Ac}, cl(\Delta)_{I0}, B \underset{\nu}{\rightarrow} \bullet\gamma, j, j]\ | \nu = \epsilon\ or\ \nu = [\delta_{Ac}]\}$$

$$\mathcal{D}^{PrPush} = \{[\Delta_{Ac1}, \Delta_{I0}, A \rightarrow \alpha \bullet B\beta, i, j]\ \vdash$$
$$[(\delta_2, j+1), (\delta_2, j+1), B \underset{\delta_2}{\rightarrow} \bullet\gamma, j, j] \wedge$$
$$[push, (\delta_2, j+1), \Delta_{Ac1}, \Delta_{I0}] \mid \delta_2 \in I\},$$

$$\mathcal{D}^{PrPop1} = \{[(\delta_1, p_1)+, \Delta_{I0}, A \rightarrow \alpha \bullet B\beta, i, j]\ \vdash$$
$$[(\bar{\delta}_1, -p_1), (\bar{\delta}_1, -p_1), B \underset{\bar{\delta}_1}{\rightarrow} \bullet\gamma, j, j] \wedge$$
$$[pop, (\bar{\delta}_1, -p_1), (\delta_1, p_1), \Delta_{I0}] \mid \delta \in \bar{I}]\},$$

$$\mathcal{D}^{Path_a} = \{[pop, \bar{\Delta}_1-, \Delta_1+, \Delta_R], [P, \Delta_1+, \Delta_2+, \Delta_{R1}] \vdash$$
$$[path, \bar{\Delta}_1 -_, \Delta_2+, \Delta_{R2}] \mid$$
$$(\Delta_R = \Delta_1+, \Delta_{R2} = \Delta_{R1}, P = push) \vee$$
$$(\Delta_R < \Delta_1+, \Delta_{R2} = \Delta_R, P = pushC)\}$$

$$\mathcal{D}^{Path_b} = \{[pop, \bar{\Delta}_1-, \Delta_1+, \Delta_R], [P, \Delta_1+, \Delta_3-, \Delta],$$
$$[path, \Delta_3-, \Delta_2+, \Delta_{R2}] \vdash$$
$$[path, \bar{\Delta}_1-, \Delta_2+, \Delta_{R1}] \mid - \textit{same conditions } Path_a\}$$

$$\mathcal{D}^{PrPop2} = \{[\Delta_{Ac}-, \Delta_I, A \rightarrow \alpha \bullet B\beta, i, j],\ [path, \Delta_{Ac}-, \Delta_1+, \Delta_{R1}] \vdash$$
$$[\bar{\Delta}_1, \bar{\Delta}_1, B \underset{\bar{\delta}_1}{\rightarrow} \bullet\gamma, j, j] \ \wedge [pop, \bar{\Delta}_1, \Delta_{Ac}-, \Delta_I] | \delta_1 \in \bar{I}\},$$

$$\mathcal{D}^{Path_c} = \{[pop, \bar{\Delta}_1-, \Delta_2-, \Delta_R], [path, \Delta_2-, \Delta_1+, \Delta_{R1}],$$
$$[P, \Delta_1+, \Delta_3+, \Delta_{R2}] \vdash [path, \bar{\Delta}_1 -_, \Delta_3+, \Delta_{R3}] \mid$$
$$(\Delta_{R1} = \Delta_1+, \Delta_{R3} = \Delta_{R2}, P = push) \vee$$
$$(\Delta_{R1} < \Delta_1+, \Delta_{R3} = \Delta_{R1}, P = pushC)\}$$

$$\mathcal{D}^{Path_d} = \{[pop, \bar{\Delta}_1-, \Delta_2-, \Delta_R], [path, \Delta_2-, \Delta_1+, \Delta_{R1}],$$
$$[P, \Delta_1+, \Delta_3-, \Delta],\ [path, \bar{\Delta}_3 -_, \Delta_4+, \Delta_{R4}] \vdash$$
$$[path, \bar{\Delta}_1 -_, \Delta_3+, \Delta_{R3}] \mid$$
$$(\Delta_{R1} = \Delta_1+, \Delta_{R3} = \Delta_{R4}, P = push) \vee$$
$$(\Delta_{R1} < \Delta_1+, \Delta_{R3} = \Delta_{R1}, P = pushC)\}$$

$$\mathcal{D}^{CompA} = \{[\Delta_{Ac}, \Delta_{In}, B \rightarrow \gamma\bullet, k, j],\ [\Delta_{In}, -, \Delta_{I0}]$$
$$[\Delta_{Ac1}, CL(\Delta)_{I0}, A \rightarrow \alpha \bullet B\beta, i, k] \vdash$$
$$[\Delta_{Ac}, \Delta_{In}, A \rightarrow \alpha B \bullet \beta, i, j]\},$$

$$\mathcal{D}^{CompB} = \{[\Delta_{Ac}+, \Delta_{In}+, B \rightarrow \gamma \bullet k, j],\ [\Delta_{In}+, -, \Delta_{I0}]$$
$$[\Delta_{Ac1}, \Delta_{I0}, A \rightarrow \alpha \bullet B\beta, j, k] \vdash$$
$$[\Delta_{Ac}+, \Delta_{I0}, A \rightarrow \alpha B \bullet \beta, i, j] \ \wedge \ [pushC, \Delta_{In}+, \Delta_{I0}]\}$$

$$\mathcal{D}^{CompC} = \{[\Delta_{Ac}-, \Delta_{In}+, B \rightarrow \gamma\bullet, k, j],\ [\Delta_{In}+, -, \Delta_{I0}]$$
$$[\Delta_{Ac1}, \Delta_{I0}, A \rightarrow \alpha \bullet B\beta, i, k], [pushC, \bar{\Delta}_I, \bar{\Delta}_I] \vdash$$

$$[\Delta_{Ac}+, \Delta_{I0}, A \rightarrow \alpha B \bullet \beta, i, j]\}$$

$$\mathcal{D}^{CompD} = \{[\bar{\Delta}_I-, \Delta_{In}+, B \rightarrow \gamma \bullet, k, j],\ [\Delta_{In}+, \_, \Delta_{I0}]$$
$$[\Delta_{Ac1}, \Delta_{I0}, A \rightarrow \alpha \bullet B\beta, i, k]\ \vdash$$
$$[\Delta_{Ac1}, \Delta_{I0}, A \rightarrow \alpha B \bullet \beta, i, j]\}$$

$$\mathcal{D}^{CompE} = \{[\Delta_{Ac}-, \Delta_{In}-, B \rightarrow \gamma \bullet, k, j],\ [\Delta_{In}-, \bar{\Delta}_{In}-, \Delta_{I0}]$$
$$[\bar{\Delta}_{In}-, \Delta_{I0}, A \rightarrow \alpha \bullet B\beta, i, k], [path2, \Delta_{Ac}-, \Delta_{Ac2}-]\ \vdash$$
$$[\Delta_{Ac2}-, \Delta_{I0}, A \rightarrow \alpha B \bullet \beta, i, j]\ \}$$

$$\mathcal{D}^{CompF} = \{[\Delta_{Ac}, \Delta_{In}-, B \rightarrow \gamma \bullet, k, j],\ [\Delta_{In}-, \_, \Delta_{I0}+]$$
$$[\Delta_{Ac1}, \Delta_{I0}+, A \rightarrow \alpha \bullet B\beta, i, k]\ \vdash$$
$$[\Delta_{Ac}, \Delta_{I0}, A \rightarrow \alpha B \bullet \beta, i, j] \wedge$$
$$[pushC, \Delta_{In}, \Delta_{I0}]\ |\Delta_{Ac} > \Delta_{In}\}$$

$$\mathcal{D}^{Path2_a} = [pushC, \bar{\Delta}_1-, \Delta_1+], [pushC, \Delta_1+, \Delta_2-] \vdash$$
$$[path2, \bar{\Delta}_1\ -\ , \Delta_2-]$$

$$\mathcal{D}^{Path2_b} = [pushC, \bar{\Delta}_1-, \Delta_1+], [pushC, \Delta_1+, \Delta_2-],$$
$$[path2, \Delta_2-, \Delta_3-]\ \vdash$$
$$[path2, \bar{\Delta}_1-, \Delta_3-]$$

$$\mathcal{D}^{CompG} = \{[\Delta_{Ac}, \Delta_{In}-, B \rightarrow \gamma \bullet, k, j],\ [\Delta_{In}-, \_, \Delta_{I0}-]$$
$$[\Delta_{Ac1}, \Delta_{I0}-, A \rightarrow \alpha \bullet B\beta, i, k]\ \vdash$$
$$[\Delta_{Ac}, \Delta_{I0}-, A \rightarrow \alpha B \bullet \beta, i, j]\ \wedge\ [pushC, \Delta_{In}, \Delta_{I0}]\}$$

$$\mathcal{D}^{CompH} = \{[\Delta_{Ac1}, CL(\Delta)_{I0}, B \rightarrow \gamma \bullet, k, j],$$
$$[\Delta_{Ac1}, \Delta_{I0}, A \rightarrow \alpha \bullet B\beta, i, k]\ \vdash$$
$$[\Delta_{Ac1}, \Delta_{I0}, A \rightarrow \alpha B \bullet \beta, i, j]\ \}$$

A correctness proof method for parsing schemata

This method is defined in [76]

- Define the set of viable items $\mathcal{W} \subseteq I$ as the set of items such that the invariant property holds.
- Proof of soundness for all the deduction steps: $\mathcal{V}(\mathbb{P}) \subseteq \mathcal{W}$. It has to be shown that for all $\eta_1 \cdots \eta_k \vdash \xi \in D$ with $\eta_i \in H \cup \mathcal{W}, 1 \leq i \leq k$, it holds that $\xi \in W$. In other words that the deduction steps preserve the invariant property.
- Proof of completeness: $\mathcal{W} \subseteq \mathcal{V}(\mathbb{P})$. Show that all the viable items (all possible derivations are deduced by the algorithm). This proof uses a so-called *deduction length function* (dlf), to use in the induction proof. Assume that any item η in $\mathcal{W}$ with derivation length less than m is obtained (if $d(\eta) < m$ then $\eta \in \mathcal{V}(\mathbb{P})$), then prove that all ξ with $d(\xi) = m$, $\xi \in \mathcal{V}(\mathbb{P})$.

Definition 13 (*derivation length function*) [10]

Let $(\mathbb{P})$ *be a parsing schema, :* $\mathcal{W} \subseteq \mathcal{I}$ *a set of items. A function* $d : \mathcal{H} \cup \mathcal{W} \to \mathbb{N}$ *is a* dlf *if*

(i) $d(h) = 0$ *for* $h \in \mathcal{H}$.

(ii) for each $\xi \in \mathcal{W}$ *there is some* $\eta_1, \cdots, \eta_k \vdash \xi \in D$ *such that*

$\{\eta_1, \cdots, \eta_k\} \subseteq \mathcal{W}$ *and* $d(\eta_i) < d(\xi)$ *for* $1 \leq i \leq k$.

The derivation length function is defined according to the properties of the parsing algorithm.

Because we introduced another class of items, the items corresponding to the control language, $\mathcal{IC}_{EGIG}$, we need also an invariant for the computation of the control properties.[11]

Proposition 7 (Invariant of the Path function) .

An item $[path, (x, -p_2), (y, p_1), (z, p_0)]$ *is valid for a string* $a_1 \cdots a_j$ *if and only if the following holds:*

1. $\langle S, \# \rangle \overset{*}{\Rightarrow} \langle a_1 ... a_j \gamma, \# \nu x \rangle$ *for some* $\gamma \in NV^*$ *and some* $\nu \in (I \cup \bar{I})^*$ *and* $x \in \bar{I}$.
2. $p_2 > p_1 \geq p_0$
3. $\nu x = \mu y \pi$ *such that* $y \in I$, *and* $\pi \in C$

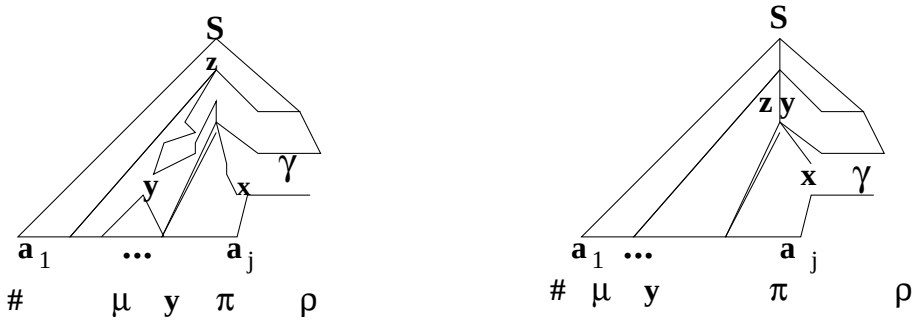


Figure 5.22: Path invariant

Lemma 7 (GIG items soundness is preserved under Scan, Predict Completer) .

$\mathcal{V}(\mathbb{P}) \subseteq \mathcal{W}$. *For all* $\eta_1 \cdots \eta_k \vdash \xi \in D$ *with* $\eta_i \in H \cup \mathcal{W}, 1 \leq i \leq k$ *it holds that* $\xi \in \mathcal{W}$

This can be rephrased as follows in terms of the algorithm presented in section 5.2, where correct *means that the proposition 2. (the invariant property) holds.*

Let $G = (N, T, I, \#, P, S)$ *be a GIG and let* $w = a_1 \cdots a_n, n \geq 0$ *where each* $a_k \in T$ *for* $1 \leq k \leq n$. *Let* $0 \leq i, j \leq n$.

- *If* η_i *is a correct item for grammar* G *and input* w *and* $\eta_i \vdash \eta_j$ *then* η_j *is a correct item.*
- *If* η_i, η_k *are correct items, for grammar* G *and input* w *and* $\eta_i, \eta_k \vdash \eta_j$ *then* η_j *is a correct item.*

Proof .

- Case 1. Scanner.

  An item η_j: $[Ac(y, p_1), I(z, p_0),\ A \rightarrow \alpha a \bullet \beta, i]$ is in S_j if and only if

  η_{j-1}: $[Ac(y, p_1), I(z, p_0),\ A \rightarrow \alpha \bullet a\beta, i]$ is in S_{j-1} and $\eta_{j-1} \underset{scan}{\vdash} \eta_j$

  Since η_{j-1} is correct, then

  a. $\langle S, \# \rangle \stackrel{*}{\Rightarrow} \langle a_1 ... a_i A\gamma, \#\nu \rangle$ for some $\gamma \in V^*$ and some $\nu \in (I \cup \bar{I})^*$.

  b.$(A \underset{\delta}{\rightarrow} \alpha a \beta) \in P$

  c. $\langle \alpha, \#\nu\delta \rangle \stackrel{*}{\Rightarrow} \langle a_{i+1} ... a_{j-1}, \#\nu z' \mu y' \pi \rangle$

[10] From [76].

[11] The correctness of the push items is straightforward. Regarding the pop items it is derived from the path function.

Now $\langle \alpha a_j, \#\eta\delta\rangle \overset{*}{\Rightarrow} (a_{i+1}...a_j, \#\nu z'\mu y'\pi\rangle$ by b. and c. This together with a. allows us to conclude that η_j is a correct item in S_j, and we conclude the proof for Case1.

- Case 2. Predictors

  An item η_{jj}: $[Ac(u, p_2), I(x, p_3),\ B \rightarrow \bullet\xi, j]$ is in S_j if and only if:

  η'_{ij}: $[Ac(y, p_1), I(z, p_0),\ A \rightarrow \alpha \bullet B\beta, i]$ is in S_j.

  $(B \underset{\delta}{\rightarrow} \xi) \in P$ such that

  Since η'_{ij} is correct,

  a. $\langle S, \#\rangle \overset{*}{\Rightarrow} \langle a_1...a_i A\gamma, \#\nu\rangle$ for some $\gamma \in V^*$ and some $\nu \in (I \cup \bar{I})^*$.

  b.$(A \underset{\delta'}{\rightarrow} \alpha B\beta) \in P$

  c. $\langle \alpha, \#\nu\delta'\rangle \overset{*}{\Rightarrow} \langle a_{i+1}...a_j, \#\nu z\mu y'\pi\rangle$

  Therefore: (this is common to all the subcases)

  d. $\langle S, \#\rangle \overset{*}{\Rightarrow} \langle a_1 \cdots a_i a_{i+1} \cdots a_j B\beta\gamma, \#\nu z\mu y\pi\rangle$ from η'_{ij} (a. b. and c.)

  and the following deduction steps apply according to the value of δ

  i. $\delta = \epsilon$, $x = cl(z)$, $p_3 = p_0$, $u = y$, $p_1 = p_2$.

  $\eta'_{ij} \underset{predict}{\vdash} \eta_{jj}$

  e. $\langle \epsilon, \#\nu z\mu y\pi\rangle \overset{*}{\Rightarrow} \langle \epsilon, \#\nu z\mu y\pi\rangle$ from $(B \underset{\epsilon}{\rightarrow} \xi) \in P$

  ii. $\delta = x \in I$, $(B \underset{x}{\rightarrow} a\xi) \in P$, $u = x$, $p_3 = j + 1$, $p_2 = j + 1$

  $\eta'_{ij} \underset{predict}{\vdash} \eta_{jj} \ \wedge\ \eta_{push}$

  $\eta_{push} = [push, (x, j + 1), (y, p_1), (z, p_0))$

  e. $\langle \epsilon, \#\nu z\mu y\pi x\rangle \overset{*}{\Rightarrow} \langle \epsilon, \#\nu z\mu y\pi x\rangle$

  iii. $\delta = \bar{y}$, $(B \underset{\bar{y}}{\rightarrow} \xi) \in P$, $x = \bar{y}$, $u = \bar{y}$, $p_3 = -p_1$, $p_2 = -p_1$

  $\eta'_{ij} \underset{predict}{\vdash} \eta_{jj} \ \wedge\ \eta_{pop}$

  $\eta_{pop} = [pop, (\bar{y}, -p_1), (y, p_1), (z, p_0))$

  e. $\langle \epsilon, \#\nu z\mu y\pi\bar{y}\rangle \overset{*}{\Rightarrow} \langle \epsilon, \#\nu z\mu y\pi\bar{y}\rangle$

  It should be noted also due to η_{pop} is obtained in this derivation, the following derivations follow:

  $\eta_{pop}, \eta_{push} \vdash \eta_{path}[path, \bar{y}, x', z'')$

where $\eta_{pop} = [pop, \bar{y}, y, z)$ and $\eta_{push} = [push, y, x', z')$ such that $x' \in I$ and z'' is either z or z', by $\mathcal{D}^{path_a}$ or

$\eta_{pop}, \eta_{push}\ \eta_{path1} \vdash \eta_{path2}$ by $\mathcal{D}^{path_b}$ where

$\eta_{pop} = [pop, \bar{y}, y, z)$, $\eta_{push} = [push, y, \bar{x}, z')$, $\eta_{path1} = [path, \bar{x}, x'', z'']$, $\eta_{path2} = [\bar{y}, x'', z'']$, $x, x'' \in I$.

iv. $y \in \bar{I}$, $\delta = \bar{x'}$, $(B \underset{\bar{x'}}{\rightarrow} \xi) \in P$, $p_2 = p_3, p_3 < 0$, $u = x$, $x = \bar{x'}$.

$\eta'_{ij}, \eta_{path} \underset{predict}{\vdash} \eta_{jj} \ \wedge\ \eta_{pop}$

$\eta_{path}((y, p_1), (x', abs(p_3)), z'')$

$\eta_{pop} = [pop, (\bar{x}, -p_3), (y, p_1), (z, p_0)]$

Since $\eta_{path} = [path, (y, p_1), (x', abs(p_3)), z'']$ is correct, then

d. $\#\nu z \mu y \pi = \#\nu' x' \pi'$ such that $\pi' \in C$.

Therefore

$\langle S, \# \rangle \overset{*}{\Rightarrow} \langle a_1 \cdots a_i a_{i+1} \cdots a_j B \beta \gamma, \#\nu z \mu y \pi \rangle$ from η'_{ij} (a. b. and c. in the previous page)

$\langle \epsilon, \#\nu z' \mu y' \pi \delta \rangle \overset{*}{\Rightarrow} \langle \epsilon, \#\nu z \mu y \pi \bar{x} \rangle$ from $(B \underset{\delta}{\rightarrow} \xi) \in P$

such that $\#\nu z \mu y \pi \bar{x} = \#\nu' x \pi' \bar{x}$ from d.

It should be noted also that as a consequence of η_{pop}, the same deductions as in the previous case are triggered:

$\eta_{pop}, \eta_{push} \vdash \eta_{path}$

$\eta_{pop}, \eta_{push}\ \eta_{path1} \vdash \eta_{path2}$ by $\mathcal{D}^{path_b}$

Case 3. Completer

An item η_{ij}: $[Ac(y', p_2'), I(z, p_0),\ A \rightarrow \alpha B \bullet \beta, i]$ is in S_j if and only if

an item η'_{kj}: $[Ac(y, p_2), I1(x, p_3),\ B \rightarrow \xi \bullet, k]$ is in S_j and

an item η''_{ik}: $[Ac1(u, p_1), I(z, p_0),\ A \rightarrow \alpha \bullet B \beta, i]$ is in S_k and

an item η_p: $[p, I1(x, p_3), \_\Delta_{Ac1}, I(z, p_0)]$ where p is either *push* or *pop*. (This item corresponds to the edges that relate the nodes in η_{ij} and η_{kj}

Given η''_{ik} and η'_{kj} are valid items we obtain:

a. $\langle S, \# \rangle \overset{*}{\Rightarrow} \langle a_1 ... a_i A \gamma, \#\nu \rangle$ for some $\gamma \in V^*$ and some $\nu \in (I \cup \bar{I})^*$.

b. $\langle \alpha, \#\nu\delta' \rangle \stackrel{*}{\Rightarrow} \langle a_{i+1}...a_k, \#\nu z'\mu u'\pi \rangle$ for some $(A \underset{\delta'}{\rightarrow} \alpha B\beta) \in P$

from η''_{ik} and

c. $\langle \xi, \#\nu'\delta'' \rangle \stackrel{*}{\Rightarrow} \langle a_{k+1}...a_j, \#\nu' x'\mu y'\pi' \rangle$ for some $(B \underset{\delta''}{\rightarrow} \xi) \in P$

Now compose some of the pieces:

d. $\langle \alpha B, \#\nu\delta' \rangle \stackrel{*}{\Rightarrow} \langle a_{i+1}...a_k B\beta, \#\nu z'\mu u'\pi \rangle$

e. $\Rightarrow \langle a_{i+1} \cdots a_k \xi, \#\nu z'\mu u'\pi\delta'' \rangle$ from $(B \underset{\delta''}{\rightarrow} \xi) \in P$

f. $\stackrel{*}{\Rightarrow} \langle a_{i+1} \cdots a_k a_{k+1} \cdots, a_j \#\nu z'\mu u'\pi x'\mu y'\pi' \rangle$

This derivation together with a. prove that η_j: $[Ac(y, p_2), I(z, p_0),\ A \rightarrow \alpha B \bullet \beta, i]$ is a correct item in S_j.

The crucial step is checking the consistency of: $\#\nu z\mu u\pi x\mu' y\pi'$ in f. The individual properties of $z\mu u$ and $x\mu y$ are checked using the information stored at:

$Ac1(u, p_1), I(z, p_0)$ in η''_{ik} $Ac(y, p_2), I1(x, p_3)$ in η'_{kj}.

Both substrings are substrings of the Control language but their concatenation might not be. We need to check the information in the corresponding nodes to validate the substring properties.

Those conditions are checked as follows according to the corresponding deduction step:

A. $\eta''_{ik}, \eta'_{kj}, \eta_p \underset{Complete}{\vdash} \eta_{ij}$ (η''_{ik} is a *transmitted* item).

if and only if:

$z \in cl(i)$, the control language spans ϵ in positions i, k, therefore the

Consequently the result subtree contains $\#\nu' x\mu' y\pi' =$ which is in the control language.

B. $\eta''_{ik}, \eta'_{kj}, \eta_p \underset{Complete}{\vdash} \eta_{ij} \wedge \eta_{pushC}$ (η'_{kj} has positive initial and active nodes)

if and only if:

$y, x \in I$ and $p_2 \geq p_3 > 0$

$\eta_{pushC} = [pushC, (x, p_3), (z, p_0)]$

therefore given $\#\nu'(x\mu'(y\pi'$ is in the control language for some ρ then $\#\nu z\mu u\pi x\mu' y\pi'$ is also in the control language for some ρ'

C. $\eta''_{ik}, \eta'_{kj}, \eta_p, \eta_{p'} \underset{Complete}{\vdash} \eta_{ij}$ (η'_{kj} initial node:+, active node:-)

if and only if:

$x \in I$, $p_3 > 0$, $y \in \bar{I}$, $abs(p_2) < p_3$

$\eta_{p'} = [popC, (\bar{x}, -p_3), (\bar{x}, -p_3)]$ requirement that x has to be reduced.

Therefore: $\#\nu z \mu u \pi (x \mu' y)\pi' = \#\nu z \mu u \pi (x \mu'' \bar{x}) \mu''' y)\pi'$

D. $\eta''_{ik}, \eta'_{kj}, \eta_p \underset{Complete}{\vdash} \eta_{ij}$ ($x\mu' y\pi'$ is in C. A reduction is performed.)

if and only if:

$x \in I$, $p_3 > 0$, $y = \bar{x}$, $p_2 = -p_3$.

Therefore $\#\nu z \mu u \pi (x \mu' \bar{x})\pi' = \#\nu z \mu u \pi'''$ and $\pi''' \in C$.

E. $\eta''_{ik}, \eta'_{kj}, \eta_p, \eta_{path2} \underset{Complete}{\vdash} \eta_{ij}$ (x reduces u. The current active node is negative).

if and only if:

$\eta_p = [p, (\bar{u}, -p_1), (u, p_1), (z, p_0)]$

$\eta_{path2} = [path2, (y, p_2), (y', p_2')]$, $abs(p_2') \leq abs(p_0)$

$x = \bar{u}$, $p_3 = -p_1$

Result string: $\#\nu z \mu (u \pi \bar{u}) \mu' y' \pi$ which is equivalent to $\#\nu z \mu \pi'' \mu' y' \pi'$.

η_{path2} guarantees that the result is in the control language.

F. $\eta''_{ik}, \eta'_{kj}, \eta_p \underset{Complete}{\vdash} \eta_{ij} \wedge \eta_p'$

if and only if:

$\eta_p' = [popC, (x, p_3), (x, p_3)]$

$z \in I$, $x \in \bar{I}$, $p_3 < p_2$

Result is $\nu z \mu u \pi x) \mu' \mathbf{y}' \pi'$

G. $\eta''_{ik}, \eta'_{kj}, \eta_p \underset{Complete}{\vdash} \eta_{ij} \wedge \eta_p 2$

if and only if

$\eta_p 2 = [popC, (x, p_3), (x, p_3)]$

$z, x \in \bar{I}$, $p_3 < 0$, $p_0 < 0$,

Result is $\nu z \mu u \pi x) \mu' \mathbf{y}' \pi'$

H. $\eta''_{ik}, \eta'_{kj}, \eta_p \underset{Complete}{\vdash} \eta_{ij}$

if and only if

$x \in cl(DI)$, the initial node is a transmitted node.

$cl(x)\mu'\mathbf{y}'\pi' = \epsilon$

Therefore the result is $\nu z\mu u\pi$

□

The following table 5.1 summarizes the conditions specified by the Completer Operations.

CASE	**COMPLETED ITEM**		**TARGET ITEM**		**Result**
	$I1(x, p_3)$	$Ac(y, p_2)$	$Ac1(u, p_1)$	$I(z, p_0)$	
A	+/-	+/-		$cl(z'\mu u'\pi) = \epsilon$	$x'\mu y'\pi'$
B	+	− +		+/-	$\#\nu z'\mu u'\pi(x'\mu'(y'\pi'$
C	+	−		+/-	$\#\nu z'\mu u'\pi(x'\mu' y')\pi'$, $\mu' = \mu''\bar{x'})\mu'''$
D	+	$- : y = \bar{x}$		+/-	**reduce:** $\#\nu z'\mu'' u'\pi''$
E	- $x = \bar{u}$	-	$+ : u$	+/-	$\#\nu z'\mu(u'\pi\bar{u'})\mu' y'\pi'$ **reduce:** $\#\nu z'\mu'' y''\pi''$
F	-	+/- $(x < y)$		+	$\#\nu z'\mu u'\pi x')\mu'\mathbf{y}'\pi'$
G	-	+/ -		-	**pushC**$(\Delta_{In}, -)$
H	$cl(x\mu y\pi') = \epsilon$			+/-, ϵ	$\#\nu z\mu u\pi$

Table 5.1: Conditions on the completer Deductions

Lemma 8 (Second direction, Completeness) .

For each $\xi \in \mathcal{W}$ there is some $\eta_1 \cdots \eta_k \vdash \xi \in D$ such that $\{\eta_1 \cdots \eta_k\} \in \mathcal{W}$ and $d(\eta_i) < d(\xi)$ for $1 \leq i \leq k$.

This lemma is equivalent to the following claim in terms of the algorithm presented in section 5.2 and Proposition 6. It states that all the derivations in the grammar will have the corresponding item in the set of states.

If

1. $\langle S, \# \rangle \stackrel{*}{\Rightarrow} \langle a_1 \cdots a_i A\gamma, \#\nu \rangle$
2. $(A \underset{\delta}{\rightarrow} \alpha\beta) \in P$
3. $\langle \alpha, \#\nu\delta \rangle \stackrel{*}{\Rightarrow} \langle a_{i+1} \cdots a_j, \#\nu z\nu' y\nu'' \rangle\}$ where $\delta = z$ if $\delta \in I \cup \bar{I}$ else $\delta = \epsilon$ such that $\#\nu z\nu' y\nu''$ satisfies the conditions imposed in proposition 2.

then an item

$\xi = [Ac(y, p_1), I(z, p_0),\ A \rightarrow \alpha \bullet \beta, i]$

is inserted in the Set S_j.

The proof by induction on the *derivation length function d* applied to the derivations specified in 1. and 3.

The derivation length function is based on the treewalk that the Earley algorithm performs. The modification of the Earley algorithm for GIGs does not modify the treewalk that the algorithm performs. Therefore the *derivation length function* is the same as defined by [76] as follows: [12]

$$d([Ac(y,p_1), I(z,p_0),\ A \rightarrow \alpha \bullet \beta, i, j]) =$$
$$min\{\pi + 2\lambda + 2\mu + j \mid \langle S, \# \rangle \stackrel{\pi}{\Rightarrow} \langle \delta A \gamma, \# \nu \rangle \wedge$$
$$\langle \delta, \# \nu \rangle \stackrel{\lambda}{\Rightarrow} a_1 \cdots a_i, \# \nu' \rangle \wedge$$
$$\langle \alpha, \# \nu' \rangle \stackrel{\mu}{\Rightarrow} \langle a_{i+1} \cdots a_j, \nu' z \nu'' y \nu''' \rangle\}$$

- Scanner: $\xi = [Ac(y,p_1), I(z,p_0),\ A \rightarrow \alpha a \bullet \beta, i]$

  Let

  1. $\langle S, \# \rangle \stackrel{*}{\Rightarrow} \langle a_1 \cdots a_i A \gamma', \# \nu \rangle$
  2. $(A \underset{\delta}{\rightarrow} \alpha\beta) \in P$
  3. $\langle \alpha, \# \nu \delta \rangle \stackrel{*}{\Rightarrow} \langle a_{i+1} \cdots a_j, \# \nu z \nu'' y \nu''' \rangle\}$ if $\delta \in I \cup \bar{I}$ else $\delta = \epsilon$

  Then $\eta = [Ac(y,p_1), I(z,p_0),\ A \rightarrow \alpha \bullet a \beta, i]$ is in S_{j-1} and from $\zeta = [a, j-1, j] \in \mathcal{H}^{GIG}$ $d(\zeta) = 0$, $d(\eta) = d(\xi) - 1$ and $\zeta,\ \eta \vdash \xi$.

- Completers: $\xi = [Ac(y,p_2), I(z,p_0),\ A \rightarrow \alpha B \bullet \beta, i]$ in S_k

  Let $\langle S, \# \rangle \stackrel{*}{\Rightarrow} \langle a_1 \cdots a_i A \gamma', \# \nu \rangle$,

  $\langle \alpha, \# \nu \rangle \stackrel{\mu}{\Rightarrow} \langle a_{i+1} \cdots a_j, \# \nu' z \nu'' u \nu''' \rangle$ and

  $\langle B, \# \nu' z \nu'' u \nu''' \rangle \underset{\delta}{\Rightarrow} \langle \gamma, \# \nu' z \nu'' u \nu''' \delta \rangle$

  $\langle \gamma, \# \nu' z \nu'' u \nu''' \delta \rangle \stackrel{\rho}{\Rightarrow} \langle a_{j+1} \cdots a_k, \# \nu' z \nu'' u \nu''' x \mu y \pi'' \rangle$

  with minimal $\mu + \rho$

  then if $\delta = x \in I \cup \bar{I}$

  $\eta = [Ac(u,p_1), I(z,p_0),\ A \rightarrow \alpha \bullet B \beta, i]$ is in S_j,

[12] Notice that induction on the strict length of the derivation would not work, because of the Completer operation. The conception of the *derivation length function* is similar to the kind of induction used by [39] on the order of the items to be introduced in the set S_i, such that the invariant holds for those items previously introduced in the item set.

$\eta' = [p, (x, p_3), (u, p_1), (z, p_0)]$ such that $p = push$ or $p = pop$.

$\zeta = [Ac(y, p_2), I(x, p_3),\ B \to \gamma\bullet, j]$ is in S_k and

$d(\eta) = d(\eta') - 1 < d(\zeta) = d(\xi) - 1$.

$\eta, \eta', \zeta \vdash \xi$

if $\delta = \epsilon$ and $x\mu y\pi'' = \epsilon$

$\zeta = [Ac(u, p_1), I(cl(z), p_0),\ B \to \gamma\bullet, j]$ is in S_k and

$d(\eta) < d(\zeta) = d(\xi) - 1$.

$\eta, \eta', \zeta \vdash \xi$

if $\delta = \epsilon$ and $x\mu y\pi'' \neq \epsilon$

$\eta' = [p, (x, p_3), (u, p_1), (z, p_0)]$ such that $p = push$ or $p = pop$.

$\zeta = [Ac(y, p_2), I(x, p_3),\ B \to \gamma\bullet, j]$ is in S_k and

$d(\eta) < d(\eta') < d(\zeta) = d(\xi) - 1$.

$\eta, \eta', \zeta \vdash \xi$

- Predict: $\xi = [Ac(y, p_1), I(z, p_0),\ B \to \bullet\gamma, j]$ in S_j

  Let $\langle S, \# \rangle \overset{*}{\Rightarrow} \langle a_1 \cdots a_j B\gamma', \#\nu \rangle$

  and let $\delta, \gamma'', \pi, \lambda, \mu$ such that

  $\langle S, \# \rangle \overset{\pi}{\Rightarrow} \langle \delta A\gamma'', \#\nu \rangle$,

  $\langle \delta, \#\nu \rangle \overset{\lambda}{\Rightarrow} a_1 \cdots a_i, \#\nu' \rangle$

  and $\langle A, \#\nu \rangle \underset{\delta}{\Rightarrow} \langle \alpha B\beta, \#\nu'\delta \rangle$

  $\langle \alpha, \#\nu'\delta \rangle \overset{\mu}{\Rightarrow} \langle a_{i+1} \cdots a_j, \nu' z \nu'' y \nu''' \rangle\}$ with $\pi + 2\lambda + 2\mu$ minimal.

  Then $\eta = [Ac(y, p_1), I(z, p_0),\ A \to \alpha\bullet B\beta, j]$

  $\eta \vdash \xi$ if $\delta = \epsilon$ and

  $\eta \vdash \xi\ \wedge\ \eta_p$

  $\eta_p = [p, (\delta, p_3), (y, p_1), (z, p_0)]$

  $d(\eta) = d(\xi) - 1 = d(\eta_p)$

□

Therefore from Lemma 7 ($\mathcal{V}(\mathbb{P}) \subseteq \mathcal{W}$), and Lemma 8 $\mathcal{W} \subseteq \mathcal{V}(\mathbb{P})$, we have proved the correctness of the Parsing Schema $\mathcal{V}(\mathbb{P}) = \mathcal{W}$.

5.5 Implementation

In order to perform some tests and as a proof of concept, the parsing algorithm described in section 5.2 was implemented in Sicstus Prolog on top of a general-purpose, agenda-based inference engine described in [72]. Encodings of the parsing schema as inference rules are interpreted by the inference engine. Given that the parsing schema abstracts from implementation details of an algorithm, the inference engine provides the control and data types. The Prolog database is used as a *chart* which plays the same role as the *state sets* in Earley's algorithm. The chart holds unique items in order to avoid applying a rule of inference to items to which the rule of inference has already applied before. This is performed using the subsumption capabilities of prolog: those items not already subsumed by a previously generated item are asserted to the database (the *chart*).

Items should be added to the chart as they are proved. However, each new item may itself generate new consequences. Any new potential items are added to an *agenda* of items for temporarily recording items whose consequences under the inference rules have not been generated yet. When an item is removed from the agenda and added to the chart, its consequences are computed and themselves added to the agenda for later consideration. Those new potential items are triggered by items added to the chart. Because the inference rules are stated explicitly, the relation between the abstract inference rules described in the previous section as parsing schema and the implementation we used is extremely transparent. However, as a meta-interpreter, the prototype is not particularly efficient.

The general form of an agenda-driven, chart-based deduction procedure is as follows (from [72]):

1. Initialize the chart to the empty set of items and the agenda to the axioms of the deduction system.

2. Repeat the following steps until the agenda is exhausted:

   (a) Select an item from the agenda, called the *trigger item*, and remove it.

   (b) Add the trigger item to the chart, if necessary.

(c) If the trigger item was added to the chart, generate all items that are new immediate consequences of the trigger item together with all items in the chart, and add these generated items to the agenda.

3. If a goal item is in the chart, the goal is proved (and the string recognized); otherwise it is not.

The axioms correspond to the initial item in the Parsing Schema:

$\mathcal{D}^{Init} = \{\vdash [(\#, 0), (\#, 0), S' \rightarrow \bullet S\$, 0, 0]\}$

and the encoding of the input string.

The goal corresponds to the final item in the Parsing Schema:

$\mathcal{F}_{EGIG} = \{[(\#, 0), (\#, 0), S' \rightarrow S\bullet, 0, n]\}$.

In this implementation the control language was not represented by separate items but as side conditions on the inference rules, which where performed by procedures which implemented the path function over the graph representation.

Tests on the following languages were performed:

- $\{a^n b^n c^n | n \geq 1\}$, $\{a^n b^n c^n d^n | n \geq 1\}$, $\{a^n b^n c^n d^{e\,n} | n \geq 1\}$, $\{ww | w \in \{a, b\}^*\}$
- $\{w(cw)^+ | w \in \{a, b\}^+\}$ $\{a^n (b^n c^n)^+ | n \geq 1\}$,
- $\{a^n b^n c^m d^m e^n f^n g^m h^m | n \geq 1\}$, $\{a^n b^n c^m d^m e^n f^n g^m h^m i^n j^n | n \geq 1\}$
- $\{a^n b^n w w c^n d^n | n \geq 1, w \in \{a, b\}^+\}$
- Grammar G_{amb4} which generates *aabbcdcd* but does not generate *aabbcddc* nor *aabbdccd*.
- Grammar G_{amb5b} which generates *aacdbbcd* and *aadcbbdc* but does not generate *aacdbbdd* nor *aacdbbdc* (a variation of G_{amb5} presented above).
- The mix language $\{w | w \in \{a, b, c\}^*$ *and* $|a|_w = |b|_w = |c|_w \geq 1\}$

Critically the most demanding was the mix language due to the grammar we chose: G_{mix} presented in chapter 2. This grammar has a high degree of ambiguity.

The number of items generated for the Mix and the copy languages were as follows:

Length	n^4	Mix L.	Copy L.	MCopy L.
6	1296	$\approx$ 500	$\approx$ 100	$\approx$ 300
9/10	10,000	$\approx$ 2,000	$\approx$ 400	$\approx$ 1,500
12	20,736	$\approx$ 9,000	$\approx$ 700	$\approx$2,100
15/16	50,625	$\approx$ 16,000	$\approx$ 1,500	$\approx$5,000
18	104,976	$\approx$ 28,000	$\approx$ 2,100	$\approx$6,800

5.6 LR parsing for GILs

An LR parser for a CFL is essentially a compiler that converts an LR CFG into a DPDA automaton (cf. [41], [2]). In the GIG case, the technique we present here converts an LR GIG into a deterministic LR-2PDA according to the definition presented in Chapter 2. This approach can be extended to a Generalized LR parsing approach, if multiple values are allowed in the parsing table (cf. [80]).

Both [75] and [3] describe transformations of Earley Parsing Schemata to LR Parsing Schemata. The similarities of Earley Parsing and LR parsing are made explicit in those transformations. We will refer to the connections between Earley parsing and LR parsing to clarify this presentation.

5.6.1 LR items and viable prefixes

We assume the reader has knowledge of LR parsing techniques (cf. [2]). The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes in context free LR-parsing. An item $A \rightarrow \beta_1 \bullet \beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$. We use this notion of viable prefixes to describe the computation of the main stack in a LR-2PDA. The notion of valid items for GILs is more complex because a GIG LR-item has to encode the possible configurations of the auxiliary stack. It is also more complex because GIL derivation requires leftmost derivation for the stack operations.

We say a GIG item index pair $[\ \delta, A \underset{\mu}{\rightarrow} \beta_1 \bullet \beta_2]$ where δ is in $I \cup \{\#\}$ is valid for a viable GIG prefix $\alpha\beta_1$ if there is a derivation using the GIG context free backbone: $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$ and there is a GIG derivation $S' \overset{*}{\Rightarrow} \gamma_i a_1 ... a_i A\beta \overset{*}{\Rightarrow} \delta\gamma_j a_1 ... a_j \beta_2\beta$.

Therefore the set of items has to encode the corresponding *top-down* prediction that affects the stack of indices. We will do this in such a way that the set of items contain information not only of what possible items belong to the set but also of the possible indices that might be associated to the set of parsing configurations that the items describe. The sets that define the states in GIG LR parsing are composed by a pair of sets (IT, IN).

5.6.2 The *Closure* Operation.

The *Closure* operation is equivalent to the Predict function in Earley parsing (cf.[3]). However in Earley's algorithm this computation is done in run-time while in LR parsing it is precompiled in the LR states. For the GIG algorithm the set of items and possible index values cannot be determined only by the *Closure* operation. They are also determined by the *Goto* operation.

If IT is a set of items for a grammar G, then $closure(IT)$ is the set of items constructed from IT by the two rules:

1. Initially, every item in IT is added to $closure(IT)$.

2. If $A \underset{\mu}{\rightarrow} \alpha \bullet B\beta$ is in $closure(IT)$ and (i) $B \underset{\delta}{\rightarrow} \gamma$ is a production where δ is in $I \cup \{\epsilon\}$, or (ii) $B \underset{\delta}{\rightarrow} a\gamma$ then add the item $B \underset{\delta}{\rightarrow} \bullet\gamma$ (case 1) or the item $B \underset{\delta}{\rightarrow} \bullet a\gamma$ (case 2) to IT if it is not already there. Apply this rule until no more new items can be added to the closure of IT.

Note that the two conditions of the *Closure* operation rule out the possible index operations $\bar{\delta}$ and $[\delta]$ when the right-hand side of the production starts with a non-terminal. Such productions are considered in case 2 of the *Goto* function.

For CFG parsing (cf. [41]) the *Closure* operation is equivalent to obtaining the set of states of a NFA that recognizes the viable prefixes.The transitions of this NFA are of the following form:

$\delta(A \rightarrow \alpha \bullet B\beta, \epsilon) = \{B \rightarrow \bullet\gamma \mid B \rightarrow \gamma \text{ is a production } \}$ (a NFA transition)

In the GIG case the transitions should be as follows, if the index operation is for instance $\bar{\iota}$:

$\delta(A \rightarrow \alpha \bullet B\beta, \epsilon, \iota) = \{(B \underset{\bar{\iota}}{\rightarrow} \bullet\gamma, \epsilon) \mid B \underset{\bar{\iota}}{\rightarrow} \gamma \text{ is a production } \}$ (a PDA transition)

It is apparent that the two items should not be conflated because there is a change in the stack configuration.

The following example uses a grammar for the language L_{wcw} to depict the construction of the set of items, the *goto* function and the parsing table.

Example 21 $L(G_{wcw}) = \{wcw \mid w \in \{a,b\}^*\}$, $G_{wcw} = (\{S,R\},\{a,b\},\{i,j\},S,\#,P)$ *and* P *is:*

1. $S \underset{i}{\rightarrow} aS$ *2.* $S \underset{j}{\rightarrow} bS$ *3.* $S \rightarrow cR$

4. $R \underset{\bar{i}}{\rightarrow} Ra$ *5.* $R \underset{\bar{j}}{\rightarrow} Rb$ *6.* $R \underset{[\#]}{\rightarrow} \epsilon$

The set of states for the grammar G_{wcw} should encode the possible configurations of the stack of indices as follows:

I_0:	#	$S' \rightarrow \bullet S$	I_1:	i	$S \underset{i}{\rightarrow} a \bullet S$	I_2:	j	$S \underset{j}{\rightarrow} b \bullet S$	I_3:	$\{i,j,\#\}$	$S \rightarrow c \bullet R$
		$S \underset{i}{\rightarrow} \bullet aS$			$S \underset{i}{\rightarrow} \bullet aS$			$S \underset{i}{\rightarrow} \bullet aS$	I_4:	$\{i,j,\#\}$	$R \underset{\bar{i}}{\rightarrow} \bullet Ra$
		$S \underset{j}{\rightarrow} \bullet bS$			$S \underset{j}{\rightarrow} \bullet bS$			$S \underset{j}{\rightarrow} \bullet bS$	I_5:	$\{i,j,\#\}$	$R \underset{\bar{j}}{\rightarrow} \bullet Rb$
		$S \rightarrow \bullet cR$			$S \rightarrow \bullet cR$			$S \rightarrow \bullet cR$	I_6:	#	$R \underset{[\#]}{\rightarrow} \bullet$
I_7 :	#	$S \underset{j}{\rightarrow} aS \bullet$	I_8:	#	$S \underset{j}{\rightarrow} bS \bullet$	I_9:	#	$S \underset{j}{\rightarrow} cR \bullet$	I_{10}:	#	$R \underset{\bar{i}}{\rightarrow} R \bullet a$
I_{11}:	#	$R \underset{\bar{j}}{\rightarrow} R \bullet$ b	I_{12}:	#	$R \underset{\bar{i}}{\rightarrow} Ra \bullet$	I_{13}:	#	$R \underset{\bar{j}}{\rightarrow} Rb \bullet$			

Figure 5.23: Set of States for G_{wcw}

In the same way that the dot implies that some input symbols have been consumed, the new sets I_4 and I_5 (as compared to context free sets) imply that an index i or j has been consumed from the stack. The set I_6 requires that the stack be empty.

5.6.3 The *Goto* Operation.

The CF function $goto(IT, X)$, where IT is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items $[A \underset{\mu}{\rightarrow} \alpha X \bullet \beta]$ such that $[A \underset{\mu}{\rightarrow} \alpha \bullet X \beta]$ is in IT. Intuitively, if I is the set of items that are valid for some viable prefix γ, then $goto(I, X)$ is the set of items that are valid for the viable prefix γX. The CF *Goto* function is a DFA transition function where IT is in the state set and X is in the vocabulary.

In the GIG case, if I is the set of items that are valid for some viable prefix-index pairs (γ,ι), then $goto(I, X, \iota) = (I_j, \iota_j)$, is the set of items that are valid of the viable prefix-index pairs (γX,ι_j) in other words, the *Goto* function in the GIG case is a PDA and the set of viable prefixes with associated indices is a context-free language.

We will describe the *Goto* function adding the possibility of computing the possible stack values of the PDA for each state. The GIG function *Goto* has three parameters a) IT is a set of items b) X is a grammar symbol and c) IN is a set of pairs (i, s) where i is an index symbol and s is a state defined by a set of items closure. We will say an index i is in IN if (i, s) is in IN. The operation $goto(IT, X, IN)$ is defined to be the pair (IT_2,IN_2) where either 1 or 2 applies:

1. IT_2 is the closure of the set of all items $[A \underset{\mu}{\rightarrow} \alpha X \bullet \beta]$ such that $[A \underset{\mu}{\rightarrow} \alpha \bullet X\beta]$ is in IT

   and IN_2 is defined to be the set of indices with an associated state such that:

   Case A: α is ϵ and X is a terminal a (i.e. $[A \underset{\mu}{\rightarrow} a \bullet \beta]$ is in IT_2):

   If μ is δ in I then (δ, IT_2) is in IN_2 and IN is in $predecessor(\delta, IT_2)$.

   If μ is ϵ then IN is in IN_2

   If μ is $[\delta]$ and δ is in IN then every (δ, IT_i) in IN is in IN_2

   If μ is $\bar{\delta}$ and δ is in IN then $predecessor(\delta, IT_i)$ is in IN_2.

   Case B: If X is a non-terminal B then IN_3 is in IN_2 such that

   $[B \underset{\mu}{\rightarrow} \gamma \bullet]$ is in (IT_3, IN_3).

   In other words, the auxiliary stack has the configuration corresponding to the completed item B.

   Case C: If α is not ϵ X is a terminal a then $IN = IN_2$(i.e. $[A \underset{\mu}{\rightarrow} \alpha a \bullet \beta]$ is in IT_2)

   When the second parameter of the *Goto* function is ϵ then $goto(IT, \epsilon, IN)$ is defined to be:

2. IT_2 is the closure of the set of all items

   $[B \underset{\bar{\delta}}{\rightarrow} \bullet \beta]$ such that $[A \underset{\mu}{\rightarrow} \alpha \bullet B\beta]$ is in IT and δ is in IN and IN_2 is defined such that $predecessor(\delta, IT_i)$ is in IN_2 for every (δ, IT_i) in IN

   $[B \underset{[\delta]}{\rightarrow} \bullet \beta]$ such that $[A \underset{\mu}{\rightarrow} \alpha \bullet B\beta]$ is in IT and δ is in IN and IN_2 is defined such that every (δ, IT_i) in IN is in IN_2

The *Goto* function for the grammar G_{wcw} is depicted in figure 5.24 with a PDA. The first element of the pair is the symbol X and the second indicates the operation on the PDA stack (e.g.: i indicates a *push*, $\bar{i}$ indicates a *pop*, [#] indicates a transition that requires an empty stack and ϵ indicates that the content of the stack is ignored). The indices associated in each state are those indices that are represented also in figure 5.23.

The construction of the collection of sets of items for a GIG grammar is made following the algorithm used for context free grammars, (e.g. [2]). The only changes are those made to the *Closure* and *Goto* operations as described above. The construction of an LR parsing table is shown in the following algorithm. The action table has three parameters, state, input (or ϵ) and auxiliary stack, the possible values are: a) pairs of shift action and operations on the auxiliary stack: *push,*

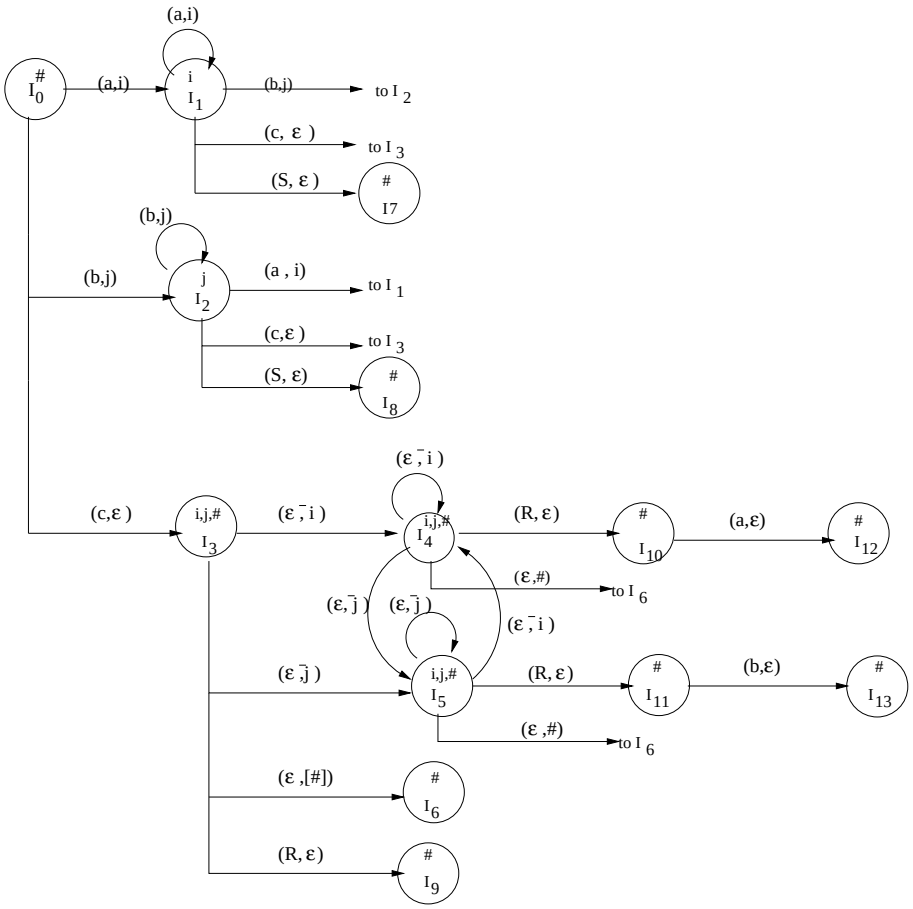


Figure 5.24: Transition diagram for the PDA recognizing the set of valid prefixes, annotated with indices for the grammar G_{wcw}

pop or none; b) reduce action (in this case no operation on the auxiliary stack is performed). The *Goto* table has the usual properties, and no operation on the auxiliary stack. The notation used is a follows: capital letters (A, B) denote non-terminals lower case letters (a) terminals, α is a non empty sequence of non terminals and terminals, β is a possible empty sequence of non terminals and terminals, δ is an index and μ is either δ, $\bar{\delta}$, $[\delta]$.

Algorithm 7 (Constructing an SLR parsing table) Input. An augmented GIG grammar G'.

Output. The SLR parsing table functions *action* and *goto*

1. Construct $C = \{I_0, I_1, ..., I_n\}$, the collection of sets of LR(0) items for G'.
2. State i is constructed from I_i. The parsing actions for state i are determined as follows:
   a. If $[A \underset{\delta}{\rightarrow} \bullet a\beta]$ is in I_i and $goto(I_i, a, IN_i) = (I_j, IN_j)$, and δ is in IN_j, then set $action[i, a, \epsilon]$ to ("shift j", "push δ"). Here a must be a terminal.
   b. If $[A \underset{\bar{\delta}}{\rightarrow} \bullet a\beta]$ is in I_i and $goto(I_i, a, IN_i) = (I_j, IN_j)$, and δ is in IN_i, then set $action[i, a, \delta]$ to ("shift j", "pop δ"). Here a must be a terminal.
   c. If $[A \underset{\bar{\delta}}{\rightarrow} \bullet B\beta]$ is in I_i and $goto(I_i, \epsilon, IN_i) = (I_j, IN_j)$, and δ is in IN_i, then set $action[i, \epsilon, \delta]$ to (shift j, "pop δ"). Here B must be a non-terminal.
   d. If $[A \underset{[\delta]}{\rightarrow} \bullet B\beta]$ is in I_i and $goto(I_i, \epsilon, IN_i) = (I_j, IN_j)$, and δ is in IN_i and IN_j then set $action[i, \epsilon, \delta]$ to (shift j, -). Here B must be a non-terminal.
   e. If $[A \underset{\epsilon}{\rightarrow} \bullet a\beta]$ is in I_i and $goto(I_i, a, IN_i) = (I_j, IN_j)$, then set $action[i, a, \epsilon]$ to ("shift j", -).
   f. If $[A \underset{[\delta]}{\rightarrow} \bullet a\beta]$ is in I_i and $goto(I_i, a, IN_i) = (I_j, IN_j)$, and δ is in IN_i and IN_j then set $action[i, a, \delta]$ to ("shift j", -).
   g. If $[A \underset{\mu}{\rightarrow} \alpha \bullet a\beta]$ is in I_i and $goto(I_i, a, IN_i) = (I_j, IN_j)$, then set $action[i, a, \epsilon]$ to ("shift j",-).

      Here a must be a terminal and α must be non-empty.
   h. If $[A \underset{\mu}{\rightarrow} \beta \bullet]$ is in I_i , then set $action[i, a, \epsilon]$ to "reduce $A \underset{\mu}{\rightarrow} \alpha$" for all a in FOLLOW(A); where A may not be S'.
   i. If $[S' \rightarrow S \bullet]$ is in I_i then set $action[i, \$, \#]$ to "accept"

   If any conflicting actions are generated by the above rules, we say the grammars is not SLR(1), but it is GLR(1).
3. The goto transitions for state i are constructed for all non-terminals A using the rule: If $goto(I_i, A, IN_i) = (I_j, IN_j)$, then $goto[i, A, in_j] = j$, for every index in IN_j.
4. the initial state of the parser is the one constructed from the set of items

The output transition table for the grammar G_{wcw} is shown in figure 5.2. This table is used with an obvious extension of the context free algorithm, adding the additional stack to compute the corresponding actions depicted in the table.

State	Action							Goto	
	(a,ϵ)	(b,ϵ)	(c,ϵ)	(ϵ,i)	(ϵ,j)	(ϵ,#)	($,#)	(S,#)	(R,#)
0	(s1, p i)	(s2, p j)	(s3, _)					14	
1	(s1, p i)	(s2, p j)	(s3, _)					7	
2	(s1, p i)	(s2, p j)	(s3, _)					8	
3				(s4, pop)	(s5, pop)	(s6, _)			9
4				(s4, pop)	(s5, pop)	(s6, _)			10
5				(s4, pop)	(s5, pop)	(s6, _)			11
6	r6	r6					r6		
7							r1		
8							r2		
9							r3		
10	(s12, _)								
11	(s13, _)								
12			r4						
13			r5						
14							acc		

Table 5.2: Action/Goto Table for G_{wcw}

The parsing algorithm for GIGs is essentially the same algorithm for CFGs. The only difference is that it takes into account the operations on the auxiliary stack. We follow the notation from [2].

Algorithm 8 (LR parsing for GIGs) .

Input. An input string w and an LR parsing action $-$ goto table for a grammar G.

Output. If w is in $L(G)$, a parse for w otherwise an error indication.

Initiate Stack1 with s_0. Initiate Stack2 with #. Set ip to point to the first symbol of $w\$$;

repeat forever begin

let s be the state on top of the Stack1 and

i the symbol on top of the Stack2

a the symbol pointed by ip.

if *action $[s, a, (i \cup \epsilon)]$ = shift (s', pi')* **then begin**

push a then s' on top of the Stack1

push i' on top of Stack2 ;

advance ip to the next input symbol.

else if *action $[s, a, (i \cup \epsilon)]$ = shift $(s', \_)$* **then begin**

push a then s' on top of the Stack1

advance ip to the next input symbol.

else if *action $[s, a, i]$ = shift (s', pop) then begin*

push a then s' on top of the Stack1

pop i from Stack2

advance ip to the next input symbol.

else if *action $[s, \epsilon, i]$ = shift (s', pop) then begin*

push a then s' on top of the Stack1

pop i from Stack2

else if *action $[s, a, i]$ = reduce $A \rightarrow \beta$* **then begin**

pop $2 * |\beta|$ symbols from Stack1;

let s' be the state now on top of the stack;

push A then goto$[s', A, i]$ on top of Stack1

output the production $A \underset{\mu}{\rightarrow} \beta$

else if *action $[s, a, i]$ = accept* **then**

return

else *error*

end

The following table shows the moves of the LR parser on the input string *abcab*.

	Stack 1	Stack2	Remaining input	Comments
1	0	#	abcab$	Initial ID
2	0a1	#i	bcab$	Shift and push
3	0a1b2	#ij	cab$	Shift and push
4	0a1b2c3	#ij	ab$	Shift
5	0a1b2c35	#i	ab$	Shift ϵ and Pop
6	0a1b2c354	#i	ab$	Shift ϵ and Pop
7	0a1b2c3546	#	ab$	Shift ϵ
8	0a1b2c354R10	#	ab$	Reduce by 6
9	0a1b2c354R10a	#	b$	Shift
9	0a1b2c35R11	#	b$	Reduce by 4
11	0a1b2c35R11b	#	$	Shift
12	0a1b2c3R9	#	$	Reduce by 5
13	0a1b2S8	#	$	Reduce by 3
14	0a1S7	#	$	Reduce by 2
15	0S14	#	$	Reduce by 1
16	-	#	$	Accept

Table 5.3: Trace of the LR-parser on the input *abcab* and grammar G_{wcw}

Chapter 6

Applicability of GIGs

6.1 Linear Indexed Grammars (LIGs) and Global Index Grammars (GIGs)

Gazdar [29] introduces Linear Indexed Grammars and discusses their applicability to Natural Language problems. His discussion is addressed not in terms of weak generative capacity but in terms of strong-generative capacity. Similar approaches are also presented in [82] and [44] (see [56] concerning weak and strong generative capacity). In this section, we review some of the abstract configurations that are argued for in [29]. The goal of this section is to show how the properties of GILs are related to the peculiarities of the *control* device that regulates the derivation. Though this mechanism looks similar to the control device in Linear Indexed Grammars , the formalisms differ in the kid of trees they generate.

GIGs offer additional descriptive power as compared to LIGs (and weakly equivalent formalisms) regarding the canonical NL problems mentioned above, at the same computational cost in terms of asymptotic complexity. They also offer additional descriptive power in terms of the structural descriptions they can generate for the same set of string languages, because they can produce *dependent paths*[1]. However those dependent paths are not obtained by encoding the dependency in the path itself.

[1]For the notion of dependent paths see for instance [82] or [44].

6.1.1 Control of the derivation in LIGs and GIGs

Every LIL can be characterized by a language $L(G, C)$, where G is a labeled grammar (cf. [87]), $G = (N, T, L, S, P)$, and C is a control set defined by a CFG (a Dyck language):

$\{a_1 ... a_n | < S, \epsilon > \stackrel{*}{\Rightarrow} < a_1, w_1 > ... < a_n, w_n >, a_i \in T \cup \{\epsilon\}, w_1, ..., w_n \in C\}$.

In other words, the *control* strings w_i are not necessarily *connected* to each other. Those control strings are encoded in the derivation of each *spine* as depicted in figure 6.1 at the left, but every substring encoded in a *spine* has to belong to the control set language. Those control words describe the properties of a path (a *spine*) in the tree generated by the grammar G, and every possible *spine* is independent.

We gave an alternative definition of the language of a GIG in chapter 5 to be the control language:

$L(G, C)$: $\{w \mid \langle S, \# \rangle \stackrel{*}{\Rightarrow} \langle w, \#\delta \rangle$ and w is in T^*, δ is in C $\}$

C is defined to be the Dyck language over the alphabet $I \cup \bar{I}$ (the set of stack indices and their complements).

It is easy to see that no control substring obtained in a derivation subtree is necessarily in the control language, as is depicted in the figure 6.1 at the right. In other words, the control of the derivation can be distributed over different paths, however those paths are connected transversely by the leftmost derivation order.

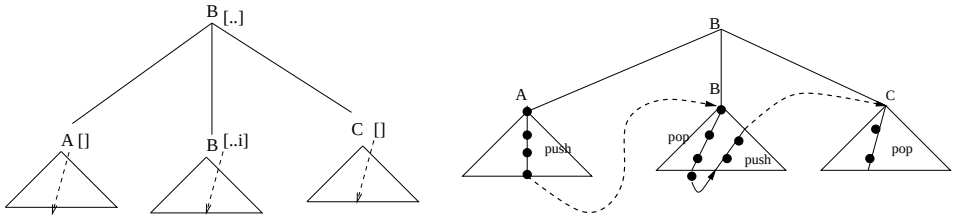


Figure 6.1: LIGs: multiple spines (left) and GIGs: leftmost derivation

6.1.2 The palindrome language

CFGs generate structural descriptions for palindrome languages (e.g. the language $\{ww^R | w \in \Sigma^*\}$) only of the nested (center embedded) sort. CFGs cannot generate the structural descriptions depicted in figures 6.2 and 6.3. (we follow Gazdar's notation: the leftmost element within the brackets corresponds to the top of the stack).

Gazdar suggests that the configuration in figure 6.2 would be necessary to represent Scandinavian

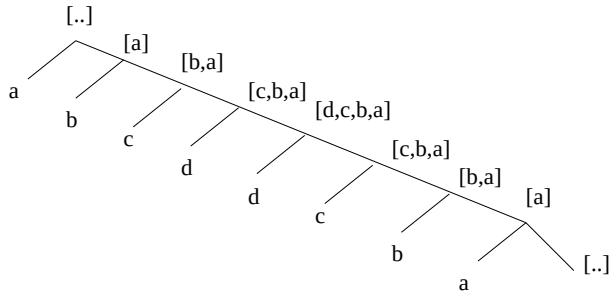


Figure 6.2: A non context-free structural description for the language ww^R

unbounded dependencies. Such an structure can be obtained using a GIG (and of course an LIG). But the exact mirror image of that structure, (i.e. the structure of figure 6.3) cannot be generated by a GIG because it would require *push* productions with a non terminal in the first position of the right-hand side. (i.e. in productions that are not in Greibach Normal Form).

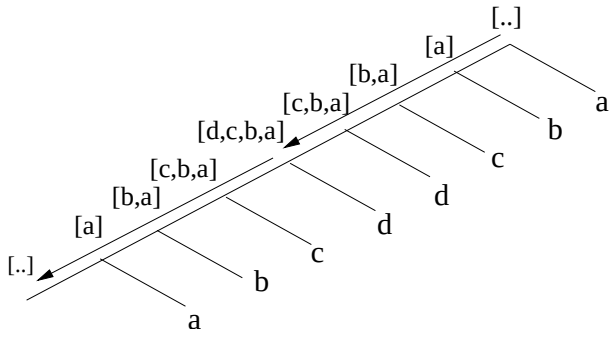


Figure 6.3: Another non context-free structural description for the language ww^R

However GIGs generate a similar structural description as depicted in figure 6.4 at the left. In such structure the dependencies are introduced in the leftmost derivation order. The English adjective constructions that Gazdar uses to motivate the LIG derivation are generated by the following GIG grammar. The corresponding structural description is shown in figure 6.4:

Example 22 (Comparative Construction) .

$G_{adj} = (\{AP, NP, \bar{A}, A\}, \{a, b, c\}, \{i, j\}, AP, \#, P)$ *where* P *is:*

$AP \rightarrow AP\ NP \qquad AP \rightarrow \bar{A} \qquad \bar{A} \rightarrow \bar{A}\ A$

$A \underset{i}{\rightarrow} a \qquad A \underset{j}{\rightarrow} b \qquad A \underset{k}{\rightarrow} c \qquad NP \underset{\bar{i}}{\rightarrow} a\ NP$

$NP \underset{\bar{j}}{\rightarrow} b\ NP \qquad NP \underset{\bar{k}}{\rightarrow} c\ NP$

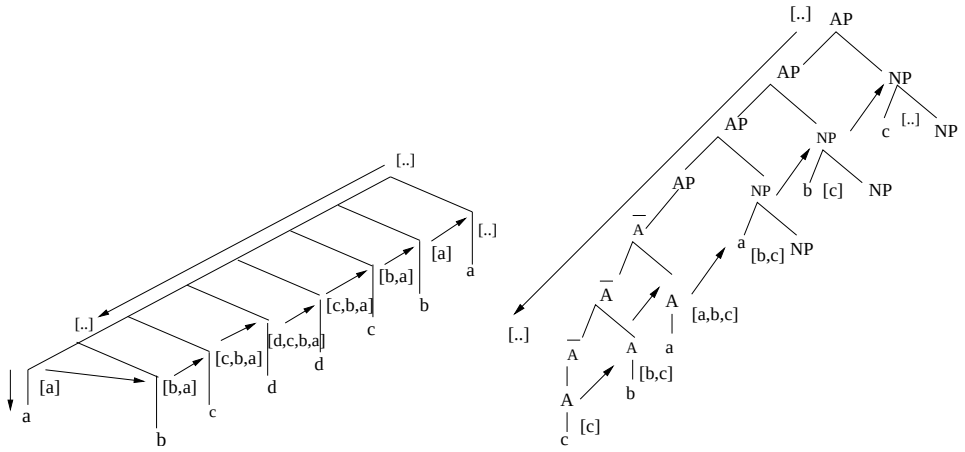


Figure 6.4: GIG structural descriptions for the language ww^R

It should be noted that the operations on indices are reversed as compared to the LIG case shown in figure 6.3. On the other hand, the introduction of indices is dependent on the presence of lexical information and its *transmission* is not carried through a top-down *spine*, as in the LIG case. The arrows show the leftmost derivation order that is required by the operations on the stack.

6.1.3 The Copy Language

Gazdar presents the following two possible LIG structural descriptions for the copy language (depicted in figure 6.5). The structural description at the left can be generated by a GIG, but not the one at the right. However, a similar one can be obtained using the same strategy as we used with the comparative construction.

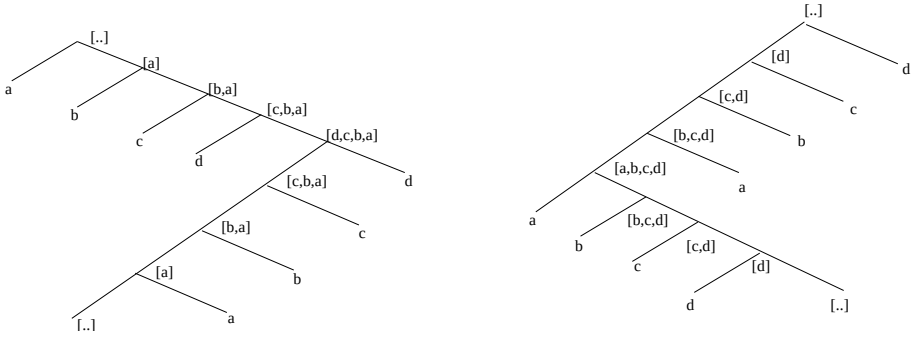


Figure 6.5: Two LIG structural descriptions for the copy language

Gazdar further argues that the following tree structure, shown in figure 6.6 at the left, could

be more appropriate for some Natural Language phenomenon that might be modeled with a copy language. Such structure cannot be generated by a LIG, but can be generated by an IG.

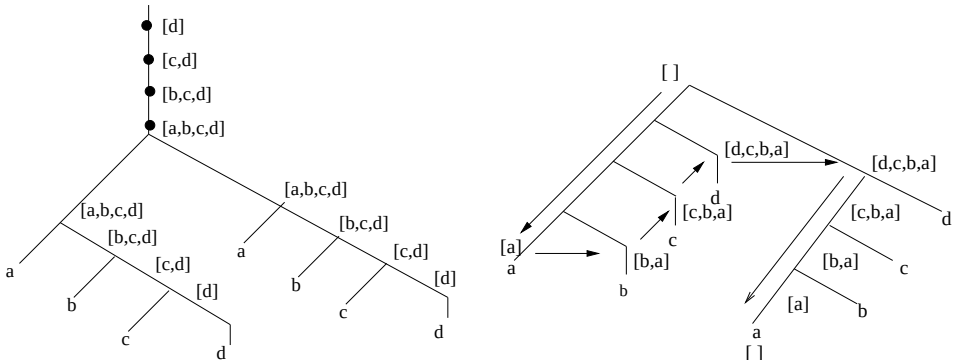


Figure 6.6: An IG and GIG structural descriptions of the copy language

GIGs cannot produce the structural description of figure 6.6 at the left either, but they can generate the one presented in the figure 6.6 (right), where the arrows depict the leftmost derivation order. GIGs can also produce similar structural descriptions for the language of multiple copies (the language $\{ww^+ \mid w \in \Sigma^*\}$ as shown in figure 6.7, corresponding to the grammar shown in example 10.

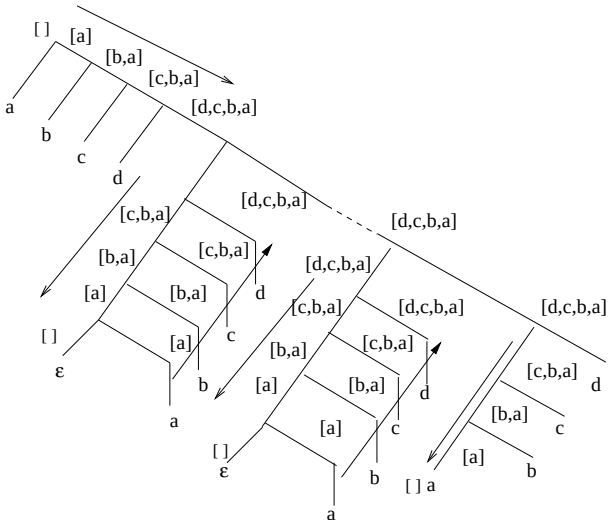


Figure 6.7: A GIG structural description for the multiple copy language

6.1.4 Multiple dependencies

Gazdar does not discuss of the applicability of multiple dependency structures in [29]. The relevant structures that can be produced by a LIG are depicted in figures 6.8 and 6.9 (left). GIGs can generate the same structures as in 6.8 and the somehow equivalent in 6.9 (right).

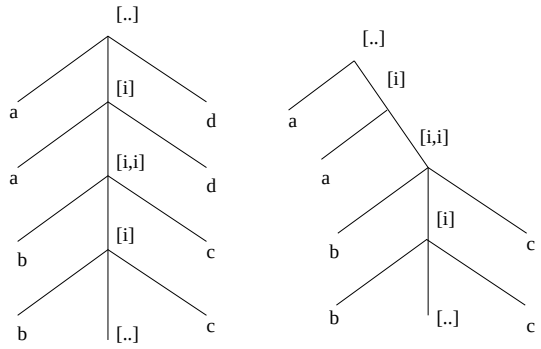


Figure 6.8: LIG and GIG structural descriptions of 4 and 3 dependencies

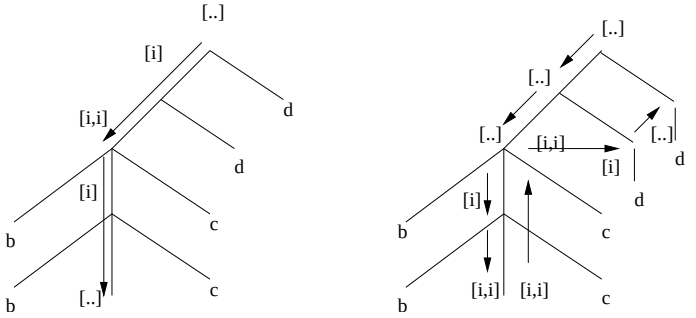


Figure 6.9: A LIG and a GIG structural descriptions of 3 dependencies

GIGs can also produce other structures that cannot be produced by a LIG, as we show in figures 6.10 and 6.11. The following language cannot be generated by LIGs. It is mentioned in [82] in relation to the definition of composition in [79] Categorial Grammars, which permits composition of functions with unbounded number of arguments and generates tree sets with dependent paths. The corresponding structure is depicted in the figure 6.10 (at the right).

Example 23 (Dependent branches) .

$L(G_{sum}) = \{\ a^n b^m c^m d^l e^l f^n \mid n = m + l \geq 1\}$,

$G_{sum} = (\{S, R, F, L\}, \{a, b, c, d, e, f\}, \{i\}, S, \#, P)$ where P is:

$$S \underset{i}{\rightarrow} aSf \mid R \qquad R \rightarrow F\ L \mid F \mid L \qquad F \underset{\bar{i}}{\rightarrow} b\ F\ c \mid b\ c \qquad L \underset{\bar{i}}{\rightarrow} d\ L\ e \mid d\ e$$

The derivation of $aabcdeff$:

$$\#S \Rightarrow i\#aSf \Rightarrow ii\#aaSff \Rightarrow ii\#aaRff \Rightarrow ii\#aaFLff \Rightarrow i\#aabcLff \Rightarrow \#aabcdeff$$

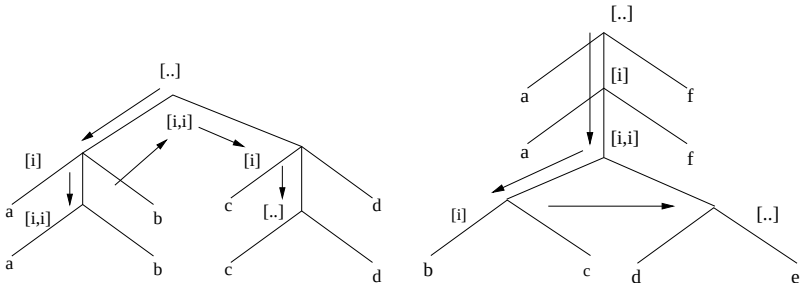


Figure 6.10: GIG structural descriptions of 4 dependencies

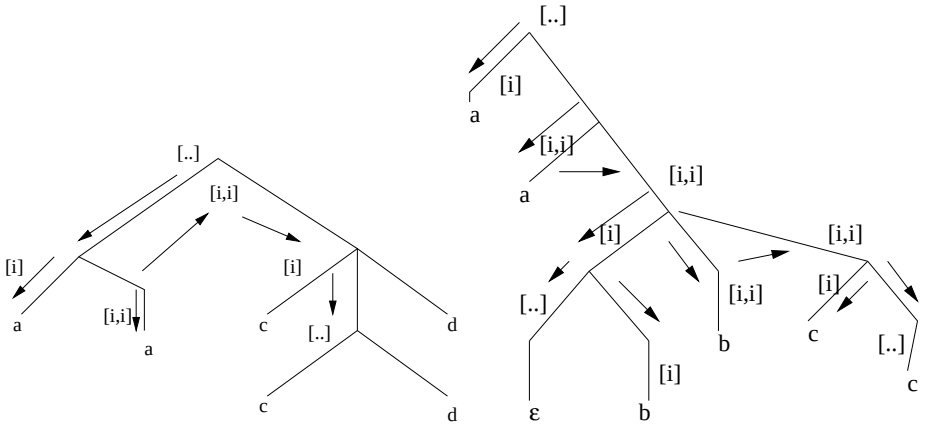


Figure 6.11: GIG structural descriptions of 3 dependencies

6.1.5 Crossing dependencies

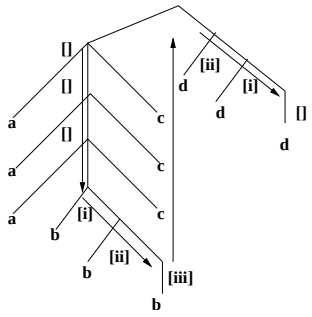
Crossing dependencies in LIGs and TAGs have the same structural description as the copy language.

GIGs can produce crossing dependencies with some alternative structures as depicted in the following structures. The first example is generated by the grammars G_{cr} presented in Chapter 3 and repeated here.

Example 24 (Crossing Dependencies) .

$L(G_{cr}) = \{a^n b^m c^n d^m |\ n, m \geq 1\}$, *where* $G_{cr} = (\{S, A, B, C\}, \{a, b, c, d\}, \{x\}, S, \#, P)$ *and* P *is:*

$S \rightarrow A\ D \qquad A \rightarrow a\ A\ c \mid a\ B\ c \qquad 3.\ B \underset{x}{\rightarrow} b\ B \mid b \qquad D \underset{\bar{x}}{\rightarrow} d\ D \mid d$

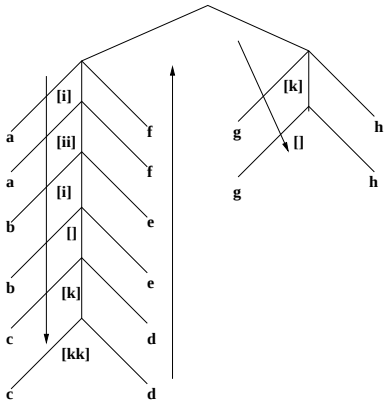


Additional number of crossing dependencies can be obtained (pairs or triples). The following kind of dependencies can be obtained:

Example 25 (Crossing Dependencies 2) .

$L(G_{cr8}) = \{a^n b^n c^m d^m e^n f^n g^m h^m |\ n, m \geq 1\}$, *where*
$G_{cr8} = (\{S, A, B, C\}, \{a, b, c, d, e, f, g, h\}, \{x, y\}, S, \#, P)$ *and* P *is:*

$S \rightarrow A\ L \qquad A \underset{x}{\rightarrow} a\ A\ f \mid a\ B\ f \qquad B \underset{\bar{x}}{\rightarrow} b\ B\ e \mid b\ C e \qquad C \underset{y}{\rightarrow} c\ C\ d \mid c\ d \qquad L \underset{\bar{y}}{\rightarrow} g\ L\ h \mid g\ h$



Such structures might be relevant to be able to generate the following language(discussed in [86]), related to the operation of generalized composition in CCGs.

$L_{crm} = \{a^n_1 a^m_2 b^n_1 c^n_1 b^m_2 c^m_2 d^m_2 d^n_1 \mid n, m \geq 1\}$

The language L_{crm} is generated by the following GIG:

Example 26 (Crossing Dependencies 3) .

$G_{crm} = (\{S, A, A_2, B, C, D, L, L_2\}, \{a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2\}, \{x, y\}, S, \#, P)$ *and* P *is:*

$S \rightarrow A\ L \qquad A \rightarrow a_1\ A\ B_1 \mid a_1\ A_2\ B_1 \qquad A_2 \underset{y}{\rightarrow} a_2\ A_2 \qquad B_1 \underset{x}{\rightarrow} b\ B \mid b$

$L \underset{\bar{x}}{\rightarrow} c_1\ L\ d_1 \mid c_1\ L_2\ d_1 \qquad L2 \rightarrow C\ D \qquad C \underset{\bar{y}}{\rightarrow} b_2\ C\ C_2 \mid b_2\ C_2 \qquad C_2 \underset{y}{\rightarrow} c_2 C_2$

$D \underset{\bar{y}}{\rightarrow} d_2 d_2$

The derivation of $a_1a_1a_2a_2a_2b_1b_1c_1c_1b_2b_2b_2c_2c_2c_2d_2d_2d_2d_1d_1$

$S \Rightarrow AL \Rightarrow \#a_1AB_1L \Rightarrow \#a_1a_1A_2B_1B_1L \overset{*}{\Rightarrow}$

$yyy\#a_1a_1a_2a_2a_2B_1B_1L \overset{*}{\Rightarrow} xxyyy\#a_1a_1a_2a_2a_2b_1b_1L \overset{*}{\Rightarrow}$

$yyy\#a_1a_1a_2a_2a_2b_1b_1c_1c_1CDd_1d_1 \overset{*}{\Rightarrow} \#a_1a_1a_2a_2a_2b_1b_1c_1c_1b_2b_2b_2C_2C_2C_2Dd_1d_1 \overset{*}{\Rightarrow}$

$yyy\#a_1a_1a_2a_2a_2b_1b_1c_1c_1b_2b_2b_2c_2c_2Dd_1d_1 \overset{*}{\Rightarrow} \#a_1a_1a_2a_2a_2b_1b_1c_1c_1b_2b_2b_2c_2c_2c_2d_2d_2d_2d_1d_1$

6.2 Set Inclusion between GIGs and LIGs

The last section showed similarities and differences between LIGs and GIGs in terms of the structural descriptions. In this section we are going to address the issue of whether LILs and GILs are comparable under set inclusion. Our conjecture is that LILs are properly included in GILs.

LIGs and Greibach Normal Form

The definition of GIGs requires *push* productions in Greibach Normal Form. If an equivalence between LIGs and LIGs with *push* productions in GNF could be obtained, we could proceed as follows.

Definition 14 *A LIG is in push Greibach Normal Formal if every push production is in Greibach Normal Form (where* $A, B \in N, a \in T$ *and* $\gamma \in (N \cup T)^*$ *:*

- $A[..] \rightarrow a\ B[..i]\ \gamma$

Lemma 9 *For any LIG in push-Greibach Normal Formal there is an Equivalent GIG that generates the same language.*

This is easy to prove, just a change in the interpretation of the derivation, from a LIG derivation to a GIG derivation and the equivalence is easy obtained.

GILs contain languages which are proven not to be in the LIL set. Therefore, if a lemma such as the following statement could be proved, the proper inclusion of LILs in GILs would be obtained:

- For any LIG there is an equivalent LIG in push-Greibach Normal Formal.

However we show in the next section that such a claim does not seem to hold.

6.2.1 LIGs in push-Greibach Normal Form

We can apply the techniques for left recursion elimination used in the conversion to Greibach Normal Form in CFGs.

The following lemma applies to productions that do not *carry* stack information. A similar lemma for CFGs is lemma 4.3 in [41]. The proof of this lemma holds for the GIG case because there is no stack information affected.

Lemma 10 *Define an $A-$production to be a production with variable A on the left. Let $G = (N, T, I, P, S)$ be a LIG. Let $A[..] \rightarrow B[]\alpha$ be a production in P and $B[..] \rightarrow [..]\beta_1|\beta_2|\cdots|\beta_r$ be the set of all B-productions (they do not affect the stack). Let $G_1 = (N, T, I, P, S)$ be obtained from G by deleting the production $A \rightarrow B\alpha_2$ from P and adding the productions $A \rightarrow \beta_1\alpha_2|\beta_2\alpha_2|\beta_r\alpha_2$. Then $L(G) = L(G_1)$.*

An adaptation of the lemma 4.4 in [41] as follows also holds for LIGs and it would allow to transform left recursion into right recursion, as long as the left recursion involves the whole spine.

Lemma 11 *Let $G = (N, T, I, P, S)$ be a LIG. Let $A[..] \rightarrow A[..\gamma_1]\alpha_1|A[..\gamma_2]\alpha_2|\cdots|A[..\gamma_r]\alpha_r$ be the set of A-productions for which A is the leftmost symbol of the right hand side. Let $A[..\gamma_i] \rightarrow \beta_1|\beta_2|\cdots|\beta_s$ be the remaining $A-$productions. Let $G_1 = (N \cup \{B\}, T, I \cup \{\gamma'\}, P1, S)$ be the LIG formed by adding the variable B to N, index γ' to I and replacing the $A-$productions by the productions:*

$A[..\gamma'_i] \rightarrow \beta_i, \quad A[..\gamma'_i] \rightarrow \beta_i B[..]$ *such that* $1 \leq i \leq s$

$B[..\gamma'_i] \rightarrow \alpha_i, \quad B[..\gamma'_i] \rightarrow \alpha_i B[..]$ *such that* $1 \leq i \leq r$

In this way, the equivalence of the structures in figure 6.12 involving the palindrome language would be obtained.

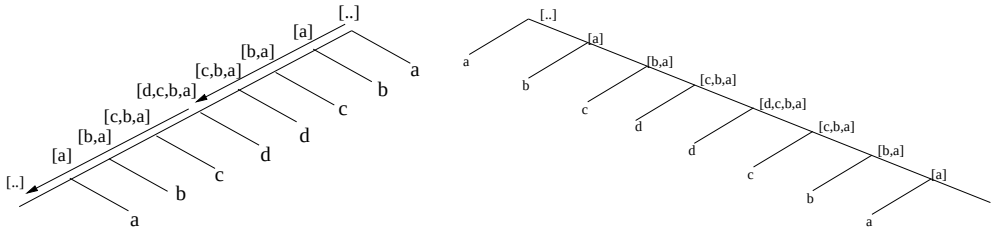


Figure 6.12: Left and right recursive equivalence in LIGs

The previous lemma would not work in the following structure, because it would need some additional changes, such as index renaming:

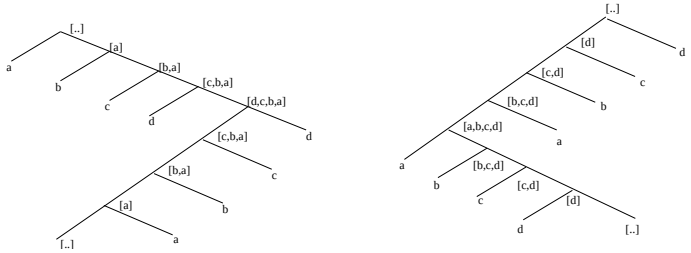


Figure 6.13: Left and right recursive equivalence in LIGs, second case

The structures depicted in figure 6.14 would require even more additional machinery. There is no way that a LIG like the one generating the three dependencies could be transformed into a LIG in push-GNF, without some mayor changes in the mapping algorithm.

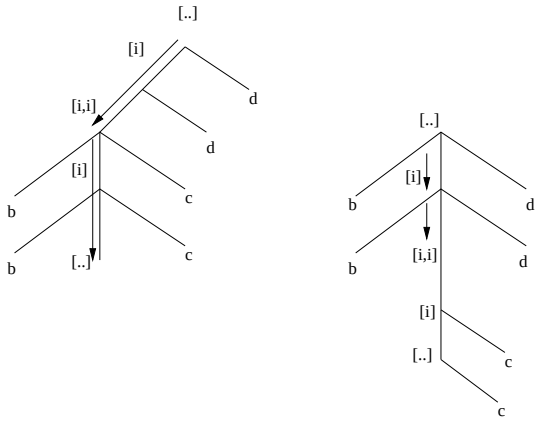


Figure 6.14: LIGs not in push-GNF I

Moreover some additional problems arise as in the following languages:

$\{w_1 w_2 w_1 w_2{}^R | w_i \in \Sigma^*\}$

$\{w_1 w_2 w_2 w_1{}^R | w_i \in \Sigma^*\}$

which can be depicted by the following structures:

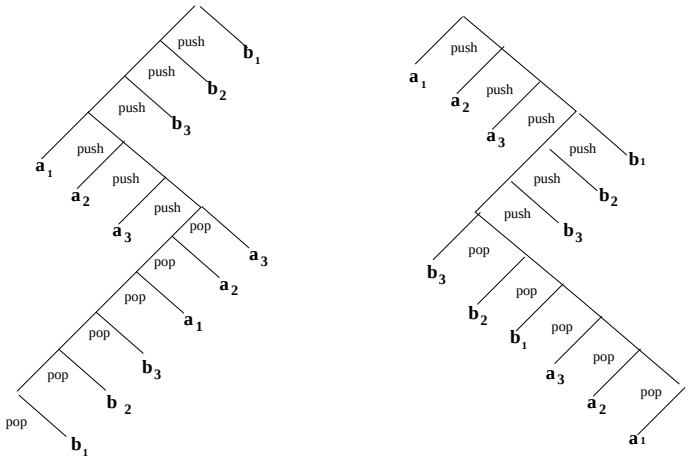


Figure 6.15: LIGs not in push-GNF II

These languages do not seem to be obtainable with a LIG in push-GNF. The problem is that they require the two directions of recursion (i.e. both right and left) for *push* productions. They are however generated by GIGs as shown in figure 6.16

It seems that some of the above-mentioned issues would still arise if we try to obtain the inclusion of LILs by simulating either an EPDA or Becker's 2-SA with a LR-2PDA. However, this alternative

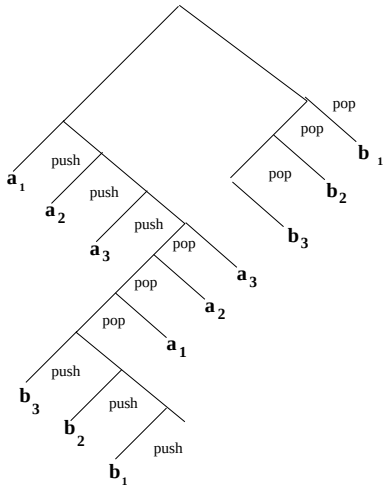


Figure 6.16: LIGs not in push-GNF II in GIGs

should be considered in detail.

6.3 GIGs and HPSGs

We showed in the last section how GIGs can produce structural descriptions similar to those of LIGs, and others which are beyond the descriptive power of LIGs and TAGs. Those structural descriptions corresponding to figure 6.2 were correlated to the use of the SLASH feature in GPSGs and HPSGs. In this section, we will show how the structural descriptive power of GIGs is not only able to capture those phenomena but also additional structural descriptions compatible with those generated by HPSGs [61] schemata. This follows from the ability of GIGs to capture dependencies through different paths in the derivation.

There has been some work compiling HPSGs into TAGs (cf. [48], [8]). One of the motivations was the potential to improve the processing efficiency of HPSG, by performing HPSG derivations at compile time. Such compilation process allowed the identification of significant parts of HPSG grammars that were mildly context-sensitive.

We will introduce informally some slight modifications to the operations on the stacks performed by a GIG. We will allow the productions of a GIG to be annotated with finite strings in $I \cup \bar{I}$ instead of single symbols. This does not change the power of the formalism. It is a standard change in PDAs (cf. [39]) to allow to push/pop several symbols from the stack. Also the symbols will be interpreted relative to the elements in the top of the stack (as a Dyck set). Therefore, different derivations

might be produced using the same production, according to what is on the top of the stack. This is exemplified with the productions $X \underset{\bar{n}v}{\rightarrow} x$ and $X \underset{[n]v}{\rightarrow} x$, in particular in the first three cases where different actions are taken (the actions are explained in the parenthesis) :

$nn\delta\#wX\beta \underset{\bar{n}v}{\Rightarrow} vn\delta\#wx\beta$ (pop n and push v)

$n\bar{v}\delta\#wX\beta \underset{\bar{n}v}{\Rightarrow} \delta\#wx\beta$ (pop n and $\bar{v}$) $vn\delta\#wX\beta \underset{\bar{n}v}{\Rightarrow} v\bar{n}vn\delta\#wx\beta$ (push $\bar{n}$ and v) $n\delta\#wX\beta \underset{[n]v}{\Rightarrow} vn\delta\#wx\beta$ (check and push)

We exemplify how GIGs can generate similar structural descriptions as HPSGs do in a very oversimplified and abstract way. We will ignore many details and try give an rough idea on how the *transmission* of features can be carried out from the lexical items by the GIG stack, obtaining very similar structural descriptions.

Head-Subj-Schema

Figure 6.17 depicts the tree structure corresponding to the Head-Subject Schema in HPSG [61].

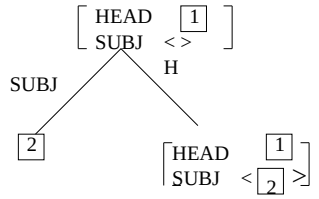


Figure 6.17: Head-Subject Schema

(1)
$$\left[\begin{array}{l} \mathrm{S}\left[\begin{array}{l} \mathrm{L\,|\,C}\left[\begin{array}{l}\mathrm{HEAD}\ \boxed{1}\\ \mathrm{SUBJ}\ \langle\rangle\\ \mathrm{COMPS}\ \boxed{3}\end{array}\right]\end{array}\right] \\ \mathrm{D}\left[\begin{array}{l} \mathrm{HEAD\_DTR}\left[\begin{array}{l}\mathrm{S\,|\,L\,|}\left[\begin{array}{l}\mathrm{HEAD}\ \boxed{1}\\ \mathrm{SUBJ}\ \left\langle\boxed{2}\right\rangle\\ \mathrm{COMPS}\ \boxed{3}\end{array}\right]\end{array}\right] \\ \mathrm{COMP\text{-}DTR}\left[\mathrm{S}\ \boxed{2}\right]\end{array}\right]\end{array}\right]$$

Figure 6.18 shows an equivalent structural description corresponding to the GIG productions and derivation shown in the next example (which might correspond to an intransitive verb). The arrows indicate how the transmission of features is encoded in the leftmost derivation order, an how the elements contained in the stack can be correlated to constituents or lexical items (terminal symbols) in a constituent recognition process.

Example 27 (intransitive verb) $XP \rightarrow YP\ XP \quad XP \rightarrow X \quad YP \rightarrow Y \quad X \underset{\bar{n}v}{\rightarrow} x \quad Y \underset{n}{\rightarrow} y$

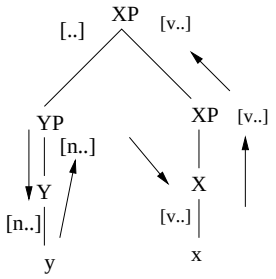


Figure 6.18: Head-Subject in GIG format

$\#XP \Rightarrow \#YPXP \Rightarrow \#yXP \Rightarrow n\#YXP \Rightarrow n\#yX \Rightarrow v\#yx$

Head-Comps-Schema

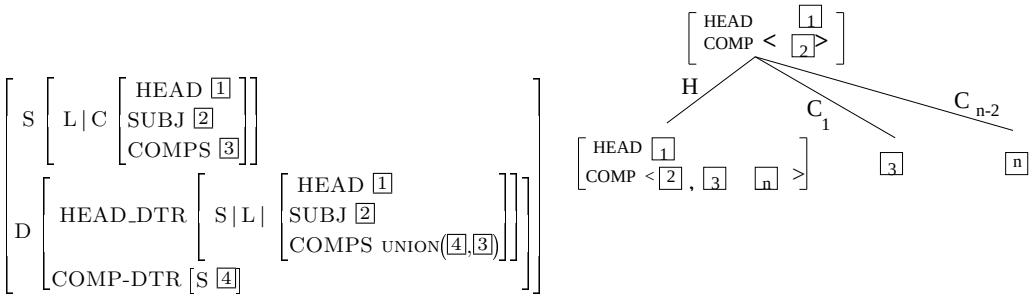


Figure 6.19: Head-Comps Schema tree representation

Figure 6.19 shows the tree structure corresponding to the Head-Complement schema in HPSG.

The following GIG productions generate the structural description corresponding to figure 6.20, where the initial configuration of the stack is assumed to be $[n]$:

Example 28 (transitive verb) .

$$XP \rightarrow X\ CP \quad CP \rightarrow Y\ CP \quad X \underset{\bar{n}v\bar{n}}{\rightarrow} x \quad CP \rightarrow \epsilon \quad Y \underset{n}{\rightarrow} y$$

The derivation:

$n\#XP \Rightarrow n\#XCP \Rightarrow \bar{n}v\#xCP \Rightarrow \bar{n}v\#xYCP \Rightarrow v\#xyCP \Rightarrow v\#xy$

The productions of example 29 (which uses some of the productions corresponding to the previous schemas) generate the structural description represented in figure 6.21, corresponding to the deriva-

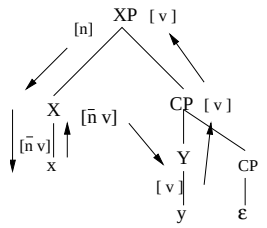


Figure 6.20: Head-Comp in GIG format

tion given in example 29. We show the contents of the stack when each lexical item is introduced in the derivation.

Example 29 (SLASH in GIG format) .

$XP \rightarrow YP\ XP \quad XP \rightarrow X\ CP \quad XP \rightarrow X\ XP$

$CP \rightarrow YP\ CP \quad X \underset{\bar{n}v\bar{n}}{\rightarrow} hates \quad CP \rightarrow \epsilon$

$X \underset{\bar{n}\bar{v}}{\rightarrow} know \quad X \underset{\bar{n}v\bar{v}}{\rightarrow} claims$

$YP \underset{n}{\rightarrow} Kim|Sandy|Dana|we$

A derivation of 'Kim we know Sandy claims Dana hates'*:*

$\#XP \Rightarrow \#YP\ XP \Rightarrow n\#Kim\ XP \Rightarrow$

$n\#Kim\ YP\ XP \Rightarrow nn\#Kim\ we\ XP \Rightarrow$

$nn\#Kim\ we\ X\ XP \Rightarrow \bar{v}n\#Kim\ we\ know\ XP \Rightarrow$

$\bar{v}n\#Kim\ we\ know\ YP\ XP \Rightarrow$

$n\bar{v}n\#Kim\ we\ know\ Sandy\ XP \Rightarrow$

$n\bar{v}n\#Kim\ we\ know\ Sandy\ X\ XP \Rightarrow$

$\bar{v}n\#Kim\ we\ know\ Sandy\ claims\ XP \Rightarrow$

$\bar{v}n\#Kim\ we\ know\ Sandy\ claims\ YP\ XP \Rightarrow$

$n\bar{v}n\#Kim\ we\ know\ Sandy\ claims\ Dana\ XP \stackrel{*}{\Rightarrow}$

$\#Kim\ we\ know\ Sandy\ claims\ Dana\ hates$

Finally the last example and figure 6.22 shows how coordination combined with SLASH can be encoded by a GIG.

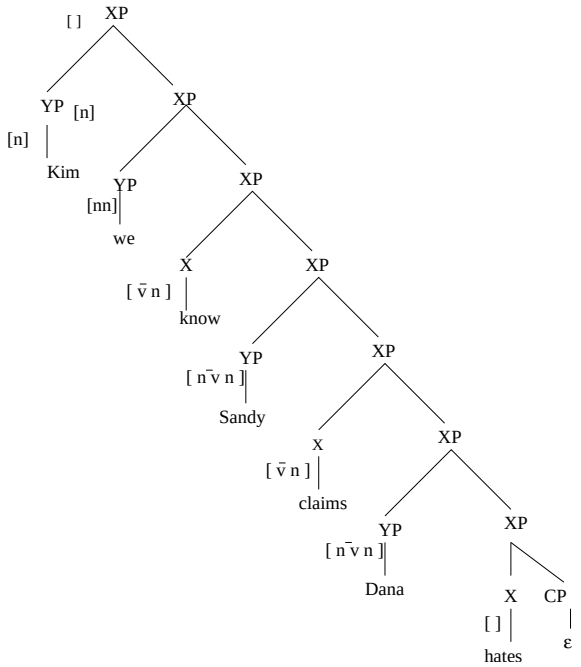


Figure 6.21: SLASH in GIG format

Example 30 (SLASH and Coordination2) .

$XP \rightarrow YP\ XP \quad XP \rightarrow X\ CP \quad XP \rightarrow X\ XP$

$CP \rightarrow YP\ CP \quad CP \rightarrow \epsilon \quad X \underset{[n\bar{v}n]_C}{\rightarrow} visit$

$X \underset{\bar{n}v\bar{n}}{\rightarrow} talk\ to \quad C \rightarrow and \quad CXP \rightarrow XP\ CXP$

$CXP \rightarrow C\ XP \quad X \underset{\bar{n}\bar{v}}{\rightarrow} did \quad YP \underset{n}{\rightarrow} Who|you$

A derivation of 'Who did you visit and talk to'*:*

$\#XP \Rightarrow \#YP\ XP \Rightarrow n\#Who\ XP \Rightarrow$

$n\#Who\ YP\ XP \Rightarrow \bar{v}n\#Who\ did\ XP \Rightarrow$

$\bar{v}n\#Who\ did\ YP\ XP \Rightarrow n\bar{v}n\#Who\ did\ you\ CXP \Rightarrow$

$n\bar{v}n\#Who\ did\ you\ XP\ CXP \Rightarrow$

$cn\bar{v}n\#Who\ did\ you\ visit\ CXP \Rightarrow$

$cn\bar{v}n\#Who\ did\ you\ visit\ C\ XP \Rightarrow$

$n\bar{v}n\#Who\ did\ you\ visit\ and\ XP \stackrel{*}{\Rightarrow}$

#Who did you visit and talk to

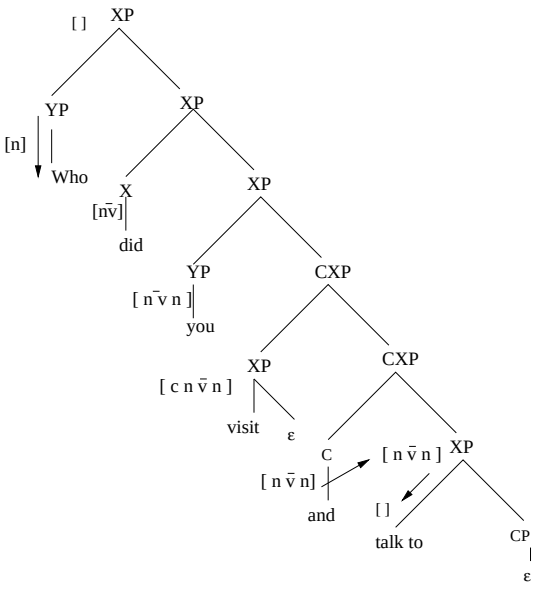


Figure 6.22: SLASH and coordination in GIG format

Chapter 7

Conclusions and Future Work

We have presented LR-2PDAS, GIGs, GILs and their properties. We showed the equivalence of LR-2PDA and GIGs. However we did not discuss the (possible) equivalence of sLR-2PDAs and trGIGs. We presented a Chomsky-Schützenberger representation theorem for GILs. We showed the descriptive power of GIGs regarding the three phenomena concerning natural language context sensitivity: *reduplication*, *multiple agreements* and *crossed agreements*. We introduced trGIGs (and sLR-2PDAs) which are more restricted than GIGs. We conjectured that trGIGs are not able to capture *reduplication phenomena*, and that are more limited regarding *crossed agreements*. An Earley type algorithm for the recognition problem of GILs was presented with a bounded polynomial time result $O(n^6)$ and $O(n^4)$ space. We analyzed the algorithm providing an account of grammar-size, time-complexity and space complexity properties. The proof of correctness of this algorithm was also presented using the Parsing Schemata framework. A proof of semilinearity showed that GILs share *mildly context sensitive* properties. The strong similarity between GIGs and LIGs suggests that LILs might be included in GILs. However, it might be difficult to prove it, or it might not be the case and both languages turn out to be incomparable. We also showed that both the weak and the strong descriptive power of GIGs is beyond LIGs. We presented a comparison of the structural descriptions that LIGs and GIGs can be generate. We have shown that GIGs generate structural descriptions for the copy and multiple dependency languages which can not be generated by LIGs. Finally, we have shown also that the extra power that characterizes GIGs, corresponds to the ability of GIGs to

generate dependent paths without *copying* the stack but *distributing* the control in different paths through the leftmost derivation order. We have shown also that those non-local relationships which are usually encoded in HPSGs as feature transmission can be encoded in GIGs using its stack, exploiting the ability of GIGs to encode dependencies through *connected* dependent paths and not only through a spine.

7.1 Future Work

The results described in the previous section are encouraging enough to continue further work concerning Global Index Languages. We mention now some of the issues that might be addressed in future research.

We have not addressed most of the decision problems, except the emptiness problem, which follows as a corollary from the semi-linearity property, and the membership problem. We have established some coarse-grain relations with other families of languages (CFLs and CSLs). The relationships between GILs LILs and ILs is still an open question. GILs and GCSLs are incomparable but trGILs might be a properly included in GCSLs. The similarity of GIGs and counter automata, coupled context-free grammars among other formalisms, should be addressed.

A candidate pumping lemma for GILs was suggested, however it is not clear to us if such a pumping lemma would provide an additional tool beyond the result already obtained on semilinearity.

We defined deterministic LR-2PDAs and deterministic GILs, and we gave an algorithm to parse $LR(k)$ GILs. However, we did not explore the properties of the class of deterministic GILs. They seem to be a proper subset of GILs and they also seem to be closed under complementation (which could be proved following the proof for CFLs); however, the proof should be considered in detail. We also mentioned the issue of ambiguity and we defined GILs with deterministic indexing. In this case, it is harder to determine whether GILs with deterministic indexing are a proper subset of the Global Index Languages. Although they do seem to be, this class might not have any theoretical import besides the fact that the recognition problem is in $O(n^3)$.

We gave an Earley algorithm for GIGs , however there are other variations on Earley's algorithm that were not considered and that might improve the impact of the grammar size on the parsing

complexity. There are also some details on the introduction of indices which would improve its performance. Another alternative to be considered is extension of the $LR(k)$ algorithm to generalized $LR(k)$ à la Tomita. If a Chomsky Normal Form for GILs is obtainable, a CYK bottom-up algorithm could be considered.

We considered some of the issues related to Natural Language from a formal and abstract point of view, i.e. language sets and structural descriptions. In the same perspective it might be interesting to see if GILs are able to describe some phenomena that are beyond mildly-context sensitive power (e.g. scrambling). Also, we only compared GILs with LIGs; there are other formalisms to be considered. Minimalist Grammars [78] share some common properties with GILs. It would be interesting to determine if minimalist grammars can be compiled into GILs.

In the area of practical applications, it would be interesting to implement the HPSG compilation ideas. Another relevant question is how much of TAG systems can be compiled into GIGs and how do the performances compare. It should be noted that the capabilities of TAGs to define *extended domains of locality* might be lost in the compilation process.

We mentioned in the introduction some of the approaches to use generative grammars as models of biological phenomena. The capabilities of GIGs to generate the multiple copies language (or multiple repeats in the biology terminology) as well as crossing dependencies (or pseudo-knots) might be used in the computational analysis of sequence data.

Bibliography

[1] A. V. Aho. Indexed grammars - an extension of context-free grammars. *Journal of the Association for Computing Machinery*, 15(4):647–671, 1968.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] M. A. Alonso, D. Cabrero, and M. Vilares. Construction of efficient generalized LR parsers. In Derick Wood and Sheng Yu, editors, *Automata Implementation*, volume 1436, pages 7–24. Springer-Verlag, Berlin-Heidelberg-New York, 1998.

[4] M. A. Alonso, E. de la Clergerie, J. Graña, and M. Vilares. New tabular algorithms for LIG parsing. In *Proc. of the Sixth Int. Workshop on Parsing Technologies (IWPT'2000)*, pages 29–40, Trento, Italy, 2000.

[5] J. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 1, Word Language Grammar, pages 111–174. Springer-Verlag, Berlin, 1997.

[6] G. Barton, R. Berwick, and E. Ristad. *Computational Complexity and Natural Language*. MIT Press, Cambridge, 1987.

[7] T. Becker. *HyTAG: a new type of Tree Adjoining Grammars for Hybrid Syntactic Representation of Free Order Languages*. PhD thesis, University of Saarbruecken, 1993.

[8] T. Becker and P. Lopez. Adapting HPSG-to-TAG compilation to wide-coverage grammars. In *Proceedings of TAG+5*, pages 47–54. 2000.

[9] R. Book. Confluent and other types of thue systems. *J. Assoc. Comput. Mach.*, 29:171–182, 1982.

[10] P. Boullier. A generalization of mildly context-sensitive formalisms. In *TAG Workshop, Philadelphia*, 1998.

[11] P. Boullier. Chinese numbers, mix, scrambling, and range concatenation grammars. Research Report RR-3614, INRIA, Rocquencourt, France, January 1999. 14 pages.

[12] P. Boullier. Range concatenation grammars. In *Proceedings of the Sixth International Workshop on Parsing Technologies (IWPT2000)*, pages 53–64, Trento, Italy, February 2000.

[13] G. Buntrock and K. Lorys. On growing context-sensitive languages. In *Automata, Languages and Programming*, pages 77–88, 1992.

[14] G. Buntrock and K. Lorys. The variable membership problem: Succinctness versus complexity. In *Symposium on Theoretical Aspects of Computer Science*, pages 595–606, 1994.

[15] G. Buntrock and G. Niemann. Weakly growing context-sensitive grammars. *Chicago Journal of Theoretical Computer Science*, 1996.

[16] G. Buntrock and F. Otto. Growing context sensitive languages and church-rosser languages. In *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science*, volume 900, 1995.

[17] J. Castaño. GIGs: Restricted context-sensitive descriptive power in bounded polynomial-time. In *Proc. of Cicling 2003, Mexico City, February 16-22*, 2003.

[18] J. Castaño. LR Parsing for Global Index Languages (GILs). In *In Proceeding of CIAA 2003, Santa Barbara,CA.*, 2003.

[19] J. Castaño. On the applicability of global index grammars. In Sandra K
übler Kotaro Funakoshi and Jahna Otterbacher, editors, *Proceedings of the ACL-2003 Student Research Workshop*, pages 15–22, 2003.

[20] J. Castaño. Global index grammars and descriptive power. In R. Oehrle and J. Rogers, editors, *Proc. of Mathematics of Language, MOL 8*, 2003b. Bloomington, Indiana, June.

[21] A. Cherubini, L. Breveglieri, C. Citrini, and S. Reghizzi. Multipushdown languages and grammars. *International Journal of Foundations of Computer Science*, 7(3):253–292, 1996.

[22] N. Chomsky and M.-P. Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 118–161. North-Holland, Amsterdam, The Netherlands, 1963.

[23] C. Culy. The complexity of the vocabulary of bambara. *Linguistics and Philosophy*, 8:345–351, 1985.

[24] E. Dahlhaus and M. K. Warmuth. Membership for growing context sensitive grammars is polynomial. In *Colloquium on Trees in Algebra and Programming*, pages 85–99, 1986.

[25] J. Dassow and G. Păun. *Regulated Rewriting in Formal Language Theory.* Springer, Berlin, Heidelberg, New York, 1989.

[26] J. Dassow, G. Păun, and A. Salomaa. Grammars with controlled derivations. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Vol. 2.* Springer, Berlin,, 1997.

[27] J. Earley. An Efficient Context-free Parsing Algorithm. *Communications of the ACM*, 13:94–102, 1970.

[28] G. Pitsch G. Hotz. On parsing coupled-context-free languages. *Theor. Comp. Sci*, pages 205–233, 1996.

[29] G. Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94. D. Reidel, Dordrecht, 1988.

[30] S. Ginsburg. *The Mathematical Theory of Context Free Languages.* Mc Graw Hill, New York, 1966.

[31] S. Ginsburg and M. Harrison. One-way nondeterministic real-time list-storage. *Journal of the Association for Computing Machinery*, 15, No. 3:428–446, 1968.

[32] S. Ginsburg and E. H. Spanier. Finite-turn pushdown automata. *SIAM Journal on Control*, 4(3):429–453, 1966.

[33] S. Ginsburg and E. H. Spanier. AFL with the semilinear property. *Journal of Computer and System Sciences*, 5(4):365–396, 1971.

[34] A. Groenink. Literal movement grammars. In *Proceedings of the 7th EACL Conference, University College, Dublin, 1995.* 1995.

[35] A. Groenink. Mild context-sensitivity and tuple-based generalizations of context-free grammar. In *104*, page 22. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 30 1996.

[36] K. Harbush. Parsing contextual grammars with linear, regular and context-free selectors. In Martin-Vide and V. Mitrana, editors, *Words, sequences, languages: where computer science, biology and linguistics meet*. Springer, Berling, New York, Tokio, 2000.

[37] T. Harju, O. Ibarra, J. Karhumäki, and A. Salomaa. Decision questions concerning semilinearity, morphisms and commutation of languages. In *LNCS 2076*, page 579ff. Springer, 2001.

[38] H. Harkema. A recognizer for minimalist grammars. In *IWPT 2000*, 2000.

[39] M. H. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1978.

[40] T. Head. Formal language theory and dna: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, pages 737–759, 1987.

[41] J. E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[42] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM (JACM)*, 25(1):116–133, 1978.

[43] A. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural description? In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural language processing: psycholinguistic, computational and theoretical perspectives*, pages 206–250. Chicago University Press, New York, 1985.

[44] A. Joshi. Relationship between strong and weak generative power of formal systems. In *Proceedings of the Fifth International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+5)*, pages 107–114, Paris, France, 2000.

[45] A. Joshi and Y. Schabes. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Vol. 3*. Springer, Berlin, 1997.

[46] A. Joshi, K. Vijay-Shanker, and D. Weir. The convergence of mildly context-sensitive grammatical formalisms. In Peter Sells, Stuart Shieber, and Thomas Wasow, editors, *Foundational issues in natural language processing*, pages 31–81. MIT Press, Cambridge, MA, 1991.

[47] Laura Kallmeyer. Local tree description grammars: A local extension of tag allowing underspecified dominance relations. *Grammars*, pages 85–137, 2001.

[48] R. Kasper, B. Kiefer, K. Netter, and K. Vijay-Shanker. Compilation of HPSG into TAG. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 92–99. Cambridge, Mass., 1995.

[49] N. A. Khabbaz. A geometric hierarchy of languages. *Journal of Computer and System Sciences*, 8(2):142–157, 1974.

[50] Dan Klein and Christopher D. Manning. Parsing and hypergraphs. In *Proceedings of the 7th International Workshop on Parsing Technologies*. 2001.

[51] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages and Programming, 2nd Colloquium*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269, Saarbrücken, 1974.

[52] Carlos Martín-Vide and et al. Sewing contexts and mildly context-sensitive languages. *TUCS Technical Report No. 257*, March 1999.

[53] R. McNaughton. An insertion into the chomsky hierarchy? In *Jewels are Forever*, pages 204–212, 1999.

[54] J. Michaelis. Derivational minimalism is mildly context-sensitive. In *Proceedings of Logical Aspects of Computational Linguistics, LACL'98. Grenoble*, 1998.

[55] J. Michaelis and M. Kracht. Semilinearity as a syntactic invariant. In Christian Retoré, editor, *LACL'96: First International Conference on Logical Aspects of Computational Linguistics*, pages 329–345. Springer-Verlag, Berlin, 1997.

[56] P. Miller. *Strong Generative Capacity*. CSLI Publications, Stanford University, Stanford CA, USA, 1999.

[57] G. Niemann and F. Otto. The church-rosser languages are the deterministic variants of the growing context-sensitive languages. In *Foundations of Software Science and Computation Structure*, pages 243–257, 1998.

[58] F. Otto. On the connections between rewriting and formal language theory. In *Rewriting Techniques and Applications: 10th International Conference, RTA-99, Trento, Italy, LNCS 1631*. Springer-Verlag, 1999.

[59] M.A. Alonso Pardo. *Interpretación Tabular de Autómatas para lenguajes de Adjunción de Árboles*. PhD thesis, Universidade da Coruña, 2000.

[60] Rohit J. Parikh. On context-free languages. *Journal of the Association for Computing Machinery*, 13:570–581, 1966.

[61] C. Pollard and I. A. Sag. *Head-driven Phrase Structure Grammar*. University of Chicago Press, Chicago, IL, 1994.

[62] S. Rajasekaran. Tree-adjoining language parsing in $o(n^6)$ time. *SIAM J. of Computation*, 25, 1996.

[63] O. Rambow. *Formal and Computational Aspects of Natural Language Syntax*. PhD thesis, University of Pennsylvania, Dept of CIS, Philadelphia, 1994.

[64] E. Rivas and S.R. Eddy. A dynamic programming algorithm for rna structure prediction including pseudoknots. *J. Mol Biol*, pages 2053–2068, 1999.

[65] E. Rivas and S.R. Eddy. The language of rna: A formal grammar that includes pseudoknots. *Bioinformatics*, pages 334–340, 2000.

[66] Gh. Paun S. Marcus, C. Martin-Vide. Contextual grammars as generative models of natural languages. *Computational Linguistics*, pages 245–274, 1998.

[67] G. Satta. Tree-adjoining grammar parsing and boolean matrix multiplication. *Computational linguistics*, 20, No. 2, 1994.

[68] D. B. Searls. String variable grammar: A logic grammar formalism for the biological language of DNA. *Journal of Logic Programming*, 24(1 and 2):73–102, 1995.

[69] D.B. Searls. The computational linguistics of biological sequences. In L. Hunter, editor, *Artificial Intelligence and Molecular Biology*, pages 47–120. AAAI Press, 1993.

[70] D.B. Searls. Formal language theory and biological macromolecules. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, page 117ff, 1999.

[71] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer. Science*, pages 191–229, 1991.

[72] S. Shieber, Y. Schabes, and F. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24:3–36, 1995.

[73] S.M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343, 1985.

[74] S.M. Shieber and Y. Schabes. An alternative conception of tree-adjoining derivation. *Computational Linguistics*, 20:91–124, 1994.

[75] K. Sikkel. *Parsing schemata.* Springer-Verlag, 1997.

[76] K. Sikkel. Parsing schemata and correctness of parsing algorithms. *Theoretical Computer Science*, 199(1–2):87–103, 1998.

[77] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, Massachusetts, 1997.

[78] E. P. Stabler. Derivational minimalism. *Lecture notes in computer science*, 1328, 1997.

[79] M. Steedman. Dependency and coordination in the grammar of dutch and english. *Language*, pages 523–568, 1985.

[80] M. Tomita. An efficiente augmented-context-free parsing algorithm. *Computational linguistics*, 13:31–46, 1987.

[81] E. V. de la Clergerie and M. A. Pardo. A tabular interpretation of a class of 2-stack automata. In *COLING-ACL*, pages 1333–1339, 1998.

[82] K. Vijay-Shanker, D. J. Weir, and A. K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *Proc. of the 25th ACL*, pages 104–111, Stanford, CA, 1987.

[83] D. A. Walters. Deterministic context-sensitive languages: Part II. *Information and Control*, 17:41–61, 1970.

[84] C. Wartena. Grammars with composite storages. In *Proceedings of the Conference on Logical Aspects of Computational Linguistics (LACL '98)*, pages 11–14, Grenoble, 1998.

[85] Ch. Wartena. On the concatenation of one-turn pushdowns. *Grammars*, pages 259–269, 1999.

[86] D. Weir. *Characterizing mildly context-sensitive grammar formalisms.* PhD thesis, University of Pennsylvania, 1988.

[87] D. J. Weir. A geometric hierarchy beyond context-free languages. *Theoretical Computer Science*, 104(2):235–261, 1992.

[88] D. A. Workman. Turn-bounded grammars and their relation to ultralinear languages. *Information and Control*, 32 No. 2:188–200, 1976.