Current Issues in Parsing Technology,
M. Tomita ed., Kluwer Academic Publishers,
1991, pp 153-171.

Towards a Uniform Formal Framework for Parsing

Bernard Lang
INRIA
B.P. 105, 78153 Le Chesnay, France
lang@inria.inria.fr

1 Introduction

Many of the formalisms used to define the syntax of natural (and programming) languages may be located in a continuum that ranges from propositional Horn logic to full first order Horn logic, possibly with non-Herbrand interpretations. This structural parenthood has been previously remarked: it lead to the development of Prolog [Col-78, Coh-88] and is analyzed in some detail in [PerW-80] for Context-Free languages and Horn Clauses. A notable outcome is the parsing technique known as Earley deduction [PerW-83].

These formalisms play (at least) three roles:

descriptive: they give a finite and organized description of the syntactic structure of the language,

analytic: they can be used to analyze sentences so as to retrieve a syntactic structure (i.e. a representation) from which the meaning can be extracted,

generative: they can also be used as the specification of the concrete representation of sentences from a more structured abstract syntactic representation (e.g. a parse tree).

The choice of a formalism is essential with respect to the descriptive role, since it controls the perspicuity with which linguistic phenomena may be understood and expressed in actual language descriptions, and hence the tractability of these descriptions for the human mind.

However, computational tractability is required by the other two roles if we intend to use these descriptions for mechanical processing of languages.

The aim of our work, which is partially reported here, is to obtain a uniform understanding of the computational aspects of syntactic phenomena within the continuum of Horn-like formalisms considered above, and devise general purpose algorithmic techniques to deal with these formalisms in practical applications.

To attain this goal, we follow a three-sided strategy:

- Systematic study of the lower end of the continuum, represented by context-free (CF) grammars (simpler formalisms, such as propositional Horn logic do not seem relevant for our purpose).
- Systematic study of the higher end of the continuum, i.e. first order Horn clauses,
- Analysis of the relations between intermediate formalisms and Horn clauses, so as to reuse for intermediate formalisms the understanding and algorithmic solutions developed for the more powerful Horn clauses.

This strategy is motivated by two facts:

- the computational properties of both CF grammars and Horn clauses may be expressed with the same computational model: the non-deterministic pushdown automaton,
- the two formalisms have a compatible concept of syntactic structure: the parse-tree in the CF case, and the proof-tree in the Horn clause case.

CThe greater simplicity of the CF formalism helps us in understanding more easily most of the computational phenomena. We then generalize this knowledge to the more powerful Horn clauses, and finally we specialize it from Horn clauses to the possibly less powerful but linguistically more perspicuous intermediate formalisms.

In this chapter we present two aspects of our work:

1. a new understanding of shared parse forests and their relation to CF grammars, and
2. a generalization to full Horn clauses, also called *Definite Clause (DC) programs*, of the push-down stack computational model developed for CF parsers.

2 Context-Free Parsing

Though much research has been devoted to this subject in the past, most of the practically usable work has concentrated on deterministic push-down parsing which is clearly inadequate for natural language applications and does not generalize to more complex formalisms. On the other hand there has been little formal investigation of general CF parsing, though many practical systems have been implemented based on some variant of Earley's algorithm.

Our contribution has been to develop a formal model which can describe these variants in a uniform way, and encompasses the construction of parse-trees, and more generally of parse-forests. This model is based on the *compilation paradigm* common in programming languages and deterministic parsing: we use the non-deterministic[1] *Pushdown Automaton (PDA)* as a virtual parsing machine which we can simulate with an Earley-like construction; variations on Earley's algorithm are then expressed as variations in the compilation schema used to produce the PDA code

[1]In this chapter, the abbreviation PDA always implies the possibility of non-determinism

```
(1)   S    ::=  NP    VP
(2)   S    ::=  S     PP
(3)   NP   ::=  n
(4)   NP   ::=  det   n
(5)   NP   ::=  NP    PP
(6)   PP   ::=  prep  NP
(7)   VP   ::=  v     NP
```

Figure 1: A Context-Free Grammar

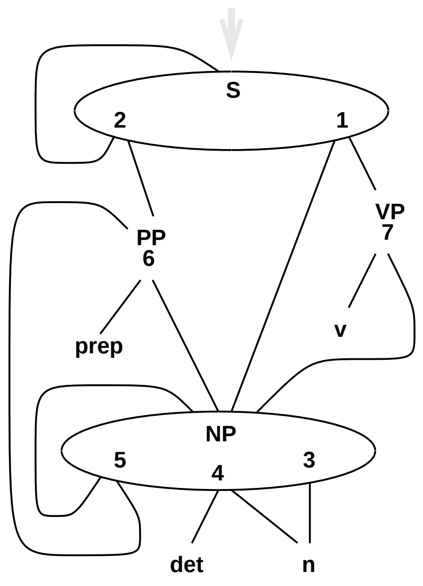

Figure 2: Graph of the Grammar

from the original CF grammar[2]. This uniform framework has been used to compare experimentally parsing schemata w.r.t. parser size, parsing speed and size of shared forest, and in reusing the wealth of PDA construction techniques to be found in the literature.

This work has been reported elsewhere [Lan-74, BilL-89, Lan-88a]. An essential outcome, which is the object of this section, is a new understanding of the relation between CF grammars, parse-trees and parse-forests, and the parsing process itself. The presentation is informal since our purpose is to give an intuitive understanding of the concepts, which is our interpretation of the earlier theoretical results.

Essentially, we shall first show that both CF grammars and shared parsed forest may be represented by AND-OR graphs, with specific interpretations. We shall then argue that this representational similarity is not accidental, and that there is no difference between a shared forest and a grammar.

2.1 Context-free Grammars

Our running example for a CF grammar is the pico-grammar of English, taken from [Tom-87], which is given in figure 1.

In figure 2 we give a graphical representation of this grammar as an AND-OR graph. The notation for this AND-OR graph is unusual and emphasizes the difference between AND and OR nodes. OR-nodes are represented by the non-terminal categories of the grammar, and AND-nodes

[2]Many variants of Earley's algorithm published in the literature [BouPS-75, Tom-87], including Earley's own [Ear-70], could be viewed as special cases of that approach.

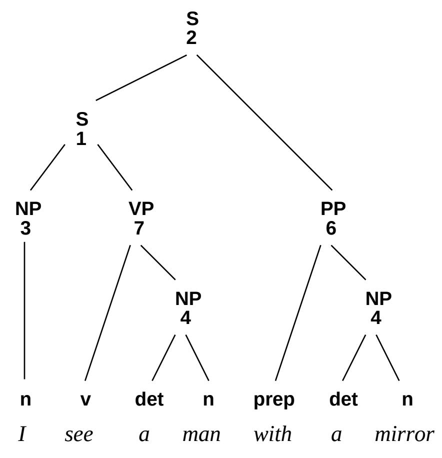

Figure 3: A parse tree

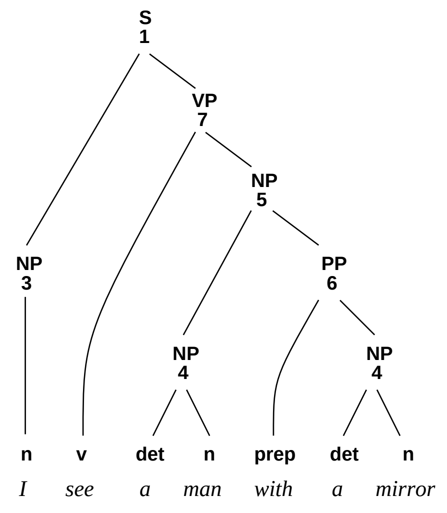

Figure 4: Another parse tree

are represented by the rules (numbers) of the grammar. There are also leaf-nodes corresponding to the terminal categories.

The OR-node corresponding to a non-terminal `X` has exiting arcs leading to each AND-node `n` representing a rule that defines `X`. This arc is not explicitly represented in the graphical formalism chosen. If there is only one such arc, then it is represented by placing `n` immediately under `X`. This is the case for the OR-node representing the non-terminal `PP`. If there are several such arcs, they are implicitly represented by enclosing in an ellipse the OR-node `X` above all its son nodes `n`, `n'`, ... This is the case for the OR-node representing the non-terminal `NP`.

The sons of an AND-node (i.e. a rule) are the grammatical categories found in the right-hand-side of the rule, *in that order*. The arcs leading from an AND-node to its sons are represented explicitly. The convention for orienting the arcs is that they leave a node from below and reach a node from above.

This graph accurately represents the grammar, and is very similar to the graphs used in some parsers. For example, LR(0) parsing uses a graph representation of the grammar that is very similar, the main difference being that the sons of AND-nodes are linked together from left to right, rather than being attached separately to the AND-node [AhoU-72, DeR-71]. More simply, this graph representation is very close to the data structures often used to represent conveniently a grammar in a computer memory.

A characteristic of the AND/OR graph representing a grammar is that all nodes have different labels. Conversely, any labelled AND/OR graph *such that all node labels are different* may be read as — translated into — a CF grammar such that AND-node labels are rule names, OR-node labels represent non-terminal categories, and leaf-node labels represent terminal categories.

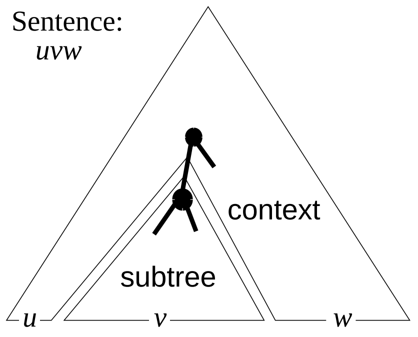

Figure 5: Context and Subtree

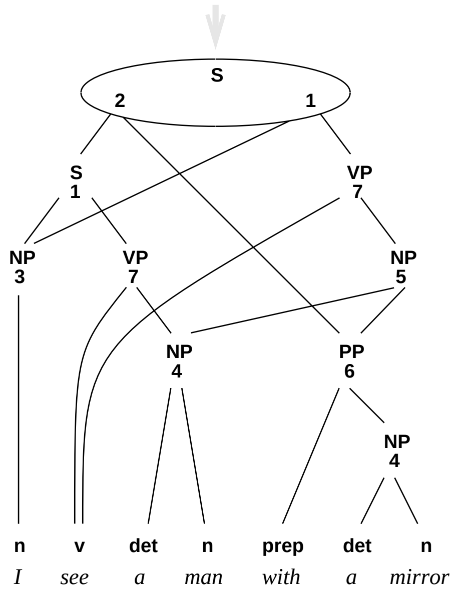

Figure 6: A shared parse forest

2.2 Parse trees and parse forests

Given a sentence in the language defined by a CF grammar, the parsing process consists in building a tree structure, *the parse tree*, that shows how this sentence can be constructed according to the grammatical rules of the language. It is however frequent that the CF syntax of a sentence is ambiguous, i.e. that several distinct parse-trees may be constructed for it.

Let us consider the grammar of figure 1.

If we take as example the sentence "`I see a man with a mirror`", which translate into the terminal sequence "`n v det n prep det n`", we can build the two parse trees given in figures 3 and 4. Note that we label a parse tree node with its non-terminal category and with the rule used to decompose it into constituents. Hence such a parse tree could be seen as an AND-OR tree similar to the AND-OR grammar graph of figure 2. However, since all OR-nodes are degenerated (i.e. have a unique son), a parse tree is just an AND-tree.

The number of possible parse trees may become very large when the size of sentences increases: it may grow exponentially with that size, and may even be infinite for cyclic grammars (which seem of little linguistic usefulness [PerW-83, Tom-85], except may-be for analyzing ill-formed sentences [Lan-88a]). Since it is often desirable to consider all possible parse trees (e.g. for semantic processing), it is convenient to merge as much as possible these parse trees into a single structure that allows them to share common parts. This sharing save on the space needed to represent the trees, and also on the later processing of these trees since it may allows to share between two trees the processing of some common parts[3]. The shared representation of all parse trees is called *shared*

[3]The direct production of such shared representation by parsing algorithms also corresponds to sharing in the parsing computation [Lan-74, Tom-87, BilL-89].

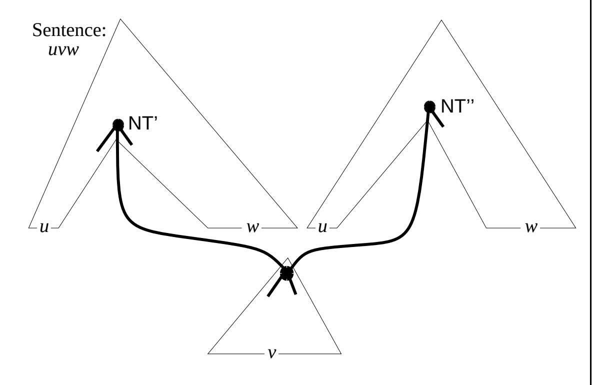

Figure 7: Two parses sharing a subtree

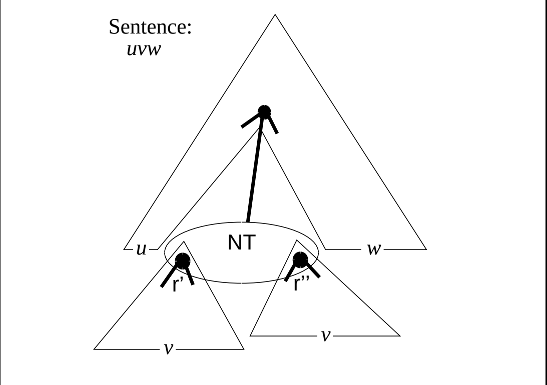

Figure 8: Two parses sharing a context

parse forest, or just *parse forest*.

To analyze how two trees can share a (*connected*) part, we first notice that such a part may be isolated by cutting the tree along an edge (or arc) as in figure 5. this actually give us two parts: a *subtree* and a *context* (cf. figure 5). Either of these two parts may be shared in forests representing two trees. When a subtree is the same for two trees, it may be shared as shown in figure 7. When contexts are equal and may thus be shared, we get the structure depicted in figure 8.

The sharing of context actually corresponds to ambiguities in the analyzed sentence: the ellipse in figure 8 contains the head nodes for two distinct parses of the same subsentence v, that both recognize v in the same non-terminal category `NT`. Each head node is labelled with the (number of the) rule used to decompose v into constituents in that parse, and the common syntactical category labels the top of the ellipse. Not accidentally, this structure is precisely the structure of the OR-nodes we used to represent CF grammars. Indeed, an ambiguity is nothing but a choice between two possible parses of the same sentence fragment v as the same syntactic category `NT`.

Using a combination of these two forms of sharing, the two parse trees of figures 3 and 4 may be merged into the shared parse forest[4] of figure 6. Note that, for this simple example, the only context being shared is the empty outer context of the two possible parse tree, that still states that a proper parse tree must belong to the syntactic category `S`.

In this representation we keep our double labelling of parse tree nodes with both the non-terminal category and the rule used to decompose it into its constituents. As indicated above, ambiguities are represented with context sharing, i.e. by OR-nodes that are the exact equivalent of those of figure 2. Hence *a shared parse forest is an AND-OR graph*[5]. Note however that the same rule (resp. non-terminal) may now label several AND-nodes (resp. OR-nodes) of the shared parse forest graph.

If we make the labels distinct, for example by indexing them so as not to lose their original

[4]This graphical representation of shared forests is not original: to our knowledge it was first used by Tomita [Tom-87]. However, we believe that its comparative understanding as context sharing, as AND-OR tree or as grammar has never been presented. Context sharing is called *local ambiguity packing* by Tomita.

[5]This graph may have cycles for infinitely ambiguous sentences when the grammar of the language is itself cyclic.

information, we can then read the shared forest graph of a sentence s as a grammar $\mathcal{F}_s$. The language of this grammar contains only the sentence s, and it gives s the same syntactic structure(s) — i.e. the same parse tree(s) and the same ambiguities — as the original grammar, up to the above renaming of labels.

It is easily shown that the space complexity of the shared forest representation is $\mathcal{O}(n^{p+1})$ where n is the length of the sentence analyzed and p is the length of the longest right-hand side of a rule. Hence, efficient representation requires a trivial transformation of the original grammar into a *2-size grammar*, i.e. a grammar such that the right-hand sides of rules have a length at most equal to 2. This generalizes/simplifies the use of *Chomsky Normal Form* [Pra-75] or *2-form* [She-76]. Such a transformation amounts effectively to allowing two forest nodes to share a common sublist of their respective lists of sons.

This type of transformation is explicitly or implicitly performed/required by parsing algorithms that have $\mathcal{O}(n^3)$ complexity. For example in [Lan-74, BilL-89], the transformation is implicit in the construction of the PDA transitions.

2.3 Parse forests for incomplete sentences

Our view of parsing may be extended to the parsing of incomplete sentences [Lan-88a].

An example of incomplete sentence is "`...  see ...  mirror`". Assuming that we know that the first hole stands for a single missing word, and that the second one stands for an arbitrary number of words, we can represent this sentence by the sequence "`?  v * n`". The convention is that "`?`" stands for one unknown word, and "`*`" for any number of them.

Such an incomplete sentence s may be understood as defining a sublanguage $\mathcal{L}_s$ which contains all the correct sentences matching s. Any parse tree for a sentence in that sublanguage may then be considered a possible parse tree for the incomplete sentence s. For example, the sentences "`I see a man with a mirror`" and "`You see a mirror`" are both in the sublanguage of the incomplete sentence above. Consequently, the two parse trees of figures 3 and 4 are possible parse trees for this sentence, along with many others.

All parse trees for the sentence s = "`?  v * n`" may be merged into a shared parse forest that is represented in figure 9.

The graph of this forest has been divided into two parts by the horizontal grey line $\alpha - \beta$.

The terminal labels underscored with a "`*`" represent any word in the corresponding terminal category. This is also true for all the terminal labels in the bottom part of the graph.

The forest fragment below the horizontal line is a (closed) subgraph of the original grammar of figure 2 (which we have completed in grey to emphasize the fact). It corresponds to parse trees of constituents that are completely undefined, within their syntactical categories, and may thus be any tree in that category that the grammar can generate. This occurs once in the forest for non-terminal `PP` at arc marked α and twice for `NP` at arcs marked β.

This bottom part of the graph brings no new information (it is just the part of the original grammar reachable from nodes `PP` and `NP`). Hence the forest could be simplified by eliminating this bottom subgraph, and labelling the end node of the α (resp. β) arc with `PP*` (resp. `NP*`), meaning an arbitrary `PP` (resp. `NP`) constituent.

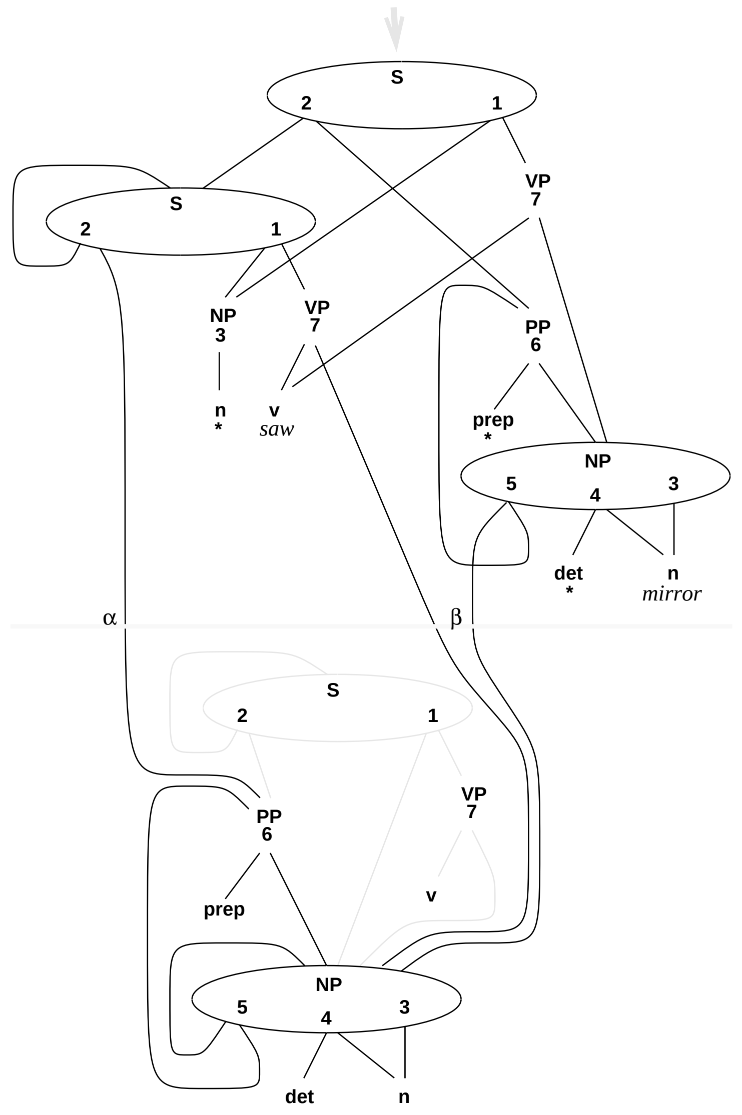

Figure 9: Full parse forest for an incomplete sentence

The complete shared forest of figure 6 may be interpreted as a CF grammar $\mathcal{G}_s$. This grammar is precisely a grammar of the sublanguage $\mathcal{L}_s$ of all sentences that match the incomplete sentence s. Again, up to renaming of nonterminals, this grammar $\mathcal{G}_s$ gives the sentences in $\mathcal{L}_s$ the same syntactic structure as the original grammar of the full language.

If the sentence parsed is the completely unknown sentence u = "*", then the corresponding sublanguage $\mathcal{L}_u$ is the complete language considered, and the parse forest for u can be quite naturally the original grammar of the full language: *The grammar of a CF language is a parse-forest of the completely unknown sentence, i.e. the syntactic structure of all sentences in the language, in a non-trivial sense.* In other words, all one can say about a fully unknown sentence assumed to be correct, is that it satisfies the syntax of the language. This statement does take a stronger signification when shared parse forests are actually built by parsers, and when such a parser does return the original grammar for the fully unknown sentence[6].

Parsing a sentence according to a CF grammar is just extracting a parse tree (or a sub-forest) fitting that sentence from the CF grammar considered as a parse forest.

Looking at these issues from another angle, we have the following consequence of the above discussion: given a set of parse trees (i.e. appropriately decorated trees), they form the set of parses of a CF language iff they can be merged into a shared forest that is finite.

In [BilL-89, Lan-88a] Billot and the author have proposed parsers that actually build shared forests formalized as CF grammar. This view of shared forests originally seemed to be an artifact of the formalization chosen in the design of these algorithms, and appeared possibly more obfuscatory than illuminating. It has been our purpose here to show that it really has a fundamental character, *independently of any parsing algorithm.*

This close relation between sharing structures and context-freeness actually hints to limitations of the effectiveness of sharing in parse forests defined by non-CF formalisms.

From an algorithmic point of view, the construction of a shared forest for a (possibly incomplete) sentence may be seen as a specialization of the original grammar to the sublanguage defined by that sentence. This shows interesting connections with the general theory of partial evaluation of programs [Fut-88], which deals with the specialization of programs by propagation of known properties of their input.

In practice, the published parsing algorithms do not always give shared forest with maximum sharing. This may result in forests that are larger or more complex, but does not invalidate our presentation.

3 Horn Clauses

The PDA based compilation approach proved itself a fruitful theoretical and experimental support for the analysis and understanding of general CF parsing à la Earley. In accordance with our strategy of uniform study of the "Horn continuum", we extended this approach to general Horn

[6]However, if it does not produce optimal sharing in the forest, the parser may return a structurally equivalent, but larger grammar.

clauses, i.e. Definite Clause (DC) programs.

This lead to the definition of the *Logical Push-Down Automaton (LPDA)* which is an operational engine intended to play for Horn clauses the same role as the usual PDA for CF languages. Space limitations prevent giving here a detailed presentation of LPDAs, and we only sketch the underlying ideas. More details may be found in [Lan-88b, Lan-88c].

As in the CF case, the evaluation of a DC program may be decomposed into two phases:

- a compilation phase that translate the DC program into a LPDA. Independently of the later execution strategy, the compilation may be done according to a variety of evaluation schemata: top-down, bottom-up, predictive bottom-up, ... Specific optimization techniques may also be developed for each of these compilation schemata.
- an execution phase that can interpret the LPDA according to some execution technique: backtrack (depth-first), breadth-first, dynamic programming, or some combination [TamS-86].

This separation of concerns leads to a better understanding of issues, and should allow a more systematic comparison of the possible alternatives.

In the case of dynamic programming execution, the LPDA formalism uses to very simple structures that we believe easier to analyze, prove, and optimize than the corresponding direct constructions on DC programs [PerW-83, Por-86, TamS-86, Vie-87b], while remaining independent of the computation schema, unlike the direct constructions. Note that predictive bottom-up compilation followed by dynamic programming execution is essentially equivalent to Earley deduction as presented in [PerW-83, Por-86].

The next sections include a presentation of LPDAs and their dynamic programming interpretation, a compilation schema for building a LPDA from a DC program, and an example applying this top-down construction to a very simple DC program.

3.1 Logical PDAs and their dynamic programming interpretation

A LPDA is essentially a PDA that stores logical atoms (i.e. predicates applied to arguments) and substitutions on its stack, instead of simple symbols. The symbols of the standard CF PDA stack may be seen as predicates with no arguments (or more accurately with two argument similar to those used to translate CF grammars into DC in [PerW-80]). A technical point is that we consider PDAs without "finite state" control: this is possible without loss of generality by having pop transitions that replace the top two atoms by only one (this is standard in LR(k) PDA parsers[AhoU-72]).

Formally a LPDA $\mathcal{A}$ is a 6-tuple: $\mathcal{A} = (\mathbf{X}, \mathbf{F}, \mathbf{\Delta}, \overset{\circ}{\$}, \$_f, \mathbf{\Theta})$
where $\mathbf{X}$ is a set of variables, $\mathbf{F}$ is a set of functions and constants symbols, $\mathbf{\Delta}$ is a set of stack predicate symbols, $\overset{\circ}{\$}$ and $\$_f$ are respectively the initial and final stack predicates, and $\mathbf{\Theta}$ is a finite set of *transitions* having one of the following three forms:

horizontal transitions:	$B \mapsto C$	– – replace B by C on top of stack
push transitions:	$B \mapsto CB$	– – push C on top of former stack top B
pop transitions:	$BD \mapsto C$	– – replace BD by C on top of stack

where B, C and D are $\mathbf{\Delta}$-atoms, i.e. atoms built with $\mathbf{\Delta}$, **F** and **X**.

Intuitively (and approximately) a pop transition $\mathrm{BD} \mapsto \mathrm{C}$ is applicable to a stack configuration with atoms A and A′ on top, iff there is a substitution s such that $\mathrm{B}s = \mathrm{A}s$ and $\mathrm{D}s = \mathrm{A}'s$. Then A and A′ are removed from the stack and replaced by $\mathrm{C}s$, i.e. the atom C to which s has been applied. Things are similar for other kinds of transitions. Of course a LPDA is usually non-deterministic w.r.t. the choice of the applicable transition.

In the case of dynamic programming interpretations, all possible computation paths are explored, with as much sub-computation sharing as possible. The algorithm proceeds by building a collection of *items* (analogous to those of Earley's algorithm) which are pairs of atoms. An item $<\mathrm{A}\ \mathrm{A}'>$ represents a stack fragment of two consecutive atoms [Lan-74, Lan-88a]. If another item $<\mathrm{A}'\ \mathrm{A}''>$ was also created, this means that the sequence of atoms $\mathrm{AA}'\mathrm{A}''$ is to be found in some possible stack configuration, and so on (*up to the use of substitutions, not discussed here*). The computation is initialized with an initial item $\overset{\circ}{U}= <\ \overset{\circ}{\$}\ \dashv\ >$. New items are produced by applying the LPDA transitions to existing items, until no new application is possible (an application may often produce an already existing item). The computation terminates under similar conditions as specialized algorithms [PerW-83, TamS-86, Vie-87b]. If successful, the computation produces one or several *final items* of the form $<\$_\mathrm{f}\ \ \overset{\circ}{\$}\ >$, where the arguments of $\$_\mathrm{f}$ are an answer substitution of the initial DC program. In a parsing context, one is usually interested in obtaining parse-trees rather than "answer substitutions". The counterpart of CF a parse tree is here a proof tree corresponding to proofs with the original DC program. Such proof trees may be obtained by the same techniques that are used in the case of CF parsing [Lan-74, BilL-89, Bil-88], and that actually interpret the items and their relations as a shared parse forest structure.

Substitutions are applied to items as follows (we give as example the most complex case): a pop transition $\mathrm{BD} \mapsto \mathrm{C}$ is applicable to a pair of items $<\mathrm{A}\ \mathrm{A}'>$ and $<\mathrm{E}\ \mathrm{E}'>$, iff there is a unifier s of $<\mathrm{A}\ \mathrm{A}'>$ and $<\mathrm{B}\ \mathrm{D}>$, and a unifier s' of $\mathrm{A}'s$ and E. This produces the item $<\mathrm{C}ss'\ \mathrm{E}'s'>$.

3.2 Top-down compilation of DC programs into LPDAs

Given a DC program, *many different compilation schemata may be used to build a corresponding LPDA* [Lan-88c]. We give here a very simple and unoptimized top-down construction. The DC program to be compiled is composed of a set of clauses γ_k: $\mathrm{A}_{k,0}$:- $\mathrm{A}_{k,1}, \ldots, \mathrm{A}_{k,n_k}$, where each $\mathrm{A}_{k,i}$ is a logical literal. The query is assumed to be the head literal $\mathrm{A}_{0,0}$ of the first clause γ_0.

The construction of the top-down LPDA is based on the introduction of new predicate symbols $\triangledown_{k,i}$, corresponding to positions between the body literals of each clause γ_k. The predicate $\triangledown_{k,0}$ corresponds to the position before the leftmost literal, and so on. Literals in clause bodies are refuted from left to right. The presence of an instance of a position literal $\triangledown_{k,i}(\mathrm{t}_k)$ in the stack indicates that the first i subgoals corresponding to the body of some instance of clause γ_k have already been refuted. The argument bindings of that position literal are the partial answer substitution computed by this partial refutation.

For every clause γ_k: $\mathrm{A}_{k,0}$:- $\mathrm{A}_{k,1}, \ldots, \mathrm{A}_{k,n_k}$, we note t_k the vector of variables occurring in the clause. Recall that $\mathrm{A}_{k,i}$ is a literal using some of the variables in γ_k, while $\triangledown_{k,i}$ is only a predicate which needs to be given the argument vector t_k to become the literal $\triangledown_{k,i}(\mathrm{t}_k)$.

Then we can define the top-down LPDA by the following transitions:

1. $\overset{\circ}{\$} \mapsto \nabla_{0,0}(\mathrm{t}_0)\ \overset{\circ}{\$}$
2. $\nabla_{k,i}(\mathrm{t}_k) \mapsto \mathrm{A}_{k,i+1}\, \nabla_{k,i}(\mathrm{t}_k)$ — *for every clause γ_k and for every position i in its body: $0 \leq i < n_k$*
3. $\mathrm{A}_{k,0} \mapsto \nabla_{k,0}(\mathrm{t}_k)$ — *for every clause γ_k*
4. $\nabla_{k,n_k}(\mathrm{t}_k)\, \nabla_{k',i}(\mathrm{t}_{k'}) \mapsto \nabla_{k',i+1}(\mathrm{t}_{k'})$[7] — *for every pair of clauses γ_k and $\gamma_{k'}$ and for every position i in the body of $\gamma_{k'}$: $0 \leq i < n_{k'}$*

The final predicate of the LPDA is the stack predicate ∇_{0,n_0} which corresponds to the end of the body of the first "query clause" of the DC program. The rest of the LPDA is defined accordingly.

The following is an informal explanation of the above transitions:

1. *Initialization*: We require the refutation of the body of clause γ_0, i.e. of the query.
2. *Selection of the leftmost remaining subgoal*: When the first i literals of clause γ_k have been refuted, as indicated by the position literal $\nabla_{k,i}(\mathrm{t}_k)$, then select the $i+1^{st}$ literal $\mathrm{A}_{k,i+1}$ to be now refuted.
3. *Selection of clause γ_k*: Having to satisfy a subgoal that is an instance of $\mathrm{A}_{k,0}$, eliminate it by resolution with the clause γ_k. The body of γ_k is now considered as a sequence of new subgoals, as indicated by the position literal $\nabla_{k,0}(\mathrm{t}_k)$.
4. *Return to calling clause $\gamma_{k'}$*: Having successfully refuted the head of clause γ_k by refuting successively all literals in its body as indicated by position literal $\nabla_{k,n_k}(\mathrm{t}_k)$, we return to the calling clause $\gamma_{k'}$ and "increment" its position literal from $\nabla_{k',i}(\mathrm{t}_{k'})$ to $\nabla_{k',i+1}(\mathrm{t}_{k'})$, since the body literal $\mathrm{A}_{k',i+1}$ has been refuted as instance of the head of γ_k.

Backtrack interpretation of a LPDA thus constructed essentially mimics the Prolog interpretation of the original DC program.

3.3 A very simple example

The following example has been produced with a prototype implementation realized by Eric Villemonte de la Clergerie and Alain Zanchetta [VilZ-88]. This example, as well as the top-down construction above, are taken from [Lan-88c].

The definite clause program to be executed is given in figure 11. Note that a search for all solutions in a backtrack evaluator would not terminate.

[7]If $k = k'$ then we rename the variable in $\mathrm{t}_{k'}$ since the transition corresponds to the use of two distinct variants of the clause γ_k.

Note also that we need not define such a transition for all triples of integer k k' and i, but only for those triples such that the head of γ_k unifies with the literal $\mathrm{A}_{k',i+1}$.

```
********* PUSH Transitions B->BC ***********

   predicate :nabla.2.0
nabla.2.0(X1) -> q(f(X1)) nabla.2.0(X1)

   predicate :nabla.0.0
nabla.0.0(X2) -> q(X2) nabla.0.0(X2)

   predicate :dollar0
dollar0() -> nabla.0.0(X2) dollar0()

********* Horizontal Transitions B->C ******

   predicate :q
q(f(f(a))) -> nabla.1.0()
q(X1) -> nabla.2.0(X1)

   predicate :query
query(X2) -> nabla.0.0(X2)

   predicate :nabla.0.1
nabla.0.1(X2) -> answer(X2)

********* POP Transitions BD->C ************

   predicate :nabla.2.1
nabla.2.1(X1) nabla.0.0(X2) -> nabla.0.1(X2)
nabla.2.1(X4) nabla.2.0(X1) -> nabla.2.1(X1)

   predicate :nabla.1.0
nabla.1.0() nabla.0.0(X2) -> nabla.0.1(X2)
nabla.1.0() nabla.2.0(X1) -> nabla.2.1(X1)

   predicate :nabla.0.1
nabla.0.1(X3) nabla.0.0(X2) -> nabla.0.1(X2)
nabla.0.1(X2) nabla.2.0(X1) -> nabla.2.1(X1)
```

Figure 10: Transitions of the LPDA.

```
Clauses:  q(f(f(a))):-.
          q(X1):-q(f(X1)).
Query:    q(X2)
```

Figure 11: The Definite Clause program.

```
dollar0() , ()()
nabla.0.0(X5) , dollar0()
q(X6) , nabla.0.0(X6)
nabla.2.0(X7) , nabla.0.0(X7)
nabla.1.0() , nabla.0.0(f(f(a)))
q(f(X8)) , nabla.2.0(X8)
nabla.0.1(f(f(a))) , dollar0()
nabla.2.0(f(X9)) , nabla.2.0(X9)
nabla.1.0() , nabla.2.0(f(a))
nabla.2.1(f(a)) , nabla.0.0(f(a))
nabla.0.1(f(a)) , dollar0()
q(f(f(X10))) , nabla.2.0(f(X10)) *
nabla.2.1(f(a)) , nabla.2.0(a)
nabla.2.1(a) , nabla.0.0(a)
nabla.0.1(a) , dollar0()
answer(a) , dollar0()
answer(f(a)) , dollar0()
answer(f(f(a))) , dollar0()
```

* subsumed by: q(f(X8)),nabla.2.0(X8)

Figure 12: Items produced by the dynamic programming interpretation.

The solutions found by the computer are:

```
X2 = f(f(a))
X2 = f(a)
X2 = a
```

These solutions were obtained by first compiling the DC program into an LPDA according to the schema defined in section 3.2, and then interpreting this LPDA with the general dynamic programming algorithm defined in section 3.1.

The LPDA transitions produced by the compilation are in figure 10. The collection of items produced by the dynamic programming computation is given in the figure 12.

In the transitions printout of figure 10, each predicate name `nabla.i.j` stands for our $\nabla_{i,j}$.

According to the construction of section 3.2, the final predicate should be `nabla.0.1`. For better readability we have added a horizontal transition to a final predicate noted `answer`.

4 Other linguistic formalisms

Pereira and Warren have shown in their classical paper [PerW-80] the link between CF grammars and DC programs. A similar approach may be applied to more complex formalisms than CF grammars, and we have done so for Tree Adjoining Grammars (TAG) [Lan-88d].

By encoding TAGs into DC programs, we can specialize to TAGs the above results, and easily build TAG parsers (using at least the general optimization techniques valid for all DC programs). Furthermore, control mechanisms akin to the agenda of chart parsers, together with some finer properties of LPDA interpretation, allow to control precisely the parsing process and produce Earley-like left-to-right parsers, with a complexity $O(n^6)$.

We expect that this approach can be extended to a variety of other linguistic formalisms, with or without unification of feature structures, such as head grammars, linear indexed grammars, combinatory categorial grammars. This is indeed suggested by the results of of Joshi, Vijay-Shanker and Weir that relate these formalisms and propose CKY or Earley parsers for some of them [VijWJ-87, VijW-89].

The parse forests built in the CF case correspond to proof forests in the Horn case. Such proof forests may be obtained by the same techniques that we used for CF parsing [BilL-89]. However it is not yet fully clear how parse trees or derivation trees may be extracted from the proof forest when DC programs are used to encode non-CF syntactic formalisms. On the basis of our experience with TAGs, we conjecture that for non-CF formalisms, the proof forest obtained corresponds to derivation forests (i.e. containing derivation trees as defined in [VijWJ-87]) rather than to forest representing the possible superficial syntactic structure of object trees.

5 Concluding Remarks

Our understanding of syntactic structures and parsing may be considerably enhanced by comparing the various approaches in similar formal terms, even though they may differ in power and/or superficial appearance. Hence we attempt to formally unify the problems in two ways:

— by considering all formalisms as special cases of Horn clauses

— by expressing all parsing strategies with a unique operational device: the pushdown automaton.

Systematic formalization of problems often considered to be pragmatic issues (e.g. structure and construction of parse forests) has considerably improved our understanding and has been an important success factor. It is our belief that such formalization is essential to harness the algorithmic intricacies of language processing, even if the expressive power of a formal system cannot not fully cover the richness of natural language constructions.

The links established with problems in other areas of computer science (e.g. partial evaluation, database recursive queries) could also be the source of interesting new approaches.

Acknowledgements

This work has been partially supported by the Eureka Software Factory (ESF) Project.

References

[AhoU-72] Aho, A.V.; and Ullman, J.D. 1972 *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, Englewood Cliffs, New Jersey.

[Bil-88] Billot, S. 1988 *Analyseurs Syntaxiques et Non-Déterminisme.* Thèse de Doctorat, Université d'Orléans la Source, Orléans (France).

[BilL-89] Billot, S.; and Lang, B. 1989 The structure of Shared Forests in Ambiguous Parsing. *Proc. of the* 27th *Annual Meeting of the Association for Computational Linguistics*, Vancouver (British Columbia), 143-151. Also INRIA Research Report 1038.

[BouPS-75] Bouckaert, M.; Pirotte, A.; and Snelling, M. 1975 Efficient Parsing Algorithms for General Context-Free Grammars. *Information Sciences* 8(1): 1-26

[Coh-88] Cohen, J. 1988 A View of the Origins and Development of Prolog. *Communications of the ACM* 31(1) :26-36.

[Col-78] Colmerauer, A. 1978 Metamorphosis Grammars. in *Natural Language Communication with Computers*, L. Bolc ed., Springer LNCS 63. First appeared as *Les Grammaires de Métamorphose*, Groupe d'Intelligence Artificielle, Université de Marseille II, 1975.

[DeR-71] DeRemer, F.L. 1971 Simple LR(k) Grammars. *Communications ACM* 14(7): 453-460.

[Ear-70] Earley, J. 1970 An Efficient Context-Free Parsing Algorithm. *Communications ACM* 13(2): 94-102.

[Fut-88] Futamura, Y. (ed.) 1988 Proceedings of the Workshop on Partial Evaluation and Mixed Computation. *New Generation Computing* 6(2,3).

[Lan-74] Lang, B. 1974 Deterministic Techniques for Efficient Non-deterministic Parsers. *Proc. of the* 2^{nd} *Colloquium on Automata, Languages and Programming*, J. Loeckx (ed.), Saarbrücken, Springer Lecture Notes in Computer Science 14: 255-269.
Also: Rapport de Recherche 72, IRIA-Laboria, Rocquencourt (France).

[Lan-88a] Lang, B. 1988 Parsing Incomplete Sentences. *Proc. of the* 12^{th} *Internat. Conf. on Computational Linguistics (COLING'88)* Vol. 1 :365-371, D. Vargha (ed.), Budapest (Hungary).

[Lan-88b] Lang, B. 1988 Datalog Automata. *Proc. of the* 3^{rd} *Internat. Conf. on Data and Knowledge Bases*, C. Beeri, J.W. Schmidt, U. Dayal (eds.), Morgan Kaufmann Pub., pp. 389-404, Jerusalem (Israel).

[Lan-88c] Lang, B. 1988 *Complete Evaluation of Horn Clauses: an Automata Theoretic Approach.* INRIA Research Report 913.

To appear in the International Journal of Foundations of Computer Science.

[Lan-88d] Lang, B. 1988 *The Systematic Construction of Earley Parsers: Application to the Production of* $\mathcal{O}(n^6)$ *Earley Parsers for Tree Adjoining Grammars.* In preparation.

[PerW-80] Pereira, F.C.N.; and Warren, D.H.D. 1980 Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13: 231-278.

[PerW-83] Pereira, F.C.N.; and Warren, D.H.D. 1983 Parsing as Deduction. *Proceedings of the* 21^{st} *Annual Meeting of the Association for Computational Linguistics*: 137-144, Cambridge (Massachusetts).

[Por-86] Porter, H.H. 3^{rd} 1986 *Earley Deduction.* Tech. Report CS/E-86-002, Oregon Graduate Center, Beaverton (Oregon).

[Pra-75] Pratt, V.R. 1975 LINGOL — A Progress Report. In *Proceedings of the* 4^th *IJCAI*: 422-428.

[She-76] Sheil, B.A. 1976 Observations on Context Free Parsing. in *Statistical Methods in Linguistics*: 71-109, Stockholm (Sweden), Proc. of Internat. Conf. on Computational Linguistics (COLING-76), Ottawa (Canada).
Also: Technical Report TR 12-76, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard Univ., Cambridge (Massachusetts).

[TamS-86] Tamaki, H.; and Sato, T. 1986 OLD Resolution with Tabulation. *Proc. of* 3^rd *Internat. Conf. on Logic Programming*, London (UK), Springer LNCS 225: 84-98.

[Tom-85] Tomita, M. 1985 *An Efficient Context-free Parsing Algorithm for Natural Languages and Its Applications.* Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

[Tom-87] Tomita, M. 1987 An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics* 13(1-2): 31-46.

[Vie-87b] Vieille, L. 1987 *Recursive Query Processing: The power of Logic.* Tech. Report TR-KB-17, European ComputerIndustry Research Center (ECRC), Munich (West Germany).

[VijWJ-87] Vijay-Shankar, K.; Weir, D.J.; and Joshi, A.K. 1987 Characterizing Structural Descriptions Produced by Various Grammatical Formalisms. *Proceedings of the* $25^r d$ *Annual Meeting of the Association for Computational Linguistics*: 104-111, Stanford (California).

[VijW-89] Vijay-Shanker, K.; and Weir, D.J. 1989 Polynomial Parsing of Extensions of Context-Free Grammars. *In this book.*

[VilZ-88] Villemonte de la Clergerie, E.; and Zanchetta, A. 1988 *Evaluateur de Clauses de Horn.* Rapport de Stage d'Option, Ecole Polytechnique, Palaiseau (France).