Computational Aspects of a Grammar Formalism for Languages with Freer Word Order

Oliver Suhre

November 19, 1999

Contents

Chapter 1

Introduction

Your order is your anarchy
Psychotic Waltz

Independence limited
Freedom of choice
Choice is made
For you my friend
Metallica

Generally speaking, computational linguistics is concerned with the use of computers to “understand” natural, i.e., human, languages. This thesis concentrates on a more concrete subtask: Given some natural language utterance, compute the syntactic structure of this utterance, which, in turn, is related to its semantic structure. Solving this task requires answering two questions: 1. What does this structure look like? and 2. How can it be computed efficiently?

Context-free grammars (CFGs) have been a popular choice to tackle the first question. The syntactic structure of a natural language utterance, i.e., a string of words, is then regarded as being the derivation tree(s) of this string w.r.t. some given CFG. Using context-free rules for writing grammars is also attractive from a computational point of view because they are known to be recognisable in $O(n^3)$. If, however, one wants to write a context-free grammar for a language with freer word order, those rules either typically overgenerate or undergenerate, i.e., word order is freer than in CFGs but not completely free (see section 1.1.2 for some examples). One can use machine learning techniques to acquire syntactic roles but the generated rules are often arbitrary with regard to semantic composition (see also section 1.2.1).

There have been attempts to relax word order implicitly, notably ID/LP grammars (cf. Gazdar, Klein, Pullum & Sag 1985), and Johnson (1985), which requires total freedom of order on an ID rule. On the other hand, recognition with CFG-like rules with no linear precedence have been shown to be $\mathcal{NP}$-complete

(Huynh 1983). Formalisms like Johnson (1985) that allow discontinuities in addition are, in practice, even less tractable. For a more detailed discussion see section 1.2.2.

The Linear Specification Language (LSL) was proposed in Götz & Penn (forthcoming) as an extension of CFGs designed to "fill the gap", linguistically and computationally, between those two extremes by naturally expressing syntactic combinations in natural languages with freer word order (see section 1.1).

In this thesis, I will investigate mainly three aspects of LSL grammars. Since they define a class of formal languages, I will take a look at how LSL behaves in terms of some classic questions of formal language theory (chapter 2), in particular, the Chomsky hierarchy, closure properties, and decidability. Section 2.8 is devoted to the complexity theoretic properties of the decidable problems.

The membership (or recognition) problem is the most relevant one for practical purposes and is analysed in detail in chapter 3. A generalisation of an Earley parser is described, extending the concepts of chart parsing in a straightforward way. That parser is shown to need exponential time in the worst case (which is not surprising since the membership problem is $\mathcal{NP}$-complete). However, a sufficient condition for a fixed LSL grammar to be parseable in polynomial time can be identified. This condition can be stated more generally so as to suit applications to natural languages better. A corresponding extension to LSL is proposed as well. The presented algorithm is not being presented as an even near-optimal solution for practical purposes, but only to establish the theoretical result.

In chapter 4, I will describe a C++ implementation of the parsing algorithm. It will be particularly concerned with issues which are not of interest from a theoretical point of view, but do play a crucial role for a practical system.

Chapter 5 contains a summary, a short discussion about what role the theoretical results play in practice, and an outlook on extensions of LSL.

In the next section, I will informally describe what an LSL grammar looks like and how strings can be derived by it. For a more formal account, see chapter 2.

1.1 The Linear Specification Language LSL

LSL grammars are a generalisation of context-free grammars in that they allow arbitrary partial orders and discontinuities on the right hand side of a grammar rule.

Throughout this thesis, I will consider an idealisation of the formalism proposed in Götz & Penn (forthcoming). In chapter 5, I will briefly present important additional parts of this original proposal. Aldag (1998) gives a formal semantics of LSL in the context of a typed feature logic (cf. King 1994).

1.1.1 Introduction to the Formalism

An LSL grammar G consists of a set of nonterminals N, a set of terminals T, a set of LSL rules P, a set of lexical entries L, and a start symbol S ($G = (N, T, P, L, S)$).

A lexical entry is a pair $Y \to a$ where $Y \in N$ and $a \in T$. Keeping the lexical entries separate from the rules is realistic, because in linguistic applications, this distinction is often made, too.

Informally, an LSL rule consists of two parts, a two-place relation between a nonterminal and a set of nonterminals (called *immediate dominance*), and some linear precedence constraints (*LP constraints*) between those nonterminals. For instance

$$\begin{aligned} S \;\to\; & A\,B\,C\,D\,; \\ & A < B, B \ll C, \langle A \rangle \end{aligned} \tag{1.1}$$

S, A, B, C, and D are nonterminals. The terms after the semicolon make statements about the ordering and discontinuity of the nonterminals on the right hand side (RHS) and left hand side (LHS) of the rule, or rather the terminal string (or *terminal yield*) derived from those nonterminals. They define relations between *occurrences* of nonterminals on the RHS of the rule, rather than nonterminals as such. If the same nonterminal occurs more than once, I will use indices to distinguish these occurrences. There are three kinds of LP constraints:

1. *(Weak) precedence* (written as $A < B$): The terminal yield of A is completely to the left of the terminal yield of B.

2. *Immediate precedence* ($A \ll B$): The rightmost terminal derived from A stands immediately to the left of the leftmost terminal derived from B.

3. *Isolation* ($\langle A \rangle$): The terminal yield of A is continuous (has no discontinuities). It is also possible to isolate the LHS of a rule which means that the terminal yield derived by a nonterminal expanded by that rule must be continuous.

If no LP constraints are imposed (denoted by "ε" after the semicolon), the rule allows any arbitrary ordering of A, B, C, and D *and* discontinuities.

So suppose, the terminal yield of A is *aa*, that of B is *bb*, that of C is *c*, and that of D is *dd*. If the rule had no LP constraints, all permutations of *aabbcdd* would be licensed by the rule.

For a terminal string to be grammatical according to rule 1.1, it must hold that

1. The rightmost a must occur somewhere to the left of the leftmost b.

2. The rightmost b must occur immediately to the left of c.

$$\begin{array}{rclllll}
\text{S} & \rightarrow & \text{NP} & \text{VP} & & ; & \text{NP} \ll \text{VP}. \\
\text{VP} & \rightarrow & \text{NP}_1 & \text{NP}_2 & \text{V} & ; & \text{NP}_1 < \text{V}, \quad \text{NP}_2 < \text{V} \\
 & & & & & & \langle\text{NP}_1\rangle, \quad\ \ \langle\text{NP}_2\rangle. \\
\text{V} & \rightarrow & \text{gibt}. & & & &
\end{array}$$

Figure 1.1: *LSL grammar for a German subordinate clause. For simplicity, it is assumed that "Fabian", "der Lisa", and "die Principia Mathematica" are NPs. The VP rule allows arbitrary ordering of the two objects of the verb.*

3. The two *a*'s must be continuous (next to each other).

These three conditions hold for the following terminal strings (there are still more): *aabbcdd*, *aadbdbc*, *aadbbcd*, *daabbcd*, *aadbdbc*. Note that D is not constrained in any way. On the other hand, one condition is violated in each of the following cases:

- *adabbcd* (A is not isolated)
- *aadbbdc* (B does not immediately precede C)
- *bddbcaa* (A does not precede B)

In what follows, I will use the term *derivation tree* (or *parse tree*) to refer to the tree expressing only the ID relations, i.e., two derivation trees which use exactly the same rules (but maybe have different word orders) are considered to be the same. In this sense, the derivation trees of all the strings generated by rule 1.1 are considered to be the same.

1.1.2 Examples

Let us now turn to more linguistically motivated examples from German which demonstrate what the term freer word order means in the context of natural language.

Subordinate Clauses

The LSL grammar in Fig. 1.1 generates the two subordinate clauses "(daß) Fabian der Lisa die Principia Mathematica gibt" and "(daß) Fabian die Principia Mathematica der Lisa gibt". The derivation trees for these two sentences are the same (see Fig. 1.2). Note how the VP rule allows for two orderings of the objects of the verb.

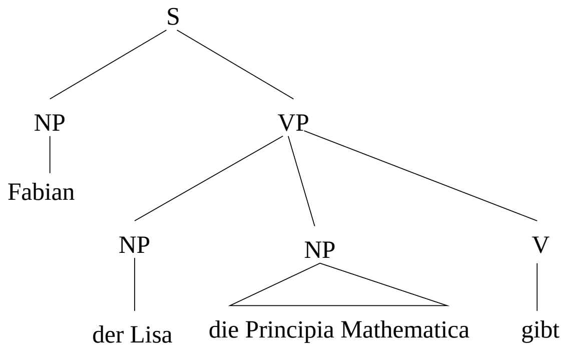


Figure 1.2: *Derivation tree for "(daß) Fabian der Lisa die Principia Mathematica gibt" and "(daß) Fabian die Principia Mathematica der Lisa gibt".*

$$\begin{array}{rcll}
\text{S} & \rightarrow & \text{NP VP} & ; \ \varepsilon \\
\text{NP} & \rightarrow & \text{D } \bar{\text{N}} & ; \ \text{D} \ll \bar{\text{N}} \\
\bar{\text{N}} & \rightarrow & \text{N CP} & ; \ \text{N} < \text{CP}, \langle \text{CP} \rangle \\
\text{D} & \rightarrow & \text{der} & \\
\text{N} & \rightarrow & \text{Mann} & \\
\text{VP} & \rightarrow & \text{stirbt} &
\end{array}$$

Figure 1.3: *LSL grammar for extraposition. For simplicity, it is assumed that "der zögert" is a CP.*

Extraposition

Consider the LSL grammar in Fig. 1.3 capturing extraposition phenomena.

This grammar derives the grammatical German sentences "Der Mann, der zögert, stirbt", "Der Mann stirbt, der zögert" and "Stirbt der Mann, der zögert" which all have the same derivation tree (see Fig. 1.4). Note, in particular, that the NP "der Mann, der zoegert" has a discontinuity in the two latter sentences.

On the other hand, the string "der stirbt Mann der zögert" is ruled out because "der" is required to immediately precede "Mann", i.e., nothing is allowed to occur in between them.

1.1.3 Encoding CFGs as LSL Grammars

It is possible to express CFGs in LSL. Consider the simple context-free rule $S \rightarrow ABC$. In LSL terminology, this rule expresses the following:

1. An S consists of (immediately dominates) an A, a B, and a C.

2. A (immediately) precedes B, and B (immediately) precedes C.

3. A, B, and C are isolated.

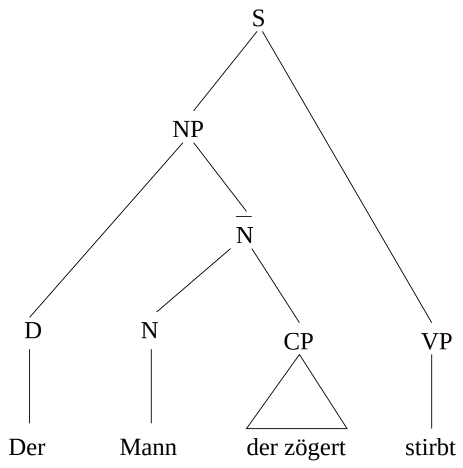


Figure 1.4: *Derivation tree of "Der Mann, der zögert, stirbt", "Der Mann stirbt, der zögert", and "Stirbt der Mann, der zögert". The three sentences only differ w.r.t. word order but are all generated with the same rules.*

4. S is isolated.

There are a couple of different ways to express these constraints. What they all have in common is the immediate dominance part, which looks just like the context-free rule: $S \to ABC; \varphi$. And here are some possibilities for φ (the LP constraints):

1. $A < B, B < C, \langle S \rangle, \langle A \rangle, \langle B \rangle, \langle C \rangle$
2. $A < B, B < C, \langle S \rangle$
3. $A \ll B, B \ll C, \langle A \rangle, \langle B \rangle, \langle C \rangle$
4. $A \ll B, B \ll C, \langle S \rangle$

Note that (1) and (2) are equivalent in that the isolation of the ordered A, B, and C is implied by the isolation of the LHS (but not the converse). If the isolation of S is left out, i.e., $A < B, B < C, \langle A \rangle, \langle B \rangle, \langle C \rangle$, this does *not* imply (2), since there could be a "hole" between the A and B, or B and C, respectively.

1.2 Motivation

The reader might ask at this point: "Why add another beast to the formal language zoo?". To answer this question, I will first briefly discuss why CFGs (one of the best known and understood grammar formalisms) are inappropriate for describing freer word order phenomena. Secondly, I will describe some alternative approaches to freer word order and show their weaknesses.

1.2.1 Why not CFGs?

There are mainly two reasons why CFGs cannot be used sensibly for describing languages with freer word order.

Expressive Power

It has often been argued that the context-free languages (CFL) are simply to weak to describe some "non context-free" phenomena in natural languages like cross-serial dependencies (cf. Shieber 1985) or copying (cf. Radzinski 1990). Furthermore, a CFG for a language with freer word order must have a single rule for every possible word ordering. Since word order is not completely free (thus the term *freer* word order), these rules typically either overgenerate or undergenerate severely. This is a fact that is not acceptable.

Parsing and Semantics

The ultimate goal of parsing an expression in natural language is not only to find out if it is licensed by a specified grammar, which is a simple yes/no question, but to somehow create the semantics of this expression. This is crucial for applications using Natural Language Processing techniques like machine translation or database querying. The semantics[1] of a natural language utterance can be computed from its derivation tree.

For expository purposes, I will use some kind of λ-terms as semantics. For instance, the German word "gibt" might have a semantics like $\lambda z \lambda y \lambda x.gives(x,y,z)$. This semantics is stored somewhere in the lexicon. Consider the German subordinate clause "(daß) Fabian der Lisa die Principia Mathematica gibt". The derivation tree of this sentence w.r.t. the grammar in Fig. 1.1 is shown in Fig. 1.2.

Assume that the semantics of "Fabian" is the constant "$fabi$", that of "der Lisa" is "$lisa$", and that of "die Principia Mathematica" is "$principia$", respectively. Then the semantics of the sentence is computed by functional application in such a way that every inner node is assigned a λ-term as follows: the semantics of VP is the semantics of the V node successively applied to the semantics of the two NP nodes (the two objects of the verb) in the order from right to left:

$$\begin{aligned} & (\lambda z \lambda y \lambda x.gives(x,y,z))(principia)(lisa) \\ = \; & \lambda y \lambda x.gives(x,y,principia)(lisa) \\ = \; & \lambda x.gives(x,lisa,principia) \end{aligned}$$

The semantics of the S node can be computed analogously:

$$(\lambda x.gives(x,lisa,principia))(fabi) = gives(fabi,lisa,principia),$$

[1]The term "semantics" should be understood as precise, detailed semantic representation rather than a vague conception of meaning such as keyword-occurrences.

which is the semantics of the sentence. Now note that this semantics is the same for a variation of this sentence, e.g., “(daß) Fabian die Principia Mathematica der Lisa gibt”. The VP rule of the grammar in Fig. 1.1 allows both orders mentioned above. Hence, the two sentences have the same derivation trees.

If, on the other hand, one assumes that “(daß) Fabian der Lisa die Principia Mathematica gibt” and “(daß) Fabian die Principia Mathematica der Lisa gibt” have different derivation trees, the rules for the semantics composition cannot be applied straightforwardly. One would then be forced to transform one tree into the other[2] to compute the semantics. This situation would certainly arise if one used a context-free grammar which has a different rule for every possible word ordering. With an adequate LSL grammar, on the other hand, it is possible for the derivation trees of all those sentences to be the same and that their respective word orders satisfy the LP constraints of that grammar. Then, the semantics can be computed uniformly.

1.2.2 Previous Work

This section will briefly cover a selection of previous work on formalisms (mostly extensions of CFGs) to handle natural languages with freer word order.

ID/LP Grammars

In Gazdar et al. (1985), ID/LP grammars were introduced. These kind of grammars has two kinds of rules: Immediate dominance (ID) and linear precedence (LP) rules. ID rules have the form $X_0 \to X_1, \ldots, X_n$, which is supposed to mean “X_0 immediately dominates X_1 to X_n”. No statement, however, is made as to what the linear order of the X_i is. LP rules are pairs of nonterminals like $A < B$ saying “If A and B are sisters, i.e., if they occur both on the RHS of a rule, then A has to precede B”. These LP statements are not attached to an ID rule as in LSL, but are valid “globally”. Furthermore, discontinuity is *not* allowed, or in LSL terminology: every nonterminal is isolated. It is easy to see that for every ID/LP grammar one can construct an equivalent CFG. This is done by computing every permutation of the RHS of an ID rule which satisfies all the LP rules. The lack of the possibility to specify discontinuity, however, makes it hard to write ID/LP grammars for languages with freer word order.

An important formal difference between ID/LP and LSL grammars is that LP constraints are attached to a particular *rule* in LSL, thus define relations between occurrences of nonterminals of that particular rule, whereas in ID/LP, they define relations between nonterminals as such. It is thus not possible to directly specify an LP rule like $X < X$[3]. In LSL, however, one can easily specify a rule $Y \to X_1 X_2; X_1 < X_2$.

[2] In some (pure) linguistic theories this is actually done.

[3] What should that mean, anyway?

UCFGs

Unordered context-free grammars (UCFGs), also called commutative context-free grammars, are a special case of ID/LP grammars in that there are only ID rules and no LP rules, i.e., there are no ordering constraints. Still, strings derived by a nonterminal are continuous. In Huynh (1983) and Barton, Berwick & Ristad (1987), it was shown that the general membership problem for UCFGs is $\mathcal{NP}$-complete.

Johnson's Extension of DCGs

In Johnson (1985), a natural extension of definite clause grammars (DCGs) was proposed for parsing "non-configurational languages", i.e., languages with completely free word order. DCGs (cf. Gazdar & Mellish 1989) are generalised context-free grammars in that nonterminals are first order terms and that a rule can also have procedural attachments. They are strongly connected to the Prolog programming language, which allows direct parsing of DCGs using the built-in depth-first search strategy. For instance, one can write the following DCG rule:

```
np --> det, n.
```

This clause plays the role of a context-free rule, expressing the fact that an `np` consists of a `det`erminer and a `n`oun. Internally, the Prolog compiler translates this DCG rule into the following clause:

```
np(X,Y) :- det(X,Z), n(Z,Y).
```

The two added arguments represent string positions. This clause now means: "To find an `np` from position `X` to `Y`, we have to find a `det` from `X` to some position `Z`, and an `n` from this `Z` to `Y`".

Generalising DCGs to languages with discontinuous constituents makes it impossible to take just two integers as string positions. Rather, we use just one extra argument instead of two which contains the so called *location* of the nonterminal, indicating which string positions belong to the constituent to be parsed. This location is implemented as a *bit pattern* (bit vector) (see also section 3.2.1). All locations on the RHS of a rule may be combined to yield the location of the LHS, i.e., the bit patterns are ORed bitwise. This is done by the three place predicate `combines`. So, the example from above would be rewritten as

```
np(L) :- det(L1), n(L2), combines(L1,L2,L).
```

where `L`, `L1`, and `L2` are bit patterns, implemented as e.g. lists.

Writing a grammar in this fashion, it is easy to construct a parser. One can either use Prolog's built-in strategy or any other strategy for processing Prolog programs, e.g., Earley Deduction. No specialised parser needs to be written. Furthermore, one can modify the definition of `combines` so as to suit other needs.

For instance, it is very straightforward to implement a predicate that checks if two bit patterns precede or immediately precede each other. Hence, one can simulate LSL with such predicates.

However, Johnson's framework does not have any notion of derivation nor any formal backbone. It describes how a parser might be implemented using Prolog as the programming language, no more and no less[4].

Generalisations of Known Parsing Algorithms

In Reape (1991), some common parsing algorithms are generalised for the parsing of so called *permutation complete* languages[5]. A language L is permutation complete if $w \in L$ implies that every permutation of w is also in L.

Starting with simple parsing algorithms, like top-down, left corner, and shift reduce parsers, chart parsers like CYK which use *codes* (another word for bit patterns) are also investigated. For all generalisations, it is proven that they have the minimality (all parses are found exactly once), soundness (a found parse is licensed by the grammar), and completeness (if the language is decidable, all parses are found) properties. As one can easily imagine, the complexity of all those algorithms is exponential.

Instead of giving up here, Reape tries to analyse the complexity of languages with a "nonconcatenative" nature, i.e., where the string of a mother might be created in a more complicated way than being simply the concatenation of the strings of the daughters. He therefore uses the linear context-free rewriting systems (LCFRS) of Vijay-Shanker, Weir & Joshi (1987). It turns out that the complexity then depends on what these "string combining" operators look like. If the operator is a function, e.g., append in the context-free case, we have polynomial complexity. If this operator, on the other hand, is relational, i.e., may have several solutions, it seems[6] that this makes the problem $\mathcal{NP}$-complete. An example of such a relational operator is shuffle, which "nondeterministically" mixes two lists.

Since LSL languages are not permutation complete in the general case, the results of Reape (1991) are of restricted value to us. Furthermore, trying to define operators to embed LSL into the mentioned LCFRS does not work in a straightforward way because it is not enough to order the string of a mother satisfying some LP constraints and then forget about those constraints for a reason described in section 2.2.

[4]To see how Johnson's framework can be described more formally, see section 3.4.

[5]Johnson (1985) does also but his framework does not rely on this property.

[6]Reape does not give formal proof for this, in fact he uses the word "seems" himself.

Dependency Grammars

In Holan, Kuboň & Plátek (1995), *non-projective context-free dependency grammars* (NCFDGs) are introduced. Generally, dependency grammars are quite popular for Slavic languages which have a high degree of freedom of word order. NCFDGs have rules of the form $A \rightarrow_L BC$ or $A \rightarrow_R BC$ which mean that a sentential form like $\alpha_1\alpha_2A\beta_1\beta_2$ can be rewritten as $\alpha_1B\alpha_2C\beta_1\beta_2$ or $\alpha_1\alpha_2B\beta_1C\beta_2$, respectively, i.e., the left (right) nonterminal on the RHS can be inserted somewhere to the left (right) of the rewritten nonterminal.

NCFDGs have a notion of discontinuity but a different notion of precedence than LSL. This is because a rule $A \rightarrow_L BC$ does *not* imply that B (weakly) precedes C (in LSL terminology) since with the rule $C \rightarrow_L DE$, the derivation $A \Rightarrow BC \Rightarrow DBE$ is possible. There is also no notion of immediate precedence or isolation, it is thus impossible to *enforce* context-freeness for some parts of the grammar which is no problem with LSL. Holan, Kuboň, Oliva & Plátek (1998) proposes an enhancement of NCFDGs (the *free order dependency grammars* (FODGs) introduced herein are basically equivalent to NCFDGs) by introducing rules of the form $A \rightarrow_X^0 BC$ $(X \in \{L, R\})$ which means that A is isolated, i.e., A has 0 discontinuities. (Note that the isolation constraint can only be attached to the LHS of a rule whereas in LSL, any nonterminal on the RHS can be isolated.)

Another important notion of NCFDGs and linguistic dependency theory in general is the *dependency tree* of a string which is rather different from a derivation tree of CFGs or LSL.

Overall, dependency theory is a very different approach to natural language than the kind of theories LSL builds upon.

Other Extensions of CFGs

The formal language literature is full of extensions of CFGs, which have emerged from different needs and motivations. For instance, Dassow & Păun (1989) is particularly concerned with adding context-sensitive features to a context-free backbone. Examples for such extensions are indexed, matrix, programmed, and random context grammars. However, none of those formalisms has either a notion of discontinuity nor partial ordering of a RHS - notions likely to be needed for natural languages with freer word order.

Chapter 2

Formal Language and Complexity Aspects

In this chapter, I will formally define LSL grammars and investigate some of the classic questions in formal language and complexity theory. In particular, I will be concerned with the Chomsky hierarchy, closure properties, decidability, and the complexity of some of the decidable problems. Although LSL languages might seem close to the context-free languages (CFL), not all results of CFLs do hold for them.

Apart from the academic interest in formal language properties of LSL (since it is, after all, a class of formal languages), there is another, maybe more interesting aspect of this chapter. If it turns out that LSL is a (more or less) adequate formalism to express natural language, one might get some insights into the properties of natural languages themselves.

Note: In a lot of the definitions, I use statements like "If $x \in A$ then $y \in B$" to describe how to construct a set B, when a set A is given. To prevent B from containing any "junk" which is allowed due to the implicational form of the statement (What happens if $x \notin A$?), this is always intended to mean "Let B be the smallest set such that: If $x \in A$, then $y \in B$".

2.1 LSL Grammars

In CFGs, the right hand side (RHS) of a context-free production $A \rightarrow BCD$ implies that the yields of B, C, and D occur in the input string in this order and that they are adjacent to each other. In LSL, that is no longer the case. We can have arbitrary *partial orders* on the RHS and must distinguish between a weak notion of order (precedence) and a strong one (immediate precedence). Precedence is, in some sense, weaker then immediate precedence because: If A immediately precedes B then A precedes B, but not necessarily the converse. In other words, immediate precedence implies weak precedence.

Thus, RHSs of LSL productions are represented as directed acyclic graphs (DAGs). There are two kinds of edges in such graphs: *i-edges* indicate obligatory immediate precedence, *p-edges* indicate obligatory precedence which may or may not be immediate. Lack of an edge indicates that precedence of any kind is not obligatory, but not that it is prohibited. I will call such a graph an *IP-graph.*

I assume an infinite, enumerable set Node.

Definition 1 (IP-Graph) *A tuple (V, E_P, E_I) is an* IP-graph*, where $V \subseteq \mathsf{Node}$ is the finite set of nodes, $E_P, E_I \subseteq V \times V$ are the disjoint sets of p-edges and i-edges, and $(V, E_P \cup E_I)$ is a DAG.*

Let IG be the set of all IP-graphs over Node. As for general graphs, one can define the notion of a path for IP-graphs as well. It is useful to distinguish between two different kinds of paths.

Definition 2 *Let $R = (V, E_P, E_I)$ be an IP-graph. A sequence $v_1 \dots v_n \in V^*$ is a*

- *Path in R if for all $i = 1, \dots, n-1$: $(v_i, v_{i+1}) \in E_P \cup E_I$.*
- *I-Path in R if for all $i = 1, \dots, n-1$: $(v_i, v_{i+1}) \in E_I$.*

Every i-path is also a path, but not necessary conversely. I will write $v \overset{R}{\rightsquigarrow} v'$, $v \overset{R}{\rightsquigarrow}_i v'$, to express that there is a path, or an i-path from v to v' in R, respectively.

Later, I will also need the following notions of start nodes and end nodes of an IP-graph.

Definition 3 *Let $R = (V, E_P, E_I) \in IG$. Define*

- $\mathsf{start}(R) := \{v \in V | \neg\exists u : (u, v) \in E_P \cup E_I\}$
- $\mathsf{end}(R) := \{v \in V | \neg\exists u : (v, u) \in E_P \cup E_I\}$

Since IP-graphs do not have cycles, these two sets are empty iff the IP-graph itself is empty.

Given an IP-graph, one can, analogously to DAGs, define the notion of a topological sort. This sorting can be seen as embedding the partial order given by the DAG into a total order. Since i-edges represent immediate precedence, we should, additionally, ensure that nodes connected by i-edges are adjacent to each other in this topological sort.

Definition 4 *Let $R = (\{v_1, \dots, v_n\}, E_P, E_I)$ be an IP-graph. A topological sort of R is a bijective function $\sigma : \{1, \dots, n\} \to \{v_1, \dots, v_n\}$ such that:*

1. *If $(v_i, v_j) \in E_P$, then $\sigma^{-1}(v_i) < \sigma^{-1}(v_j)$.*
2. *If $(v_i, v_j) \in E_I$, then $\sigma^{-1}(v_i) = \sigma^{-1}(v_j) - 1$.*

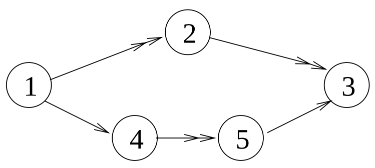


Figure 2.1: *An IP-graph which has indegree and outdegree ≤ 1 w.r.t. E_I, but is not topologically sortable. P-edges are drawn as single, i-edges as double arrows.*

The sequence $\sigma(1) \ldots \sigma(n)$ is called the topological ordering of R w.r.t. σ.

Although σ is only defined by its inverse, it is well defined because σ is a bijection and, thus, has a unique inverse σ^{-1}.

Not all IP-graphs have such a topological ordering. However, for the IP-graphs to represent the partial orders of the RHSs of LSL productions, they must be topologically sortable.

Definition 5 (Rule Graph) *An IP-graph is called a* rule graph *iff it has a topological ordering.*

Let $RG \subset IG$ be the set of all rule graphs.

One can identify a necessary condition for an IP graph to be topologically sortable.

Proposition 1 *Let $R = (V, E_P, E_I) \in IG$. If $R \in RG$, then every $v \in V$ has indegree and outdegree ≤ 1 w.r.t. E_I.*

Proof:
Let σ be a topological sort of R. Now assume that the outdegree of V w.r.t. to E_I is ≥ 2, thus there are different $v, v_1, v_2 \in V$ such that $(v, v_1) \in E_I$ and $(v, v_2) \in E_I$. Since σ is a bijection, $\sigma^{-1}(v_1) \neq \sigma^{-1}(v_2)$. But it must hold that $\sigma^{-1}(v) = \sigma^{-1}(v_1) - 1$ and $\sigma^{-1}(v) = \sigma^{-1}(v_2) - 1$, thus $\sigma^{-1}(v_1) = \sigma^{-1}(v_2)$, which is a contradiction.

The proof for the indegree follows from symmetry.

□

By contraposition, we have the following corollary:

Corollary 1 *Let $R = (V, E_P, E_I) \in IG$. If there is a $v \in V$ which has indegree and outdegree ≥ 2 w.r.t. E_I, then R does not have a topological ordering.*

The condition of Proposition 1 is only necessary, not sufficient, i.e., there are IP-graphs with indegree and outdegree ≤ 1 w.r.t. E_I which are, however, not topologically sortable. For instance, consider Fig. 2.1.

If a rule graph has a topological ordering, it can be computed in polynomial time (just as for ordinary DAGs).

Proposition 2 *There is a function* topsort : $IG \to$ Node* *which computes a topological ordering of an IP-graph in polynomial time if there is one.*

Proof:
Let $R = (V, E_P, E_I)$. We can assume that the indegree and outdegree of R w. r. t. E_I is ≤ 1. Otherwise, R does not have a topological ordering (cf. Corollary 1). Furthermore, w.l.o.g. we can assume that no two nodes which are connected by an i-path also have a p-edge between them.

It was shown in, e.g., Cormen, Leiserson & Rivest (1990) that topologically sorting a DAG (N, E) can be done by performing a depth first search which takes time $O(|N| + |E|)$. Sorting $(V, E_P \cup E_I)$ with this method makes condition 1 of Definition 4 true, but condition 2 may still be false. To get this right, one can simply consider all maximal chains in E_I as atomic nodes because for all possible topological orderings, the order of such a chain is the same. These maximal chains can be computed by the function maxichains:

```
**function** maxichains($(V, E_P, E_I) : IG$): $\mathcal{P}(V^+)$;
$S := \emptyset$;
**for all** $v \in V$ such that there is no $y$ s.t. $(y, v) \in E_I$ **do**
  $\alpha := v$;
  $u := v$;
  **while** $\exists x : (u, x) \in E_I$ **do**
    $\alpha := \alpha x$; (* *concatenate* $\alpha$ *and* $x$ *)
    $u := x$;
  **endwhile**
  $S := S \cup \{\alpha\}$;
**endfor**
**return** $S$;
```

maxichains first examines each node (takes time $O(|V|)$), checks if it is a start node w.r.t. E_I (takes time $O(|E_I|)$), and then follows the i-chain, if any. Note that by Corollary 1, this i-chain is unique if there exists a topological ordering, and thus, following it takes time $O(|E_I|)$. The overall runtime is thus $O(|V||E_I|^2)$, a polynomial.

We can now construct the graph

$$\begin{array}{rcll} D & = & (\mathsf{maxichains}(R), E_D) & \\ E_D & = & \{(v_1 \ldots v_n, u_1 \ldots u_m) \mid & \text{there are } i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \\ & & & \text{such that } (v_i, u_j) \in E_P\} \end{array}$$

In D, all (maximal) chains in E_I are treated as a single node. Creating D can also be done in polynomial time. If D has a cycle, R has no topological ordering and we fail (cyclicity can be tested in polynomial time). (Due to our assumption above, D cannot introduce loops, i.e., edges of the form (v, v).)

Sorting this graph topologically yields a sequence of nodes of D which is also a sequence of nodes of R, and is thus a topological ordering of R. The overall runtime is then polynomial.

$\square$

Having the notion of a rule graph, one can define LSL productions. Rule graphs labelled with nonterminals are used as RHSs in an LSL production. I assume N to be a fixed, finite set of nonterminals.

Definition 6 (LSL Production) *An LSL production is of the form*

$$(v \rightarrow (V, E_P, E_I), \theta, I)$$

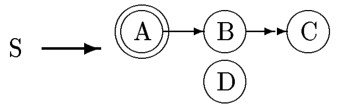
where

- $v \notin V$ *is an unused symbol*
- $(V, E_P, E_I) \in RG$
- $\theta : V \cup \{v\} \rightarrow N$
- $I \subseteq V \cup \{v\}$

An LSL production has a partial order on the RHS, represented as a rule graph, which is labelled with nonterminals via θ.

The edges of E_P (E_I respectively) represent the $<$ ($\ll$) relation originally defined in the LSL formalism, see section 1.1.1. The set I plays the role of the $\langle\rangle$ relation.

As above, p-edges are drawn as single, and i-edges as double arrows. Isolated nodes are drawn as double circles, the labels of the nodes are written within them. For instance



By letting the graph on the RHS be empty ($V = \emptyset$), we have an ε-production.

There can be three kinds of LP constraints on the RHS:

1. (Weak) Precedence, $(x, y) \in E_P$: x, i.e., the terminal yield of the category $\theta(x)$, is realised to the left of y.

2. Immediate precedence, $(x, y) \in E_I$: The rightmost terminal of the yield of x must occur immediately to the left of the leftmost terminal of y, i.e., the yields are adjacent.

3. Isolation, $x \in I$: x is contiguously realised. Note that also the left hand side (LHS) of a production may be isolated.

LSL productions are also called LSL rules. Now we are ready to define LSL grammars.

Definition 7 (LSL Grammar) *An LSL grammar is a tuple $G = (N, T, P, L, S)$ where N is the set of nonterminals, T is the set of terminals, $L \subseteq N \times T$ is the set of lexical entries, P is a set of LSL productions, and $S \in N$ is the start symbol.*

Let LSLG be the set of all LSL grammars. Note that by definition, LSL grammars are in some kind of "normal form" in that terminals do not appear in productions other than of the form $X \rightarrow a$. This is realistic in that grammars used in linguistic applications are usually separated into grammar rules and the lexicon.

From now on, I assume a fixed LSL grammar $G = (N, T, P, L, S)$.

Next, I will define what derivations are formally and prove some properties of this derivation relation.

2.2 Derivations

With CFGs, a derivation step works as follows: If we have a string (sentential form) $\alpha A \beta$ where A is a nonterminal, we can choose some production $A \rightarrow \gamma$ out of the set of defined productions and replace A with the RHS (γ) yielding the sentential form $\alpha\gamma\beta$. This can be extended straightforwardly for UCFGs (cf. section 1.2.2) in that A can also be replaced by some permutation of γ.

For LSL derivations, a similar mechanism would not work, because LP constraints introduced at some point in the derivation may have to be remembered until the very end. One cannot simply order a sequence of nonterminals in a sentential form in a way which satisfies all LP constraints currently imposed and then forget about those constraints because, discontinuities might appear later in the derivation.

For instance, consider the set of LSL productions $S \rightarrow AB; \varepsilon$, $A \rightarrow CD; C < D, \langle C \rangle$, and lexical entries $C \rightarrow c$, $D \rightarrow d$, and $B \rightarrow b$. This grammar can derive the string *cbd*. In a "Chomsky" style sentential form, we only have two possibilities of expanding S, namely AB or BA. However, the possible discontinuity of A makes it important to somehow remember the LP constraints $C < D$ and $\langle C \rangle$ until the very end of the derivation.

To overcome this problem, nonterminals in a sentential form are tagged with a finite set of integers indicating which string positions this nonterminal must eventually derive. I will call those sets *index sets*. Let $\mathsf{Fin}(\mathbb{N}) = \{M \subseteq \mathbb{N} | M \text{ is finite}\}$.

A derivation step splits the index sets among the the items on the RHS so as to satisfy the LP constraints. When we arrive at a stage where all index sets are singleton, we may end the derivation, apply lexical entries and in this fashion, generate a string. I will write these two "phases" as $\Rightarrow$ and $\rhd$.

Isolated nonterminals are tagged with *contiguous* index sets.

Definition 8 *Let $M \in \mathsf{Fin}(I\!N)$. M is called* contiguous *(written as $\mathsf{cont}(M)$) iff $M = \emptyset$ or for all $\min(M) < k < \max(M)$, it holds that $k \in M$.*

Since the order in which the tagged nonterminals appear in a sentential form should not matter (only the index sets determine the order in the derived terminal string), we might use sets as containers. This would be correct for nonempty index sets, but we might derive a nonterminal A with the empty index set at two different points in a derivation which would only appear once in the sentential form if we used sets. Hence, we use bags (also called "multisets") instead.

The bags over a domain Q are written as $\mathcal{B}(Q)$. I use "[" and "]" as opening and closing brackets of bags. Formally, a bag B over a domain Q is a mapping from Q to $I\!N_0$. If $B(q) = 0$, q is not in B, and if $B(q) = i$, i copies of q are in B. The cardinality of a bag is defined as $|B| = \sum_{q \in Q} B(q)$. The union of two bags $B, B' \in \mathcal{B}(Q)$ is defined such that $B \cup B' \in \mathcal{B}(Q)$ and for all $x \in Q$: $(B \cup B')(x) = B(x) + B'(x)$. The empty bag is written as $[]$.

Here is the definition of the derivation relation $\Rightarrow$, the first phase of a derivation.

Definition 9 *The derivation relation*

$$\Rightarrow_G \subseteq \mathcal{B}(N \times \mathsf{Fin}(I\!N)) \times \mathcal{B}(N \times \mathsf{Fin}(I\!N))$$

with respect to an LSL grammar $G = (N, T, P, L, S)$ is defined to be such that

$$D \cup [(X_0, M_0)] \Rightarrow_G D \cup [(X_1, M_1), \ldots, (X_n, M_n)]$$

iff

1. *$(v_0 \to \underbrace{(\{v_1, \ldots, v_n\}, E_P, E_I)}_{=:R}, \theta, I) \in P$ and $\theta(v_i) = X_i$ for all $i = 0, 1, \ldots, n$.*
2. *$M_0 = \bigcup_{i=1}^{n} M_i$ and all M_i are pairwise disjoint.*
3. *For all $i, j \in \{0, 1, \ldots, n\}$:*
   (a) *If $v_i \in I$ then $\mathsf{cont}(M_i)$.*
   (b) *If $v_i \overset{R}{\leadsto} v_j$ and $M_i \neq \emptyset \neq M_j$ then $\max(M_i) < \min(M_j)$.*
   (c) *If $v_i v_{i_1} \ldots v_{i_k} v_j$ is an i-path in R, $M_i \neq \emptyset \neq M_j$, and $M_{i_q} = \emptyset$ for all $q = 1, \ldots, k$, then $\max(M_i) = \min(M_j) - 1$.*

Condition 1 of the definition requires that indeed a production of G is applied. The second condition expresses that the index set is partitioned among the nonterminals on the RHS. Note that it is allowed that some of the M_i are empty to enable derivations with ε-productions. Finally, the index sets have to satisfy certain conditions imposed by the LP constraints (condition 3):

(3a) If a nonterminal X_i is isolated, then its index set M_i must be contiguous.

(3b) If a nonterminal X_i tagged with index set M_i precedes a nonterminal X_j tagged with M_j then the biggest member of M_i must be smaller than the smallest member of M_j, i.e., all terminals derived by X_i occur to the left of all terminals derived by X_j, provided that M_i and M_j are not empty.

(3c) If the rule graph has an i-path between two nodes v_i and v_j which have nonempty index sets, and all nodes on this i-path have empty index sets, then the immediate precedence relation should also hold between v_i and v_j. If $k = 0$, there is an i-edge between v_i and v_j, i.e., v_i immediately precedes v_j.

Note that $M_0 = \emptyset$ if $n = 0$, i.e., if the production has an empty rule graph on the RHS, the tagged nonterminal $(X_0, \emptyset)$ is deleted from the sentential form.

As usual, $\Rightarrow^i_G$ denotes a derivation in exactly i steps, and $\Rightarrow^*_G$ the reflexive and transitive closure of $\Rightarrow_G$. When I want to express that a fixed production p was used for a derivation step, I write $\Rightarrow_p$.

For instance, the grammar G_{ex} with productions

$$\begin{array}{llcll}
(\mathrm{P1}) & S & \rightarrow & ABCD & ; \ \langle A\rangle, A < B, B \ll C \\
(\mathrm{P2}) & A & \rightarrow & A'A' & ; \ \varepsilon \\
(\mathrm{P3}) & B & \rightarrow & B'_1B'_2B'_3 & ; \ B'_1 < B'_2, B'_2 < B'_3 \\
(\mathrm{P4}) & B' & \rightarrow & \varepsilon & ; \ \varepsilon \\
(\mathrm{P5}) & C & \rightarrow & C' & ; \ \varepsilon \\
(\mathrm{P6}) & D & \rightarrow & D'D' & ; \ \varepsilon
\end{array}$$

gives rise to the derivation

$$\begin{array}{ll}
 & [(S,\{1,2,3,4,5,6\})] \\
\Rightarrow_{P1} & [(A,\{2,3\}),(B,\{5\}),(C,\{6\}),(D,\{1,4\})] \\
\Rightarrow_{P2} & [(A',\{2\}),(A',\{3\}),(B,\{5\}),(C,\{6\}),(D,\{1,4\})] \\
\Rightarrow_{P3} & [(A',\{2\}),(A',\{3\}),(B',\emptyset),(B',\{5\}),(B',\emptyset),(C,\{6\}),(D,\{1,4\})] \\
\Rightarrow_{P5} & [(A',\{2\}),(A',\{3\}),(B',\emptyset),(B',\{5\}),(B',\emptyset),(C',\{6\}),(D,\{1,4\})] \\
\Rightarrow_{P6} & [(A',\{2\}),(A',\{3\}),(B',\emptyset),(B',\{5\}),(B',\emptyset),(C',\{6\}),(D',\{1\}),(D',\{4\})] \\
\Rightarrow_{P4} & [(A',\{2\}),(A',\{3\}),(B',\emptyset),(B',\{5\}),(C',\{6\}),(D',\{1\}),(D',\{4\})] \\
\Rightarrow_{P4} & [(A',\{2\}),(A',\{3\}),(B',\{5\}),(C',\{6\}),(D',\{1\}),(D',\{4\})]
\end{array}$$

Note that application of $(P3)$ introduces two nonterminals $(B', \emptyset)$ which would be "merged" if sentential forms were sets, thus the usage of bags. Those two nonterminals are then deleted by applying $(P4)$ twice.

The second phase of the derivation strips off the (now singleton) index sets from the nonterminals and applies lexical entries.

Definition 10 *The terminating derivation relation*

$$\triangleright_G \subseteq \mathcal{B}(N \times \mathsf{Fin}(I\!N)) \times T^*$$

with respect to G is defined to be such that

$$[(X_1, M_1), \ldots, (X_n, M_n)] \triangleright_G w_1 \ldots w_n$$

with $w_i \in T$ iff

1. $|M_i| = 1$ *for all* $i = 1, \ldots, n$
2. *If* $M_i = \{j\}$, *then* $X_i \to w_j \in L$.

Suppose G_{ex} has lexical entries

$$\begin{array}{lcl} A' & \to & a \\ B' & \to & b \\ C' & \to & c \\ D' & \to & d \end{array}$$

then the last sentential form from the example above derives a terminal string as follows:

$$[(A', \{2\}), (A', \{3\}), (B', \{5\}), (C', \{6\}), (D', \{1\}), (D', \{4\})] \triangleright_{G_{ex}} daadbc$$

What now remains to be defined is the notion of the language generated by an LSL grammar.

Definition 11 *The language generated by an LSL grammar $G = (N, T, P, L, S)$ is defined as*

$$L(G) := \{w \in T^* | \exists D \in \mathcal{B}(N \times \mathsf{Fin}(I\!N)) : [(S, \{1, \ldots, |w|\})] \Rightarrow_G^* D \triangleright_G w\}.$$

Let $LSLL$ be the set of all languages generated by LSL grammars.

2.2.1 A Note on Graph Grammars

Derivations of CFGs are defined as rewriting sentential forms as follows: Select a nonterminal in a sentential form and a production which has this nonterminal on the LHS and replace the nonterminal in the sentential form with the RHS of the selected production. Defining derivations of LSL grammars in an analogous way would result in some kind of graph rewriting. There is a vast amount of literature on graph rewriting systems or graph grammars. In Nagl (1979), a very general notion of graph grammars is introduced. Analogously to Chomsky grammars, there are also regular, context-free and context-sensitive graph grammars. The inclusion properties of these grammar types are, however, quite different than for Chomsky grammars. For LSL, we would like something like context-free graph rewriting because LHSs of LSL productions consist of only one nonterminal symbol. Context-free graph grammars are, e.g., discussed in Engelfriet & Rozenberg

$$\begin{array}{llcll}
(R1) & S & \to & ABC & ; \ A \ll B, B \ll C \\
(R2) & B & \to & B'B' & ; \ \varepsilon \\
(R3) & B & \to & \varepsilon & ; \ \varepsilon \\
(R4) & B' & \to & \varepsilon & ; \ \varepsilon
\end{array}$$

Figure 2.2: *LSL grammar which poses a problem for graph rewriting.*

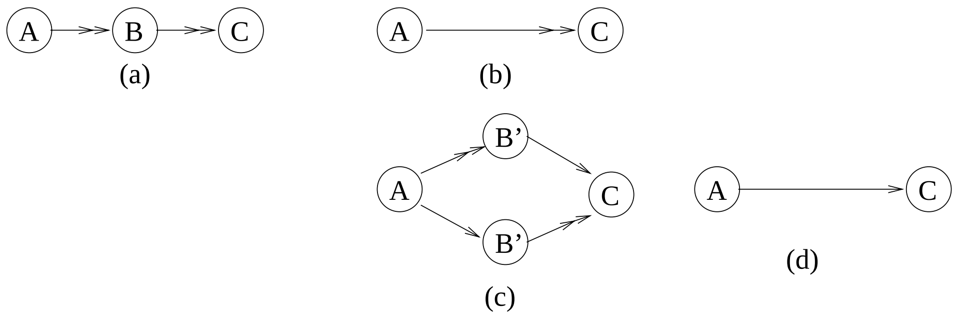


Figure 2.3: *Possible derivations with an LSL graph grammar.*

(1997) (called node-replacement grammars). Intuitively, a derivation should work as follows: We start with a graph with a single node, which is labelled with the start symbol. A rewriting step should then remove a node which is labelled with the same nonterminal as the LHS of some production, plug in the corresponding RHS rule graph, and connect the rule graph to the environment of the removed node of the original graph in a natural manner. At some point we have to convert that graph back into a string. This should be done by "guessing" some topological ordering of the graph yielding a sequence of nonterminals and applying a lexical entry to each of those nonterminals. The result is then a string.

The following problem arises when one wants to define LSL derivations in such a way. Consider the LSL grammar in Fig. 2.2 where S is the start symbol. In Fig. 2.3 you can see two possible derivations, namely (a) $\Rightarrow_{R3}$ (b) and (a) $\Rightarrow_{R2}$ (c) $\Rightarrow^2_{R4}$ (d). In (c), the idea of how to connect the two nodes labelled with B' with the A and C nodes is that we nondeterministically choose a (start) node which immediately precedes A and one (end) node which immediately precedes C. Now, whereas (b) is the desired result, (d) is *not*, because a further derivation might allow terminals neither derived by A nor C to stand between them.

However, I do *not* claim that modifying context-free graph grammars to define LSL derivations cannot be done but it seems that the machinery needed would be far more complicated than the one presented in Definition 11.

2.3 Simplifications of LSL Grammars

In this section, I will present two constructions for simplifying LSL grammars. Special cases of these constructions are well known for CFGs. First, I will show how to eliminate ε-productions from a grammar. Secondly, an algorithm is shown which transforms a grammar into an equivalent one which does not have unit productions.

2.3.1 Elimination of ε-Productions

I will now show how to construct, given an arbitrary LSL grammar, an equivalent LSL grammar without ε-productions. The construction itself is quite similar to the one presented in Hopcroft & Ullman (1979) for CFGs. Substituting ε for a nonterminal on the RHS of a "Chomsky" style production is easy: Just delete it. For LSL productions, however, this is a little bit trickier, because we have to delete a node in a rule graph.

For this purpose, I define the function delete.

Definition 12 *Define a function* delete : Node $\times$ *RG* $\rightarrow$ *RG such that*

$$\mathsf{delete}(v, (V, E_P, E_I)) = (V', E'_P, E'_I)$$

if

1. $v \in V, V' = V - \{v\}$
2. $E_P - \{(x, y) \in E_P | x = v \vee y = v\} \subseteq E'_P$
   $E_I - \{(x, y) \in E_I | x = v \vee y = v\} \subseteq E'_I$
3. *If* $(x, v) \in E_I$ *and* $(v, y) \in E_I$*, then* $(x, y) \in E'_I$.
4. *If* $(x, v) \in E_P \cup E_I$ *and* $(v, y) \in E_P \cup E_I$ *(but not both* $(x, v) \in E_I$ *and* $(v, y) \in E_I$*), then* $(x, y) \in E'_P$.

For instance, consider the graph of Fig. 2.4 (a). Suppose we want to delete node 2. With condition 3, we have to connect 1 and 3 with an i-edge (Fig. 2.4 (b)). Deleting node 3 subsequently yields Fig. 2.4 (c) due to condition 4.

If we want to delete two nodes v_1 and v_2 from a graph, it does not matter in which order they are deleted, in other words delete is associative.

Lemma 1 *Let* $R = (V, E_P, E_I) \in RG$ *and* $v_1, v_2 \in V$.

$$\mathsf{delete}(v_2, \mathsf{delete}(v_1, R)) = \mathsf{delete}(v_1, \mathsf{delete}(v_2, R))$$

Proof:
Let $R_1 = (V_1, E^1_P, E^1_I) = \mathsf{delete}(v_1, R)$, $R_{12} = (V_{12}, E^{12}_P, E^{12}_I) = \mathsf{delete}(v_2, R_1)$, $R_2 = (V_2, E^2_P, E^2_I) = \mathsf{delete}(v_2, R)$, and $R_{21} = (V_{21}, E^{21}_P, E^{21}_I) = \mathsf{delete}(v_1, R_2)$. We must now show that $R_{12} = R_{21}$.

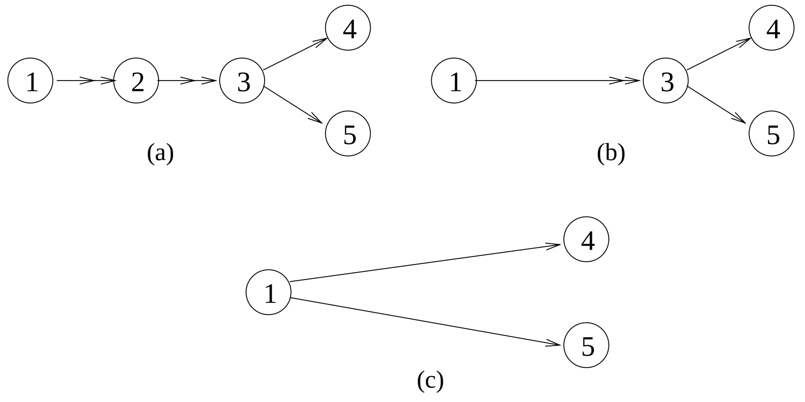


Figure 2.4: *Example for deleting nodes in a rule graph.*

1. It obviously holds that $V_{12} = V_{21} = V - \{v_1, v_2\}$.
2. Show that $(x,y) \in E_P^{12}$ iff $(x,y) \in E_P^{21}$:

   ($\Rightarrow$): Suppose $(x,y) \in E_P^{12}$. Then there are two cases to consider:

   Case 1: $(x,y) \in E_P$. It must be the case that $v_1 \neq x \neq v_2$ and $v_1 \neq y \neq v_2$ which implies that $(x,y) \in E_P^{21}$ since only edges adjacent to v_1 and v_2 are affected by delete.

   Case 2: $(x,y) \notin E_P$. One of the following two cases must be true:

   (a) $(x,v_1) \in E_P \cup E_I$ and $(v_1,y) \in E_P \cup E_I$ (where at least one of those two edges is not in E_I). Thus also $(x,y) \in E_P^{21}$ (with Definition 12, 4).

   (b) $(x,v_2) \in E_P \cup E_I$ and $(v_2,y) \in E_P \cup E_I$ (where at least one of those two edges is not in E_I). Analogously to (a), $(x,y) \in E_P^{21}$.

   (c) $(x,v_1),(v_1,v_2),(v_2,y) \in E_P \cup E_I$ (where at least one of those edges is not in E_I). Then also $(x,y) \in E_P^{21}$ with Definition 12, 4. (Analogously for the roles of v_1 and v_2 reversed.)

   ($\Leftarrow$): Completely symmetric to ($\Rightarrow$).
3. Show that $(x,y) \in E_I^{12}$ iff $(x,y) \in E_I^{21}$:

   ($\Rightarrow$): Suppose $(x,y) \in E_I^{12}$. Then there are again two cases to consider:

   Case 1: $(x,y) \in E_I$. $(x,y) \in E_I$ for the same reason as above.

   Case 2: $(x,y) \notin E_I$. Again, one of the two following cases must be true:

   (a) $(x,v_1) \in E_I$ and $(v_1,y) \in E_I$. Thus also $(x,y) \in E_I^{21}$ (with Definition 12, 3).

   (b) $(x,v_2) \in E_I$ and $(v_2,y) \in E_I$. Analogously to (a), $(x,y) \in E_I^{21}$.

(c) $(x, v_1), (v_1, v_2), (v_2, y) \in E_I$. Then also $(x, y) \in E_P^{21}$ with Definition 12, 3. (Analogously for the roles v_1 and v_2 reversed.)

($\Leftarrow$): Again, completely symmetric to ($\Rightarrow$).

□

With Lemma 1, delete can be extended to delete a set of nodes from a graph in a well-defined way:

$$\mathsf{delete}(\{v_1, \ldots, v_n\}, R) := \mathsf{delete}(v_1, \mathsf{delete}(v_2, \ldots, \mathsf{delete}(v_n, R) \ldots))$$

If two nodes are connected by a path in a rule graph, deletion of a third node does not influence this property.

Lemma 2 *Let* $R = (V, E_P, E_I) \in RG$*,* $u \in V$*, and* $R_d = \mathsf{delete}(u, R) = (V', E'_P, E'_I)$*. Let* $V \ni v \neq u \neq v' \in V$*. Then*

1. $v \overset{R}{\rightsquigarrow} v'$ *iff* $v \overset{R_d}{\rightsquigarrow} v'$

2. $v \overset{R}{\rightsquigarrow}_i v'$ *iff* $v \overset{R_d}{\rightsquigarrow}_i v'$

Proof:

1. ($\Rightarrow$) Suppose, $\alpha = vv_1 \ldots v_m v'$ is a path in R.

   (a) $v_i \neq u$ for all $i = 1, \ldots, m$: Then α is also a path in R_d with Definition 12, 2.

   (b) $v_i = u$ for some $i \in \{1, \ldots, m\}$. Then by Definition 12, 3 and 4: $(v_{i-1}, v_{i+1}) \in E'_P \cup E'_I$, thus $v \overset{R'}{\rightsquigarrow} v'$.

   ($\Leftarrow$) Suppose there is no path between v and v' in R. delete only adds an edge between two nodes which are already connected by a path. But since v and v' are not, they cannot be connected in R_d either.

2. ($\Rightarrow$) Suppose, $\alpha = vv_1 \ldots v_m v'$ is an i-path in R.

   (a) $v_i \neq u$ for all $i = 1, \ldots, m$: Then α is also an i-path in R_d with Definition 12, 2.

   (b) $v_i = u$ for some $i \in \{1, \ldots, m\}$. Then by Definition 12, 3: $(v_{i-1}, v_{i+1}) \in E'_I$, thus $v \overset{R_d}{\rightsquigarrow}_i v'$.

($\Leftarrow$) Suppose there is no i-path between v and v' in R. delete only adds an i-edge between two nodes which are already connected by an i-path. But since v and v' are not, they cannot be connected in R_d either.

□

The last fact we need about delete is the fact that a derivation with some production plus a series of ε-production is also possible by deleting the erased node from that production directly.

Lemma 3 *Let G be an LSL grammar. Let $P \ni p = (v_0 \to (V, E_P, E_I), \theta, I)$ where $V = \{v_1, \ldots, v_n\}$, $n \geq 1$, and $\theta(v_i) = X_i$. Then*

$$\begin{aligned}[(X_0, M_0)] \quad &\Rightarrow_p \quad \underbrace{[(X_1, M_1), \ldots, (X_k, \emptyset), \ldots, (X_n, M_n)]}_{=:D} \\ &\Rightarrow_G^+ \quad \underbrace{[(X_1, M_1), \ldots, (X_{k-1}, M_{k-1}), (X_{k+1}, M_{k+1}), \ldots, (X_n, M_n)]}_{=:D'}\end{aligned}$$

iff

$$[(X_0, M_0)] \Rightarrow_{p'} D'$$

where $p' = (v_0 \to \mathsf{delete}(v_k, (V, E_P, E_I)), \theta|_{V-\{v_k\}}, I - \{v_k\})$.

Proof:
Let $R_d = \mathsf{delete}(v_k, (V, E_P, E_I)) = (V', E'_P, E'_I)$. In the derivation step from D to D', the nonterminal X_k was deleted by applying one or more ε-productions.

With Lemma 2, it holds that for all $v_i, v_j \in V' \subseteq V$: $v_i \overset{R}{\leadsto} v_j$ iff $v_i \overset{R_d}{\leadsto} v_j$ and $v_i \overset{R}{\leadsto}_i v_j$ iff $v_i \overset{R_d}{\leadsto}_i v_j$. In particular, $v_i \alpha v_k \beta v_j$ is a path (i-path) in R iff $v_i \alpha\beta v_j$ is a path (i-path) in R_d.

Thus, the lemma holds.

□

With Lemma 3, we can finally present the transformation of an LSL grammar into an equivalent one without ε-productions.

Proposition 3 *For every LSL grammar G, there is an LSL grammar G' with no ε-productions such that $L(G) - \{\varepsilon\} = L(G')$.*

Proof:
Let $G = (N, T, P, L, S)$. Then construct grammar $G' = (N', T, P', L, S)$ as follows:

1. Construct the set $N_0 \subseteq N$ (the set of *nullable* nonterminals, i.e., all nonterminals which may derive the empty string) as follows: If $X \to \varepsilon; \varphi \in P$, then $X \in N_0$. If there is a production $X \to X_1 \ldots X_n; \varphi \in P$ and all $X_i \in N_0$, then $X \in N_0$. Repeat this until no new nonterminals are added to N_0. Then $N' := N - N_0$.

2. Construct P' as follows: If $(v \rightarrow (V, E_P, E_I), \theta, I) \in P$ where $V \neq \emptyset$, then add all productions $(v \rightarrow R', \theta|_{V-V^*}, I - V^*)$ to P', where there is $V^* = \{v_1^*, \ldots, v_m^*\} \subset V$ with $m \geq 0$, such that $\theta(v_i^*) \in N_0$, and $R' = \mathsf{delete}(V^*, (V, E_P, E_I))$.

I will now prove the following claim (this proof is completely analogous to the corresponding one in Hopcroft & Ullman (1979)).

Claim: $[(A, M)] \Rightarrow_{G'}^* D$ iff $[(A, M)] \Rightarrow_G^* D$ and $D \neq \emptyset$

Proof of the claim:

($\Leftarrow$) Let $[(A, M)] \Rightarrow_G^i D$ and $D \neq \emptyset$. Induction over i:

(IB) $i = 0$: trivial

(IH) $[(A, M)] \Rightarrow_G^{i-1} D$ implies $[(A, M)] \Rightarrow_{G'}^* D$

(IS) Let $[(A, M)] \Rightarrow_G [(X_1, M_1), \ldots, (X_n, M_n)] \Rightarrow_G^{i-1} D$ with production $(v \rightarrow (V, E_P, E_I), \theta, I) \in P$, $V = \{v_1, \ldots, v_n\}$ such that $\theta(v_j) = X_j$. Write $D = D_1 \cup \ldots \cup D_n$ such that for each j: $[(X_j, M_j)] \Rightarrow_G^* D_j$ in fewer than i steps in the derivation above. If $D_j \neq \emptyset$ then by (IH) we have $[(X_j, M_j)] \Rightarrow_{G'}^* D_j$. If $D_j = \emptyset$ then X_j is nullable and $M_j = \emptyset$. Let $j_1, \ldots, j_m \in \{1, \ldots, n\}$ be pairwise different such that $D_{j_k} \neq \emptyset$ for all $k = 1, \ldots, m$. Thus, there is a production $(v \rightarrow (V', E'_P, E'_I), \theta', I') \in P'$ such that there is $V^* \subset V$: $(V', E'_P, E'_I) = \mathsf{delete}(V^*, (V, E_P, E_I))$ such that for all j: $v_j \in V^*$ if $M_j = \emptyset$ and $v_j \in V'$ if $M_j \neq \emptyset$. Hence, with Lemma 3 there is a derivation

$$[(A, M) \Rightarrow_{G'} [(X_{j_1}, M_{j_1}), \ldots, (X_{j_m}, M_{j_m})] \Rightarrow_{G'}^* D_{j_1} \cup \ldots \cup D_{j_m} = D$$

($\Rightarrow$) Suppose $[(A, M)] \Rightarrow_{G'}^i D$. Surely $D \neq \emptyset$, since G' has no ε productions. Proof by induction over i.

(IB) $i = 0$: trivial

(IH) $[(A, M)] \Rightarrow_{G'}^{i-1} D$ implies $[(A, M)] \Rightarrow_G^* D$

(IS) Let $[(A, M)] \Rightarrow_{G'} [(X_1, M_1), \ldots, (X_n, M_n)] \Rightarrow_{G'}^{i-1} D$ with production $(v' \rightarrow (V', E'_P, E'_I), \theta', I') \in P'$, $V' = \{v'_1, \ldots, v'_n\}$, $\theta(v') = A$, and w.l.o.g. $\theta(v'_j) = X_j$.
There must be a production $p = (v \rightarrow (V, E_P, E_I), \theta, I) \in P$ such that there is $V^* \subset V$: $(V', E'_P, E'_I) = \mathsf{delete}(V^*, (V, E_P, E_I))$ (if $(V, E_P, E_I) = (V', E'_P, E'_I)$, V^* is set to $\emptyset$). Thus with Lemma 3:

$$\begin{array}{rcl} [(A, M)] & \Rightarrow_p & [(X_1, M_1), \ldots, (X_n, M_n), (Y_1, \emptyset), \ldots, (Y_s, \emptyset)] \\ & \Rightarrow_G^* & [(X_1, M_1), \ldots, (X_n, M_n)] \end{array}$$

with $s = |V^*|$.

Write $D = D_1 \cup \ldots \cup D_n$ such that for all j: $[(X_j, M_j)] \Rightarrow^*_{G'} D_j$ by fewer than i steps in the assumed derivation above. Then by (IH): $[(X_j, M_j)] \Rightarrow^*_G D_j$. Thus

$$[(A, M)] \Rightarrow^*_G [(X_1, M_1), \ldots, (X_n, M_n)] \Rightarrow^*_G D_1 \cup \ldots \cup D_n = D$$

□

To see that the proposition holds just note that $D \triangleright_G w$ iff $D \triangleright_{G'} w$, because G and G' have exactly the same set of lexical entries.

□

As for CFGs, elimination of ε-productions might add exponentially many new productions.

2.3.2 Elimination of Unit Productions

A unit production is a production $(v \to (V, E_P, E_I), \theta, I)$ where $|V| = 1$. As in the previous section, the construction for eliminating unit productions is quite similar to the one for CFGs. The idea is to construct a set $U(A)$ for each nonterminal A which contains all nonterminals A can be replaced by, using only unit productions. Then we add a production $A \to \alpha$ for every $B \in U(A)$ where $B \to \alpha$ is a production.

In our case, we have to be a little bit careful here, because we must distinguish two different kinds of "unit production chains", depending on whether one unit production in the chain has an isolation constraint attached to it. If $B \in U(A)$ and one of the derived productions has such an isolation constraint, then for every production $B \to \alpha; \varphi$, we have to add $A \to \alpha; \varphi, \langle A \rangle$. The exact construction is shown in the proof of the next proposition.

Proposition 4 *Let $G = (N, T, P, L, S)$ be an LSL grammar. Then there is an LSL grammar G', which can be constructed in polynomial time, such that $L(G) = L(G')$ and G' has no unit productions.*

Proof:
In some systematic fashion, we can construct all sequences

$$A_1 \rho_1 A_2 \rho_2 \ldots A_{m-1} \rho_{m-1} A_m$$

such that all $A_i \in N$ are pairwise different, there are unit productions $p_i = (v_i \to (\{u_i\}, \emptyset, \emptyset), \{v_i \mapsto A_i, u_i \mapsto A_{i+1}\}, I_i)$ for $i = 1, \ldots, m$, and $\rho_i = 0$ if $I_i = \emptyset$ and $\rho_i = 1$, otherwise. These sequences represent the mentioned unit production chains. There are only polynomially many of those chains since all A_i are required to be pairwise different. During the construction of those sequences, we can in parallel build the following sets for each $A \in N$. If there is a sequence

$A\rho_1 \dots \rho_m B$ then $B \in U(A)$ iff $\rho_i = 0$ for all $i = 1, \dots, m$ and $B \in U_I(A)$ iff there is an $i \in \{1, \dots, m\}$ such that $\rho_i = 1$. Constructing $U(A)$ and $U_I(A)$ can be done in polynomial time.

We can now construct the new grammar $G' = (N, T, P', L, S)$ as follows

1. Add all non unit productions of P to P'.
2. If there is $(v \rightarrow (V, E_P, E_I), \theta, I) \in P$ with $|V| \geq 2$ and $\theta(v) = A$. then
   (a) add $(v \rightarrow (V, E_P, E_I), \theta[v \mapsto B], I)$ to P' if $B \in U(A)$
   (b) add $(v \rightarrow (V, E_P, E_I), \theta[v \mapsto B], I \cup \{v\})$ to P' if $B \in U_I(A)$

It should be clear that G' "simulates" all possible unit production chains and, additionally, takes care of the isolation constraints. Furthermore, since $U(A)$ and $U_I(A)$ are polynomial in size, constructing G' takes polynomial time (in the size of G).

□

2.4 Context-Free Subsets

It is possible to construct, for a given LSL grammar G, a context-free grammar $\mathsf{cf}(G)$ with $L(\mathsf{cf}(G)) \subseteq L(G)$. Loosely speaking, this is done by topologically sorting the rule graph on the RHS. This grammar has a lot of useful properties.

Definition 13 *Define a function* $\mathsf{cf} : LSLG \rightarrow CFG$ *as follows:*

$$\mathsf{cf}((N, T, P, L, S)) = (N, T, P', S)$$

if

1. *If* $(v \rightarrow (V, E_P, E_I), \theta, I) \in P$ *then* $\theta(v) \rightarrow \theta(\sigma(1)) \dots \theta(\sigma(n)) \in P'$ *where* $\sigma : \{1, \dots, |V|\} \rightarrow V$ *is such that* $\sigma(1) \dots \sigma(|V|) = \mathsf{topsort}((V, E_P, E_I))$.
2. *If* $X \rightarrow a \in L$ *then* $X \rightarrow a \in P'$

cf is constructed in two steps: First, all rule graphs are sorted using $\mathsf{topsort}$ and, second, all lexical entries are simply added to the set of new productions.

If $X \rightarrow X_1 \dots X_n$ is a context-free production being the result of this construction, all X_i are isolated and each X_i immediately precedes X_{i+1}. In particular, the LP constraints of the original LSL production are satisfied. Thus, the following holds:

Proposition 5 $L(\mathsf{cf}(G)) \subseteq L(G)$.

Proof:
Let $G = (N, T, P, L, S)$ be an LSL grammar. Suppose there is a derivation $S \Rightarrow^*_{\mathsf{cf}(G)} w = w_1 \dots w_n$. W.l.o.g. we can assume that this derivation is of the form

$$\begin{aligned} S = \alpha_0 &\Rightarrow_{\mathsf{cf}(G)} \alpha_1 \Rightarrow_{\mathsf{cf}(G)} \dots \Rightarrow_{\mathsf{cf}(G)} \alpha_k = A_1 \dots A_n \\ &\Rightarrow_{\mathsf{cf}(G)} a_1 A_2 \dots A_n \Rightarrow_{\mathsf{cf}(G)} a_1 a_2 A_3 \dots A_n \Rightarrow_{\mathsf{cf}(G)} \dots \Rightarrow_{\mathsf{cf}(G)} a_1 \dots a_n = w \end{aligned}$$

with $\alpha_i \in N^*$, $a_i \in T$. Let $\mathsf{span}_i(j)$ be the set of string positions the jth nonterminal from the left in sentential form α_i derives, e.g., $\mathsf{span}_k(l) = l$ for all $l = 1, \dots, n$. Construct a sequence $D_0, D_1, \dots, D_k \in \mathcal{B}(N \times \mathsf{Fin}(I\!N))$ as follows:

If $\alpha_i = X_1 \dots X_m$ then $D_i = [(X_1, \mathsf{span}_i(1)), \dots, (X_m, \mathsf{span}_i(m))]$. Thus, $D_0 = [(S, \{1, \dots, n\})]$, $D_k = [(A_1, \{1\}), \dots, (A_n, \{n\})]$, and $D_k \triangleright_G a_1 \dots a_n = w$.

For all $i = 1, \dots, k$ it holds by definition of span:

1. For all $l = 1, \dots, |\alpha_i|$: $\mathsf{cont}(\mathsf{span}_i(l))$.
2. If $\mathsf{span}_i(l) \neq \emptyset \neq \mathsf{span}_i(q)$ with $l < q \leq |\alpha_i|$, then $\max(\mathsf{span}_i(l)) < \min(\mathsf{span}_i(q))$.
3. If $\mathsf{span}_i(l) \neq \emptyset \neq \mathsf{span}_i(q)$ with $l < q \leq |\alpha_i|$, and for all $d = l+1, \dots, q-1$: $\mathsf{span}_i(d) = \emptyset$, then $\max(\mathsf{span}_i(l)) = \min(\mathsf{span}_i(q)) - 1$.

It now remains to show that $D_0 \Rightarrow^*_G D_k$. Proof by induction over length of this derivation i.

(IB) i=0: D_0 is, of course, a derivation.

(IH) $D_0 \Rightarrow^*_G D_i$

(IS) Let $\alpha_i = X_1 \dots X_{j-1} X_j X_{j+1} \dots X_s$ and $\alpha_{i+1} = X_1 \dots X_{j-1} Y_1 \dots Y_r X_{j+1} \dots X_s$ be the result of applying the production $p = X_j \to Y_1 \dots Y_r$ of $\mathsf{cf}(G)$. Then

$$D_i = [(X_1, M_1), \dots, (X_j, M_j), \dots, (X_s, M_s)]$$

and

$$D_{i+1} = [(X_1, M_1), \dots, (Y_1, M'_1), \dots, (Y_r, M'_r), \dots, (X_s, M_s)]$$

where $M'_l = \mathsf{span}_{i+1}(l + j - 1)$ for $l = 1, \dots, r$, by definition of the D_i. Let $(v \to \underbrace{(V, E_P, E_I)}_{=:R}, \theta, I) \in P$ be the production p was constructed from where $V = \{v_1, \dots, v_r\}$ and $\theta = \{v_l \mapsto Y_l | l = 1, \dots, r\}$. It holds that

1. $\theta(v_l) = Y_l$ for all $l = 1, \dots, r$.

2. $M_j = \mathsf{span}_i(j) = \bigcup_{l=1}^{r} \mathsf{span}_{i+1}(l+j-1) = \bigcup_{l=1}^{r} M'_j$ by definition of span.
3. (a) $\mathsf{cont}(M'_l)$ for all $l = 1, \ldots, r$ with (1) from above
   (b) If $v_l \overset{R}{\rightsquigarrow} v_q$, $l < q$ and $M'_l \neq \emptyset \neq M'_q$, then with (2) from above: $\max(M'_l) < \min(M'_q)$.
   (c) If $v_l v_{l_1} \ldots v_{l_m} v_q$ is i-path in R with $M'_l \neq \emptyset \neq M'_q$ and $M_{l_t} = \emptyset$ for all $t = 1, \ldots, m$, then with (3) from above: $\max(M'_l) = \min(M'_q) - 1$.

Thus, all conditions of Definition 9 are satisfied.

□

Obviously, $L(G) \not\subseteq L(\mathsf{cf}(G))$ in the general case, but it can be shown that $L(\mathsf{cf}(G)$ is letter equivalent to $L(G)$.

Definition 14 (Letter equivalence) *Two words $w_1, w_2 \in T^*$ are called* letter equivalent *if for all $a \in T$: $\#_a(w_1) = \#_a(w_2)$ (where $\#_a(w)$ is the number of occurrences of the terminal a in word w).*

Two languages $L_1, L_2 \subseteq T^*$ are called letter equivalent if for each $w_1 \in L_1$, there is a letter equivalent $w_2 \in L_2$ and vice versa.

Proposition 6 *$L(G)$ is letter equivalent to $L(\mathsf{cf}(G))$.*

Proof:
($\Rightarrow$) Suppose there is a derivation

$$[(S, \{1, \ldots, n\})] \Rightarrow_G^* [(A_1, M_1), \ldots, (A_n, M_n)] \triangleright_G w_1 \ldots w_n = w$$

with $w_j \in T$. Let $t_i = w_j$ if $M_i = \{j\}$. Then there also exists a derivation

$$S \Rightarrow_{\mathsf{cf}(G)}^* A_1 \ldots A_n \Rightarrow_{\mathsf{cf}(G)}^* t_1 \ldots t_n = w'$$

Since all M_i are pairwise disjoint, it holds that w' is a permutation of w. In other words: for all $a \in T$: $\#_a(w) = \#_a(w')$.

($\Leftarrow$) If $w \in L(\mathsf{cf}(G))$ then with Proposition 5, $w \in L(G)$. Since w is letter equivalent to itself, the proposition holds.

□

As direct corollaries, we have

Corollary 2 *$L(G) = \emptyset$ iff $L(\mathsf{cf}(G)) = \emptyset$.*

Corollary 3 *$L(G)$ is finite iff $L(\mathsf{cf}(G))$ is finite.*

When we consider a unary terminal alphabet, it holds that two words are letter equivalent iff they are equal.

Corollary 4 *Let G be an LSL grammar over a unary alphabet. Then $L(G) = L(\mathsf{cf}(G))$.*

In Salomaa (1973), theorem 7.3. (p. 68) proves that context-free languages over a unary terminal alphabet are regular. Combining this with Corollary 4 yields

Corollary 5 *An LSLL over a unary alphabet is regular.*

A remark about the complexity of cf is in order. With Proposition 2, it immediately follows that

Corollary 6 cf *can be computed in polynomial time.*

Taken together, Propositions 5 and 6 can be used to show

Proposition 7 *Let G be an LSL grammar. If for all n, it holds that $|\{w \in L(G) | |w| = n\}| \leq 1$, then $L(\mathsf{cf}(G)) = L(G)$.*

Proof:
"$\subseteq$" follows directly from Proposition 5. We now have to show that $L(G) \subseteq L(\mathsf{cf}(G))$. Let $w \in L(G)$, $|w| = n$. Then there is a letter equivalent $w' \in L(\mathsf{cf}(G))$ to w, where $|w'| = n$. Since, in turn, $w' \in L(G)$ and w is the only element in $L(G)$ with length n, it must be the case that $w = w'$.

□

2.5 LSL in the Chomsky Hierarchy

First, I will take a look at where $LSLL$ stand in relation to CFL. It turns out that LSL grammars are a proper extension of CFGs.

Proposition 8 $CFL \subset LSLL$

Proof:

($\subseteq$) Let $G = (N, T, P, S)$ be a context-free grammar in Chomsky normal form. Then it is possible to effectively construct an equivalent LSL grammar $G_L = (N, T, P_L, L, S)$ as follows:

1. For all productions in P of the form $X \to a$, $a \in T$, add $X \to a$ to L.
2. For all productions in P of the form $X \to AB$, $A, B \in N$, add $X \to AB; A \ll B$ to P_L.

($\neq$) Consider the following LSL grammar G_{eq}:

$$\begin{array}{lcl} S & \rightarrow & ABCS; \varepsilon \\ S & \rightarrow & ABC; \varepsilon \\ A & \rightarrow & a \\ B & \rightarrow & b \\ C & \rightarrow & c \end{array}$$

G_{eq} generates the language: $L_{eq} := L(G_{eq}) = \{w \in (a|b|c)^+ | \#_a(w) = \#_b(w) = \#_c(w)\}$ and this language is *not* context-free. This is because CFL is closed under intersection with regular languages. So if $L(G_{eq})$ were context-free, then $L(G_{eq}) \cap a^*b^*c^*$ would be context-free. But $L(G_{eq}) \cap a^*b^*c^* = \{a^nb^nc^n | n \geq 1\}$ which is not context-free.

□

Apparently, LSL grammars can "count" more than CFGs, but they are not able to order (more than two) terminals "globally". As a result, $a^nb^nc^n$ cannot be generated by any LSL grammar.

Proposition 9 $\{a^nb^nc^n | n \geq 1\} \notin LSLL$

Proof:
Suppose $L := \{a^nb^nc^n | n \geq 1\} \in LSLL$ with LSL grammar G. L does not contain two words with the same length. Then with Proposition 7, it holds that L can be generated by the context-free grammar $\mathsf{cf}(G)$, which is a contradiction since L cannot be generated by any context-free grammar.

□

It follows directly that the context-sensitive languages (CSL) are not contained in the LSL languages:

Corollary 7 $CSL \not\subseteq LSLL$

2.6 Closure Properties

In this section, I want to investigate some of the "classic" formal language closure properties of $LSLL$. Most of the proofs are very similar to the corresponding ones in Hopcroft & Ullman (1979) for CFGs.

Proposition 10 *$LSLL$ is closed under union, concatenation, and Kleene closure.*

Proof:
Let $L_1 = L(G_1)$ and $L_2 = L(G_2)$ with the two LSL grammars $G_1 = (N_1, T_1, P_1, L_1, S_1)$ and $G_2 = (N_2, T_2, P_2, L_2, S_2)$. We can assume that $N_1 \cap N_2 = \emptyset$.

For $L_1 \cup L_2$ we can construct the LSL grammar $G_U = (N_1 \cup N_2 \cup \{S_U\}, T_1 \cup T_2, P_U, L_1 \cup L_2, S_U)$ where S_U is a new symbol and $P_U = P_1 \cup P_2 \cup \{S_U \to S_1, S_U \to S_2\}$. It obviously holds that $L(G_U) = L(G_1) \cup L(G_2)$.

The language generated by LSL grammar $G_C = (N_1 \cup N_2 \cup \{S_C\}, T_1 \cup T_2, P_C, L_1 \cup L_2, S_C)$ where S_C is a new symbol and $P_C = P_1 \cup P_2 \cup \{S_C \to S_1 S_2; S_1 \ll S_2\}$ is $L_1 L_2$. The immediate precedence constraint ensures that all terminals derived from S_1 appear to the left of all terminals derived from S_2.

For generating the Kleene closure of L_1, consider the LSL grammar $G_K = (N_1 \cup \{S_K\}, T_1, P_K, L_1, S_K)$ where S_K is an unused symbol and $P_K = P_1 \cup \{S_K \to S_1 S_K; S_1 \ll S_K, S_K \to \varepsilon\}$. Analogously to above, it holds that $L(G_K) = L(G_1)^*$.

□

Proposition 11 *LSLL is not closed under intersection.*

Proof:
As shown in Hopcroft & Ullman (1979), the languages $L_1 = \{a^i b^i c^j | i, j \geq 1\}$ and $L_2 = \{a^j b^i c^i | i, j \geq 1\}$ are context-free. Thus they are also in $LSLL$ (Proposition 8). Their intersection $L = L_1 \cap L_2 = \{a^i b^i c^i | i \geq 1\}$ is not in $LSLL$ (Proposition 9).

□

Proposition 12 *LSLL is not closed under complement.*

Proof:
Let $L_1, L_2 \in LSLL$. Suppose $LSLL$ is closed under complement. Then $L_1 \cup L_2 = \overline{\bar{L}_1 \cup \bar{L}_2}$. Since $LSLL$ is closed under union, it follows that it is closed under intersection which is a contradiction to Proposition 11.

□

Proposition 13 *LSLL is closed under substitutions.*

Proof:
Let $L \subseteq T^*$ be an $LSLL$ and for each $a \in T$ let L_a be an $LSLL$. We have thus LSL grammars $G = (N, T, P, L, S)$ with $L = L(G)$ and for each $a \in T$: $G_a = (N_a, T_a, P_a, L_a, S_a)$ with $L_a = L(G_a)$.

One can then construct an LSL grammar $G' = (N', T', P', L', S)$ with

$$\begin{aligned}
N' &= N \cup \bigcup_{a \in T} N_a \\
T' &= \bigcup_{a \in T} T_a \\
P' &= P \cup \bigcup_{a \in T} P_a \cup \{A \to S_a; \langle A \rangle | A \to a \in L\} \\
L' &= \bigcup_{a \in T} L_a
\end{aligned}$$

This grammar uses the productions of G to derive some sentential form but instead of terminating with terminal a, it "invokes" G_a. Since the LHSs of the new productions in P' are isolated, it is ensured that a is substituted by a contiguous word of L_a. More formally:

Suppose there is a derivation

$$[(S, \{1, \ldots, n\})] \Rightarrow_G^* [(A_1, \{1\}), \ldots, (A_n, \{n\})] \triangleright_G a_1 \ldots a_n$$

Then the following is also a valid derivation

$$[(S, M_1 \cup \ldots \cup M_n)] \Rightarrow_G^* [(A_1, M_1), \ldots, (A_n, M_n)]$$

where the M_i are contiguous index sets, pairwise disjoint, and $M_1 \cup \ldots \cup M_n$ is contiguous. Note that if some $M_i = \emptyset$, this is still true. This derivation is also possible in G' by construction.

Now if there are derivations

$$[(S_{a_i}, M_i)] \Rightarrow_{G'}^* D_i \triangleright_{G'} w_i$$

for all i, there is a derivation

$$\begin{aligned}
[(S, M_1 \cup \ldots \cup M_n)] &\Rightarrow_{G'}^* [(A_1, M_1), \ldots, (A_n, M_n)] \\
&\Rightarrow_{G'} [(S_{a_1}, M_1), \ldots, (S_{a_n}, M_n)] \\
&\Rightarrow_{G'}^* D_1 \cup \ldots \cup D_n \triangleright_{G'} w_1 \ldots w_n
\end{aligned}$$

□

Since substitutions are a generalisation of homomorphisms, it immediately follows that

Corollary 8 *LSLL is closed under homomorphism.*

The last results also hold for CFL, but unlike CFL, $LSLL$ is not closed under intersection with regular sets.

Proposition 14 *LSLL is not closed under intersection with regular sets.*

Proof:
This follows from the fact that $L_{eq} = \{(a|b|c)^+ | \#a = \#b = \#c\} \in LSLL$ (see proof of Proposition 8) and $L_{eq} \cap a^*b^*c^* = L_n \notin LSLL$ (Proposition 9).

□

This result suggests that it is difficult to find a straightforward automaton model which recognises LSL languages.

Another important property of formal languages (especially in the context of AFL theory) is closure under inverse homomorphism, i.e., if every member of a class of languages is the homomorphic image of another member of that class. In Ginsburg (1975) (theorem 3.7.2, p. 74), the following was shown:

	$\cup$	$\cdot$	$*$	$\cap$	co	subs (hom)	$\cap REG$	inv hom
CFL	y	y	y	n	n	y	y	y
$LSLL$	y	y	y	n	n	y	n	n

Table 2.1: *Closure properties of $LSLL$.*

Proposition 15 (Ginsburg) *Let $\mathcal{L}$ be a family of ε-free languages. If $\mathcal{L}$ is not closed under intersection with regular sets, then it is not true that $\mathcal{L}$ is closed under concatenation, ε-free homomorphism and inverse homomorphism.*

Since $LSLL$ is not closed under intersection with regular sets and closed under concatenation and ε-free homomorphism, it cannot be the case that it is closed under inverse homomorphism.

Corollary 9 *$LSLL$ is not closed under inverse homomorphisms.*

Table 2.1 provides an overview of all of the closure properties in contrast to CFL.

2.7 Decidability

When talking about decidability issues, we always assume that the underlying alphabet has at least two elements, since otherwise the considered language is regular (see Corollary 5).

As for CFGs, the emptiness, finiteness, and membership problem are decidable.

Proposition 16 *Emptiness and Finiteness of LSL grammars is decidable.*

Proof:
With Corollaries 2 and 3 one can decide emptiness and finiteness of an LSL grammar G by computing $\mathsf{cf}(G)$ and deciding the problem for $\mathsf{cf}(G)$. Emptiness and finiteness are decidable for context-free grammars (cf. Hopcroft & Ullman (1979)). Since topsort and thus $\mathsf{cf}(G)$ is decidable, the proposition holds.

$\square$

Proposition 17 *The membership problem for LSL grammars is decidable.*

Proof:
I will show how a nondeterministic Turing machine (NTM) $\mathcal{M}$ can decide if a given string x belongs to $L(G)$ for a given LSL grammar G. First, we transform G into a grammar G' with no ε and no unit productions. If $x = \varepsilon$ and S is nullable in G, we accept. Otherwise, $x = x_1 \ldots x_n \neq \varepsilon$.

$\mathcal{M}$ will simulate a derivation. It starts with writing $(S, \{1, \ldots, |x|\})$ onto the tape. A derivation step is simulated in a straightforward way: Suppose the tape looks like this:

$$(X_1, M_1) \ldots (X_n, M_n)$$

$\mathcal{M}$ then nondeterministically "guesses" some production $Y \to Y_1 \ldots Y_r; \varphi$ such that $Y = X_k$ for some $k \in \{1, \ldots, n\}$ and simulates the derivation completely analogous to Definition 9, thus the tape will look like this:

$$(X_1, M_1) \ldots (Y_1, M_1') \ldots (Y_r, M_r') \ldots (X_n, M_n)$$

Now the following holds: Since G' has no unit productions, $r \geq 2$, in other words the number of tagged nonterminals strictly increases with every simulated derivation step. Since G' also has no ε-productions, $M_i' \neq \emptyset$ for all $i = 1, \ldots, r$. This means that it will eventually be the case that every index set is singleton, thus every possible computation will in a finite number of steps end with a situation like this:

$$(C_1, S_1) \ldots (C_m, S_m)$$

where w.l.o.g. $S_i = \{i\}$. If $m = |x|$ and there are lexical entries $S_i \to x_i \in L$, $\mathcal{M}$ accepts, otherwise it rejects.

Since, every simulation of $\mathcal{M}$ halts, $\mathcal{M}$ can be simulated by a deterministic Turing machine. Hence, the membership problem is decidable.

□

Note however, that the simulation of the derivation does need more than linear space because there are n slots for nonterminals each requiring space $O(n)$ for storing the index sets. Thus, $O(n^2)$ space is needed overall which does *not* prove that $LSLL \subseteq CSL$.

All undecidability results for CFLs of course also hold for $LSLL$. The next proposition presents some of those.

Proposition 18 *Let $T^* \supseteq L, L' \in LSLL$, $R \in REG$. It is undecidable if $L = T^*$, $L = L'$, $L \subseteq L'$, $L = R$, $L \subseteq REG$, and $L \cap L' = \emptyset$.*

Proof:
That follows directly from the corresponding undecidability results for CFL and that $CFL \subseteq LSLL$. In particular, the proof of Proposition 8 shows an effective construction, given a context-free grammar, of an equivalent LSL grammar.

□

Until now, we basically have the same decidability results for CFL and $LSLL$. We will, however, see later that the complexity of especially the membership problem differs remarkably.

A question that remains is this: Given an LSL grammar G, is $L(G)$ context-free? Unfortunately, it is not possible to answer this question in the general case. The next Proposition was proven in Holzer (1999).

	$L = T^*$	$L_1 = L_2$	$L_1 \subseteq L_2$	$L = R$, $R \in REG$	$L \subseteq REG$
CFL	undec.	undec.	undec.	undec.	undec.
$LSLL$	undec.	undec.	undec.	undec.	undec.

	$L_1 \cap L_2 = \emptyset$	$w \in L$	$L = \emptyset$	$\|L\| < \infty$	$L \in CFL$
CFL	undec.	dec.	dec.	dec.	triv.
$LSLL$	undec.	dec.	dec.	dec.	undec.

Table 2.2: *Decidability properties of LSLL (entries mean undecidable, decidable, trivially decidable)*

Proposition 19 *Let G be an LSL grammar. It is undecidable whether $L(G)$ is context-free.*

Table 2.2 provides an overview over the decidability properties.

2.8 Complexity

In this section I want to take a look at complexity theoretic aspects of LSL grammars, especially at the problems that were shown to be decidable in section 2.7. It will be shown that from this point of view, LSL grammars are a "proper" generalisation of CFGs, as LSL recognition is more difficult than the polynomial time recognition for CFGs (in fact, the general membership problem for CFGs is $\mathcal{P}$-complete, cf. Greenlaw, Hoover & Ruzzo (1995) whereas the fixed membership problem is $LOGCFL$-complete, cf. Johnson (1990)).

First, I will take a look at the emptiness and finiteness problems.

2.8.1 Emptiness and Finiteness

Proposition 20 *The emptiness and finiteness problems for LSL grammars are $\mathcal{P}$-complete.*

Proof:
Let E_{CFL} (F_{CFL}) be the emptiness (finiteness) problem for context-free languages and E_{LSLL} (F_{LSLL}), for LSL languages.

(in $\mathcal{P}$): Let G be an LSL grammar. With Corollaries 2 and 3, the problem is decidable by computing $\mathsf{cf}(G)$ (which is possible in polynomial time according to Corollary 6) and solving the problem for $\mathsf{cf}(G)$. Since E_{CFL} and F_{CFL} are $\mathcal{P}$-complete (cf. Greenlaw et al. 1995) and thus also in $\mathcal{P}$, this algorithm needs polynomial time.

($\mathcal{P}$-hardness): (I will only show the proof for E_{LSLL} since the one for F_{LSLL} is completely analogous.) Since E_{CFL} is $\mathcal{P}$-complete, it holds that for all $L \in \mathcal{P}$, L reduces to E_{CFL} with a logspace reduction f such that $x \in L \iff f(x) \in E_{CFL}$, i.e., $f(x)$ is (the encoding of) a context-free grammar G where $L(G) = \emptyset$ iff $x \in L$.

Let g be a function which takes (an encoding of) a context-free grammar and "transforms" it into (an encoding of) an LSL grammar as follows:

First, a new CFG G' is constructed which has only productions of the form $N \times N^*$ or $N \times T$. This can be done by replacing every terminal a on the RHS of a production by a new nonterminal $\bar{a}$ and adding a new production $a \to \bar{a}$ to G'. Building G' is possible in constant space.

Secondly, an LSL grammar is constructed from G' by copying every production of G' and connecting all nonterminals on the RHS by an i-edge in their occurring order. To do that, the number of nonterminals on this RHS must be counted which takes logarithmic space.

(This construction was also shown in the proof of Proposition 8).

Since it holds that

$$x \in L \iff g(f(x)) \in E_{LSLL}$$

and the composition of logspace reductions is also a logspace reduction (Papadimitriou 1995), we have shown that every $L \in \mathcal{P}$ is reducible to E_{LSLL}.

□

2.8.2 Membership

In this section I will investigate the complexity both of the general and fixed membership problem. These two differ in that for the former, the input is considered to consist of the LSL grammar plus the string to be recognised whereas for the latter, the input consists of the string only.

Proposition 21 *The general membership problem for LSL grammars is $\mathcal{NP}$-complete.*

Proof:
($\in \mathcal{NP}$) Let $G = (N, T, P, L, S)$ be an LSL grammar. We can assume that G has no unit productions (cf. Proposition 4). The NTM $\mathcal{M}$ deciding if a string $x \in L(G)$ works exactly as the one described in the proof of Proposition 17. There is, however, a small difficulty: the grammar resulting from eliminating all ε-productions might be exponentially bigger than G. Instead, $\mathcal{M}$ computes the

set of nullable symbols (in polynomial time) and each time a nonterminal $(X, \emptyset)$ appears on the tape, $\mathcal{M}$ checks if X is nullable, and if so, it is deleted from the tape and the computation continues, otherwise $\mathcal{M}$ halts and rejects.

Since the derivation tree is only polynomial in size and G does not contain unit productions, all computations of $\mathcal{M}$ take polynomial time.

($\mathcal{NP}$-hardness) Proof by reduction from the general membership problem for UCFGs (see section 1.2.2).

It is easy to see that UCFGs are a special case of LSL grammars: a UCFG is an LSL grammar with no precedence constraints and all nonterminals being isolated. A UCFG rule $X \rightarrow \{X_1, \ldots, X_n\}$ (which means that X may be rewritten by any permutation of the X_i) can be simulated by the LSL rule $X \rightarrow X_1 \ldots X_n; \langle X_1 \rangle, \ldots, \langle X_n \rangle$.

Since the general membership for UCFGs is $\mathcal{NP}$-complete, as was shown in Huynh (1983) and Barton et al. (1987), the proposition holds.

$\square$

What happens with the complexity if we hold the grammar fixed, i.e., the grammar is no longer part of the input? One could suspect that this makes the problem easier. For UCFGs, the fixed membership problem is indeed in $\mathcal{P}$ (which for instance follows from the complexity analysis in section 3.3). But, unfortunately, for LSL grammars this is not the case.

Proposition 22 *The fixed membership problem for LSL grammars is $\mathcal{NP}$-complete.*

Proof:
($\in \mathcal{NP}$) This follows directly from the fact that the general membership problem is in $\mathcal{NP}$, see Proposition 21.

($\mathcal{NP}$-hardness) This was proven in Holzer & Suhre (1999) by reduction from the tripartite matching problem (cf. Garey & Johnson 1979).

$\square$

2.9 Unification-Based Grammars

Unification-Based (context-free) Grammars (UBGs) are like CFGs, but they use more complex structures than nonterminals (*complex categories*). Basically, only one operation on these structures needs to be defined: unification. In this section I will briefly define UBGs and show how LSL can be extended towards a similar notion. For simplicity, I will use a well known instance of complex categories, namely first order terms.

Definition 15 (UBG) *A Unification-Based Grammar (UBG) is a tuple $(\mathcal{F}, T, P, t_S)$ where $\mathcal{F}$ is the set of first order terms over some fixed signature, T is the set of terminal symbols, $P \subseteq \mathcal{F} \times (\mathcal{F} \cup T)^*$ is the set of rules, and $t_S \in \mathcal{F}$ is the start term.*

Note that $\mathcal{F}$ which is the analogue to the set of nonterminals in conventional CFGs, is infinite in the general case.

Derivations are defined analogously to CFGs with the difference that not only nonterminals in a sentential form which are equal to the LHS of some rule can be expanded, but also those who unify with it.

Definition 16 *Let $UG = (\mathcal{F}, T, P, t_S)$ be a UBG. The derivation relation $\Rightarrow_{UG} \subseteq (\mathcal{F} \cup T)^* \times (\mathcal{F} \cup T)^*$ is defined as follows: $\alpha t \beta \Rightarrow_{UG} \gamma$ iff*

- *$t' \to t_1 \dots t_n$ is a copy of a rule in $r \in P$ such that all variables of r are replaced by fresh ones.*
- *t unifies with t' with most general unifier (mgu) μ.*
- *$\gamma = (\alpha t_1 \dots t_n \beta)\mu$.*

If one considers strings as being constants (first order terms with no arguments and no variables), CFGs can be considered as a special case of UBGs because two constants unify iff they are equal. The language of a UBG is the set of strings derived by the start term.

Definition 17 *Let $UG = (\mathcal{F}, T, P, t_S)$ be a UBG. Then the language of UG is defined as $L(UG) := \{w \in T^* | t_S \Rightarrow^*_{UG} w\}$.*

2.9.1 An Example

For the following example, I will use Prolog notation, i.e., constants start with a lowercase, variables with an uppercase letter.

```
np(Gen,Case)    →  det(Gen,Case) n(Gen,Case)
det(masc,nom)   →  der
n(masc,_)       →  mann
det(fem,nom)    →  die
n(fem,_)        →  frau
```

The first rule expresses the fact that the gender and case of a determiner must match those of the noun to form a valid nominal phrase. Suppose the start term is `np(X,Y)`. An example derivation is then

```
np(X,Y)  ⇒G  det(X,Y) n(X,Y)    {X/X,Y/Y}
         ⇒G  det(masc,Y) mann   {X/masc,Y/Y}
         ⇒G  der mann           {X/masc,Y/nom}
```

On the right, you see the current instantiations of the variables.

Now, if the grammar is used for parsing, the string `der mann` will yield the instantiated term `np(masc,nom)`.

Definite clause grammars (DCGs, see Gazdar & Mellish 1989) are basically UBGs over first order terms. Additionally, they provide the possibility to attach arbitrary procedures, i.e., Prolog predicates, to a rule.

2.9.2 Beyond First Order Terms

UBGs can be defined over more complex structures than first order terms. A prominent example is the typed feature structures used in HPSG (cf. Pollard & Sag 1994), formally described in Carpenter (1992) or King (1994). The ALE system (cf. Carpenter & Penn 1998) is an example of an efficient Prolog implementation of UBGs using typed feature structures.

For more on UBGs, see Gerdemann (1991), Shieber (1986), Covington (1995), or Gazdar & Mellish (1989).

It is easy to extend LSL towards a unification-based approach in a similar fashion. Since for some structures, the unification operation might not even be decidable, the formal language and complexity-theoretic properties of LSL with strings as nonterminals are not applicable any more.

The next chapter presents a parsing algorithm for LSL grammars. With Propositions 21 and 22 we cannot expect that this algorithm has less than exponential runtime. However, a reasonable condition on LSL grammars will be identified such that the fixed recognition problem with grammars satisfying this condition is solvable in polynomial time.

Chapter 3

Parsing

This chapter presents a parsing algorithm for LSL grammars which is based on Earley's algorithm for CFGs. A complexity analysis of this algorithm reveals a sufficient condition on grammars which allows parsing grammars with this property in polynomial time.

I will now briefly describe Earley's algorithm mainly to introduce the notation I will use later. This version is similar to the presentation in, e.g., Gazdar & Mellish (1989).

3.1 Earley's Algorithm

I assume a fixed CFG (N, T, P, S') being start-separated such that there is only one rule $S' \to S$. I furthermore assume that every rule in P looks either like $X \to X_1 \ldots X_n$ where $X_i \in N$ or $X \to w$ where $w \in T$.

Earley's algorithm operates on a chart (well-formed substring table). It makes one single pass through the input string x. $x[k]$ is the terminal at position k in x. Every chart entry looks either like

1.
$$\langle i, j, X \to X_1 \ldots X_m \bullet X_{m+1} \ldots X_n \rangle$$
where $i, j \in I\!N$, $X \to X_1 \ldots X_n \in P$, or

2.
$$\langle i, i+1, X \to w \bullet \rangle$$
where $i \in I\!N$, $X \to w \in P$, and $w \in T$.

In the following, I write α, β, γ for strings in $(N \cup T)^*$, X, Y for nonterminals, and w for a terminal.

A chart entry of the form

$$\langle i, j, X \to \alpha \bullet \rangle$$

is called an *inactive edge* (or *passive edge*), every other entry is called an *active edge*.

At initialisation, k is set to 0, and an initial edge

$$\langle 0, 0, S' \to \bullet S \rangle$$

is inserted into the chart.

At each step, three operations are performed until there are no more to perform.

1. Scanning (Fig. 3.1):
   If $x[k] = w$ and there is $X \to w \in P$ then add
   $$\langle k, k+1, X \to w \bullet \rangle$$
   to the chart.

2. Prediction (Fig. 3.2):
   If there is an edge
   $$\langle i, k, X \to \alpha \bullet Y \beta \rangle$$
   in the chart then for every rule $Y \to \gamma$, add edge
   $$\langle k, k, Y \to \bullet \gamma \rangle$$
   to the chart

3. Completion (Fig. 3.3):
   If there is an inactive edge
   $$\langle j, k, X \to \alpha \bullet \rangle$$
   and an active edge
   $$\langle i, j, Y \to \beta \bullet X \gamma \rangle$$
   in the chart then add edge
   $$\langle i, k, Y \to \beta X \bullet \gamma \rangle$$
   to the chart

At each string position k, one scanning step is performed and prediction plus completion are repeated until they yield no new edges. Then k is incremented by one and these steps are repeated. When $k = |x| + 1$, the algorithm terminates. If there is an edge

$$\langle 0, |x| + 1, S' \to S \bullet \rangle$$

at that point, x is in the language and we accept.

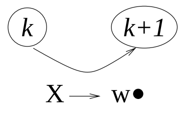


Figure 3.1: *Scanning in Earley's algorithm: If there is w at position k, we add the corresponding inactive edge.*

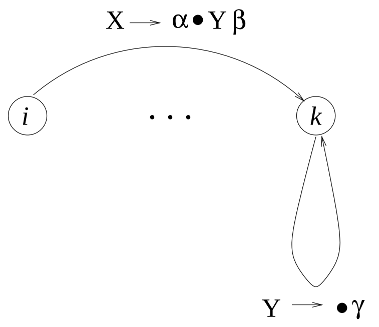


Figure 3.2: *Prediction in Earley's algorithm: If we want to find a Y starting at position k, we add edges for all rules expanding Y.*

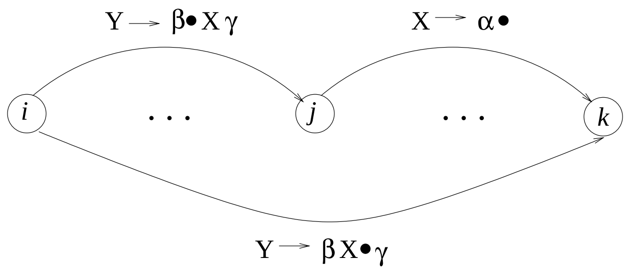


Figure 3.3: *Completion in Earley's algorithm: If we have found an X from j to k and look for an X at j, then we can extend the active edge to k.*

3.2 Parsing LSL Grammars

The proposed parsing algorithm is a natural generalisation of Earley's algorithm. The concept of chart edges in Earley's algorithm must be extended to suit the different needs. I assume a fixed LSL grammar $G = (N, T, P, L, S)$.

3.2.1 Bit Vectors as String Positions

As shown above, Earley's algorithm (and any chart parser which uses well-formed-substring tables) uses chart edges to indicate substrings that have already been or still must be parsed, cf., Gazdar & Mellish (1989), e.g., $\langle i, j, A \rightarrow B \bullet C \rangle$. Two integers suffice because we know that the yield of nonterminal A must be continuous and is hence defined by the start and end position in the input string. This condition does not necessarily hold for LSL grammars.

This difference is captured in the LSL parser by using *bit vectors* instead of these two integers. Using bit vectors for this purpose is a very natural idea also considered, e.g., in Johnson (1985) or Reape (1991). Consider, again, the example sentence from section 1.1.2 and the bit vectors associated with some nonterminals during the parsing process:

		Der	Mann	stirbt	der	zögert
(B1)	D	1	0	0	0	0
(B2)	N	0	1	0	0	0
(B3)	CP	0	0	0	1	1
(B4)	$\bar{\mathrm{N}}$	0	1	0	1	1
(B5)	NP	1	1	0	1	1
(B6)	VP	0	0	1	0	0
(B7)	S	1	1	1	1	1

The bit vectors have the same length as the input string. If the terminal belongs to the yield, the corresponding entry in the bit vector is set to 1, and 0 otherwise. (B6) could occur in an inactive edge indicating that the third terminal in the input string forms the (complete) VP 'stirbt', and similarly for the NP 'Der Mann der zögert'. Note (B5) as it does not consist of a continuous block of 1s indicating that the NP is not continuous.

Now instead of "concatenating" the chart edges, the completion operation will perform a bitwise OR on these bit vectors, provided they do not overlap (this is the case iff the bitwise AND only consists of 0s). Overlapping bit vectors are not allowed because a terminal cannot be derived by two different nonterminals. For instance, combining (B1) and (B3) yields an NP (because NP $\rightarrow$ D $\bar{\mathrm{N}}$), with bit vector (B5). Similarly, we take (B5) and (B6) (the NP and VP) to build an S yielding (B7). Since (B7) contains only 1s, we know that this S yields the complete input. Provided we take S to be the "start symbol" of our grammar, we can accept the input. It is also during completion where we check the precedence and isolation constraints.

In what follows, I will write BV for the set of all bit vectors. Since for one parse, the length of all bit vectors occurring in the edges is fixed I assume that all mentioned bit vectors are of the right length. $\beta \wedge \beta'$ ($\beta \vee \beta'$, respectively) is the result of the bitwise AND (OR, respectively) of the bit vectors β and β'. For instance:

$$\begin{array}{cc} & 0100100 \\ \vee & 0011001 \\ \hline = & 0111101 \end{array}$$

Let $\mathsf{right}(\beta)$ ($\mathsf{left}(\beta)$, respectively) be the position, i.e., an integer value, of the rightmost (leftmost, respectively) bit set to 1 in the bit vector β. If β consists just of 0s (written as $\beta = \vec{0}$) we define $\mathsf{left}(\beta)$ to be 0 and $\mathsf{right}(\beta)$ to be $|\beta| + 1$ where $|\beta|$ is the length of the bit vector. For instance (the crucial bits are underlined):

$\mathsf{left}(0\underline{1}001100) = 2$, $\mathsf{right}(01001\underline{1}00) = 6$

Note that bit vectors are only a different representation for the index sets from chapter 2.

3.2.2 Edges

As in Earley's algorithm, our edges consist of the string position information (bit vectors in our case) and the rules. Earley's algorithm keeps a dot in the RHS of a rule to indicate which constituent is to be parsed next. This single dot suffices because of the total order on the RHSs of the rules. For a rule graph representing a partial order we must probably put the dot before more than one node, i.e., we have a *dot set*. After having completed a constituent, we can "move" the dot over the corresponding node by adding all successors of this node to the dot set. If the dot set is empty, i.e., all constituents were found, the edge is inactive.

Definition 18 (Edge) *An edge is a tuple*

$$\langle \beta, p, D, \pi \rangle$$

where

- $\beta \in BV$
- $p = (v \to (V, E_P, E_I), \theta, I) \in P$
- $D \subseteq V$
- $\pi : V \to I\!N$

D is the dot set. π maps every node that has already been found to its rightmost string position or if it is empty, to the rightmost string position of every nonempty predecessor of it. Every active edge will be initialised with π^n, the function that maps every input to $n+1$. This convention identifies every node v such that $\pi(v) = n+1$ as a node which has not already been found yet. π is needed to check the precedence constraints. I will write $\pi[x \mapsto a]$ for the mapping that is equal to π except that x is mapped to a.

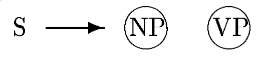
In what follows, I will use a more readable notation for edges:

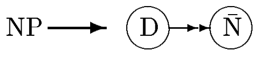
β	$(v \to (V, E_P, E_I), \theta, I)$	D	π

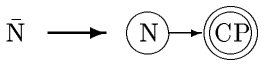
Before presenting the algorithm formally, let me work through it with the simple example from above.

3.2.3 A Worked Example

In this section, I will show how the algorithm will parse the sentence "Der Mann stirbt, der zögert" with the grammar of section 1.1.1. I repeat the grammar here using rule graphs for the RHSs:

(R1) 

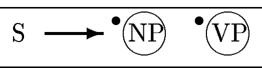
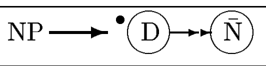
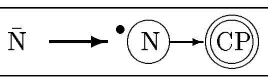
(R2) 

(R3) 

I will leave out θ and I from the formal definition and rather write the non-terminal into the node and use double circles to indicate isolation.

π^n is the constant function that maps every input to $n+1$ as mentioned above. Since the input sentence is of length $n = 5$, we will use π^5 mapping every input to 6. A '$\bullet$' left of a node indicates that it is in the dot set (if there is an incoming i-edge into this node it must come next). If a node was found, the '$\bullet$' gets shifted over it and is then placed to the left of a succeeding node (if there is one) or the right of it if there is no such node. So, for instance $\text{S} \longrightarrow \text{NP}^{\bullet}\ {}^{\bullet}\text{VP}$ indicates that the NP has been found where the VP is still to be.

The parser initialises the chart before the actual parsing process by inserting an edge for every rule with dots before all start nodes with the bit vector $\vec{0}$. In a way, we predict every rule at every string position.

(I1)	00000		π^5
(I2)	00000		π^5
(I3)	00000		π^5

One single pass through the input is performed. Each step consists of two operations:

1. Scanning: skipping one terminal of the input and creating an edge from the lexical entry. This edge has an empty rule graph on the RHS to indicate that it is inactive.

2. Completion: combining an active edge with a matching inactive edge.

The latter is repeated until no more edges can be added to the chart (see also section 3.2.6).

A scan operation advances a pointer on the input string. Completion just produces new edges. These new edges are listed in each step with an identifying number (to the left) and the edges or rules which were used creating them (to the right).

Having initialised the chart, we start the pass over the input with the input pointer at the very beginning:
→ Der Mann stirbt der zögert

1. (a) Scan: Der → Mann stirbt der zögert

(1.1)	10000	$\mathrm{D} \longrightarrow$	π^5	(L1)

   (b) Completion:

(1.2)	10000	$\mathrm{NP} \longrightarrow \textcircled{D} \rightarrow \bullet \textcircled{\bar{N}}$	$\pi^5[\mathrm{D} \mapsto 1]$	(1.1) + (I2)

2. (a) Scan: Der Mann → stirbt der zögert

(2.1)	01000	$\mathrm{N} \longrightarrow$	π^5	(L2)

   (b) Completion:

(2.2)	01000	$\bar{\mathrm{N}} \longrightarrow \textcircled{N} \rightarrow \bullet \textcircled{\textcircled{CP}}$	$\{ \mathrm{N} \mapsto 2, \mathrm{CP} \mapsto 6\}$	(I3)+(2.1)

3. (a) Scan: Der Mann stirbt → der zögert

(3.1)	00100	$\mathrm{VP} \longrightarrow$	π^5	(L3)

   (b) Completion:

(3.2)	00100	$\mathrm{S} \longrightarrow \bullet\textcircled{NP} \quad \textcircled{VP}\bullet$	$\{ \mathrm{NP} \mapsto 6, \mathrm{VP} \mapsto 3\}$	(R1)

4. I assume that the CP is recognised during the next two scanning steps, i.e., (4.1) is inactive.

   (a) Scan: Der Mann stirbt der zögert →

(4.1)	00011	$\mathrm{CP} \longrightarrow$	π^5

(b) Completion:

(4.2)	01011	$\bar{\mathrm{N}} \longrightarrow \textcircled{\small N} \rightarrow \textcircled{\textcircled{\small CP}}^{\bullet}$	$\{ \mathrm{CP} \mapsto 5, \mathrm{N} \mapsto 2\}$	(2.2) + (4.1)
(4.3)	11011	$\mathrm{NP} \longrightarrow \textcircled{\small D} \twoheadrightarrow \textcircled{\small $\bar{\mathrm{N}}$}^{\bullet}$	$\{\mathrm{D} \mapsto 1, \bar{\mathrm{N}} \mapsto 5\}$	(1.2) + (4.2)
(4.4)	11111	$\mathrm{S} \longrightarrow \textcircled{\small NP}^{\bullet}\ \textcircled{\small VP}^{\bullet}$	$\{\mathrm{NP} \mapsto 5, \mathrm{VP} \mapsto 3\}$	(3.2) + (4.3)

(4.2) is possible because the CP is isolated and it occurs at position 4 which is bigger than required ($\pi(N) = 2$).
Similarly, in (4.3.), the leftmost string position of N̄ (2) is exactly one bigger than the rightmost of D ($1 = \pi(D)$), i.e., D immediately precedes N̄.

Since the bit vector of (4.4) only consists of 1s and S is the start symbol of the grammar, we halt and accept.

3.2.4 The LSL Parsing Algorithm

In this section I will present the parsing algorithm in detail. The pseudocode for the main parsing procedure is shown in Fig. 3.4. As shown in the last section, the chart is initialised first. Then, a single pass over the input takes place. At each string position, scanning (also called lexical lookup) is performed, and the corresponding inactive edges are inserted into the chart. Then, completion is applied as long as new edges can be added to the chart.

I use the following functions:

- set(k, i) returns a bit vector of length k with the ith bit being set to 1 and all other bits being set to 0.
- lex_entries(w) returns a set of nonterminals $\{C_1, \dots, C_m\}$ such that $C_i \rightarrow w \in L$ for all $i = 1, \dots, m$.
- add_edge(e) adds edge e to the chart.

Initialisation

The prediction operation in Earley's algorithm can be seen as a kind of top down guidance at a string position k. This is established by inserting an edge from k to k. With bit vectors this is not possible in the same way. An "empty" edge would bear the bit vector $\vec{0}$, which gives absolutely no indication *where* in the input string this edge is meant to start.

What we do instead is perform prediction "off-line", i.e., add an edge for every rule in R bearing $\vec{0}$ before making a pass over the input. The bitwise OR

```
procedure parse($x$:list)
1  $n := |x|$;
2  initialise();
3  for $i := 1$ to $n$ do      (* Add initial edges for $x[i]$ *)
4      $\beta :=$ set$(n, i)$;
5      (* Scanning *)
6      for all $L \in$ lex_entries$(x[i])$ do
7          add_edge$(\langle\beta, (u \to (\emptyset,\emptyset,\emptyset), \{u \mapsto L\}, \emptyset), \emptyset, \pi^n\rangle)$;
8      repeat
9          completion();
10     until no more new edges can be added
11 endfor
12 if there is $\langle\vec{1}, (v \to R, \theta, I), \emptyset, \pi\rangle$ with $\theta(v) = S$ then
13     accept; (* If $S$ spans the whole string $\Rightarrow$ accept *)
14 else reject;
```

Figure 3.4: *Top level loop of the parser*

```
procedure initialise()
1  for each $(v \to R, \theta, I) \in P$ do
2      add_edge$(\langle\vec{0}, (v \to R, \theta, I), $start$(R), \pi^n\rangle)$;
```

Figure 3.5: *The initialisation procedure*

operation of completion will then automatically use these edges correctly[1]. Such a predicted edge will be initialised with the set of start nodes as the dot set.

Note also that with this mechanism, there is an inactive edge for every ε-rule with bit vector $\vec{0}$. This perfectly makes sense, because an empty nonterminal could really appear anywhere in the string.

You can see the pseudocode for initialisation in Fig. 3.5.

Completion

During the completion operation, we may check if parsed substrings are continuous. For this purpose I modify the definition of cont so as to suit bit vectors.

Definition 19 *The predicate* cont(β) *is true iff the bit vector* β *contains exactly*

[1] An actual implementation might, on the other hand, use some kind of index to be able to do prediction at a certain string position. For the complexity analysis, however, it does not make a difference.

```
procedure completion()
1   select inactive edge $\langle\beta', (v' \rightarrow (V', E'_P, E'_I), \theta', I'), \emptyset, \pi'\rangle$;
2   select active edge $\langle\beta, (v \rightarrow (V, E_P, E_I), \theta, I), D, \pi\rangle$;
3   if $\beta \wedge \beta' \neq \vec{0}$ then return;
4   for all $x \in D$ with $\theta(x) = \theta'(v')$ do
5       $B := \mathsf{pred}_{(V, E_P, E_I)}(x)$;
6       $F := \mathsf{succ}_{(V, E_P, E_I)}(x)$;
7       if $\max_{y \in B}(\pi(y)) > n$ then return;
8       $D^* := (D - \{x\}) \cup F$;
9       Check that all of the following conditions hold, else return:
10          1. If $\beta' \neq \vec{0} \neq \beta$:
11              a) If $(y, x) \in E_I$ then $\pi(y) = \mathsf{left}(\beta') - 1$
12              b) If $(y, x) \in E_P$ then $\pi(y) < \mathsf{left}(\beta')$
13          2. If $x \in I$ then $\mathsf{cont}(\beta')$
14          3. If $v \in I$ and $D^* = \emptyset$ then $\mathsf{cont}(\beta \vee \beta')$
15      $\pi^* := \pi[x \mapsto \min(\max_{y \in B}(\pi(y)), \mathsf{right}(\beta'))]$;
16      $\mathsf{add\_edge}(\langle\beta \vee \beta', (v \rightarrow (V, E_P, E_I), \theta, I), D^*, \pi^*\rangle)$;
17  endfor
```

Figure 3.6: *The completion procedure*

one continuous block of 1s or $\beta = \vec{0}$.

If $\mathsf{cont}(\beta)$[2] is true we know that the constituent associated with β is isolated.

The pseudocode of the completion procedure is shown in Fig. 3.6.

First an inactive and an active edge are selected from the chart (line 1 and 2). These edges can only be possible candidates if their bit vectors do not overlap which is ensured by demanding that $\beta \wedge \beta' = \vec{0}$ (line 3), i.e., there is no position where both β and β' have a 1. Then in line 4, every node x of the active edge which has "the dot to the left of it", i.e., which is in the dot set D, is examined. The nonterminal of that node and the nonterminal of the LHS of the inactive edge must be equal ($\theta(x) = \theta'(v')$). Additionally, the π value of all predecessors must be $\leq n$, because otherwise they have not been found yet (line 7). D^* is the set of the union of $D - \{x\}$ and all the successor nodes of x, i.e., we "shift" the dot over x (line 8). The conditions to be checked ensure that the LP constraints are satisfied, namely

1. LP constraints need only to be checked if $\beta' \neq \vec{0} \neq \beta$, because otherwise, at least one of the edges belongs to an empty category which trivially satisfies all such constraints.

[2] Compare with Definition 8.

   (a) If y immediately precedes x, the π value of y (the rightmost position of y) has to be exactly the value of the leftmost string position of x - 1.

   (b) If y precedes x, then the π value of y has to be smaller than the leftmost position of x.

2. An isolated node must have a continuous bit vector associated with it.

3. If the last node of the active edge was found and the LHS of this edge is isolated, combining the two edges must yield a continuous bit vector.

In line 15, the new π^* is equal to π except for x. $\pi^*(x) = \mathsf{right}(\beta')$ if $\beta \neq \vec{0}$. If not, we let it be the value of the rightmost position of one of its predecessors. If x has no predecessors and is empty, $\pi^*(x) = 0$. The new edge that is added bears the bitwise OR of β and β', the same rule as the active edge, the new dot set D^*, and a modified π^* (line 16).

Note that empty nonterminals do not cause any problems. If there is an empty nonterminal, the corresponding rule is just inserted during initialisation with $\vec{0}$. If a rule is recognised as having only empty nonterminals, completion takes care that the corresponding bit vector is also $\vec{0}$.

3.2.5 Correctness

To show the correctness of the algorithm, one must prove that it recognises exactly those strings derived by some fixed LSL grammar $G = (N, T, P, L, S)$ in the sense of Definition 11. I consider a fixed string $w = w_1 \dots w_n$ of length n. In what follows, the term "chart" always refers to the chart after parsing w.

Every bit vector of length n can be interpreted as an index set of at most size n and vice versa. I will thus use bit vectors throughout this section, even in places where, formally, index sets should occur, e.g., expressions like $[(X, \beta)] \Rightarrow_G D$ or $j \in \beta$ are well-defined. In particular, $\emptyset$ is equivalent to $\vec{0}$.

W.l.o.g., we can assume that there is at most one lexical entry $C_i \to w_i \in L$ per terminal w_i. Then let $W(\beta) = [(C_j, \{j\}) | j \in \beta]$. It holds that $W(\beta) \triangleright_G w$.

From section 3.2.4, it should be clear that the completion procedure works correctly. By an inductive argument, one can show that a certain series of completions corresponds to a single valid derivation step and vice versa.

Lemma 4 *Let $e_a = \langle \vec{0}, (v \to R, \theta, I), \mathsf{start}(R), \pi^n \rangle$ be an active edge added during initialisation where r is the number of nodes of R. Let $e_1, \dots, e_r$ be inactive edges where $e_i = \langle \beta_i, (v_i \to R_i, \theta_i, I_i), \emptyset, \pi_i \rangle$.*

Then e_a can successively be completed with $e_1, \dots, e_r$ yielding an inactive edge iff there is a derivation

$$[(\theta(v), \beta_1 \cup \dots \cup \beta_r)] \Rightarrow_G [(\theta_1(v_1), \beta_1), \dots, (\theta_r(v_r), \beta_r)]$$

The next definition defines the inactive and active child, respectively, for some edge e, i.e., the two edges e was created from.

Definition 20 *Let e be an edge in the chart. Let $\mathsf{ic}(e)$ and $\mathsf{ac}(e)$ (inactive, active child) be the inactive and active edge, respectively, e was created from by completion. If e was created during initialisation or scanning, $\mathsf{ic}(e) = \mathsf{ac}(e) = nil$.*

The notion of an inactive child can be extended to the set of inactive children of an edge e. This set contains all inactive edges which were needed directly for building e.

Definition 21 *Let e be an edge in the chart. Define the set of inactive children of e as follows:*

$$\mathsf{ichld}(e) = \begin{cases} \emptyset & \textit{if } \mathsf{ic}(e) = \mathsf{ac}(e) = nil \\ \{\mathsf{ic}(e)\} \cup \mathsf{ichld}(\mathsf{ac}(e)) & \textit{otherwise} \end{cases}$$

The depth of an inactive edge e is similar to the depth of the parse tree that has e as a root.

Definition 22 *Let e_p be an inactive edge in the chart. Define the depth of e_p as follows:*

$$\mathsf{depth}(e_p) = \begin{cases} 0 & \textit{if } \mathsf{ic}(e_p) = \mathsf{ac}(e_p) = nil \\ 1 + \max_{e \in \mathsf{ichld}(e_p)} \mathsf{depth}(e) & \textit{otherwise} \end{cases}$$

The next lemma proves that each inactive chart edge corresponds to a derivation and vice versa.

Lemma 5 *$\langle \beta, (v \to R, \theta, I), \emptyset, \pi \rangle$ is an inactive edge in the chart iff there is a derivation $[(\theta(v), \beta)] \Rightarrow_G^* W(\beta)$.*

Proof:
($\Rightarrow$) Induction over the depth d of the edge.

(IA) $d = 0$: If $e = \langle \beta, (v \to R, \theta, I), \emptyset, \pi \rangle$ was inserted during initialisation, R is empty, thus $\beta = \vec{0}$ and $[(\theta(v), \vec{0})] \Rightarrow_G^* [] = W(\beta)$. If e was inserted during scanning, R has a single node, thus $\beta = \mathsf{set}(n, k)$ for some k and $[(\theta(v), \beta)] = W(\beta)$.

(IH) If $e = \langle \beta, (v \to R, \theta, I), \emptyset, \pi \rangle$ with $\mathsf{depth}(e) \leq d - 1$ is in the chart, then $[(\theta(v), \beta)] \Rightarrow_G^* W(\beta)$.

(IS) Let $p = (v \to R, \theta, I) \in P$ and r be the number of nodes of R. Let $e = \langle \beta, p, \emptyset, \pi \rangle$ with $\mathsf{depth}(e) = d$. Let $e_i = \langle \beta_i, (v_i \to R_i, \theta_i, I_i), \emptyset, \pi_i \rangle$ for all $i = 1, \ldots, r$ such that $\{e_1, \ldots, e_r\} = \mathsf{ichld}(e)$ which means that $\mathsf{depth}(e_i) \leq d - 1$ for all i. Then there must exist an active edge $e_a =$

$\langle\vec{0}, p, \mathsf{start}(R), \pi^n\rangle$, inserted during initialisation, such that e is the result of successive completion of e_a with $e_1, \ldots, e_r$. With Lemma 4 and (IH) it holds that

$$\begin{aligned}[(\theta(v), \beta)] &\Rightarrow_p [(\theta_1(v_1), \beta_1), \ldots, (\theta_r(v_r), \beta_r)] \\ &\Rightarrow_G^* W(\beta_1) \cup \ldots \cup W(\beta_r) = W(\beta)\end{aligned}$$

($\Leftarrow$) Induction over the length of the derivation k.

(IA) $k = 0$: If $[(X, \beta)] = W(\beta)$, it is either the case that $\beta = \vec{0}$ or $\beta = \mathsf{set}(n, j)$ for some j. In the former case, $\langle\vec{0}, (v \to R, \theta, I), \emptyset, \pi^n\rangle$ was inserted during initialisation. Scanning, on the other hand, ensures that the edge $\langle\beta, (v \to R, \theta, I), \emptyset, \pi^n\rangle$ is in the chart in the latter case.

(IH) $[(X, \beta)] \Rightarrow_G^{\leq k-1} W(\beta)$ implies that there is an edge $\langle\beta, (v \to R, \theta, I), \emptyset, \pi^n\rangle$ with $\theta(v) = X$ in the chart.

(IS) Let $p = (v \to R, \theta, I) \in P$ where R has r nodes and $\theta(v) = X$. Suppose there is a derivation

$$\begin{aligned}[(X, \beta)] &\Rightarrow_p [(X_1, \beta_1), \ldots, (X_r, \beta_r)] \\ &\Rightarrow_G^* W(\beta_1) \cup \ldots \cup W(\beta_r) = W(\beta)\end{aligned}$$

such that each (X_i, β_i) derives $W(\beta_i)$ in fewer than k steps. Then with (IH) there are passive edges $e_i = \langle\beta_i, (v_i \to R_i, \theta_i, I_i), \emptyset, \pi_i\rangle$ such that $\theta_i(v_i) = X_i$ for all $i = 1, \ldots, r$. With Lemma 4 there is $e_a = \langle\vec{0}, p, \mathsf{start}(R), \pi^n\rangle$ in the chart, inserted during initialisation, such that $e = \langle\beta, p, \emptyset, \pi'\rangle$ is the result of successively completing e_a with $e_1, \ldots, e_r$.

□

It is now easy to see that the algorithm is correct:

Proposition 23 *Let $G = (N, T, P, L, S)$ be an LSL grammar and let $w \in T^*$. The LSL parsing algorithm recognises w iff $w \in L(G)$.*

Proof:
With Lemma 5 it is the case that there is an edge $\langle\vec{1}, (v \to R, \theta, I), \emptyset, \pi\rangle$ with $\theta(v) = S$ in the chart iff there exists a derivation $[(S, \{1, \ldots, n\})] \Rightarrow_G^* W(\vec{1}) \rhd_G w$.

□

3.2.6 Termination

For the algorithm to terminate, it is important to show that completion does not loop at any string position, i.e., that it does not add an infinite number of edges to the chart. There are two things that must be ensured:

1. Two edges that already have been combined should not be considered for completion again which can be done easily, e.g., by marking those pairs[3].

2. One must check that an edge to be inserted is not already in the chart. This check is often called "subsumption check" in the literature. This possibility is not ruled out by 1. because two different completions might result in the same edge. It can simply be done by checking if any edge already in the chart is equal to the edge to be inserted.

Thus the following holds:

Proposition 24 *The LSL parsing algorithm terminates on every input.*

With the naive strategy mentioned in 2., the number of comparisons is then linear in the size of the chart and comparing two edges, particularly comparing two bit vectors, takes linear time in the size of the input string. I will thus ignore the subsumption check in the following since it does not "blow up" the runtime exponentially.

3.3 Complexity Analysis

As for the presentation of the algorithm itself, I will compare the complexity analysis of the LSL parsing algorithm with the one of Earley's algorithm. I only consider parsing with a fixed grammar, i.e., the input is only the string to be parsed rather than this string plus the grammar. Since the grammar is fixed, we assume the number of nonterminals and rules to be constant.

3.3.1 Complexity of Earley's Algorithm

Earley's algorithm uses three basic operations which are performed at each string position during a parse:

1. Scanning

2. Prediction

3. Completion

We will now analyse how long each of them takes in a single step, i.e., at some string position i.

There can only be $O(i)$ edges ending at string position i, because there are only i possible starting points[4], and the number of rules and nonterminals is constant.

[3] In the implementation a more sophisticated strategy was used using iterators (see section 4.3).

[4] This will *not* be the case for the LSL parsing algorithm.

1. At position i, only one symbol is scanned, i.e., scanning takes time $O(1)$.
2. Prediction considers each of the $O(i)$ edges and since the number of rules is constant, only $O(i)$ edges are inserted thus, prediction takes time $O(i)$.
3. If there is an inactive edge e_p ending at i, we may be able to perform completion with an active edge e_a ending where e_p starts (say position $i-j$). At this position $i-j$ there are $O(i-j)$ (active) edges. So we have $O(i(i-j)) = O(i^2)$ different possibilities for completion.

Summing over all the n string positions yields:

$$\sum_{i=1}^{n}(\underbrace{O(1)}_{\text{Scan.}} + \underbrace{O(i)}_{\text{Pred.}} + \underbrace{O(i^2)}_{\text{Compl.}}) = \sum_{i=1}^{n} O(i^2) \leq \sum_{i=1}^{n} O(n^2) = nO(n^2) = O(n^3)$$

Note that it is crucial that when adding an inactive edge from j to k, we know that *only active edges ending at j* are possible candidates for completion. This will no longer be the case for the LSL parsing algorithm since the notions of start and end point as needed for Earley's algorithm no longer exist.

3.3.2 Complexity Analysis of the LSL Parsing Algorithm

The complexity analysis consists of three parts. First, a relationship between space and time complexity is established that allows us to focus on space in the following discussion. Second, I will present a measure for space complexity which immediately gives the worst-case complexity of the algorithm (exponential). Third, a sufficient condition for parsing in polynomial time is presented.

Time and Space Bounds

In this section, I will describe a relationship between the time and space bounds of the algorithm. In particular, the space bound can be used as a measure for the time bound.

The space bound can be measured in terms of the number of edges in the chart. Note that I ignore the storage needed for a bit vector which is $O(n)$, i.e., the term "space" means "number of edges in the chart".

Suppose our algorithm needs $O(f(n))$ space with $f(n) = \Omega(n)$,i.e., for an input string of length n, the chart contains $O(f(n))$ edges after the parse. Initialisation inserts an edge with $\vec{0}$ for every rule and an empty edge for each nonterminal in ε-rule. Since the number of rules is constant, initialisation inserts $O(1)$ edges.

Let us now consider the time needed for each of the two basic operations at some string position i (analogously to the analysis above).

1. Scanning: Since scanning one terminal and adding a fixed number of rules is independent of i, scanning takes time $O(1)$.

2. Completion: As mentioned above, the notions of start and endpoints of edges no longer exist: they do not make sense for bit vectors. Instead, the algorithm performs a simple completion strategy by trying to combine each possible pair of edges. Since we have $O(f(i))$ edges in the chart at string position i and we have $O(f^2(i))$ pairs, completion occurs $O(f^2(i))$ times. The result of completion is a new edge in the chart.

As above, the overall complexity is given by summing over all string positions:

$$\underbrace{O(1)}_{\text{Init.}} + \sum_{i=1}^{n}(\underbrace{O(1)}_{\text{Scan.}} + \underbrace{O(f^2(i))}_{\text{Compl.}}) = \sum_{i=1}^{n} O(f^2(i)) \leq \sum_{i=1}^{n} O(f^2(n)) = O(nf^2(n))$$

Note that, if $f(n)$ is a polynomial, e.g., n^k, the overall runtime is also polynomial, namely $O(n^{2k+1})$. In what follows, I will identify a condition that bounds this f to a polynomial and thus enables parsing in polynomial time.

Again, I ignore the time needed to perform a bitwise OR of two bit vectors (this is possible in time $O(n)$). Taking this time into account, however, would yield $O(n^2 f^2(n))$ which is still polynomial if f is polynomial.

A Measure for Space Complexity

As was shown in Earley (1968), Earley's algorithm requires space $O(n^2)$. The only content of the edges we must consider are the start and end points. The rest is of constant size because the rule sizes and the number of nonterminals are constant. So for input length n, we can have an edge between each pair of nodes. And there are $(n+1)^2 = O(n^2)$ such pairs.

Similarly, for our algorithm, we only need to measure the possible numbers of "string positions" which in our case are the bit vectors. As above, the rule sizes etc. are constant and thus will not be considered. Since there are 2^n bit vectors of length n, the chart might contain $2^{O(n)}$ edges in the worst case.

Consider the following example to see that this number can actually be reached: Suppose we have an LSL grammar just consisting of the rule

$$A \rightarrow A\ A; \varepsilon$$

(i.e. no LP constraint on the RHS) and a lexical entry

$$A \rightarrow a$$

where A is the start symbol. Then, the number of edges for the input string a^n is actually 2^n (see Fig. 3.7).

Substituting this result in the above sum yields:

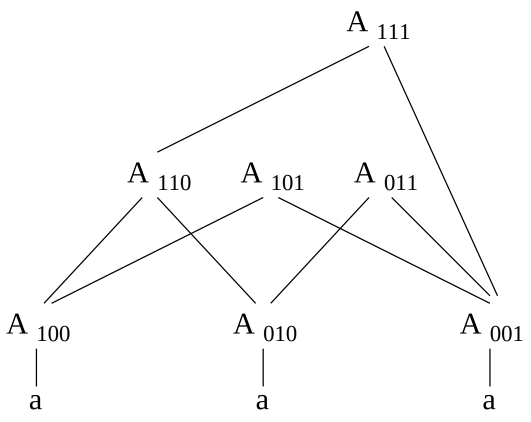


Figure 3.7: *The chart for the LSL rule $A \to A\ A; \varepsilon$, lexical entry $A \to a$, and input string aaa. The indices at the nonterminal are the corresponding bit vectors. Note that there are other possibilities than the one shown to create A_{111}.*

Proposition 25 (Worst case complexity) *The LSL parsing algorithm has a worst case time complexity of $2^{O(n)}$.*

One would suspect that only "pathological" grammars like the example in Fig. 3.7 do actually need exponential time.

In the next section, I will present a sufficient condition for LSL grammars that ensures parseability in polynomial time. A special case of this condition is directly expressible in LSL.

3.3.3 A Sufficient Condition for Polynomial Time

As seen above, the number of possible edges in the chart correlates only with the possible number of bit vectors for length n. If we want to reduce complexity to polynomial time, we must impose a restriction on all bit vectors that might occur during a parse of an LSL grammar.

Definition 23 (Block) *A* block *in a bit vector β is a string of continuous 1s.*

For instance, in

$$00'\underline{1111}'000'\underline{111}'0$$

there are two blocks.

Note that one block is exactly determined by two positions in the bit vector (in the example indicated with a " ′ "). Thus, if we consider bit vectors of length n with k blocks, we can choose $2k$ positions out of the $n+1$ positions that there are, and in this fashion completely determine the form of the bit vector.

So, for the example above $n = 13$ and $k = 2$. We chose $2k = 4$ positions out of the $n + 1 = 14$ possible ones, namely 2 and 6 for the first block and 9 and 12 for the second.

The next question is: How many possible bit vectors of length n with k blocks can there possibly be, or in other words: How many possibilities are there to pick $2k$ positions out of $n+1$? The answer is

$$\binom{n+1}{2k}^{\dagger} = O((n+1)^{2k}) = O(n^{2k})$$

And this is polynomial in n.

So suppose we have input length n and know that all bit vectors occurring in edges in the chart have (at most) k blocks, i.e., k is a constant independent of n. The maximal number of possible bit vectors in the edges is then

$$\sum_{i=1}^{k} \binom{n+1}{2i} = \sum_{i=1}^{k} O(n^{2i}) = O(n^{2k})$$

Thus, the number of possible edges as well is $O(n^{2k})$. Space complexity is then polynomial (since k is constant) and thus the run time is

$$O(n(n^{2k})^2) = O(n^{4k+1})$$

This k depends on the LSL grammar used. For CFGs, we have $k = 1$ and thus the algorithm runs in $O(n^5)$. The difference to the $O(n^3)$ of Earley's algorithm stems from the fact that continuity of edges enables better indexing of the chart.

In general, the maximal number of blocks in a parse may not be constant, i.e. independent of n, but rather be a function $k(n)$ growing with n for some grammars. For this case, the space is bounded by

$$\sum_{i=1}^{k(n)} O(\binom{n+1}{2i}) = O(2^n)$$

for $k(n) = \frac{n}{2}$ for example. The aim is now to find a condition for LSL grammars such that k can be bounded by a constant.

I will call the maximal number of blocks occurring for a fixed LSL grammar G the *block number* of G.

We can summarise the last section as

Proposition 26 *If the block number of an LSL grammar $G = (N, T, P, L, S)$ is bounded by a constant, then for all $w \in T^*$, the LSL parsing algorithm takes polynomial time parsing w w.r.t. G (G is* parseable in polynomial time*).*

In Holan et al. (1995), it is mentioned that, if the span of every nonterminal is *one-gap*, i.e., has at most one discontinuity, then space and time complexity of

[†] Rather than n^{2k}. Permutation without replacement is justified because blocks do not "overlap".

their parser (an extension of the CYK algorithm to parse dependency grammars, see section 1.2.2) is polynomial. The condition of Proposition 26 is more general and is also stated in Holan et al. (1998). It can be seen as a kind of m-gap restriction.

In the next section, I will go further and describe a condition for LSL grammars to be parseable in polynomial time which is less restrictive and more natural than posing a one-gap or m-gap restriction on *every* nonterminal.

The Condition

As shown above, we must ensure that the block number k for an LSL grammar G is a constant, i.e., does not grow with n. Therefore, we will first identify which conditions are responsible for letting k be a function of n.

Closely related to this is the question of how grammars can generate arbitrary long sentences, i.e., the language generated by the grammar is infinite. This is because if the language is finite, k is trivially constant. Only infinite languages may have the property that k grows with n.

So what makes grammars generate arbitrarily long sentences? The answer is recursion. If we did not have recursion, the language would be finite. One must somehow restrict the properties of recursive nonterminals in such a way that k can only be constant.

In this section I will prove that, if one can ensure that the yield of every recursive nonterminal has at most a constant number of blocks, we know that the block number of the grammar is a constant and thus the grammar is parseable in polynomial time. In other words, if every recursive nonterminal has a fixed block number, then we cannot add arbitrarily many blocks to a mother nonterminal.

The rest of this section consists of the formal proof of this proposition. I assume a fixed grammar $G = (N, T, P, L, S)$.

Definition 24 (Recursive nonterminal) *A nonterminal $X \in N$ is called* recursive *iff there are pairwise distinct rules*

$$\begin{array}{rcl} (v_1 \to (V_1, E_{P1}, E_{I1}), \theta_1, I_1) & \in & P \\ (v_2 \to (V_2, E_{P2}, E_{I2}), \theta_2, I_2) & \in & P \\ & \vdots & \\ (v_n \to (V_n, E_{Pn}, E_{In}), \theta_n, I_n) & \in & P \end{array}$$

such that

- *For all $i = 1, \dots, n-1$ there is a $v \in V_i$ with $\theta_i(v) = \theta_{i+1}(v_{i+1})$*
- *There is a $v \in V_n$ such that $\theta_1(v_1) = \theta_n(v) = X$.*

Let $N_R \subseteq N$ be the set of all recursive nonterminals. N_R can be constructed as in the context-free case (cf. Hopcroft & Ullman 1979) by building a graph (N, E) where there is an edge from X to Y if there is a rule $X \rightarrow \ldots Y \ldots ; \varphi$. A recursive nonterminal then is every nonterminal which lies on a cycle in this graph.

I will now define a special notion of derivation which is a restricted form of derivations we used thus far, in that it does not allow for expansion of recursive nonterminals.

Definition 25 *The nonrecursive derivation relation*

$$\overset{R}{\Rightarrow}_G \subseteq \mathcal{B}(N \times \mathsf{Fin}(I\!\!N)) \times \mathcal{B}(N \times \mathsf{Fin}(I\!\!N))$$

w.r.t. to G is defined such that

$$D_1 = D \cup [(X_0, M_0)] \overset{R}{\Rightarrow}_G D \cup [(X_1, M_1), \ldots, (X_n, M_n)] = D_2$$

if $D_1 \Rightarrow_G D_2$ *and* $X_0 \notin N_R$

It obviously holds that $\overset{R}{\Rightarrow}_G$ is a special case of $\Rightarrow_G$, formally $\overset{R}{\Rightarrow}_G \subseteq \Rightarrow_G$.

One can now define a normal form for an LSL derivation which consists of three phases as opposed to the two we had before. First, all non recursive nonterminals are expanded until only recursive nonterminals or nonterminals which are eventually expanded by a lexical entry are in the sentential form (we will use $\overset{R}{\Rightarrow}$ for that). The second phase is the usual derivation w.r.t. $\Rightarrow$. The third phase (as before) applies lexical entries and thus generates a string.

Definition 26 *A derivation* $D_0 \Rightarrow_G D_1 \Rightarrow_G \ldots \Rightarrow_G D_n \rhd_G w$ *is in* normal form *iff*

1. *There is k such that* $D_i \overset{R}{\Rightarrow}_G D_{i+1}$ *for all* $i = 0, 1, \ldots, k-1$
2. *For all* $(X, M) \in D_k$ *it holds that* $X \in N_R$ *or* $(X, M) \in D_n$

A derivation is in normal form, if all nonrecursive nonterminals are expanded first until only recursive nonterminals or those nonterminals, which will eventually be used to generate terminals, appear in the sentential form. We can w.l.o.g. assume that all derivations are in normal form $D_0 \overset{R}{\Rightarrow}{}^*_G D_k \Rightarrow^*_G D_n \rhd_G w$ since k could also be 0.

Lemma 6 *Let* $M \in \mathsf{Fin}(I\!\!N)$. *If* $[(A, M)] \overset{R}{\Rightarrow}{}^*_G D \Rightarrow^*_G D' \rhd_G w$ *is a derivation in normal form, then* $|D|$ *is bounded by a constant.*

Proof:
Since $\overset{R}{\Rightarrow}$ does not expand recursive nonterminals, there is only a finite number of possibilities to apply productions.

$\square$

Lemma 7 *If there is $m \in \mathbb{N}$ such that every recursive nonterminal has block number $\leq m$ then the block number of every nonterminal is bounded by a constant.*

Proof:
The proposition holds trivially for all nonterminals in N_R. Let $A \in N$. Let $\mathsf{bn} : \mathsf{Fin}(\mathbb{N}) \to \mathbb{N}$ be the function such that $\mathsf{bn}(M)$ is the block number of M. Define a function $f : \mathcal{B}(\mathsf{Fin}(\mathbb{N}) \times \mathbb{N}) \to \mathbb{N}$ as the block number of a sentential form as follows:

$$\begin{aligned} f(\emptyset) &= 0 \\ f([(X, M)] \cup D) &= \mathsf{bn}(M) + f(D) \end{aligned}$$

It holds that if $[(A, M)] \Rightarrow_G^* D$ then $\mathsf{bn}(M) \leq f(D)$ because splitting an index set into many index sets during a derivation step may only result in more blocks.

Let $[(A, M)] \overset{R}{\Rightarrow}{}_G^* D \Rightarrow_G^* D' \triangleright_G w$ be a derivation in normal form. It holds that[5]

$$\mathsf{bn}(M) \leq f(D) \leq |D| m$$

Since with Lemma 6, $|D|$ is bounded by some constant c, it is the case that $\mathsf{bn}(M) \leq cm$ and thus the block number of A is bounded by a constant.

$\square$

Proposition 27 *If the block numbers of all recursive nonterminals of G are bounded by a constant then G is parseable in polynomial time.*

Proof:
With Lemma 7, it holds that the block number of every nonterminal is bounded by a constant c. The bit vectors of all inactive edges used during a parse then also have at most block number c. Since each active edges only has a constant number of nonterminals on the RHS (call this constant r), its bit vector has at most rc blocks. With Proposition 26, G is parseable in polynomial time.

$\square$

There are two possibilities how Proposition 27 can be exploited.

1. LSL has a feature which can be used to set the block number of all recursive nonterminals to 1: Isolation. Since this is directly expressible in LSL, it can easily be checked. Note that it is not necessary that *every* occurrence of a recursive nonterminal X is isolated but either every occurrence of X on a LHS, or every occurrence on a RHS. If this condition is satisfied I will call X to be *isolated in the grammar G*.

[5] This is a rather rough estimate because some nonterminals that might be expanded with lexical entries have block number 1.

LP constraints on $X \rightarrow X_1 \ldots X_n$	Grammar type	Complexity
$\langle X_i \rangle, \forall i$	UCFG	$\mathcal{P}$
$X_i \ll X_{i+1}, \forall i$	CFG	$\mathcal{P}$ ($LOGCFL$-complete)
$\langle X_i \rangle$, $X_i \in N_R$	Restricted LSLG	$\mathcal{P}$
any	General LSLG	$\mathcal{NP}$-complete

Table 3.1: *Special cases of LSL grammars and their complexity for the fixed membership problem. The first column shows the LP constraints attached to every rule $X \rightarrow X_1 \ldots X_n$ in the grammar (N_R is the set of recursive nonterminals), the second the (effective) grammar type, and the third the complexity.*

Corollary 10 *An LSL grammar G is parseable in polynomial time if every recursive nonterminal in G is isolated in G.*

Corollary 10 is a specialisation of Proposition 27 in that the required condition can be directly expressed in LSL.

2. One could, however, pursue another (in a way the opposite) approach and extend LSL's notion of isolation to *m-isolation*. If a constituent is m-isolated that should mean that it can have at most m blocks (or in other words at most $m-1$ discontinuities). Such an extended LSL grammar where all recursive nonterminals are m-isolated satisfies the condition of Proposition 27 and, thus, this grammar is parseable in polynomial time.

From a linguistic point of view, the latter approach is more promising because the requirement that every recursive nonterminal has no discontinuities might be too strong. For a discussion see section 5.2. (Holan et al. (1998) mention a possible extension to their dependency grammar framework by adding rules of the form $A \rightarrow^i_X BC$ meaning that a tree dominated by A built using this rule may have at most i discontinuities (see also section 1.2.2). It is not mentioned, however, which (subset of) nonterminals should appear on the LHSs of such rules in order to decrease complexity.)

Notice that, during the whole complexity analysis, the precedence relation never came into play. The important factor which made parsing polynomial[6] is isolation. What is it then that makes parsing CFGs so efficient? Isolation, i.e., contiguity, makes it polynomial. The total order on the RHS, additionally, enables good indexing and thus Earley's result of $O(n^3)$.

Fig. 3.1 summarises the complexity results for some variations and special cases of LSL grammars.

[6]Note that this is the case for the *fixed* recognition problem.

3.3.4 Computing the Block Number

The condition of Proposition 27 requires that the block number of every recursive nonterminal is bounded by a constant. It is pretty difficult to take this into consideration when writing a grammar. What one wants is an algorithm which takes a grammar and checks if such a condition is satisfied. Furthermore, it would be nice if one could say more about the maximal block number than just "bounded by some constant".

I will now present a simple algorithm to determine the block number of a given grammar. The block number of a grammar is the maximum of the block numbers of all nonterminals in the grammar. We consider all nonterminals (even the ones that are not "reachable" from the start symbol S) because the parser may find substrings that can be derived by such a nonterminal. The block number of a nonterminal X can be computed as follows: Consider every rule p with X on the LHS and some nonterminals $X_1, \ldots, X_m$ on the RHS (in some order). If X is isolated in p, the block number is simply 1. Otherwise, we sum up all block numbers of the X_i. If X_i is isolated, the block number is again 1 and no recursion is needed. If not we recurse. At the end we subtract the number of i-edges from this sum because if two bit vectors with block numbers k_1 and k_2, respectively, are put together via immediate precedence, the block number of the resulting new bit vector is $k_1 + k_2 - 1$. Furthermore, we must remember all nonterminals we already tried to compute the block numbers of to avoid loops. You can see the pseudocode of this algorithm in Fig. 3.8. The nonterminals already inspected are stored in the argument visited which is a set.

Note that block_num() only returns the maximal number of blocks for inactive edges. If we want to compute the block number of a grammar, we also must take possible active edges into account (see Fig. 3.9).

Since LSL currently does not have a notion of m-isolation, this algorithm can only check the condition of Corollary 10, i.e., that every recursive nonterminal is isolated. If LSL is extended towards such a notion, the lines 10 and 11 in Fig. 3.8 may be modified in a straightforward way.

Note again that only a *sufficient* condition from above is checked, i.e., if the algorithm returns a number k, the block number is bounded by k. If, however, the algorithm terminates abnormally, nothing can be said about this grammar being parseable in polynomial time.

It is easy to see that the block number computed by the function block_number in Fig. 3.9 is large even when all recursive nonterminals are isolated, i.e., the time complexity is a polynomial with a high degree. Nonetheless, a line is drawn between exponential and polynomial time parsing.

```
function block_num($X$:$N$,visited:set($N$)) : integer
1  if $X \in$ visited then output("Possibly infinite block number !"); halt;
2  if there is $X \rightarrow a \in L$ then max := 1; (* Lexical nonterminal *)
3  else max := 0;
4  (* For all rules expanding $X$ *)
5  for each $(v \rightarrow (\{v_1, \ldots, v_m\}, E_P, E_I), \theta, I) \in P$ do
6     if $v \in I$ then res := 1; (* Isolation $\Rightarrow$ Block number = 1 *)
7     else
8         res := 0;
9         for $i$= 1 to $m$ do
10            if $v_i \in I$ then (* node on RHS is isolated $\Rightarrow$ block number = 1 *)
11              b := 1;
12            else (* Compute block number for all non-isolated nonterminals *)
13              b := block_num($\theta(v_i)$,$\{X\} \cup$ visited);
14            res := res + b;
15        endfor
16        res := res - $|E_I|$; (* Subtract blocks that "melt together" *)
17    endif
18    if res > max then max := res; (* Find maximum *)
19 endfor
20 return(max);
```

Figure 3.8: *Computing the block number for a nonterminal X.*

```
function block_number($(N, T, P, L, S)$: LSL grammar) : integer
1  $b_I$ := $\max_{X \in N}\{$block_num$(X, \emptyset)\}$; (* Inactive edges *)
2  (* Now for all active edges *)
3  $m$ := 0;
4  for each $(v \rightarrow (V, E_P, E_I), \theta, I) \in P$ do
5     $b_A$ := $\max_{V' \subset V}\{\sum_{x \in V'}$ block_num$(\theta(x))\}$;
6     if $b_A > m$ then $m$ := $b$;
7  endfor
8  return max($b_I, b_A$);
```

Figure 3.9: *Compute the block number of a grammar.* block_num*() only returns the block number for inactive edges. An active edge for rule $(v \rightarrow (V, E_P, E_I), \theta, I)$ can have a block number which is at most the sum of all nonterminals of all* proper *subsets of V.*

3.4 Generalisation of LSL Grammars

In this section, I will briefly show how LSL grammars can be generalised such that arbitrary LP constraints (not necessarily only $<$, $\ll$, and $\langle\rangle$) can be used and how still the condition of Proposition 27 can be applied to yield polynomial time parseability.

Definition 27 (LPC-Grammar) *An* LPC-Grammar *is a tuple* $G = (N, T, P, L, S)$, *where* N, T, L, S *are defined as for LSL grammars and* P *is a set of rules of the form* $(X_0 \to X_1 \dots X_r, \varphi)$ *where* $X_i \in N$ *and* φ *is a function from* $(\mathsf{Fin}(I\!N))^r$ *to* $\{true, false\}$.

A rule in an LPC grammar consists of a context free rule plus a function φ which checks if some LP constraints between the string positions of the nonterminals on the right hand side hold and returns *true* if so, and *false* otherwise. φ is essentially a predicate, but the functional notation should indicate its procedural nature. It is the analogue to the `combines` predicate of Johnson (1985). Derivations are a generalisation of LSL derivations.

Definition 28 (LPC Derivation) *Let* $G = (N, T, P, L, S)$ *be an LPC grammar. Then the derivation relation* $\Rightarrow_G$ *is defined such that*

$$D \cup [(X_0, M_0)] \Rightarrow_G D \cup [(X_1, M_1), \dots, (X_r, M_r)]$$

if

1. $(X_0 \to X_1 \dots X_r, \varphi) \in P$
2. $M_0 = \bigcup_{i=1}^r M_i$ *and all* M_i *are pairwise disjoint*
3. $\varphi(M_1, \dots, M_r) = true$

The terminating derivation relation $\triangleright_G$ is defined exactly as for LSL grammars, as is the language of an LPC grammar. For the description of the parsing algorithm, I will use bit vectors instead of index sets and assume that φ is defined for those.

In the generalisation of the LSL chart parser, we only use inactive edges. An edge then is simply a pair $\langle \beta, X \rangle$ where β is a bit vector and $X \in N$. The parser can also be generalised as follows: Let $w = w_1 \dots w_n$ be the string to be parsed. After scanning which works as for the LSL parser, we choose a rule $(X \to X_1 \dots X_r, \varphi) \in P$ and look at all r-tuples of edges. If the nonterminals of those edges match $X_1, \dots, X_r$, respectively, *and* their bit vectors satisfy φ, i.e., φ maps them to *true*, we can add an edge with the nonterminal X and the bitwise OR of all the bit vectors of the X_i. For simplicity, I assume that φ also checks that the bit vectors do not overlap. The criterion for acceptance is also the same as for the LSL parser. The pseudocode can be seen in Fig. 3.10.

```
**procedure** lpc_parse($x : T^*$)
1 **for** each $(X \to \varepsilon, \varphi) \in P$ **do** add_edge($\langle \vec{0}, X \rangle$);
2 **for** $i := 1$ **to** $|x|$ **do**
3     scan($x_i$);
4     **while** there are edges $\langle \beta_1, X_1 \rangle, \ldots, \langle \beta_m, X_m \rangle$ in the chart
5         and there is rule $(X_0 \to X_1 \ldots X_m, \varphi) \in P$
6         such that $\varphi(\beta_1, \ldots, \beta_m) = true$ **do** add_edge($\langle \beta_1 \vee \ldots \vee \beta_m, X_0 \rangle$);
7 **endfor**
8 **if** there is an edge $\langle \vec{1}, S \rangle$ in the chart **then** accept;
9     **else** reject;
```

Figure 3.10: *Pseudocode for parsing LPC grammars*

Now let $c(n) = \Omega(n)$ be the number of edges in the chart, and $f_\varphi(n)$ be the time complexity of the "worst" function attached to a rule. Let r be the maximal number of nonterminals occurring on the RHS of a rule. The complexity of this algorithm for the fixed parsing problem, using a similar argument as in section 3.3, is then

$$\begin{aligned} & \underbrace{O(1)}_{\text{Initialisation (line 1)}} + \sum_{i=1}^{n} (\underbrace{O(1)}_{\text{Scan. (line 3)}} + \underbrace{O((c(i))^r f_\varphi(n))}_{\text{Compl. (lines 4-6)}}) \\ = & \sum_{i=1} O((c(i))^r f_\varphi(n)) \leq \sum_{i=1} O((c(n))^r f_\varphi(n)) = O(n(c(n))^r f_\varphi(n)) \end{aligned}$$

Note that since there are no active edges, we must search through the chart finding all r tuples rather than all pairs, which makes complexity much worse.

Again, if $c(n) = n^k$ for some $k \geq 1$, and, additionally, φ has polynomial complexity, i.e., $f_\varphi(n) = n^q$ for some $q \geq 0$, we have

$$O(n(c(n))^r f_\varphi(n)) = O(n(n^k)^r n^q) = O(n^{1+kr+q})$$

Since k, r, and q are constant, this is a polynomial in n. If there is a notion of active edges, the algorithm must only search through all possible pairs of edges in the chart, and thus $r = 2$.

To ensure that $c(n)$ is a polynomial, the same sufficient condition as above holds: all recursive nonterminals should have at most a constant number m of blocks, they should be m-isolated. Every φ attached to a rule must return $false$ if this property is not satisfied in the corresponding rule.

LSL grammars are a special instance of LPC grammars. One can easily define functions implementing isolation, precedence, or immediate precedence constraints between any of their arguments.

The formal language results of LSL do not hold for LPC grammars in general. It is, for example, not even required that a function attached to a rule is decidable.

If we, however, assume that all functions attached to rules have polynomial complexity, the exponent of the polynomial parsing complexity for LPC grammars satisfying the sufficient condition from Proposition 27 is much larger due to the lack of the notion of active edges (see above), but it is still constant, thus parsing in polynomial time is possible.

Chapter 4

Implementation

This chapter describes an implementation of the LSL parsing algorithm (this implementation is called LieSL). LieSL was implemented in C++ using the LEDA library[1] (cf. Mehlhorn, Näher, Seel & Uhrig 1999). LEDA (Library of Efficient Data Types and Algorithms) provides standard data types (such as lists) as well as more complicated data structures (such as graphs). LieSL can be used on any UNIX[2] system.

As described in section 2.9, LieSL implements a unification-based version of LSL grammars. I will not describe the complete implementation in detail but rather emphasise the design decisions of problems which were underspecified in chapter 3 since they do not make any difference from a complexity theoretic point of view but play an essential role in a practical system. These issues are: organisation of the chart (indexing) (section 4.3), control of the core algorithm, i.e., which edges are selected for completion and when (section 4.4), and implementation of complex categories (section 4.2).

4.1 General Remarks

4.1.1 Usage

It is possible to use LieSL as a standalone application (using atomic or first order term categories), or in connection with the ConTroll/XTroll system (cf. Götz, Meurers & Gerdemann 1997) using its typed feature structures. In either case, LieSL provides a terminal-based text mode or an X-interface.

Format of an LSL Grammar

LieSL grammars must satisfy the BNF in Fig. 4.1.

[1]LEDA is available free of charge for research purposes.
[2]UNIX is a registered trademark of AT&T.

```
<Grammar>            ::=    ε
                        |   <NERules>.
<NERules>            ::=    <Rule>
                        |   <Rule> <NERules>.
<Rule>               ::=    <LSLRule>
                        |   <LexEntry>.
<LexEntry>           ::=    <Constant> '--->' <YVDef> '.'.
<LSLRule>            ::=    <YVDef> '==>' <YVars> ';' <LPConstraints> '.'.
<YVars>              ::=    ε
                        |   <NEYVars>.
<NEYVars>            ::=    <YVDef>
                        |   <YVDef> ',' <NEYVars>.
<YVDef>              ::=    <YV>'(' <Desc> ')'.
<LPConstraints>      ::=    ε
                        |   <NELPConstraints>.
<NELPConstraints>    ::=    <LPCons>
                        |   <LPCons> ',' <NELPConstraints>.
<LPCons>             ::=    <YV> '<' <YV>
                        |   <YV> '<<' <YV>
                        |   '[' <YV> ']'
                        |   '[' <YV> ']_' <Number>.
```

Figure 4.1: *BNF for LieSL grammars.*

`<YV>` is a yieldvariable which, by convention, looks like a Prolog constant. `<Number>` is an arbitrary natural number. `<Desc>` is an arbitrary string (a description of a category) which can also have nested parentheses. These descriptions depend on which category type was chosen. Comments are either possible in C-style, i.e., between "`/*`" and "`*/`", or as single line comments starting with "`%`".

Note that `[]` are the parentheses used for isolation and that LieSL also has a notion of m-isolation written as `[y]_m`. See Fig. 4.2 for an example (cf. section 1.2.1).

4.1.2 LieSL-ConTroll Interface

LieSL uses the C-Prolog interface of SICStus Prolog (cf. SICS 1999) for interfacing with ConTroll which is implemented in SICStus. As in the standalone application, all control stays within the C++ part of LieSL. The only calls made to SICStus are itself hook predicates to ConTroll routines, mainly unification and subsumption. ConTroll's constraint resolution interpreter is *not* used. It is therefore straightforward to use a different underlying system than ConTroll by simply changing the mentioned hook predicates.

```
% rule for sentences
s( s(Sem) ) ==> np( np(X,Gen,Case) ),
                vp( vp(lambda(X,Sem)) ) ;
np << vp, [vp].

% rule for the VP (any order of the two objects allowed)
vp( vp(Sem) ) ==> v( v(lambda(X,lambda(Y,Sem))) ),
                  npdo( np(X,GenDO,akk) ),
                  npio( np(Y,GenIO,dat) ) ;
npdo < v, npio < v,
[npdo], [npio].

% rule for NPs with determiner
np( np(Sem,Gen,Case) ) ==> d( det(Gen,Case) ),
                           n( n(Sem,Gen,Case) ) ;
d << n.

% Lexicon
der    ---> l( det(masc,nom) ).
der    ---> l( det(fem,dat) ).
die    ---> l( det(fem,akk) ).
Fabian ---> l( n(fabi,masc,_) ).
Lisa   ---> l( n(lisa,fem,_) ).
PrincipiaMathematica ---> l( n(principia,fem,_) ).
gibt   ---> l( v(lambda(X,lambda(Y,lambda(Z,gives(Z,Y,X))))) ).
```

Figure 4.2: *Example LieSL grammar with first order terms as categories. This grammar generates German complement clauses. Note that the second rule licenses both orderings of the objects of "gibt". Parsing, e.g., "der Fabian die PrincipiaMathematica der Lisa gibt" yields* `s(gives(fabi,lisa,principia))`.

4.2 Implementation of Complex Categories

The design of LieSL enables easy addition of a new concept of category. In the current system, three types of categories are supported: atomic categories (nonterminals, i.e., just strings), first order terms, and ConTrollś typed feature structures (see section 4.1.2). In general, all kinds of unification-based categories can easily be integrated into the system, i.e., categories which have a notion of subsumption and unification. For the case of atomic categories, a subsumes b iff $a = b$ and a unifies with b yielding c iff $a = b = c$. For first order terms, the classical notions of term subsumption and term unification are used whereas for typed feature structures, more complicated notions are necessary (cf. Carpenter 1992).

The Idea

Unification-based categories may have a notion of shared variables (also called structure sharing). For instance, the terms $f(a, X)$ and $g(X)$ share the variable X (if they both occur within the scope of the same rule). Once this variable is instantiated, the Xs in both terms are bound to that instantiation. Variables are, on the other hand, only shared within the scope of one rule, i.e., if X appears in two different rules, it may not be shared. Inspired by a WAM model for first order terms (cf. Ait-Kaci 1991, Warren 1983), a category in LieSL is considered as being some kind of pointer into a data structure which, in turn, realises the sharing of variables. So for example, the terms $f(a, X)$ and $g(X)$ are represented by "pointers" (in this case the integers 0 and 3, respectively) into the following WAM style heap

0	$f/2$	
1	$a/0$	
2	REF	2
3	$g/1$	
4	REF	2

The identity of the second and the first argument, respectively, of the two terms (i.e. the variable X) is established by letting the second cell after cell 0 (thus second argument of f) and the first cell after cell 3 (first argument of g) point to the same heap cell, namely 2.

Every rule, and thus every edge, has exactly one object realising this shared data structure associated with it. In turn, every category appearing in this rule (or edge) is a pointer into exactly this object.

Implementation

The abstract C++ classes `CategoryCommon` and `Category` implement the functionality of a shared data structure object and a pointer into this object, respec-

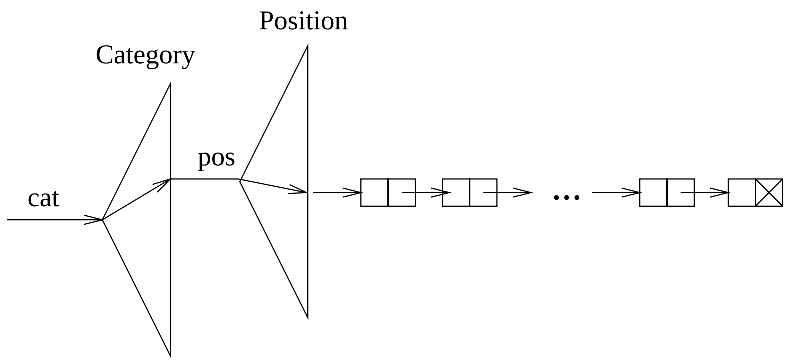


Figure 4.3: *Organisation of the chart. The index* (*cat*, *pos*) *is mapped to a list of edges by two indexing mechanisms (represented by triangles) - first by category, then by string position.*

tively. The parser itself only uses the (pure virtual) methods defined for those abstract classes. Thus, new category types can be added by new subclasses, e.g., `AtomicCategory` and `AtomicCategoryCommon` for atomic categories, without making modifications to the parser code.

4.3 Chart Organisation

From a theoretical point of view, the chart may simply be a list because in the worst case, the required access time is linear in the size of the chart. In practice, however, one might suspect that chart organisation is crucial for the speed of a running system. As a rule of thumb, unifications between categories must be avoided whenever possible because the copying and complex operations involved may slow down the runtime considerably.

4.3.1 Indexing Scheme

The chart consists of two components: the *active* chart and the *passive* chart which contain active and passive edges, respectively. Each edge is stored and is accessible under an *index*. Such an index consists of two parts: 1. a string (which represents a category, also called cat-index) and 2. an integer (a position in the input string). Given such an index (*cat*, *pos*), the respective edges are retrieved by indexing *first* by category, and *secondly* by string position. Under each index, one can, in turn, find a linked list of edges (see also Fig. 4.3).

In general, categories with different cat-indices must not be unifiable whereas, on the other hand, categories with the same cat-index are *not* guaranteed to be unifiable. Ideally, the index should be chosen such that in most cases two categories with the same cat-index are, in fact, unifiable.

The cat-index depends on the category type. For atomic categories, it can just be the category itself (since it is just a string). For first order terms, the cat-index might for example be the functor of the term. Then the two terms $f(a, b)$ and $f(c, d, e)$ have the same cat-index (namely `f`) but are not unifiable. A more sophisticated scheme might take the arity also into account (in the example above, the two cat-indices would be `f/2` and `f/3`, respectively, and would thus (correctly) be considered as not unifiable). For typed feature structures, an obvious choice would be the type of the root node. However, this might be suboptimal because when using a typical HPSG style grammar, a majority of found constituents are just `phrase`s. A more sophisticated scheme might thus also take feature values into account.

It means different things for active and passive edges to be stored under the index $ix = (cat, pos)$. For a passive edge it means that the category on the LHS has cat-index *cat* and *pos* is the leftmost bit in its bit vector, in other words, it starts at position *pos*. An active edge at *ix*, on the other hand, indicates that it is looking for a category with cat-index *cat*, i.e., a node on the RHS labelled with a category with cat-index *cat* is in the dot set, and that this category should start at string position *pos*. This latter fact will typically be the result of an LP constraint.

4.3.2 Chart Operations

There are mainly two chart operations the parsing algorithm uses for a given edge e: inserting e and retrieving all edges which are candidates for completion with e.

Inserting an Edge

An edge e is inserted into the chart as follows:

If e is passive, the cat-index *cat* of the LHS and the leftmost bit of e's bit vector *pos* is determined and e is appended to the list accessible under the index (cat, pos) in the passive chart.

If e is active, for every node v in the dot set with cat-index *cat* and for every string position *pos* where v might possibly start, append e to the list accessible at (cat, pos) in the active chart. Since the dot set in general contains more than one node, and each of those nodes may be found at more than one position, e might be stored under many different indices.

Appending e to a list in this context means appending only if e is not subsumed by an edge already in the list, i.e., LieSL uses a subsumption check (cf. section 3.2.6). (Two edges where one subsumes the other also have the same index.)

For instance, suppose we have typed feature structures as categories with a type hierarchy with two types t and t_1 where t subsumes[3] t_1. For the sake of

[3]This type subsumption has nothing to do with the subsumption check mentioned earlier.

example, we will use the type of the root node as a the cat-index.

The passive edge

$$e_p = \langle 011, t \rightarrow \ldots \rangle$$

will be inserted under the index $(t, 2)$ because it has cat-index t on the LHS and starts at string position 2 (the rule graph on the RHS is not important here, hence the '...'). (The problem that e_p should also be accessible via t_1 because t_1 is subsumed by t will be discussed in the next subsection.)

On the other hand, the active edge

$$e_a = \langle 100,\ t \longrightarrow \textcircled{..} \rightarrow\!\bullet\,\textcircled{t_1}\ \rangle$$

is inserted under the indices $(t_1, 2)$ and $(t_1, 3)$, because a t_1 may be found at position 2 or 3 (if we ignore empty categories for now). Note that the first string position belongs to the first node. If the edge in the rule graph of e_a was an i-edge, then $(t_1, 2)$ would be the only index because due to the immediate precedence constraint it must occur at string position 2 and not later.

Retrieving all Candidates For Completion

At some point, the parsing algorithm wants to perform all possible completions of a newly created edge e with all edges in the chart. The chart determines all those candidates for completion as follows:

Suppose e is passive. Roughly speaking, we determine the index of e and use *this* index to retrieve edges from the *active* chart[4]. With e_p from the example above, all active edges which can be found under the index $(t, 2)$ would be selected. With the assumed type hierarchy, however, we run into the problem that a category with cat-index t might unify with categories with cat-index t_1 so we should retrieve all edges under index $(t_1, 2)$ as well. Thus, the indexing scheme additionally provides a way to get all cat-indices for categories which might unify with a given category. In the example, all unifiable cat-indices for t would be t itself and t_1, thus e_a is (correctly) chosen as a possible candidate for completion with e_p since it is stored (among others) under index $(t_1, 2)$.

For active edges, this mechanism works analogously. In the example, e_a would consider all passive edges under the indices $(t_1, 2)$, $(t_1, 3)$, $(t, 2)$, and $(t, 3)$.

As demonstrated, there is, in general, more than one list that must be searched through. To access all those lists uniformly, the chart is able to create an *iterator* object (cf. Gamma, Helm, Johnson & Vlissides 1995) for an edge e. Dereferencing this iterator object subsequently yields all those edges which are candidates for completion with e. Additionally, the iterator is implemented such that no two edges are tried to be completed more than once.

[4]The index is thus used both for storing and for retrieving.

```
procedure process($A$: Agenda)
last_successful := nil;
$(e, it)$ := dequeue($A$);
while $e \neq$ last_successful do
    while $it$ can be dereferenced do
        $c$ := deref($it$);
        if $e$ and $c$ can be completed do
            last_successful := $e$;
            $e_c$ := complete($e$,$c$);
            add $e_c$ to the chart and to $A$
        endif
    endwhile
    enqueue($A$,($e$,$it$));
    $(e, it)$ := dequeue(A);
endwhile
```

Figure 4.4: *Pseudocode for processing the agenda, initialised after lexical lookup.*

4.4 Control

As in classical chart parsing, LieSL uses an agenda, i.e., a queue, which keeps track of new edges that were inserted into the chart. An *agenda entry* is a pair consisting of an edge and the iterator of that edge (see section 4.3). When arriving at string position i, where lexical item w is found, new edges are created which have an empty rule graph on the RHS and category C on the LHS if $C \rightarrow w$ is a lexical entry (see section 3.2.4). All those edges are then stored in the chart and enqueued into an empty agenda together with their associated chart iterators, i.e., the iterators over all possible candidates for completion (see Fig. 4.5 for the pseudocode of the top level loop which is modification of Fig. 3.4). After this lexical lookup step, the agenda is processed as follows:

Dequeue (e, it) from the agenda. While there are candidates for completion with e in the chart, perform completion with those. In other words, while it can be dereferenced, try to perform completion with those dereferenced entries. If such a completion successfully yields a new edge, insert it into the chart and into the agenda. Finally, add (e, it) to the agenda again. If a complete pass through the agenda is made without one successful completion, we are done at this string position. See Fig. 4.4 for the pseudocode of this loop.

```
procedure parse($x$:list)
1  $n := |x|$;
2  initialise();
3  for $i := 1$ to $|x|$ do    (* Add initial edges for $x[i]$ *)
4      $\beta :=$ set($n, i$);
5      $A :=$ emptyAgenda();
6      (* Scanning *)
7      for all $L \in$ lex_entries($x[i]$) do
8          $e := \langle\beta, (u \to (\emptyset,\emptyset,\emptyset), \{u \mapsto L\}, \emptyset), \emptyset, \pi_n\rangle)$;
9          add $e$ to chart
10         insert $e$ into $A$ (* together with e's chart iterator *)
11     endfor
12     process($A$); (* do all possible completions *)
13 endfor
14 if there is $\langle\vec{1}, (v \to R, \theta, I), \emptyset, \pi\rangle$ with $\theta(v) = S$ then
15     accept; (* If $S$ spans the whole string $\Rightarrow$ accept *)
16 else reject;
```

Figure 4.5: *Top level loop of the parser with an agenda*

4.5 LieSL in Practice

In this section, I will first investigate what role the theoretical predictions in chapter 3 about parseability of LSL grammars (in particular about parseability in polynomial time) play in practice. Secondly, I will present a linguistically motivated grammar which uses ConTroll-style typed feature structures.

Practical Effects of LP Constraints

To investigate what effects the different LP constraints have on the parsing times, I used variations of the simple grammar (using atomic categories):

$$\begin{array}{lcl} A & \to & AA; \varphi \\ A & \to & a \end{array}$$

In particular, (almost) all possible kinds of LP constraints were substituted for φ and LieSL was run on the inputs $a^5, a^6, \ldots$ up to a^{20}. Since atomic categories are implemented using a stringpool, copying and "unifying" those categories amounts just to pointer copying, and pointer comparison, respectively. Thus, the operations on categories themselves are kept to a minimum[5].

[5]Profiling proved that point. On the other hand, most of the overall time when parsing with first order categories was needed for copying, subsumption, and unification (in that order).

See Table 4.1 for the test results. There is one column for each data set. The column headers indicate which isolation (NI: no isolation, RI: right isol., i.e., only nonterminals on RHS isolated, LI: left isol., i.e., only LHS isolated) and precedence (NP: no precedence, WP: Weak prec., IP: Immediate prec.) constraints were attached to the first rule in the grammar above. For instance, LI/WP corresponds to the rule $A \rightarrow A_1 A_2; \langle A \rangle, A_1 < A_2$. Each row shows the parsing times for some input length (which, in turn, is indicated in the first column), the times themselves are given in seconds[6]. Under the time the size of the chart is indicated as (# of active edges, # of passive edges).

Note that the the first four data sets are effectively LSL encodings of *context-free* grammars. In fact, the charts built in those four runs are all the same. The first three of them have merely the same runtimes due to the fact that immediate precedence narrows down the number of possible completion candidates more than weak precedence (see section 4.3).

Considering this and the fact that both data sets which do not involve any isolation (either directly or indirectly) are much slower than the others, implies that the theoretical predictions are borne out (in this case). Furthermore, immediate precedence is an important factor. Note that isolation and immediate precedence are more closely related than isolation and weak precedence because if the RHS of every rule is a single "immediate precedence chain", this implies that the LHS is isolated.

Overall, parsing times decrease as the data sets get more "restrictive". This observation can also be interpreted to show that weak precedence also influences the runtime for the better which is what one might have expected. As one can see from the chart sizes, weak precedence only reduces the number of active edges, but there is still a substantial difference particularly between NI/WP and NI/NP or RI/WP and WI/NP.

A Small Grammar For German

The grammar discussed in this section was written by Frank Richter (cf. Richter & Suhre 1999). It demonstrates the possible usage of LieSL in a linguistic environment. In particular, it captures the effect of freer word order in German sentences using ConTroll-style typed feature structures as categories. In Fig. 4.6, a rule of the grammar is shown (see section A for the complete grammar). It is important to note that complete freedom of word order (including discontinuities) is never allowed, i.e., a lot of isolation and precedence constraints are used.

The parsing times for all the sentences generated by the grammar, e.g., "dass der Mann das Buch dem Kind gibt", are all around 0.6s. Considering the fact that the structures built during a parse are quite large, this is acceptable.

[6]As measured on a SUN SPARC Ultra Enterprise 450 with two 250 MHz Ultra II SPARC CPUs and 512 MB of main memory running under SunOS 5.6.

n	Data Sets (ordered by increasing runtimes in secs)								
	RI/IP	LI/IP	NI/IP	LI/WP	LI/NP	RI/WP	RI/NP	NI/WP	NI/NP
5	0.01	0.01	0.01	0.01	0.02	0.01	0.02	0.01	0.05
	(11,15)				(32,15)	(11,30)	(32,30)	(16,31)	(64,31)
6	0.01	0.01	0.01	0.01	0.03	0.02	0.05	0.03	0.17
	(16,21)				(44,21)	(16,56)	(44,56)	(32,63)	(128,63)
7	0.02	0.01	0.01	0.01	0.04	0.03	0.09	0.07	0.61
	(22,28)				(58,28)	(22,98)	(58,98)	(64,127)	(256,127)
8	0.02	0.03	0.02	0.02	0.07	0.04	0.16	0.17	2.39
	(29,36)				(74,36)	(29,162)	(74,162)	(128,255)	(512,255)
9	0.03	0.03	0.03	0.03	0.09	0.07	0.25	0.48	11.08
	(37,45)				(92,45)	(37,255)	(92,255)	(256,511)	(1024,511)
10	0.04	0.04	0.04	0.05	0.15	0.12	0.46	1.57	55.81
	(46, 55)				(112,55)	(46,385)	(112,385)	(512,1023)	(2048,1023)
11	0.04	0.04	0.04	0.06	0.21	0.17	0.74	5.89	310.66
	(56,66)				(134,66)	(56,561)	(134,561)	(1024,2047)	(4096,2047)
12	0.06	0.06	0.06	0.08	0.27	0.27	1.19	24.62	1899.400
	(67,78)				(158,78)	(67,793)	(158,793)	(2048,4095)	(8192,4095)
13	0.07	0.08	0.08	0.10	0.37	0.42	1.84	115.12	11675.00
	(79,91)				(184,91)	(79,1092)	(184,1092)	(4096,8191)	(16384,8191)
14	0.09	0.08	0.09	0.13	0.48	0.62	2.84	542.25	
	(92,105)				(212,105)	(92,1470)	(212,1470)	(8192,16383)	
15	0.12	0.11	0.10	0.17	0.61	0.93	4.26	2519.53	
	(106,120)				(242,120)	(106,1940)	(242,1940)	(16384,32767)	
16	0.14	0.14	0.15	0.21	0.77	1.35	6.28	11032.61	
	(121,136)				(274,136)	(121,2516)	(274,2516)	(32768,65535)	
17	0.18	0.16	0.16	0.25	0.94	1.96	9.16		
	(137,153)				(308,153)	(137,3213)	(308,3213)		
18	0.20	0.20	0.21	0.29	1.17	2.73	13.11		
	(154,171)				(344,171)	(154,4047)	(344,4047)		
19	0.24	0.25	0.24	0.37	1.40	3.91	18.57		
	(172,190)				(382,190)	(172,5035)	(382,5035)		
20	0.28	0.29	0.29	0.43	1.69	5.42	25.76		
	(210,401)				(422,210)	(191,6195)	(422,6195)		

Table 4.1: *Parsing times and chart sizes (active,passive) for variations of the test grammar. The charts for the first four data sets are all the same.*

```
% Rule for flat finite verb last sentences
vp( (hf_phrase, dtr1:SU, dtr2:IO, dtr3:DO, dtr4:VE, synsem:nonloc:[],
     synsem:loc:cat:(head:HD,
                     subcat:[])) )
 ==>
  np1( (SU, bin_phrase, synsem:(NP1,loc:cat:head:noun))),
  np2( (IO, bin_phrase, synsem:(NP2,loc:cat:head:noun))),
  np3( (DO, bin_phrase, synsem:(NP3,loc:cat:head:noun))),
    v( (VE, word, synsem:loc:cat:((head:(HD,dsl:[]),
                                 subcat:[(NP1),(NP2),(NP3)]))))
; np1 < v, np2 < v, np3 < v,
  [np1], [np2], [np3]
.
```

Figure 4.6: *A rule from Richter's grammar.*

Unfortunately, large scale LSL grammars were not available at the time of writing this thesis. It is thus still an open question how LieSL performs in real linguistic applications.

However, since Richter's grammar uses quite a lot LP constraints, particularly isolation, it is a reasonable assumption that this property will also hold for large grammars and thus the odds are not too bad that LieSL performs reasonably well.

Chapter 5

Conclusion

In this chapter I will summarise the basic results of this thesis and discuss some of their consequences. After the following summary, I want to briefly discuss how the sufficient condition for polynomial time parseability is practical from a linguistic point of view. The last section then presents possible extensions to LSL as proposed in Götz & Penn (forthcoming).

5.1 Summary

In this thesis I presented the LSL grammar formalism, a natural generalisation of context free grammars, to express natural language phenomena which involve freer word order. The set of LSL languages is a proper superset of the context free languages but the same decidability results hold for the two classes. However, they differ in their closure and complexity theoretic properties. In particular, both the general and fixed membership problem are $\mathcal{NP}$-complete for LSL grammars whereas for CFL, the general is $\mathcal{P}$-complete and the fixed is $LOGCFL$-complete.

I furthermore presented a generalisation of Earley's algorithm for parsing LSL grammars. The worst case complexity of this algorithm is exponential (as could be expected). It can be shown, however, that parsing in polynomial time can be ensured if the grammar satisfies the condition that the yield of every recursive nonterminal has at most a constant number of discontinuities (at most a constant number of "blocks").

A unification-based extension of this parsing algorithm was implemented, providing atomic categories, first order terms, and ConTroll-style typed feature structures. Experiments with this implementation (called LieSL) showed that the theoretical predictions are borne out in that the more isolation there is, the shorter the parsing times are. Furthermore, factors which were not so important in the theroretical analysis (precedence), have a considerable influence in practice.

A grammar using typed feature structures demonstrated how LSL can be used for writing grammars in a linguistic framework.

5.2 The Role of m-Isolation

We have seen that the condition of Corollary 10 is directly expressible in LSL. For natural language, however, this condition is too strong. One might want to relax the notion of isolation to something like *m-isolation*, meaning at most m blocks (or, in other words, at most $m-1$ discontinuities). With Proposition 27, this is sufficient to ensure parseability in polynomial time. I conjecture that there is such a number for natural languages.

To see that this number might even be bigger than 2, consider the sentence:

Von dem Mann	habe ich	viele Bücher	gelesen,	der “The Hobbit” geschrieben hat.
By the man	have I	many books	read	who “The Hobbit” written has.

“I have read many books by the man who wrote ‘The Hobbit’. ”

Here, the discontinuous NP “Viele Bücher von dem Mann, der ‘The Hobbit’ geschrieben hat” has three blocks (two discontinuities). Holan et al. (1998) argue that a number of 5 discontinuities is certainly enough for Czech.

Including m-isolation into the parsing algorithm is straightforward and was actually implemented in the LieSL system. How it will influence the formal language properties is an open question. I would suspect, however, that it does not make a big difference.

Further linguistic research should be concerned about what this m might be for different languages.

5.3 Outlook

In Götz & Penn (forthcoming), other LP constraints than isolation, weak precedence, and immediate precedence are proposed. In this section, I want to present the three most important of those and discuss them briefly. These are:

1. Liberation, written as $\langle X\rangle(:Y)$. Liberation generalises isolation in that components of an isolated constituent are explicitly allowed to violate the isolation constraint. The example means that “X is isolated except for a Y it contains”. “contains” here is equivalent to “derives”.

2. Universal Quantification $(\forall X \in Y.\varphi)$. This kind of constraint enables formulating statements like: “For all X contained in Y (for all Xs derived by Y), the constraint φ must hold”.

3. Existential Quantification $(\exists X \in Y.\varphi)$. Similarly, this means: “There must be an X contained in Y which satisfies φ”.

From a linguistic point of view, it is desirable to have such constraints. From a processing point of view, however, it is not clear how to implement them in an efficient manner: In LSL as presented in this thesis, LP constraints can be checked "locally" during parsing; the derivation tree, i.e., the "history", of a constituent can be ignored to check if it is isolated, or if it (weakly or immediately) precedes another constituent. With the constraints above, however, this is no longer the case. Those constraints require knowledge about the complete derivation tree of a constituent to be able to check, for example, if some other constraint holds for all contained constituents of a certain type (as would be needed for universal quantification). Furthermore, the effects on the formal language and complexity results are not at all obvious.

It will be an important topic in further research to integrate these new constraints into LSL and at the same time ensure efficient processing.

Bibliography

Ait-Kaci, Hassan (1991): *Warren's Abstract Machine - A Tutorial Reconstruction*. MIT Press, Cambridge, Massachusetts.

Aldag, Bjørn (1998): LSL - preliminary version. Unpublished manuscript, Seminar für Sprachwissenschaft, Universität Tübingen.

Barton, G. Edward, Robert C. Berwick & Eric Sven Ristad (1987): *Computational Complexity and Natural Language*. MIT Press.

Carpenter, Bob (1992): *The Logic of Typed Feature Structures*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.

Carpenter, Bob & Gerald Penn (1998): *ALE - The Attribute Logical Engine, Release 3.1.*. Language Technologies Institute, Carnegie Mellon University, URL: `http://www.sfs.nphil.uni-tuebingen.de/~gpenn/ale.html`.

Cormen, Thomas H., Charles E. Leiserson & Ronald L. Rivest (1990): *Introduction to Algorithms*. MIT Press.

Covington, Michael (1995): *Natural Language Processing for Prolog Programmers*. Prentice Hall.

Darnell, Peter A. & Philip E. Margolis (1991): *C - A Software Engineering Approach*. 2nd edn, Springer.

Dassow, J. & G. Păun (1989): *Regulated Rewriting in Formal Language Theory*. Springer.

Earley, Jay (1968): An Efficient Context Free Parsing Algorithm. In: *Communications of the ACM*. Vol. 13, ACM, pp. 353–391.

Engelfriet, J. & G. Rozenberg (1997): Node Replacement Graph Grammars. *In* Rozenberg (1997), chapter 1.

Gamma, Erich, Richard Helm, Ralph Johnson & John Vlissides (1995): *Design Patterns - Elements of Object-Oriented Software*. Addison-Wesley.

Garey, Michael R. & David S. Johnson (1979): *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

Gazdar, G., E. Klein, G. Pullum & I. Sag (1985): *Generalized Phrase Structure Grammar*. Harvard University Press/Blackwell's, Cambridge.

Gazdar, Gerald & Chris Mellish (1989): *Natural Language Processing in Prolog.* Addison Wesley.

Gerdemann, Dale (1991): Parsing and Generation of Unification Grammars. PhD thesis, University of Illinois at Urbana-Champaign.

Ginsburg, Seymour (1975): *Algebraic and Automata-Theoretic Properties of Formal Languages.* North Holland Publishing Company.

Götz, T. & G. Penn (forthcoming): A Linear Specification Language for Parsing with Freer Word-Order Languages. *In:* G. Penn, ed., *An Introduction to LSL: Motivation, Semantics and Computation.* Arbeitspapier des Sonderforschungsbereichs 340, Tübingen.

Götz, Thilo, Detmar Meurers & Dale Gerdemann (1997): *The ConTroll Manual, ConTroll v1.0.β, XTroll v5.0.β.* Seminar für Sprachwissenschaft, Tübingen, URL: `http://www.sfs.nphil.uni-tuebingen.de/controll.html`.

Greenlaw, R., H.J. Hoover & W.L. Ruzzo (1995): *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press.

Holan, T., V. Kuboň, K. Oliva & M. Plátek (1998): Two Useful Measures of Word Order Complexity. In: *Proceedings of Dependency-based Grammars: Proceedings of the Workshop.* COLING-ACL '98, Montreal, pp. 21–28.

Holan, T., V. Kuboň & M. Plátek (1995): Parsing Free-Word-Order Languages. *In:* M. Bartošek, J. Staudel & J. Wiedemann, eds, *SOFSEM '95: Theory and Practice of Informatics.* Vol. 1012 of *LCNS*, Springer, pp. 379–384.

Holzer, Markus (1999): It is undecidable if an LSL grammar generates a context-free language. Unpublished Ms., Universität Tübingen.

Holzer, Markus & Oliver Suhre (1999): The fixed membership problem for LSL grammars is NP-hard. Unpublished Ms., Universität Tübingen.

Hopcroft, John E. & Jeffrey D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, Massachusetts.

Huynh, D. T. (1983): 'Commutative grammars: the complexity of uniform word problems', *Information and Control* **57**, 21–39.

Johnson, David S. (1990): A Catalog of Complexity Classes. *In* van Leeuwen (1990), chapter 2, pp. 67–161.

Johnson, Mark (1985): Parsing Discontinuous Constituents. In: *Proceedings of the 23rd Annual Meeting of the ACL.* Association of Computation Linguistics, Chicago, pp. 127–132.

King, Paul John (1994): An expanded logical formalism for HPSG. Technical Report 59, Sonderforschungsbereich 340. Arbeitspapiere des SFB 340.

Mehlhorn, Kurt, Stefan Näher, Michael Seel & Christian Uhrig (1999): *The LEDA User Manual Version 3.8.* MPI Saarbrücken / LEDA Software GmbH, URL: `http://www.mpi-sb.mpg.de/leda`.

Nagl, Manfred (1979): *Graph-Grammatiken - Theorie, Implementierung, Anwendungen.* Vieweg, Wiesbaden.

Papadimitriou, Christos H. (1995): *Computational Complexity.* 2nd edn, Addison-Wesley.

Pollard, Carl & Ivan Sag (1994): *Head Driven Phrase Structure Grammar.* The University of Chicago Press.

Radzinski, Daniel (1990): 'Unbounded Syntactic Copying in Mandarin Chinese', *Linguistics & Philosophy* **13**(1), 113–127.

Reape, Mike (1991): Parsing Bounded Discontinuous Constituents: Generalisations of some common algorithms. In: *Proceedings of the First Computational Linguistics in the Netherlands Day (CLIN).* OTK, Universiteit van Utrecht.

Richter, Frank & Oliver Suhre (1999): 'The Linear Specification Language LSL: A Grammar Formalism For Parsing Languages with Freer Word Order (System Demonstration)', presented at the final workshop of the SFB 340: "Linguistic Form and Computation", Bad Teinach, 11.10. - 13.10.

Rozenberg, G., ed. (1997): *Handbook of Graph Grammars and Computing by Graph Transformations.* World Scientific Publishing Company, London.

Salomaa, Arto (1973): *Formal Languages.* Academic Press.

Schildt, Herbert (1995): *C++ - The Complete Reference.* 2nd edn, McGraw-Hill.

Shieber, Stuart (1985): 'Evidence against the context-freeness of natual language', *Linguistics & Philosophy* **8**(1), 333–343.

Shieber, Stuart (1986): *An Introduction to Unification-Based Approaches to Grammar.* Lecture Notes no. 5, CSLI, Stanford.

SICS (1999): *SICStus Prolog Manual, Release 3.7.1..* Programming Systems Group, Swedish Institute for Computer Science, URL: `http://www.sics.se/isl/sicstus.html`.

van Leeuwen, Jan, ed. (1990): *Algorithms and Complexity.* Vol. 1 of *Handbook of Theoretical Computer Science*, MIT Press.

Vijay-Shanker, K., David J. Weir & Aravind K. Joshi (1987): Characterizing structural descriptions produced by various grammatical formalisms. In: *Proceedings of the 25th Annual Meeting of the ACL.* Association of Computation Linguistics, Stanford.

Warren, David H. D. (1983): An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, CA.

Appendix A

An Example Grammar for LieSL

This section shows the grammar written by Frank Richter for a fragment of German. It uses ConTroll's typed feature structures.

Sentences parseable with this grammar are (among others):

```
dass der mann dem kind das buch gibt
dass dem kind der mann das buch gibt
dass das buch dem kind der mann gibt

der mann gibt dem kind das buch
der mann gibt das buch dem kind
das buch gibt der mann dem kind
```

A.1 The Type Hierarchy

```
type_hierarchy

bot
  sign synsem:synsem phon:list
    bin_phrase dtr1:sign dtr2:sign
    hf_phrase dtr1:sign dtr2:sign dtr3:sign dtr4:sign
    word
  synsem loc:loc nonloc:list
  loc cat:cat cont:cont
  cat head:head subcat:list
  cont index:index
  index per:num gen:gen
  num
    sg
    pl
  gen
    fem
    masc
    neut
  head
    marker spec:synsem
    art  case:case
    verb
      vfinal dsl:list
      verb2
    noun case:case
  case
    nom
    akk
    dat
  string
.
```

A.2 The Grammar

```
% Grammar rules

% Rules for definite NPs
np( (bin_phrase, dtr1:ART,
                 dtr2:NP, synsem:nonloc:[],
                 synsem:loc:cat:(head:HD,
                                 subcat:[])) )
 ==>
   a( (ART, word, synsem:(X, loc:(cat:head:art,
                                  cont:index:per:Num))) ),
   n( (NP, word, synsem:loc:(cat:(head:(HD, noun),
                                  subcat:[X]),
                             cont:index:per:Num)) )
; a << n
.

% Rule for flat finite verb last sentences
vp( (hf_phrase, dtr1:SU, dtr2:IO, dtr3:DO, dtr4:VE, synsem:nonloc:[],
     synsem:loc:cat:(head:HD,
                     subcat:[])) )
 ==>
  np1( (SU, bin_phrase, synsem:(NP1,loc:cat:head:noun))),
  np2( (IO, bin_phrase, synsem:(NP2,loc:cat:head:noun))),
  np3( (DO, bin_phrase, synsem:(NP3,loc:cat:head:noun))),
    v( (VE, word, synsem:loc:cat:((head:(HD,dsl:[]),
                                   subcat:[(NP1),(NP2),(NP3)]))))
; np1 < v,
  np2 < v,
  np3 < v,
  [np1],
  [np2],
  [np3]
.

% Rule for the dass-complementiser
s( (bin_phrase, dtr1:MA, dtr2:VP, synsem:nonloc:[],
    (synsem:loc:cat:(head:HD,
                     subcat:SC))) )
 ==>
   m( (MA, word, synsem:loc:cat:head:spec:SS) ),
  vp( (VP, hf_phrase, (synsem:(SS, loc:cat:(head:HD,subcat:SC)))) )
; m << vp,
  [vp]
.

% Rule for the complement of a V2 verb
vp( (bin_phrase,
     synsem:nonloc:NL,
     dtr1:V2, dtr2:VP,
     synsem:loc:cat:(head:VH,
                     subcat:[])) )
 ==>
  v2( (V2, word, synsem:nonloc:NL,
       synsem:loc:cat:(head:(VH,verb2),
                       subcat:[SC])) ),
   v( (VP, bin_phrase, synsem:nonloc:[],
       synsem:SC) )
; v2 << v,
  [v]
.

% 2 rules for the binary complement realisation of the verb trace
vp( (bin_phrase, synsem:nonloc:[],
     dtr1:NP, dtr2:V,
     synsem:loc:cat:(head:dsl:(X,ne_list),
                     subcat:[NP1])) )
 ==>
  np( (NP, bin_phrase, synsem:(NP2,loc:cat:head:noun)) ),
   v( (V, word, synsem:loc:cat:(head:dsl:X,
                                subcat:[(NP1),(NP2)])) )
; [np]
.
```

```
vp( (bin_phrase, synsem:nonloc:[],
     dtr1:NP, dtr2:V,
     synsem:loc:cat:(head:dsl:(X,ne_list),
                     subcat:[])) )
 ==>
  np( (NP, bin_phrase, synsem:(NP1,loc:cat:head:noun)) ),
   v( (V, bin_phrase, synsem:loc:cat:(head:dsl:X,
                                      subcat:[NP1])) )
; [np],
  [v]
.

% Rule for topicalisation in V2 sentences
s( (bin_phrase, synsem:nonloc:[],
    dtr1:NP, dtr2:VP,
    synsem:loc:cat:(head:(HD,verb2),
                    subcat:(SC,[]))) )
 ==>
  np( (NP, bin_phrase, synsem:loc:NLNP) ),
  vp( (VP, bin_phrase, synsem:(loc:cat:(head:HD,
                                        subcat:SC),
                               nonloc:[NLNP])) )
; np << vp,
  [np],
  [vp]
.


%% Trace, empty category
v( (word, synsem:nonloc:[], phon:[],
          synsem:loc:cat:(head:dsl:SC,
                          subcat:(SC, [(loc:cat:head:noun),
                                       (loc:cat:head:noun)]))) ) ==>
;
.

%% Lexicon

%  Nouns
buch ---> l( (word, (synsem:loc:((cat:(head:(noun,case:CA),
                                      subcat:[(loc:((cat:head:(art,case:CA)),
                                                    (cont:index:IN)))])),
                                 cont:index:(IN,gen:neut))),
                    phon:[buch],
                     synsem:nonloc:[]) ).
kind ---> l( (word, (synsem:loc:((cat:(head:(noun,case:CA),
                                  subcat:[(loc:((cat:head:(art,case:CA)),
                                                (cont:index:IN)))])),
                              cont:index:(IN,gen:neut))),
                     phon:[kind],
                     synsem:nonloc:[]) ).
mann ---> l( (word, (synsem:loc:((cat:(head:(noun,case:CA),
                                  subcat:[(loc:((cat:head:(art,case:CA)),
                                                (cont:index:IN)))])),
                              cont:index:(IN,gen:masc))),
                     phon:[mann],
                     synsem:nonloc:[]) ).

% determiner
der --->  l( (word, (synsem:loc:((cat:(head:(art, case:nom),
                                 subcat:[])),
                              cont:index:(per:sg,
                                          gen:masc))),
                     phon:[der],
                     synsem:nonloc:[]) ).
das --->  l( (word, (synsem:loc:((cat:(head:(art, case:(nom;akk)),
                                 subcat:[])),
                              cont:index:(per:sg,
                                          gen:neut))),
                     phon:[das],
                     synsem:nonloc:[]) ).
dem --->  l( (word, (synsem:loc:((cat:(head:(art, case:dat),
                                 subcat:[])),
                              cont:index:(per:sg,
                                          gen:(masc;neut)))),
                     phon:[dem],
                     synsem:nonloc:[]) ).

% verbs
% vfinal version of gibt
gibt ---> l( (word, (synsem:loc:cat:(head:dsl:[],
                                 subcat:[(loc:cat:(head:(noun, case:nom),
                                                  subcat:[])),
                                         (loc:cat:(head:(noun, case:dat),
                                                  subcat:[])),
                                         (loc:cat:(head:(noun, case:akk),
                                                  subcat:[]))])),
                     phon:[gibt],
                     synsem:nonloc:[]) ).
```

```
% verb2 versions of gibt
gibt ---> l( (word, (synsem:loc:cat:(head:verb2,
                                subcat:[(loc:cat:(head:dsl:[
                                        (loc:cat:(head:(noun, case:dat),
                                                  subcat:[])),
                                        (loc:cat:(head:(noun, case:akk),
                                                  subcat:[]))],
                                                  subcat:[]))])),
                     phon:[gibt],
                     synsem:nonloc:[(cat:(head:(noun, case:nom),
                                          subcat:[]))]) ).
gibt ---> l( (word, (synsem:loc:cat:(head:verb2,
                                subcat:[(loc:cat:(head:dsl:[
                                        (loc:cat:(head:(noun, case:nom),
                                                  subcat:[])),
                                        (loc:cat:(head:(noun, case:dat),
                                                  subcat:[]))],
                                                  subcat:[]))])),
                     phon:[gibt],
                     synsem:nonloc:[(cat:(head:(noun, case:akk),
                                          subcat:[]))]) ).
gibt ---> l( (word, (synsem:loc:cat:(head:verb2,
                                subcat:[(loc:cat:(head:dsl:[
                                        (loc:cat:(head:(noun, case:nom),
                                                  subcat:[])),
                                        (loc:cat:(head:(noun, case:akk),
                                                  subcat:[]))],
                                                  subcat:[]))])),
                     phon:[gibt],
                     synsem:nonloc:[(cat:(head:(noun, case:dat),
                                          subcat:[]))]) ).

% complementiser
dass ---> l( (word, (synsem:loc:cat:((head:spec:loc:cat:(head:vfinal,
                                                    subcat:[]),
                                 subcat:[]))),
                     phon:[dass],
                     synsem:nonloc:[])).
```