

# On the Transformational Derivation of an Efficient Recognizer for Algol 68

Caspar Derksen  
caspard@cs.kun.nl

August 1995

Masters Thesis no. 357  
Catholic University of Nijmegen



# Preface

This work is the result of my graduation project at the Catholic University of Nijmegen. It has been supervised by Prof. C.H.A. Koster.

I should like to thank my parents, Donald, Cindy, Walter, Joyce, Berry, and others for their friendship and support while I was working on this thesis. Thanks go to Paula, Marco, Erik and others for enlightening life at the office during work.

Special thanks go to Prof. C.H.A. Koster for awakening my interest in two level grammars and for his support and invaluable advice and patience.

Caspar Derksen  
Nijmegen, August 1995



# Abstract

During the sixties two level van Wijngaarden grammars (2VWG) were introduced for the formal definition of Algol 68. Two level van Wijngaarden grammars provide a formalism suited for writing rigorous formal definitions which read like human prose. Both the context free and context sensitive syntax of Algol 68 are defined in 2VWG. However, the specification is not executable.

The Extended Affix Grammar (EAG) formalism is closely related to 2VWG. Any 2VWG can be simulated in EAG. From a given EAG an exact recognizer for the language defined by the EAG can be derived automatically. Termination of such a recognizer is guaranteed if certain liberal well-formedness conditions are satisfied. However, in general some effort is necessary to enforce well-formedness.

CDL3 is a programming language which rides the borderline between affix grammars and programs. It can be seen as a deterministic version of EAG. From a grammar in CDL3 efficient parsers can be generated.

Specifications in 2VWG and EAG often are of a declarative nature. Our purpose is to derive efficient parsers from such a language specification in a transformational way. We have generalized well known transformations techniques for making context free grammars top down parsable to two level grammars. Furthermore, we have defined transformations for applying the well-known unfold/rearrange/fold-technique on specifications which are formulated as a two level grammar. These transformations can be used for imposing stronger well-formedness conditions on a two level grammar and for operationalizing declarative constructs in the grammar.

As a case study, we have undertaken an attempt to derive a new specification for Algol 68 in EAG by applying language preserving transformations to the original grammar. Secondly, we have tried to make the resulting EAG deterministic in order to obtain an efficient recognizer in CDL3.



# Contents

<b>Preface</b>	<b>i</b>
<b>Abstract</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The Problem . . . . .	1
1.2 Aims of the Present Work . . . . .	2
1.3 Outline of the Thesis . . . . .	2
<b>2 Two Level Grammars</b>	<b>3</b>
2.1 Two Level van Wijngaarden Grammars . . . . .	3
2.1.1 Formal Definition . . . . .	3
2.1.2 Properties of 2VWG . . . . .	8
2.1.3 Hyper-derivations and Conjugations . . . . .	9
2.1.4 Conjugation Grammars . . . . .	10
2.1.5 Transparent Two Level Grammars . . . . .	11
2.2 Extended Affix Grammars . . . . .	12
2.2.1 Formal Definition of EAG . . . . .	12
2.2.2 Properties of EAG . . . . .	15
2.2.3 Implementation . . . . .	16
2.3 CDL3 . . . . .	18
2.3.1 Formal Definition . . . . .	18
2.3.2 Properties of CDL3 . . . . .	25
2.4 Conclusions . . . . .	26

<b>3</b>	<b>Transformations on Two Level Grammars</b>	<b>27</b>
3.1	Semantical Foundations . . . . .	27
3.1.1	Equivalence of Grammars . . . . .	27
3.1.2	Equivalence of Rules . . . . .	28
3.1.3	Equivalence of Hypernotations . . . . .	28
3.2	Notational Conventions . . . . .	28
3.2.1	Notation of Transformation rules . . . . .	29
3.2.2	Notation of Derivations . . . . .	29
3.3	Basic Transformation Rules . . . . .	29
3.3.1	General Transformations . . . . .	29
3.3.2	Folding and Unfolding . . . . .	30
3.3.3	Rule Introduction and Pruning . . . . .	35
3.3.4	Predicate Introduction and Removal . . . . .	37
3.3.5	Left-factorization . . . . .	38
3.3.6	Left-recursion Removal . . . . .	40
3.3.7	Restriction . . . . .	42
3.3.8	Invariant Introduction and Removal . . . . .	43
3.3.9	Lifting and Sinking . . . . .	43
3.3.10	Embedding . . . . .	43
3.3.11	Change Order . . . . .	44
3.3.12	Symbolic Evaluation . . . . .	44
3.3.13	Renaming . . . . .	44
3.3.14	Transformation of the Metagrammar . . . . .	45
3.4	Further Transformation Rules . . . . .	46
3.4.1	Transformations using Equality Tests . . . . .	46
3.4.2	Transformations on Lists . . . . .	46
3.5	Sample Developments . . . . .	47
3.5.1	Concatenation of Lists . . . . .	47
3.5.2	Splitting of Lists . . . . .	49
3.5.3	Ravelling of Moods . . . . .	50
3.6	Conclusions . . . . .	51

<b>4</b>	<b>An Extended Affix Grammar for Algol 68</b>	<b>53</b>
4.1	Imaging 2VWG into EAG . . . . .	53
4.1.1	Strategy . . . . .	55
4.2	Deriving an EAG for Algol68 . . . . .	55
4.2.1	Overview of the process . . . . .	55
4.2.2	Separation of the First and Second Level . . . . .	56
4.2.3	From Relations to Functions . . . . .	59
4.2.4	From Flat Domains to Tree Structures . . . . .	59
4.2.5	Left-recursion Removal . . . . .	62
4.2.6	Optimizations . . . . .	62
4.2.7	Sample Derivations . . . . .	64
4.2.8	On the Terminal Objects of Algol 68 . . . . .	72
4.2.9	Implementation Status . . . . .	74
4.3	Practical Usability of <code>eag-compile</code> . . . . .	75
4.3.1	Compiling and Debugging . . . . .	75
4.3.2	The Typing System . . . . .	75
4.3.3	An Experiment . . . . .	76
4.3.4	Reliability of <code>eag-compile</code> . . . . .	79
4.4	Conclusions . . . . .	79
4.4.1	Conclusions about Transformation of Two Level Grammars . . . . .	79
4.4.2	Conclusions about the EAG Formalism . . . . .	80
4.4.3	Conclusions about <code>eag-compile</code> . . . . .	80
<b>5</b>	<b>Towards a Parser for Algol 68 in CDL3</b>	<b>81</b>
5.1	Imaging EAG into CDL3 . . . . .	81
5.1.1	Strategy . . . . .	82
5.1.2	Typing the Hyper-rules . . . . .	82
5.1.3	Rule Ordering and Algorithm Classification . . . . .	82
5.1.4	Example . . . . .	83
5.2	Additional Transformations . . . . .	84
5.2.1	Decomposition of Grammars . . . . .	84
5.2.2	Place-Holder . . . . .	84

5.3	Deriving a Parser for Algol 68 in CDL3 . . . . .	86
5.3.1	Lexical Analysis . . . . .	86
5.3.2	The Pre-Scan . . . . .	88
5.3.3	Context Free Analysis . . . . .	89
5.3.4	Imposing Two Pass Affix Evaluation . . . . .	92
5.3.5	Context Sensitive Analysis . . . . .	93
5.4	Overview of the Parser . . . . .	97
5.4.1	Module Structure . . . . .	97
5.4.2	Implementation Status . . . . .	97
5.4.3	Size of the Parser . . . . .	97
5.5	Practical Usability of CDL3 . . . . .	98
5.6	Conclusions . . . . .	98
5.6.1	Future Work . . . . .	99
<b>6</b>	<b>Conclusions</b>	<b>101</b>
6.1	Two Level Grammars . . . . .	101
6.2	Transformation of Two level Grammars . . . . .	101
6.2.1	Extended Affix Grammars . . . . .	102
6.2.2	CDL3 . . . . .	102
6.2.3	Future Work . . . . .	102
	<b>Bibliography</b>	<b>103</b>
	<b>Index</b>	<b>106</b>

# Chapter 1

## Introduction

In this chapter, the goal of this thesis and the questions addressed in it are explained. Furthermore, an outline of the thesis is given.

The reader of this thesis is advised to have read the Revised Report on the Algorithmic Language Algol 68 [Wij76] and to be familiar with context free grammars and their properties.

### 1.1 The Problem

During the sixties and early seventies various forms of two level grammars were introduced. In a two level grammar, both context free and context sensitive syntax of a language, as well as its semantics, can be described. One notable example is the grammar of Algol 68. In [Wij76], a two level van Wijngaarden grammar (2VWG) is given which completely defines the syntax of the language, including all of its context conditions. However, the grammar is not executable.

Nowadays, various implementations of two level grammars are available, compiling an input grammar into a recognizer or parser for the language described by it. These implementations may impose various restrictions on the input grammar. At least some restrictions are necessary in order for a grammar to be executable. In general, liberal restrictions allow more expressive power of the formalism. On the other hand, stronger restrictions enable more efficient implementations.

The implementor of a compiler-compiler system has to find the right balance between generality and efficiency, hereby determining the practical applicability of her system. We will examine the practical applicability of two such systems. The first system studied is an EAG-compiler imposing virtually no restrictions at all. The second system studied is a CDL3-compiler imposing quite strong restrictions.

A grammar in a liberal formalism can be viewed as the specification of an implementation grammar in a more restricted formalism. We will examine whether transformational methods can be applied successfully in a grammatical context. This question is of relevance, because obtaining a correct implementation from a language specification may be an issue in the problem of compiler correctness. Transformational methods may provide a way of dealing

with this problem.

In [Kos69], it is recognized that the rules of a two level grammar may be viewed as procedures in a parsing algorithm. This observation forms the basis of the grammatical programming paradigm on which CDL3 is based. We will also examine the applicability of this paradigm to real life grammars.

## 1.2 Aims of the Present Work

The syntax of Algol 68 is one of the most ambitious efforts ever undertaken to describe a major programming language by a formal grammar. It is therefore an ideal object on which to try modern analysis and transformation methods.

As described in this thesis, we have tried to transform the grammar for Algol 68 in 2VWG into EAG. This resulted in a method for transforming 2VWG into EAG and yielded an Extended Affix Grammar for Algol 68. Parts of this grammar have served to validate the reliability of the EAG-compiler. A prototype implementation of Algol 68 in EAG was aimed at, but this turned out to be infeasible, mainly due to lack of efficiency of the EAG-compiler and the parsers generated by it.

Next, we have tried to transform the EAG of Algol 68 into CDL3 in order to yield an efficient implementation and a general method for transforming EAG into CDL3. In this case, the restrictions imposed by CDL3 turned out to be no fundamental obstacles. Furthermore, the thesis shows that EAG can be used as a tool for experimentation with prototypes from which efficient implementations in CDL3 can be derived. The transformations needed for deriving an implementation for Algol 68 are exemplary, and should be sufficient for most other languages used in practice.

## 1.3 Outline of the Thesis

In chapter 2, three types of two level grammars (2VWG, EAG and CDL3) and their implementations are described. In chapter 3, a number of language preserving transformations on two level grammars is defined. The applicability of these transformations is demonstrated in the next chapters. In chapter 4, the problems encountered in transforming a grammar in 2VWG into EAG are described and an overview of the derivation of an EAG for Algol 68 is given. Furthermore, conclusions concerning the practical applicability of the EAG-compiler used are drawn. In chapter 5, the problems met in transforming a grammar in EAG into a CDL3 program are outlined and a description is given of the derivation of a recognizer for Algol 68 in CDL3. Furthermore, conclusions concerning the restrictions imposed by CDL3 are drawn. Finally, in chapter 6 the conclusions resulting from this work are resumed and some suggestions for possible continuations on the project are given.

## Chapter 2

# Two Level Grammars

The family of two level grammars comprises a broad range of formalisms especially suited for the definition and implementation of languages. In [Kos<sub>1</sub>] and [Kos93], various types of two level grammars are discussed and compared.

In this chapter, three particular formalisms are discussed. First, in §2.1, two level van Wijngaarden grammars are described, and some definitions which will be needed in the next chapter are given. Two level van Wijngaarden grammars are of a *generative* nature, which makes them especially suited for language definition. CDL3, described in §2.3, is a formalism of *analytical* nature, designed for implementing programming languages. In between are the Extended Affix Grammars (EAG) which are described in §2.2. Finally, a brief resumé and some conclusions are given in §2.4.

### 2.1 Two Level van Wijngaarden Grammars

Two level van Wijngaarden grammars (2VWGs) were introduced by van Wijngaarden for the formal description of Algol 68. They are a ‘pure’ form of two level grammars, without admixture of functions from outside the formalism.

In §2.1.1, a formal definition of 2VWG is given and some important notions are defined. In §2.1.2 some properties of 2VWG are discussed. In §2.1.3, more important definitions are given. Finally, in §2.1.4 and §2.1.5 two formalism closely related to 2VWG are discussed, because of their relation with the work in the rest of this thesis. Furthermore, the ideas behind some of the transformations in the next chapter originate from these formalisms.

#### 2.1.1 Formal Definition

In the definition of 2VWG, as well in the definitions of EAG and CDL3, sets of production rules are defined. Each production rule is assumed to have an unique label enabling rules with the same meaning to have multiple instances. However, in order to obtain smaller definitions, we will abstract from these labels and speak of sets of production rules where actually sets of tuples of a label and a production rule are meant.

DEFINITION 2.1.1 A two level van Wijngaarden grammar is a 6-tuple

$$G = (M, V, T, P_M, P_H, S)$$

with the following components:

$M$  is a finite alphabet of *metanotions* (*meta-nonterminals* or *meta-variables*).

$V$  is a finite alphabet of *syntactic marks*, with  $M \cap V = \emptyset$ .

Elements of  $V^+$  are called *protonotions*.

$T$  is a finite set of *terminal symbols*, with  $T \subset V_S^+$ .

$P_M$  is a finite set of *metarules*, with  $P_M \subset M \times (M \cup V)^*$ .

$P_H$  is a finite set of *hyper-rules*, with  $P_H \subset (H \setminus T) \times H^*$ , where  $H = (M \cup V)^+$ . The elements of  $H$  are called *hypernotations*.

$S$  is the *start hypernotation*, with  $S \in H$ .

As usual, a rule is composed of a *left hand side*, followed by a *right hand side*. The right hand side of a rule is called an *alternative* for the left hand side. An alternative consists of a possibly empty sequence of *members*.

Hyper-rules producing the empty string  $\varepsilon$  are called *predicates*.

### Notation

NOTATION 2.1.1 A metarule  $(x, y_1y_2 \dots y_n)$  is written as

$$x :: y_1y_2 \dots y_n. \quad (\text{without separating comma's})$$

When both  $x :: y$ . and  $x :: z$ . are metarules with the same left hand side  $x$ , it is customary to write

$$x :: y; z.$$

A hypernotation  $x = x_1x_2 \dots x_n$  is written as

$$x_1x_2 \dots x_n$$

A hyper-rule  $(x, y_1y_2 \dots y_n)$  is written as

$$x : y_1, y_2, \dots, y_n. \quad (\text{with separating comma's})$$

When both  $x : y$ . and  $x : z$ . are hyper-rules with the same left hand side  $x$

$$x : y; z.$$

is usually written.

By convention, every terminal symbol ends with `symbol`.

### Metagrammar

For every  $x \in M$ , the 4-tuple  $G_x = (M, V, P_M, x)$  forms a context free grammar, called the *metagrammar* of  $x$ . The *direct metaproduction* relation  $\rightarrow_{P_M}$  on strings in  $(M \cup V)^*$  is defined as usual. The *metaproduction* relation in the metagrammar  $\rightarrow_{P_M}^*$  is the reflexive and transitive closure of  $\rightarrow_{P_M}$ .

DEFINITION 2.1.2 Define  $L_M : (M \cup V)^* \rightarrow \mathcal{P}(V^*)$  with

$$L_M(x) = \{w \in V^* \mid x \rightarrow_{P_M}^* w\}$$

It is important to keep in mind that a subscript  $_M$  indicates normal context free rewriting, using direct productions in  $P_M$ .

The language  $L_x = L_M(x) = L(G_x)$  specifies the *domain* from which the meta-variable  $x$  may assume a value. Elements from  $L_x$  are called *terminal metaproductions* of  $x$ .

A metanotion can have a number of *synonyms*. By convention, the metagrammar is supposed to be endowed with a sufficient number of synonyms  $K$  for metanotions  $N$  and *additional metarules*  $K :: N..$ . Often, synonyms are created by appending a nonnegative integer to a metanotion. It is assumed that synonyms do not occur within the right hand side of any metarule.

In two level grammars, metanotions play a dual role as variables only assuming values in their domain, and as types delimiting the values which a variable may assume. Define  $Type : M \rightarrow M$  with  $Type(K) = N$  if  $K$  is a synonym for  $N$  and  $Type(K) = K$  otherwise.  $Type$  is used to define a homomorphism  $Form : (M \cup V)^* \rightarrow (M \cup V)^*$  replacing synonyms with their ‘real’ types.

DEFINITION 2.1.3 Define

$$Form(x) = \begin{cases} Type(x) & , \text{ if } x \in M \\ x & , \text{ if } x \in V \end{cases}$$

This definition will be needed for the definition of a particular type of substitutions in the next subsection.

### Hypernotations, Hyper-rules and Consistent Substitution

Hyper-rules in  $P_H$  are called *basic hyper-rules*. The start hypernotation and hypernotations occurring within a basic hyper-rule are termed *basic hypernotations*.

From a hypernotation a *derived hypernotations* a set of can be obtained by substituting metanotions with sentential forms derived from them according to the metagrammar. The *consistent substitution rule* demands that multiple occurrences of the same metanotion are substituted with the same value. Applying such a substitution to all hyper-notions in a hyper-rule yields a *derived hyper-rule*.

DEFINITION 2.1.4 Formally, a *substitution* is a homomorphism  $\theta : (M \cup V)^* \rightarrow (M \cup V)^*$  with

$$\theta(x) = x, \text{ if } x \in V$$

Define  $Map(\theta) = \{N \in M \mid \theta(N) \neq N\}$ .

A substitution is termed *compliant*, if

$$\forall N \in M [L_M(\theta(N)) \subseteq L_M(N)]$$

and *uncompliant* otherwise. Let  $\Sigma$  be the set of all compliant substitutions. A substitution  $\theta$  is termed *faithful*, if

$$\forall N \in M [N \rightarrow_M^* Form(\theta(N))]$$

Notice that both faithful and compliant substitutions ‘obey’ the metagrammar. It is easy to proof that a faithful substitution is also compliant.

A hypernotation can be viewed as an *abstraction* of a possibly infinite set of protonotations which can be derived from it by applying compliant substitutions. Let  $P(h)$  denote the set of protonotations which can be derived from a hypernotation  $h$ .

NOTATION 2.1.2 The result of substituting all occurrences of metanotions  $N_i$  with  $y_i$  for  $i$  ranging from 1 to  $n$  is denoted as follows:

$$x[N_1 := y_1, N_2 := y_2, \dots, N_n := y_n] = \theta_n \circ \dots \circ \theta_1(x)$$

where each  $\theta_i$  is a substitution with  $\theta_i(N_i) = y_i$  and  $Map(\theta_i) = N_i$ .

PROPOSITION 2.1.1 It is undecidable whether a substitution is compliant.

PROOF Let  $S$  and  $S'$  be the start symbols of two context free grammars. If  $[S := S']$  is valid, then  $L_{S'} \subseteq L_S$ . Because  $L_{S'} \neq \emptyset$ , it follows that  $L_S \cap L_{S'} \neq \emptyset$ . This is undecidable.  $\square$

### The Derivation Process

Hyper-rules can be viewed as *rule schemata* from which a possibly infinite set of context free productions rules can be derived by substituting all metanotions in a hyper-rule with terminal metaproducts.

DEFINITION 2.1.5 A tuple  $x_0 : x_1, x_2, \dots, x_n$ . is a *production rule*, if there exists a basic hyper-rule  $y_0 : y_1, y_2, \dots, y_n$ . and a compliant substitution  $\theta$ , such that

$$\forall_{0 \leq i \leq n} [x_i \in V^* \wedge x_i = \theta(y_i)]$$

Let  $P$  be the set of all production rules.

A *notion* is a protonotion for which a production rule can be derived. A notion is termed *productive* and any other protonotion is termed a *blind alley*.

The *direct production* relation  $\rightarrow_P$  on strings in  $H^*$  is defined as usual. The *production* relation  $\rightarrow_P^*$  is the reflexive and transitive closure of  $\rightarrow_P$ .

DEFINITION 2.1.6 Define  $L : H \rightarrow \mathcal{P}(T^*)$  with

$$L(N) = \{w \in T^* \mid \exists_{v \in P(N)} [v \rightarrow_P^* w]\}$$

The *language*  $L(G) = L(S)$  is called the *strict* language generated by a 2VWG  $G$  with start hypernotation  $S$ .

## Representation

It is customary to associate a *representation* with a 2VWG  $G$ , defining one or more representations for each terminal symbol. In formal language theory, representations are known as *substitutions*, but in order to avoid confusion the term ‘representation’ is preferred. It is clear that an ill chosen representation may cause certain problems in parsing in general. This subject will be continued in §4.2.8 which deals with lexical analysis.

## Example

The following 2VWG generates the language  $\{a^n b^n c^n \mid n \geq 1\}$ .

```

NOTION :: ABC ; NOTION ABC.
ABC :: a ; b ; ... ; z.
N :: N plus one ; one.
start : N times letter a followed by
      N times letter b followed by N times letter c.
N1 plus N2 times NOTION : N1 times NOTION followed by N2 times NOTION.
one times NOTION : NOTION symbol.
NOTION1 followed by NOTION2 : NOTION1, NOTION2.

```

Representation:

```

letter a symbol → a
letter b symbol → b
letter c symbol → c

```

This example grammar illustrates three aspects which are typical to the grammar of Algol 68 in [Wij76]. Firstly, the grammar is quite ambiguous. Secondly, the metanotion NOTION ‘lifts’ whole production rules to the second level. And thirdly, the decomposition of  $N$  into  $N1$  plus  $N2$  cannot be obtained by applying direct metaproductions. Definition and application of the metagrammar are said to be *inconsistent*.

### 2.1.2 Properties of 2VWG

#### The Recognition Problem

The *recognition problem* is the following problem [Kos70]:

Given a two level grammar  $G$  and a sentence  $w$ , decide whether  $w \in L(G)$ .

In general, the recognition problem is undecidable for two level grammars. Therefore, implementations enforce or assume restrictions making the recognition problem decidable.

#### The Referencing Problem

The *referencing problem* is the following problem (a reformulation of Koster citing Baker in [Kos70]):

Given a 2VWG  $G$  and two arbitrary hypernotations  $h$  and  $h'$ , check whether there exist compliant substitutions  $\theta$  and  $\theta'$  such that  $\theta(h) = \theta'(h')$

The referencing problem is undecidable because it can be reduced to the undecidable problem of checking whether two context free languages have an empty intersection. Solvability of the referencing problem is a sufficient condition for the recognition problem to be solvable.

#### Implementations of 2VWG

Researchers have defined subclasses of 2VWG for which the referencing problem is decidable [Weg80, Deu74, Mał84]. Transparent two level grammars [Mał84], briefly treated in section 2.1.5, are an example of this approach. Other researchers have explored related executable formalisms [Kos70, Wat74, Krä78]. Conjugation grammars [Krä78] are an example of the latter approach. However, none of these formalism is capable of dealing directly with the grammar of Algol 68 as defined in [Wij76].

#### Generative Power of Two Level Grammars

The generative power of 2VWGs and two level grammars in general, depends primarily on the power of the metagrammar. From a two level grammar with a finite choice metagrammar, a weakly equivalent context free grammar can be obtained by expanding every rule containing metanotions into a finite number of context free rules. If the metagrammar is more powerful than a finite choice grammar, the generative power of a two level grammar jumps from class 2 to class 0 in the Chomsky hierarchy (see e.g. [Kos93]).

#### 2VWGs as a Descriptive Tool

According to Krämer and Schmidt [Krä78], a description serves two different purposes: definition and implementation. However, the requirements on definition and implementation may

be conflicting. This can be illustrated with examples from the definition of Algol 68 [Wij76]. First of all, this definition is not blurred with error reporting and recovery. However, these are desirable features of an implementation. Secondly, many predicates work rather inefficient in an imaginary operational meaning. Thirdly, the definition is not over-specified, which leads to ambiguity. As an example, in many cases, it is sufficient to specify only that two modes are equivalent, leaving an infinite number of possible equivalent spellings for recursive modes. Furthermore, many predicates checking context conditions may succeed in more than one way.

It may be unavoidable to have two different equivalent descriptions of the same language, one being a brief and clear definition and the other being an efficient and maintainable implementation. It is interesting to explore whether it is possible to obtain an implementation from a description using transformational methods.

### 2.1.3 Hyper-derivations and Conjugations

The human reader of a 2VWG does not grasp the grammar in terms of production rules which are applied to notions. Instead, he applies hyper-rules directly to hypernotions, hereby abstracting from the infinite set of production rules [Krä78]. This idea leads to a new type of derivation, called *hyper-derivation*. The possibility of constructing hyper-derivations has been explored by Krämer and Schmidt [Krä78] and Małuszyńky [Mał84].

In a hyper-derivation, a sentential form

$$\alpha, h, \gamma$$

may be rewritten to

$$\theta(\alpha), \theta'(\beta), \theta(\gamma)$$

if there exists a basic hyper-rule  $(h', \beta)$  and a pair of compliant substitutions  $\theta$  and  $\theta'$  such that  $\theta(h) = \theta'(h')$ . Remark that variables in  $\alpha\gamma$  are not intended to get bound to variables occurring in  $\beta$  but not in  $h'$ . This can be prevented by appropriate renaming.

Two hypernotions are called *conjoint*, if they have a commonly derived protonotion.

DEFINITION 2.1.7 Define *Conjoint* :  $H \times H \rightarrow \mathbb{B}$  with

$$\text{Conjoint}(h, h') \Leftrightarrow \exists_{(\theta, \theta') \in \Sigma^2} [\theta(h) = \theta'(h')]$$

If two hypernotions are not conjoint, then they are said to be disjoint.

A pair  $(\theta, \theta') \in \Sigma^2$  is a *conjugation* of a pair of hypernotions  $(h, h')$ , if  $\theta(h) = \theta'(h')$ .

A set  $C$  of conjugations of a pair of hypernotions  $(h, h')$  is *complete*, if for every conjugation  $(\theta, \theta')$  of  $(h, h')$

$$\exists_{\rho \in \Sigma, (\sigma, \tau) \in C} [(\theta, \theta') = (\rho \circ \sigma, \rho \circ \tau)]$$

A complete set of conjugations of hypernotions  $h$  and  $h'$  specifies all common protonotions of  $h$  and  $h'$ . The notion of completeness of a set of conjugations is needed in the next chapter for defining some transformations on two level grammars.

A hypernotation  $g$  is a *junction* of hypernotions  $h$  and  $h'$ , if there exists a conjugation of  $h$  and  $h'$ . A junction specifies a set of common protonotions which can be derived from both  $h$  and  $h'$ . Clearly, every common protonotion can be derived from one or more of the junctions which are obtained by applying a conjugation from a complete set of conjugations.

EXAMPLE 2.1.1 The hypernotions `where THING1 and THING2` and `where (PROPSETY) is (LAB LABSETY)` and `SOID balances SOID1 and SOID2`, defined in [Wij76], have a finite and complete set of conjugations

$$\{([\text{THING1} := (\text{PROPSETY}) \text{ is } (\text{LAB LABSETY}), \\ \text{THING2} := \text{SOID balances SOID1 and SOID2}], []), \\ ([\text{THING1} := (\text{PROPSETY}) \text{ is } (\text{LAB LABSETY}) \text{ and SOID balances SOID1,} \\ \text{THING2} := \text{SOID2}], [])\}$$

It is however possible that a finite and complete set of conjugations does not exist.

EXAMPLE 2.1.2 Consider the following metagrammar.

```
TWO :: ii ; ii TWO.
THREE :: iii ; iii THREE.
```

It is clear that the pair of hypernotions (TWO, THREE) does not have a finite set of conjugations. One might conceive of one, something like  $\{([\text{TWO} := \text{SIX}, \text{THREE} := \text{SIX}]])\}$ , but the metagrammar given above does not contain a metarule `SIX :: iiii ; iiii SIX`.

PROPOSITION 2.1.2 It is undecidable whether a set of conjugations is complete.

PROOF Let  $S$  and  $S'$  be the start symbols of two context free grammars. If  $\emptyset$  is complete for  $(S, S')$ , then  $L(S) \cap L(S') = \emptyset$ . This is undecidable.  $\square$

## 2.1.4 Conjugation Grammars

In this paragraph, only a brief impression of *conjugation grammars* [Krä78] is given. A conjugation grammar is a 2VWG which is endowed with so called *conjugation rules*, explicitly indicating junctions and conjugations. A conjugation rule specifies a sequence of faithful substitutions, indicating how conjoint hypernotions  $h$  and  $h'$  can be transformed into a junction of  $h$  and  $h'$ . In constructing hyper-derivations, conjugation rules are not applied directly to the basic hyper-rules. Instead, a sequence of substitutions, called the *current meta-derivation* is kept track of. The current meta-derivation maps instances of metanotions to metaproduction trees.

All substitutions in a conjugation rule should be in accordance with the metagrammar. Otherwise, the referencing problem is encountered again when deciding whether some conjugation rule does apply in constructing a derivation.

The reader may notice the similarity between runtime application of a conjugation rule and ‘specification time’ application of the unfold transformation defined in the next chapter.

In a conjugation grammar, all conjoint hypernotations must have a finite and complete set of conjugations. Otherwise, the number of conjugation rules could not be finite. Furthermore, application and definition of the metagrammar must be consistent. This requires the use of predicates for checking context conditions. Krämer and Schmidt consider ‘hiding’ context conditions within junctions a bad practice, leading to definitions which are not very transparent.

### 2.1.5 Transparent Two Level Grammars

Constructing a hyper-derivation requires the referencing problem being solvable. Małuszyński defines a class of two level grammars, the *transparent two level grammars*, for which the referencing problem is decidable [Mał84].

DEFINITION 2.1.8 A 2VWG  $G$  is called *transparent*, if there exists a *main metanotion*  $K$  such that  $G_K$  is unambiguous and  $K \rightarrow_{P_M}^* \text{Form}(h)$ , for every basic hypernotation  $h$ .

Notice that transparency implies that definition and application of the metagrammar are consistent. The alternatives for the main metanotion can be seen as type declarations for the hypernotations of the grammar.

In [Mał84], a *hyper-replacement* is defined as a faithful substitution. Assume that two conjoint hypernotations  $h$  and  $h'$  do not contain common variables. Otherwise, rename these common variables. Then there exists a single hyper-replacement  $\theta$ , such that  $\theta(h) = \theta(h')$ .  $\theta$  is called an *unifier* of  $h$  and  $h'$ . An unifier  $\theta_1$  is called *more general* than  $\theta_2$ , if there exists a hyper-replacement  $\theta_3$ , such that  $\theta_2 = \theta_3 \circ \theta_1$ . An unifier is a *most general* unifier, if there does not exist a more general unifier. Notice the similarity between most general unifiers and complete sets of conjugations.

THEOREM 2.1.1 (Małuszyński, [Mał84]) For every pair of hypernotations in a transparent two level grammar

1. it is decidable whether two hypernotations are unifiable (conjoint), and
2. if they are unifiable, then there exists a most general unifier of these hypernotations.

The theorem is proven by setting up a correspondence between hypernotations and terms, hereby reducing grammatical unification to normal term unification [Mał84, Mał82].

Notice that in a transparent grammar there exists a complete set of conjugations containing exactly one element for all conjoint hypernotations. Furthermore, this set can be computed. This is an important observation, because in a transparent grammar applicability of many of the transformations defined in the next chapter is decidable.

CONJECTURE 2.1.1 A transparent grammar  $TG$  can be transformed automatically into a conjugation grammar  $CG$ . For all finitely many basic hypernotations  $h$  and  $h'$  of  $TG$  it can be checked whether they are conjoint. If they are, a unifier  $\theta$  is computed and a conjugation rule ' $\theta(h) = \theta(h')$ ' is added.

## 2.2 Extended Affix Grammars

Extended Affix Grammars (EAG), devised by Watt [Wat74], are an extended form of Koster's Affix Grammars [Kos70]. The main extension is that in EAG, affix expressions are allowed to occur on parameter positions. In this way, predicates defined outside the formalism become superfluous and a close resemblance to 2VWG is obtained.

In §2.2.1 the EAG formalism is described. The close relation between the EAG and 2VWG formalisms is pointed out in §2.2.2. Finally, in §2.2.3 an EAG-compiler developed at Nijmegen University is described. This implementation imposes very few restrictions, bringing a close approximation to 2VWG at hand.

### 2.2.1 Formal Definition of EAG

DEFINITION 2.2.1 An extended affix grammar is an 8-tuple

$$G = (V_N, V_T, A_N, A_T, P_A, P_H, C, S)$$

with the following components:

$V_N$  is a finite alphabet of *nonterminal symbols*.

$V_T$  is a finite alphabet of *terminal symbols*, such that  $V_N \cap V_T = \emptyset$ .

$A_N$  is a finite set of *affix nonterminal symbols* (metanotions or affixes).

$A_T$  is a finite set of *affix terminal symbols*, such that  $A_N \cap A_T = \emptyset$ .

$P_A$  is a finite set of *affix production rules* (or metarules), with  $P_A \subset A_N \times E$ .  
 $E = (A_N \cup A_T)^*$ . The elements of  $E$  are called *affix expressions*.

$P_H$  is a finite set of *hyper-rules*, with  $P_H \subset H \times (H \cup V_T)^*$ ,  
 where  $H = V_N \times E^*$ . The elements of  $H$  are called *hypernotations*.

$C = (Pos, Flow)$  is the *control*, with

$Pos : V_N \rightarrow \mathbb{N}$ ,  $Pos(x)$  is the number of *affix positions* of  $x$ .

$Flow : V_N \rightarrow \mathbb{N} \rightarrow \{\iota, \delta\}$

If  $Flow(x)(i) = \iota$  then the  $i$ -th affix position of  $x$  is called *inherited*. Otherwise, it is called *derived*.

$S$  is the *start hypernotation*, with  $S \in H$ .

Again, a rule is composed of a *left hand side*, followed by a *right hand side*. The right hand side of a rule is called an *alternative* for the nonterminal on the left hand side. An alternative consists of a possibly empty sequence of *members*. Nonterminals with empty alternatives only, are called *predicates*.

The control does not figure in the formal interpretation of EAG, but is merely used to supply information concerning affix flow.

### Notation

NOTATION 2.2.1 An affix expression  $(a_1, \dots, a_n)$  is written as

$$a_1 + \dots + a_n$$

The ‘+’ denotes concatenation (juxtaposition) at the meta-level. A metarule  $(a, (a_1, \dots, a_n))$  is therefore written as

$$a :: a_1 + \dots + a_n.$$

A hypernotation  $(x, (e_1, \dots, e_n))$  is denoted by

$$x(e_1, \dots, e_n)$$

In order to obtain a closer resemblance to 2VWG, one may alternatively write the infix notation

$$x_0(\alpha_1)x_1 \dots (\alpha_n)x_n$$

provided that  $x = x_0 \dots x_n$  and  $(e_1, \dots, e_n) = (\alpha_1, \dots, \alpha_n)$ .

A hyper-rule  $(h, (h_1, \dots, h_n))$  is written as

$$h : h_1, \dots, h_n.$$

When both  $a :: e_1.$  and  $a :: e_2.$  are metarules,

$$a :: e_1; e_2.$$

is written, and when both  $h : \alpha.$  and  $h : \beta.$  are hyper-rules, one may write

$$h : \alpha; \beta.$$

An affix expression on an inherited (derived) position may be preceded (resp. followed) by a *flow symbol* (>). Only in some cases indication of flow is obligatory. This will be explained in section 2.2.3. The denotation of a terminal symbols is enclosed in double quotes (“”).

For regular right hand sides the regular set like notation from [Mei86] is a convenient shorthand. As an example, the *semi-terminal*  $\{abc\}^+ (x)$  recognizes any nonempty sequence of as, bs and cs, and ‘assigns’ the recognized string to x. Although this is merely syntactic sugar, an implementation may take advantage of it.

### Metagrammar

Affix nonterminals may or may not occur on the left hand side of some affix production rule. If an affix nonterminal  $x$  does, then it is said to be *meta-defined*, and the context free grammar  $G_x = (A_N, A_T, P_A, x)$  is called the *metagrammar* of  $x$ .

DEFINITION 2.2.2 The *affix language* or *domain*  $L_x = L_M(x)$  of an affix  $x$  is defined as

$$L_x = \begin{cases} \{x\} & \text{if } x \in A_T \\ L(G_x) & \text{if } x \text{ is meta-defined} \\ A_T^* & \text{else} \end{cases}$$

### Hypernotations, Hyper-rules and Consistent Substitution

A hypernotation is composed of a nonterminal symbol, called its *head*, followed by a *display*. A display is a possibly empty sequence of affix expressions. Affix expressions on their turn are composed of so called *affix terms*.

The hyper-rules in  $P_H$  are called *basic hyper-rules*. A *basic hypernotation* is the start hypernotation, or a hypernotation within some basic hyper-rule.

The basic hyper-rules of an EAG may be viewed as rule schemata, from which *production rules* can be obtained by replacing affix nonterminals with elements from their affix languages. The *consistent substitution rule* ensures that in a hyper-rule multiple occurrences of the same affix nonterminal are substituted with the same terminal production.

DEFINITION 2.2.3 An *affix assignment* is a homomorphism  $\theta : (A_N \cup A_T)^* \rightarrow A_T^*$  with

$$\theta(x) \in L_x \text{ for all } x \in A_N \cup A_T$$

Let  $\theta : H \rightarrow H$  with

$$\theta(x, (x_1, \dots, x_n)) = (x, (\theta(x_1), \dots, \theta(x_n)))$$

be the extension of an affix assignment to hypernotations.

Notice that an affix assignment is a special kind of compliant substitution. Notions like ‘conjugation’ and ‘substitution’, as defined in sections 2.1.3 and 2.1.1 for 2VWG, have obvious counterparts in EAG. Therefore, such notions are considered to be defined for EAG as well.

A *protonotation* is any element of  $V_N \times (A_T^*)^*$ . Let  $P(x)$  denote the set of protonotations which can be derived from a hypernotation  $x$  by applying affix assignments.

### Derivation Process

DEFINITION 2.2.4 A tuple  $(x_0, (x_1, \dots, x_n))$  is a *production rule*, if there exist a hyper-rule  $(y_0, (y_1, \dots, y_n))$  and an affix assignment  $\theta$  such that

$$\forall_{0 \leq i \leq n} [x_i \in V_N \times (A_T^*)^* \wedge x_i = \theta(y_i)]$$

Let  $P$  be the set of all production rules.

The *direct production* relation  $\rightarrow_P$  on strings in  $(H \cup V_T)^*$  is defined as usual. The *production* relation  $\rightarrow_P^*$  is the reflexive and transitive closure of  $\rightarrow_P$ .

DEFINITION 2.2.5 Define  $L : H \rightarrow \mathcal{P}(V_T^*)$  with

$$L(x) = \{w \in V_T^* \mid \exists_{v \in P(x)} [v \rightarrow_P^* w]\}$$

The *language*  $L(G) = L(S)$  is the language generated by an EAG  $G$  with start hypernotation  $S$ .

### Example

The following Extended Affix Grammar generates the language  $\{a^n b^n c^n \mid n \geq 1\}$

```

N :: zero ; succ + N.
start : abc (succ + N).
abc (N) : a (N), b (N), c (N).
a (zero) : .
a (succ + N) : "a", a (N).
b (zero) : .
b (succ + N) : "b", b (N).
c (zero) : .
c (succ + N) : "c", c (N).

```

### 2.2.2 Properties of EAG

An EAG can be compiled automatically into an exact recognizer for the language generated by the EAG. Therefore, one may identify an EAG with a recognition algorithm. This algorithm is guaranteed to terminate if certain *well-formedness* conditions, as described in §2.2.3, are met. Meijer [Mei86] defines the semantics of an eag as a translator function, which maps an input sentence onto a set of translations. A translation consists of the (evaluated) affixes with which the root node of a parse tree may be endowed. An implementation usually writes these affixes on some output device. This enables one to write compilers in EAG.

There is no principal difference between EAG and 2VWG, in the sense that any 2VWG can be transcribed into EAG (and of course vice versa). The transcriptions defined in §4.1 yield

an EAG with the same generative meaning as any 2VWG. However, it may very well be the case that the resulting EAG is very inefficient. Furthermore, the resulting EAG may not have an operational meaning. This is always the case if the 2VWG specifies an infinite number of parsing for some sentence, or if during parsing an infinite number of blind alleys can be derived. In the next chapter, transformations are defined for transforming a two level grammar into a more operational form.

### 2.2.3 Implementation

At Nijmegen University, Marc Seutter has implemented a compiler-compiler for EAG, called `eag-compile` [Seu93]. The generated compilers are either based on a recursive backup parser [Kos74], or on a left-corner backup parser [Bra92]. Furthermore, `eag-compile` can generate a syntax directed editor from an EAG [Bra92, Seu93].

#### Head Grammar

The *head grammar* or *underlying context free grammar* of an EAG is obtained from the hyperrules by removing the displays from all hypernotations. `eag-compile` generates all possible parses according to the *head grammar*, while decorating the resulting parse trees in all possible ways. Non-predicate rules of the head grammar are required to be free of left-recursion (free of hidden left-recursion), if a top-down parser (resp. left-corner parser) is to be generated.

#### Affix Flow

When the values of all affix nonterminals in an affix expression become known, the resulting string is propagated through an affix graph. During propagation the string is matched against affix expressions, by splitting the string in all possible ways and performing context free meta parses. If meta parsing succeeds, the values of new affix nonterminals become known. `eag-compile` requires the meta rules to be free of left recursion.

Affix evaluation is interleaved with head grammar parsing, in order to reject an successful parse as soon as possible. When the values of all affix nonterminals in an affix expression become known, the resulting string is immediately propagated as far as possible. This evaluation strategy is known as *prompt evaluation*. Prompt evaluation enables three features. The first one is *affix directed parsing*, i.e. letting affix values steer the parsing process. `eag-compile` does not use affix directed parsing. Instead, affix evaluation takes place after context free recognition of a nonterminal. Secondly, affix flow is not restricted to be from left to right, but is left completely free. The direction of flow is established *dynamically*, and may even differ for different instances of the same rule. This feature is known as *free affix flow*. Thirdly, affix evaluation is not restricted to a fixed number of passes.

#### Predicates

The only place where flow annotations are taken into account is within predicates. The evaluation of a predicate remains *delayed*, until all its inherited affix positions become known.

If this were not the case, recursive predicates would never terminate. Positions which must be known in order to guarantee termination of a predicate are called *critical*, and must be preceded by a flow symbol.

For efficiency reasons, `eag-compile` does support some built-in predicates [Seu93]. However, these predicates can be simulated in the formalism itself. Therefore, the formal definition of EAG is not affected by this extension.

### Tuples, Numerals and Typing

In practice, a specification writer will often want to manipulate numerical values and complex hierarchical data structures. Although they are redundant, since they can be simulated by string-valued affixes, the natural numbers and tuples are introduced for reasons of clarity and efficiency. They have a simple isomorphic embedding in flat string domains [Mei86]. Again, this extension does not affect the formal definition of EAG.

Tuples and numerals may occur in affix expressions and on the right hand sides of affix production rules. A numeral is denoted by its decimal representation. Tuples are denoted using the *composition operator* ('\*'). A decimal number denoting  $n$  means 'i <sup>$n$</sup> ', and a  $n$ -tuple ' $\alpha_1 * \dots * \alpha_n$ ' means '(  $\alpha_1$  ±  $\dots$  ±  $\alpha_n$  )'. The special nonterminals 'i', '(', '±' and ')' are hidden from the user.

A typing system for EAG is treated in [Bra92]. In short, every affix expression is assigned one of the basic types STRING, INT, or  $n$ -TUPLE. The special affix terminal `nil` denotes the empty tuple, and the special affix nonterminal `tuple` recognizes any tuple. Now an eag is considered to be well-typed if every affix nonterminal and every affix position of each nonterminal has type STRING, INT or union of  $n$ -TUPLES. An EAG which is not well-typed is rejected by `eag-compile`.

### Well-formedness and Termination

For a given input sentence, the parser generated from an EAG will try to find all possible parses. Clearly, if the parser gets into a loop or if the number of parses is infinite, nontermination is the result. In the original article on Affix Grammars [Kos70], Koster formulates some *well-formedness conditions*, which impose restrictions on the affix flow and on the head grammar. In particular, they guarantee termination. In his dissertation [Mei86], Meijer formulates well-formedness conditions for EAG which presuppose *static* flow. For dynamic flow v.d. Brand [Bra92] suggests that 'the specification writer should be very careful in using meta-defined affix-nonterminals'.

Assume that *Flow* is defined for every affix position of every nonterminal. The inherited positions on the left hand side (right hand side), and the derived positions on the right hand side (left hand side) of a hyper-rule, are called *defining* (resp. *applying*) positions. An affix nonterminal is called *applying only*, if it occurs within applying positions only.

Meijer shows that for a well-formed EAG there are finitely many 'admissible decorated parse trees', which can all be found. A parse tree is *admissible* if its associated *dependency graph* [Mei86] does not contain cycles. An EAG is well-formed if:

1. head grammar parsing and meta parsing always terminate,
2. termination of predicates is guaranteed, and
3. the affix language of every affix nonterminal which is applying only in some hyper-rule, is finite.

In general, well-formedness is not decidable, although fairly liberal sufficient conditions can be found.

## 2.3 CDL3

CDL3 is a programming language based on Affix Grammars, in which important software engineering considerations have been incorporated. The affix rules are used to define tree types. As a balance between efficiency and generality, execution is deterministic, while affix evaluation takes place in two passes. Structured programming is supported by a module structure. Static semantic checks ensure that a CDL3 program can be executed deterministically in two passes and help the programmer prevent many errors.

CDL3 is not a specification tool. It is a language for denoting large programs whose structure can be characterized as *syntax directed* [Kos94] at a high level of abstraction. The reader is referred to [Kos94] for an introduction of CDL3 as a programming language and a software engineering tool. Here, we will summarize the features of CDL3. In §2.3.1 the CDL3 formalism is described. In §2.3.2 the relationship between EAG and CDL3 is examined.

CDL3 rides the borderline between grammars and programs. Therefore, we will use both notions of *program* and *grammar* for referring to a CDL3 program (or grammar).

### 2.3.1 Formal Definition

The following definition, with slight adaptations, is taken from [Kos70].

DEFINITION 2.3.1 An affix grammar is a 9-tuple

$$G = (V_N, V_T, V_P, A_N, A_T, P_A, P_H, C, S)$$

with the following components:

$V_N$  is a finite alphabet of *nonterminal symbols*.

$V_T$  is a finite alphabet of *terminal symbols*.

$V_P$  is a finite set of *primitive predicate symbols*.

$V_N$ ,  $V_T$  and  $V_P$  are mutually disjoint and do not contain  $\omega$ , the *forbidden symbol*.

$A_N$  is a finite set of *affix nonterminal symbols* (metanotions or affixes).

$A_T$  is a finite set of *affix terminal symbols*, such that  $A_N \cap A_T = \emptyset$ .

$P_A$  is a finite set of *affix production rules* (or metarules), with  $P_A \subset A_N \times E$ , where  $E = (A_N \cup A_T)^*$ . The elements of  $E$  are called *affix expressions*.

The context free grammar  $G_x = (A_N, A_T, P_A, x)$  is called the *metagrammar* of  $x$ . Define  $L_x = L_M(x) = L(G_x)$ , the *domain* or *affix language* of  $x$ .

$P_H$  is a finite set of *basic hyper-rules*, with  $P_H \subset H_l \times (H_r \cup V_T)^*$ , where  $H_l = V_N \times E^*$  and  $H_r = (V_N \cup V_P) \times E^*$ .

The elements of  $H = H_l \cup H_r$  are called *hypermotions*.

$C = (Pos, Flow, Type, F)$  is the *control*, with

$Pos : (V_N \cup V_P) \rightarrow \mathbb{N}$ ,

$Pos(x)$  is the number of *affix positions* of  $x$ .

$Flow : (V_N \cup V_P) \rightarrow \mathbb{N} \rightarrow \{\iota, \delta\}$

If  $Flow(x, i) = \iota$  ( $\delta$ ), then the  $i$ -th affix position of  $x$  is called *inherited*, (resp. *derived*).

$Type : (V_N \cup V_P) \times \mathbb{N} \rightarrow A_N$ ,

$Type(x, i)$  is the *domain* or *type* of the  $i$ -th affix position of  $x$ .

$F : V_P \ni x \rightarrow (L_{Type(x,1)} \times \dots \times L_{Type(x,Pos(x))} \rightarrow \{\varepsilon, \omega\})$

For every  $x \in V_P$ ,  $F_x$  is a total recursive function called the function *associated* with  $x$ .

$S$  is the *start hypermotion*, with  $S \in H$ .

The notions of *left hand side*, *right hand side*, *alternative*, *member* and *predicate* are defined as usual. In CDL terminology, the notion ‘predicate’ also refers to a special class of algorithms (see below). In order to avoid confusion, we will use PRED or PREDICATE for referring to this latter meaning.

DEFINITION 2.3.2 A CDL3 program is a tuple

$$(G, >, Class, Pass)$$

where

$G$  is an affix grammar.

$>$  is a partial ordering on  $P_H$ , with  $x > y$  or  $y > x$  if the left hand sides of  $x$  and  $y$  are composed of the same nonterminal.  $>$  defines the order in which rules are tried during parsing. Therefore, CDL3 grammar is called *ordered*.

$Class$  with  $Class : (V_N \cup V_P) \rightarrow \{\text{TEST, PRED, ACTION, FUNCTION}\}$  is an *algorithm classification*.

$Pass$  is an overloaded function with  $Pass : V_N \times \mathbb{N} \rightarrow \{1, 2\}$  indicating whether an affix position contains a *first pass* or a *second pass* parameter, and  $Pass : P_H \times \mathbb{N} \rightarrow \{1, 2\}$  indicating whether a member is to be evaluated in the first or in the second pass. Each second pass member must be a predicate.

The control, *Class* and *Pass* do not play a role in the generative interpretation of a CDL3 grammar. The control and *Pass* are used to supply information concerning affix flow and enable a compiler to verify that each parse tree can be built decorated in at most two passes. *Class* is used for verifying the programmers intentions concerning the effect and possible failure of algorithms.

### Notation

NOTATION 2.3.1 An affix expression  $(a_1, \dots, a_n)$  is written as

$$a_1 \dots a_n$$

An affix rule  $(a, (a_1, \dots, a_n))$  is written as

$$a :: a_1 \dots a_n.$$

In denoting hypernotations, a slash ('/') is used to separate first pass affix positions from second pass affix positions. Therefore, a hypernotation  $(x, (e_1, \dots, e_m, e_{m+1}, \dots, e_n))$  with  $Pass(x, i) = 1$  for  $1 \leq i \leq m$  and  $Pass(x, i) = 2$  for  $m + 1 \leq i \leq n$  is written as

$$x(e_1, \dots, e_m / e_{m+1}, \dots, e_n)$$

The infix notation of EAG may be used alternatively. An affix expression occurring on an inherited (derived) position must be preceded (resp. followed) by a flow symbol ('>').

A slash is also used to separate first pass members from second pass members. Therefore, a hyper-rule  $r = (h, (h_1, \dots, h_m, h_{m+1}, \dots, h_n))$  with  $Pass(r, i) = 1$  for  $1 \leq i \leq m$  and  $Pass(r, i) = 2$  for  $m + 1 \leq i \leq n$  is written as

$$\text{CLASS } h : h_1, \dots, h_m / h_{m+1}, \dots, h_n.$$

where CLASS is replaced with *Class*(*h*).

It is required that all first pass elements, i.e. affix positions or members, precede the second pass elements. When no second pass elements are present, no slash is written.

The members  $h_m$  and  $h_n$  ( $m$  and  $n$  as above) may be an *enclosed group*, i.e. a sequence of alternatives enclosed in parentheses, standing for implicit left-factorization (see §3.3.5 and [Kos94]) of the last members in a pass.

When both  $a :: e_1.$  and  $a :: e_2.$

$$a :: e_1; e_2.$$

is written, and when both  $h : \alpha.$  and  $h : \beta.$

$$h : \alpha; \beta.$$

is written. However, the ordering of hyper-rules is reflected by their textual ordering. Hence,  $h : \alpha; \beta.$  implies  $h : \alpha. > h : \beta..$

## Metagrammar

Again affix nonterminals play the dual role of types and variables. Ground instances of variables are implemented as tree structures (terms) of the type indicated by the variable. In order to prevent backtracking, each affix expression occurring on some affix position must represent an unique tree. Therefore, the following restrictions are imposed:

1. the metagrammar must not contain empty productions, and
2. for each affix expression  $h$  occurring on some affix position with domain  $N$  there must exist exactly one left-most derivation such that  $N \rightarrow_{P_M}^* Form(h)$ .

The first restriction enables the compiler to check the second one.

## Guards

Trees are built, decomposed and tested for equality by means of primitive predicates called *guards*. There are four types of guards:

[LAYER = LAYER1]	<i>equality</i> test of ground affix values
[LAYER -> new DECSETY LABSETY]	<i>split</i> , combined with conformity test
[new DECSETY LABSETY -> LAYER]	<i>join</i>
[LAYER -> LAYER1]	<i>assignment</i> of LAYER to LAYER1

With each type of guard, a function testing its arguments for equality is associated. Equal guards have two inherited positions, while splits, joins and assignments have one inherited followed by one derived position.

Not every affix expression is allowed in a guard. Equal guards and assignments require both affixes involved to be synonyms. Split guards and join guards contain an affix and a *conforming* affix expression. This means that only one affix rule at a time is concerned by a guard. When a sequence of affix rules is concerned, we speak of *sub-guards* embedded in a guard. For example, the split guard [LAYER -> new DECS LABSETY] is considered to stand for [LAYER -> new DECSETY LABSETY], [DECSETY -> DECS]. This expansion is always possible, because each affix expression can be obtained by a unique meta-derivation, Embedded guards can be made explicit (or vice versa) by a transformation called *predicate introduction* (resp. *predicate removal*), defined in §3.3.4.

Affix expressions occurring on positions of nonterminal symbols or primitive predicate symbols which are not a guard, stand for *implicit guards*. Implicit guards can also be made explicit (or vice versa) by predicate introduction (resp. predicate removal).

## Hypernotations, Hyper-Rules and Consistent Substitution

As usual, a nonterminal symbol is composed of a *head* followed by a *display* containing *affix expressions*.

The notions of *basic hypernotation*, *basic hyper-rule*, *consistent substitution*, *affix assignment*, etcetera, are defined in the obvious way. Remember that  $P(h)$  is the set of *protonotions* derived from a hypernotation  $h$ .

As a notational extension for obtaining a compact and practical notation for programming, an inherited and a derived affix of the same type and pass may be packed into one *transient parameter* position (see [Kos94] and example 2.3.1). Transient parameters seem to violate the consistent substitution rule, but they are considered syntactic sugar.

### Derivation Process

Before describing the semantics of an ordered grammar, we have to make an extension to the grammar. Let  $\#$  be a new terminal symbol for marking the end of a sentence, and let  $S^\# : S, \#$ . be a new hyper-rule.

The set of *production rules*  $P$  is defined as usual with the following extension. Let  $(N, \alpha)$  be a member of a production rule originally in  $P$ . If  $N$  is a primitive predicate with  $F_N(\alpha) = \varepsilon$ , then a rule  $(N, \alpha) : \varepsilon$ . is added to  $P$ . The *direct production relation*  $\rightarrow_P$  and the *production relation*  $\rightarrow_P^*$  are defined as usual.

The ordering  $>$  is extended to production rules as follows. Let  $p, p'$  be basic hyper-rules. For all  $\theta(p), \theta'(p') \in P$ , define  $\theta(p) > \theta'(p') \Leftrightarrow p > p'$ .

DEFINITION 2.3.3 Define  $Dir : P \rightarrow \mathcal{P}(V_T^*)$  with

$$Dir(h : \alpha.) = \{t \in V_T \mid \exists_{w \in V_T^*, \beta, \gamma \in H^*} [S^\# \rightarrow_P^* wh\beta \rightarrow_P w\alpha\beta \rightarrow_P^* wt\gamma]\}$$

$Dir(h : \alpha.)$  is called the *director set* of  $h : \alpha.$ .

We will adapt the definition of *LL(1)*-ness in [Wil92] to specify the language recognized by a CDL3 grammar. In an *LL(1)* grammar, all alternatives for a hypernotation have disjunct director sets, enabling top down parsers to make a unique choice for an alternative. In a CDL3 grammar, the *first* applicable alternative is chosen. For a *LL(1)* grammar this will always be the right alternative, if any.

DEFINITION 2.3.4 Define  $\vdash : (H^* \times V_T^*)^2 \rightarrow \mathbb{B}$  with

$$\begin{aligned} \langle h\beta, vw \rangle &\vdash \langle \alpha\beta, vw \rangle && \text{if } h : \alpha. \in P \wedge (h : \gamma. > h : \alpha. \Rightarrow v \notin Dir(h : \gamma.)) \\ \langle v\alpha, vw \rangle &\vdash \langle \alpha, w \rangle \end{aligned}$$

$\vdash^*$  is the reflexive and transitive closure of  $\vdash$ .

DEFINITION 2.3.5 Define  $L : H \rightarrow \mathcal{P}(V_T^*)$  with

$$L(N) = \{w \in V_T^* \mid \exists_{v \in P(N)} \langle v\#, w\# \rangle \vdash^* \langle \varepsilon, \varepsilon \rangle\}$$

The set  $L(G) = L(S)$  is the *language* generated by a CDL3 grammar  $G$ . Notice that a CDL3 grammar is *deterministic*, i.e. in any derivation step there is at most one alternative applicable.

### Affix Flow and Affix Evaluation

Again, the inherited affix positions on the left hand side (right hand side) of a rule and the derived affix positions on the right hand side (left hand side) of a rule, are called *defining* (resp. *applying*) positions. An affix has a *defining occurrence* if it appears on a defining position. Similarly, an affix has an *applying occurrence* if it appears on a applying position. When an affix has more than one defining occurrence, no implicit equality test is performed, as opposed to e.g. EAG. In CDL3, each equality test is to be specified explicitly with an equal guard.<sup>1</sup>

For every affix it is required that each applying occurrence is preceded by an defining occurrence. Furthermore, each applying occurrence on a first pass position is to be preceded by a defining occurrence which is also on a first pass position. Consequently, within each pass affix flow is restricted to be from left to right. Furthermore, above conditions are sufficient to guarantee that each affix can be evaluated by visiting each node of a parse tree at most two times.

When building a parse tree, members are evaluated from left to right. The evaluation of second pass (predicate) members remains delayed, until all first pass members have been evaluated. If a first pass member of some alternative fails, the second pass of the alternative is not executed. When a second pass member  $h$  has a second pass as well, evaluation of the second pass members of  $h$  is also delayed until the first pass members of  $h$  are evaluated successfully (in the second pass). This mechanism is applied recursively to all second pass members. Still, each node in a parse tree is visited at most two times.

### Static Semantic Checks

With an algorithm, two independent attributes are associated:

1. whether it may fail, or always succeeds, and
2. whether it has an effect.

By an *effect* is meant a change to the global environment of the algorithm.

The programmer indicates her intentions by explicitly indicating the effect and possibility of failure of each algorithm. There are four classes of algorithms:

Class	may fail	has an effect
TEST	yes	no
PRED	yes	yes
ACTION	no	yes
FUNCTION	no	no

The philosophy of effects and their control is described in [Feu78] and [Kos94].

---

<sup>1</sup>Each defining occurrence of an affix following another defining occurrence is to be understood as an occurrence of another synonymous affix, a relaxation of the consistent substitution rule to avoid the necessity of inventing synonyms.

In CDL3 notation, an (abstract) algorithm is a sequence of alternatives for a nonterminal. A nonterminal has an effect if one of its alternatives has an effect, and an alternative has an effect if it contains a member which has an effect. Similarly, a nonterminal may fail if all of its alternatives may fail, and an alternative may fail if it contains a member which may fail. Furthermore, a terminal symbol has an effect and may fail. Failure and effect of primitive predicates is determined by the implementation. There are two special (primitive) predicates '+' and '-', indicating success resp. failure. Both predicates do not have an effect.

A *defect* occurs when a member which has an effect is followed by a member which may fail, or when a second pass member may fail. In order to avoid backtracking, defects are forbidden. Absence of defects is verified by the compiler (see [Kos94]). Often, defects can be 'repaired' by adding error alternatives (see §5.1.3). This amounts to an extension of the language described with all erroneous input sentences, for which an error is reported.

### Modules and Global Variables

In CDL3, modules define abstract data types with exactly one instance. Inside a module, global variables may be introduced. A module consists of an affix expression containing the global variables of the module and the definitions of the procedures operating on this expression. The global variables of a module are considered implicit transient parameters to all procedures exported by the module and recursively to the procedures calling those procedures.

**Preludes, Interludes and Postludes** Modules may have special procedures, the *preludes*, *interludes* and *postludes*, separating the initializations and finalizations of a program from the main algorithm. These procedures are called respectively at the beginning, before the second pass and at the end of the execution of a program.

### Example

EXAMPLE 2.3.1 The following syntax is a fragment of a program parsing arithmetic expressions. An abstract syntax tree representing the expression is built.

```
(1) MINPRIO, MAXPRIO = PRIO.
    EXPR :: EXPR OP EXPR ;
           OP EXPR ;
           CONST ;
           erroneous.

    PRED (>PRIO) expression (EXPR) :
        monadic expression (EXPR),
        maxprio (MAXPRIO),
        (PRIO, MAXPRIO) expression tail (EXPR).

(2) ACTION should be (>PRIO) expression (EXPR) :
    (PRIO) expression (EXPR) ;
    syntax error ("Expression expected"),
    [erroneous -> EXPR].
```

```

PRED monadic expression (EXPR>) :
  monadic operator (OP),
    should be monadic expression (EXPR1),
    [OP EXPR1 -> EXPR] ;
  secondary (EXPR).
ACTION should be monadic expression (EXPR>) :
  monadic expression (EXPR) ;
  syntax error ("Monadic expression expected"),
  [erroneous -> EXPR].
(3) ACTION (>MINPRIO, >MAXPRIO) expression tail (>EXPR>) :
  (MINPRIO, MAXPRIO) operator (OP, PRIO),
    should be (PRIO + 1) expression (EXPR1),
    [EXPR OP EXPR1 -> EXPR],
  (MINPRIO, PRIO) expressions tail (EXPR) ;
+.
PRED (>MINPRIO, >MAXPRIO) operator (OP>, PRIO>) :
  is ahead (OP),
  where (OP) has priority (PRIO),
  lesseq (MINPRIO, PRIO),
  lesseq (PRIO, MAXPRIO),
  operator (OP).

```

The example exemplifies some more features of CDL3. Rule 1 contains two *synonym declarations*. An important difference between specification and implementation is illustrated in rule 2. If the first alternative of rule 2 fails, then the input contains an invalid expression. If this is the case, an error message is generated, and an attempt is made to recover from the error and resume parsing. In fact, we have enlarged the language recognized with all erroneous strings. We will elaborate this point in chapter 5. In rule 3, a transient parameter contains the abstract syntax tree being built. The transient parameter can be simulated with one inherited and one derived parameter, but the transient parameter allows a more compact notation. In the example, `is ahead (OP)` is a `TEST` inspecting the next symbol of the input. It succeeds if the next symbol is an operator, in which case the operator is returned. `maxprio (PRIO)` is a `FUNCTION` always yielding the maximal priority of an expression. Furthermore, the example contains an overloaded `PREDICATE` operator. Because CDL3 is strongly typed we will consider both occurrences as instances of different nonterminals.

### 2.3.2 Properties of CDL3

A CDL3 grammar is transparent in the sense that each CDL3 grammar can be transformed into a transparent grammar. The construction is as follows.

1. First, all implicit guards and all embedded guards are made explicit.
2. The metagrammar is made unambiguous by replacing each metarule  $N :: h.$  with  $N :: \text{head } h.$ , where `head` is a unique affix terminal. Furthermore, each split guard  $[N' \rightarrow h']$  with  $\text{Form}(N') = \text{Form}(N)$  and  $\text{Form}(h) = \text{Form}(h')$  is replaced with  $[N' \rightarrow \text{head } h']$ . Similarly, each join guard  $[h' \rightarrow N']$  is replaced with  $[\text{head } h' \rightarrow N']$ .

3. A main metanotion  $K$  is constructed as follows. For each nonterminal or primitive predicate  $x$  a metarule  $K :: x(\text{Type}(x, 1), \dots, \text{Type}(x, \text{Pos}(x)))$ . is added.

Clearly, the syntax for  $K$  is also unambiguous and for each basic hypernotation  $h$  we have  $K \rightarrow^* \text{Form}(h)$ .

Any  $LL(1)$  grammar can be transcribed straightforwardly into a recursive descent parser in CDL3 for the language generated by the grammar [Kos94]. Therefore, CDL3 can be seen as a deterministic and transparent version of EAG. Because CDL3 is also a programming language, parsers for any kind of grammars can be written in CDL3, provided that the parsing problem is decidable. However, look-ahead and backtracking have to be programmed explicitly by the programmer, when required. In the next chapters, we will explore transformation techniques for making two level grammars (more)  $LL(1)$ .

## 2.4 Conclusions

A two level grammar consists of a set of rule schemata (the first level), from which production rules can be derived by consistent replacement of variables in a rule schema with values from their domain. This domain can be specified by a context free grammar (the second level).

Rule schemata are built of hypernotations. From hypernotations, other hypernotations can be derived by means of consistent substitutions. Hypernotations from which one or more common protonotations can be derived are called conjoint. Conjoint hypernotations can be unified by conjugations specifying sequences of substitutions. Applying a conjugation yields a junction of two hypernotations. A set of conjugations is called complete, if every common protonotation can be derived from a junction obtained by applying a conjugation in the set. It is not decidable whether hypernotations are conjoint or whether a given set of conjugations is complete.

In conjugation grammars and transparent grammars, definition and application of the meta-grammar are consistent. This means that all substitutions in a conjugation can be obtained by applying direct metaproductions, and allows the second level to be implemented by tree-structured domains. Furthermore, in a transparent grammar it is decidable whether hypernotations are conjoint, and if so, there exists exactly one unifier constituting a complete set of conjugations. This unifier can be computed.

In this chapter, three types of two level grammars have been described. In 2VWG there is no strict border between the first and second level. An EAG may be viewed as a 2VWG in which the first level is separated from the second level. The first level of an EAG constitutes a context free grammar. CDL3 is a programming language based on Affix Grammars. CDL3 can be seen as a transparent and deterministic version of EAG.

## Chapter 3

# Transformations on Two Level Grammars

In this chapter, a set of transformations on two level grammars is defined. Unless stated otherwise, the transformations are defined for 2VWG and their counterparts in related formalisms are assumed to be defined implicitly. Only the generative meaning of a grammar is taken into account. For some transformations, preservation of well-formedness conditions is not guaranteed. This is a consequence of the objective of this project: it is examined whether a grammar can be massaged with transformations until it fits into a more restricted formalism in which stronger well-formedness conditions apply, allowing (more efficient) executability. In general, the initial grammar will not be well well-formed.

First, in §3.1 some notions of equivalence between grammars, rules and hypernotations are defined. These are needed for establishing the correctness of a transformation rule. Then, in §3.2 some notation is provided and in §3.3 a set of general transformation rules is defined. More transformations, specific to deterministic formalisms, are given in chapter 2.3. In §3.5 some sample developments are given. Finally, in §3.6 some conclusions are drawn.

### 3.1 Semantical Foundations

In this section, the semantics of grammars, rules and hypernotations are defined. Objects with the same semantics are considered to be interchangeable.

#### 3.1.1 Equivalence of Grammars

For two level grammars, many different notions of equivalence can be defined. These notions may take into account relations between the metagrammar, hyper-rules, production rules, the semantics of the sentences produced by the grammar, or any combination of them. In this thesis, only weak equivalence is considered. No method for the preservation of the grammatical structure is provided for. Furthermore, it is assumed that ambiguity is an undesirable property.

DEFINITION 3.1.1 Two level grammars  $G$  and  $G'$  are said to be *weakly equivalent*, if

$$\text{Repr}(L(G)) = \text{Repr}(L(G'))$$

If  $G$  and  $G'$  are weakly equivalent,  $G \equiv G'$  is written.

### 3.1.2 Equivalence of Rules

Let  $R$  be a set of rules. Then  $G + R$  denotes the result of adding the rules in  $R$  to grammar  $G$ . Furthermore, let  $G - R$  denote the result of deleting the rules in  $R$  from  $G$ .

DEFINITION 3.1.2 Two sets of rules  $R$  and  $R'$  are called *equivalent* in  $G$ , notation  $R \equiv R'$ , if

$$G \equiv G - R + R'$$

### 3.1.3 Equivalence of Hypernotations

A hypernotation or a sequence of hypernotations specifies some relation between instances of its metanotations and its terminal productions.

DEFINITION 3.1.3 Let  $\alpha$  and  $\beta$  be sequences of hypernotations. When for all affix assignments  $\theta$

$$\theta(\alpha) \rightarrow^* w \Rightarrow \theta(\beta) \rightarrow^* w$$

then  $\alpha \Rightarrow \beta$  is written. Furthermore, when  $\alpha \Rightarrow \beta$  and  $\beta \Rightarrow \alpha$  then  $\alpha$  and  $\beta$  are called *equivalent* and  $\alpha \equiv \beta$  is written.

## 3.2 Notational Conventions

Here, the notational conventions used in the sequel of this chapter are established. Metanotations and sets of metanotations are denoted by capital letters. However, a capital letter  $G$  always refers to a grammar. Small letters  $a, b, \dots, h, g, \dots$  denote hypernotations, while small letter  $r, s, \dots$  denote (labels of) production rules. Sequences of hypernotations are denoted by Greek letters  $\alpha, \beta, \dots$ , but  $\theta$  is used for substitutions which are assumed to be compliant. Concatenation is denoted by juxtaposition or by a comma. Semicolons are used as separators when enumerating a set of rules. When enumerating a sequence of alternatives, this sequence is assumed to contain all alternatives. Furthermore, the notational conventions for denoting rules, as defined in the previous chapter, apply.

Often, applicability of a transformation rule is limited by some conditions. For formulating these conditions, the following functions are defined. Let  $\#Occs(N, x)$  be the number of occurrences of metanotation  $N$  in  $x$ , where  $x$  is a rule or a sequence of hypernotations. The set of metanotations occurring in  $x$  is denoted by  $Var(x)$ .

The hypernotation  $a(X_1, \dots, X_n)$  denotes any hypernotation  $h$  with  $Var(h) = \cup_{1 \leq i \leq n} X_i$ . The same convention applies to sequences of hypernotations. For example, if  $\alpha = \beta(Y)$  is a sequence of hypernotations, then  $Var(\alpha) = Y$ . When  $\alpha(X)$  and  $\alpha(Y)$  occur within the same context, then it is assumed that  $Form(\alpha(X)) = Form(\alpha(Y))$  and that  $Var(X) \cap Var(Y) = \emptyset$ .

Finally, let  $x[N := y]$  denote the result of substituting all occurrences of metanotation  $N$  in  $x$  with  $y$ , where  $x$  may be a rule or a hypernotation.

### 3.2.1 Notation of Transformation rules

A *transformation rule* is a statement of equivalence between a *specification grammar*  $G$  and an *implementation grammar*  $G'$ , possibly preceded by some applicability conditions  $C$ , meaning that  $C \Rightarrow (G \equiv G')$ .

Only the rules or hypernotations involved in a particular transformation are mentioned. However, both cases are shorthand for a transformation between two grammars. Transformations between sequences of hypernotations are intended to take place within the right hand side of some rule.

### 3.2.2 Notation of Derivations

A derivation consists of a sequence  $G_0 \equiv \{\text{Comment}_1\}G_1 \dots \equiv \{\text{Comment}_n\}G_n$ , where ‘ $\text{Comment}_i$ ’ refers to one or more transformation rules. By transitivity of  $\equiv$  it follows that  $G_0 \equiv G_n$ . As a shorthand, only the rules involved in a particular derivation are mentioned.

## 3.3 Basic Transformation Rules

In this section, a set of transformation rules which may be useful in deriving an executable grammar from a specification grammar is defined. Some transformations are a generalization of well known ‘syntax improving’ transformations for context free grammars to two level grammars. Other transformations can be viewed as applications of well known program development transformations in a grammatical context.

### 3.3.1 General Transformations

The following transformations can be used for utilizing equivalences in a grammar which are not covered by any other transformation.

In the right hand side of a hyper-rule, a sequence of hypernotations may be replaced with an equivalent sequence.

TRANSFORMATION 1 *Equivalent hypernotations*

Let  $r = h : \alpha\beta\gamma$ . and  $r' = h : \alpha\beta'\gamma$ . be production rules with  $\beta \equiv \beta'$ . Then  $r \equiv r'$ .

Furthermore, a set of rules may be replaced with an equivalent set.

TRANSFORMATION 2 *Equivalent rules*

Let  $R$  and  $R'$  be sets of rules with  $R \equiv R'$ . Then

$$G \equiv G - R + R'$$

### 3.3.2 Folding and Unfolding

Folding and unfolding are two well known transformations from transformational program development (see e.g. [Par90]). In essence, an unfolding transformation replaces a definiendum by its definition, while a folding transformation does the opposite.

In a grammatical context, these transformations have obvious counterparts, with the right hand side of a rule acting as a definition for the left hand side. This leads to simple substitution transformations, such as the expansion and contraction of nonterminals. In fact, these are nothing but unfolding and folding transformations on context free grammars. For example, the corner substitution transformation (expansion) followed by a left-factorization is a well known technique for achieving  $LL(1)$ -ness.

In the case of two level grammars, these transformations become even more interesting. A two level grammar may be viewed as the specification of a program. By generalizing substitution transformations on context free grammars to two level grammars, folding and unfolding become useful transformations for both syntax improvement and program development. The combination of unfolding, simplification and subsequent folding results in a strategy for deriving recursive predicates from their declarative specifications. See §3.5 for some examples.

An unfolding transformation replaces a member with its alternatives. For every alternative of the member being expanded, a new rule is introduced. The original rule has become superfluous and is deleted from the grammar. Folding is the inverse transformation of unfolding.

#### Folding and Unfolding of Metanotions

TRANSFORMATION 3 *Fold/Unfold metanotion in metarule* (expansion)

$$\begin{array}{l}
 M \quad :: \quad \alpha N \beta. \\
 N \quad :: \quad \gamma_1; \dots; \gamma_n. \\
 \quad \quad \equiv \\
 M \quad :: \quad \alpha \gamma_1 \beta; \\
 \quad \quad \quad \vdots \\
 \quad \quad :: \quad \alpha \gamma_n \beta. \\
 N \quad :: \quad \gamma_1; \dots; \gamma_n.
 \end{array}$$

EXAMPLE 3.3.1 Unfolding of metanotions in metarules.

```

TAD :: bold TAG ;
      DYAD BECOMESETY ;
      DYAD cum NOMAD BECOMESETY .

```

```

TAM :: bold TAG ;
      MONAD BECOMESETY ;
      MONAD cum NOMAD BECOMESETY.
TAO :: TAD ;
      TAM.

```

Obviously the grammar for TAO is ambiguous, since for example a **bold TAG** has two leftmost derivations.

```

TAO :: TAD ;
      TAM.
                                                    ≡ {Unfold TAD}

TAO :: bold TAG ;
      DYAD BECOMESETY ;
      DYAD cum NOMAD BECOMESETY.
      TAM.
                                                    ≡ {Unfold TAM}

TAO :: bold TAG ;
      DYAD BECOMESETY ;
      DYAD cum NOMAD BECOMESETY.
      bold TAG ;
      MONAD BECOMESETY ;
      MONAD cum NOMAD BECOMESETY.
                                                    ≡ {Delete superfluous rule}

TAO :: bold TAG ;
      DYAD BECOMESETY ;
      DYAD cum NOMAD BECOMESETY.
      MONAD BECOMESETY ;
      MONAD cum NOMAD BECOMESETY.

```

Notice that an ambiguity has been removed. The syntax for TAO is still ambiguous, but the ambiguity can be removed by similarly unfolding DYAD.

In a two level grammar, similar hyper-rules are often grouped into one rule. This grouping is done or undone by folding or unfolding metanotations in hyper-rules. An important application of unfolding metanotations in hyper-rules is to serve as a *case introduction*. In this way, subsequent simplifications may become possible, and different cases may be given tailored treatments. On the other hand, unfolding increases the number of hyper-rules and may introduce prefix sharing between alternatives.

#### TRANSFORMATION 4 *Unfold/Fold metanotation in hyper-rule*

Let  $r$  be a hyper-rule in which a metanotation  $N$  occurs.

Let  $N :: h_1 ; \dots ; h_n$ . be all alternatives for  $N$ .

Assume that for each alternative  $h_i$  and each metanotation  $K$

$$\#Occs(K, h_i) \leq 1 \tag{3.1}$$

$$Var(h_i) \cap Var(r) \subseteq \{M\} \tag{3.2}$$

Let  $r_i = r[N := h_i]$  for  $1 \leq i \leq n$ .

Then

$$r \equiv r_1 ; \dots ; r_n$$

Conditions 3.1 and 3.2 prevent undesired consistent substitution of metanotions in a metarule. Both conditions can be fulfilled by renaming metanotions.

PROOF It will be proven that

$$PR(r) = \bigcup_{1 \leq i \leq n} PR(r_i)$$

$\subseteq$  Let  $\theta(r)$  be a production rule, say  $\theta(r) = \theta'(r[N := w])$  with  $w \in L_M(h_i)$ . Because  $\forall_{K \in M} [\#Ocs(K, h_i) \leq 1]$  (3.1), there exists some  $\xi$  with  $w = \xi(h_i)$  and  $Map(\xi) = Var(h_i)$ . Because  $Map(\xi) \cap Var(r) \subseteq \{N\}$  (3.2), we have  $r[N := \xi(h_i)] = \xi(r[N := h_i])$ . Consequently,  $\theta(r) = \theta'(r[N := w]) = \theta'(r[N := \xi(h_i)]) = \theta'(\xi(r[N := h_i]))$ . Then  $\theta(r)$  is derivable from  $r[N := h_i]$  by  $\theta' \circ \xi$ .

$\supseteq$  Let  $\theta(r[N := h_i])$  be a production rule. Say  $\theta(r[N := h_i]) = \theta(\xi(r))$  for  $\xi$  with  $Map(\xi) = \{N\}$  and  $\xi : N \mapsto h_i$ . Then  $\theta(r[N := h_i])$  is derivable from  $r$  by  $\theta \circ \xi$ .

□

Folding is defined to be the inverse transformation of unfolding. Thus, the result of folding a metanotion  $h$  in a grammar  $G$  is any grammar  $G'$  such that  $G$  is the result of unfolding  $h$  in  $G'$ . Notice that folding is nondeterministic, i.e. folding does not have an unique result.

EXAMPLE 3.3.2 Unfolding of a metanotion in hyper-rules.

```

WHETHER :: where ; unless.
WHETHER SORT balances SORT1 and SORT2 :
  where (SORT1) is (strong), WHETHER (SORT2) is (SORT) ;
  where (SORT2) is (strong), WHETHER (SORT1) is (SORT).
                                                    ≡ {Unfold WHETHER}

WHETHER :: where ; unless.
where SORT balances SORT1 and SORT2 :
  where (SORT1) is (strong), where (SORT2) is (SORT) ;
  where (SORT2) is (strong), where (SORT1) is (SORT).
unless SORT balances SORT1 and SORT2 :
  where (SORT1) is (strong), unless (SORT2) is (SORT) ;
  where (SORT2) is (strong), unless (SORT1) is (SORT).

```

## Folding and Unfolding of Hypernotations

The intuition behind unfolding hypernotations is to anticipate on every possible production. For hypernotations this is not as easy as for metanotions. Let  $h$  be a right hand side hypernotation.

Then a hyper-rule  $h' : \alpha$ . may be applied to  $h$  if and only if  $h$  and  $h'$  are conjoint. Assume that a conjugation  $(\theta, \theta')$  of  $(h, h')$  does exist, hence  $\theta(h) = \theta'(h')$ . Then (under some conditions)  $h$  may be substituted with  $\theta'(\alpha)$ , provided that the context of  $h$  is confined to  $\theta$ . Of course,  $h$  must be unfolded for every possible alternative. Furthermore, one must take care to unfold  $h$  for every possible conjugation, or else productions will be lost. It will be shown that a complete set of conjugations is sufficient for this purpose.

Both folding and unfolding of hypernotations presuppose the reference problem being solved. In addition, one must be able to find a finite and complete set of conjugations, which is also undecidable, as stated by proposition 2.1.2.

The unfolding transformation is often used as preparation for a left-factorization or left-recursion removal. Another important application is to eliminate certain hyper-rules by unfolding all of their occurrences. Folding is also used to make a newly introduced rule apply. Finally, it may be employed in an unfold/fold strategy for deriving recursive predicates.

TRANSFORMATION 5 *Unfold/Fold hypernotation in hyper-rule.*

Let  $r = g : \alpha, h, \beta$ . be a hyper-rule.

Let  $h_1 : \gamma_1; \dots; h_n : \gamma_n$ . be all hyper-rules with  $Conjoint(h, h_i)$ .

Assume that for each hyper-rule  $h_i : \gamma_i$ . with  $1 \leq i \leq n$  there exists a finite and complete set of conjugations  $\{(\theta_{i_1}, \theta'_{i_1}), \dots, (\theta_{i_{j_i}}, \theta'_{i_{j_i}})\}$  of  $(h, h_i)$ . Assume that for each hyper-rule  $h_i : \gamma_i$ . and each conjugation  $(\theta_{i_j}, \theta'_{i_j})$  with  $1 \leq i \leq n$  and  $1 \leq j \leq j_i$

$$Var(\theta_{i_j}(g, \alpha\beta)) \cap Var(\theta'_{i_j}(\gamma_i)) \subseteq Var(\theta_{i_j}(h)) \quad (3.3)$$

$$Map(\theta_{i_j}) \subseteq Var(h) \quad (3.4)$$

$$Map(\theta'_{i_j}) \subseteq Var(h_i) \quad (3.5)$$

Let

$$r_{i_j} = \theta_{i_j}(g) : \theta_{i_j}(\alpha)\theta'_{i_j}(\gamma_i)\theta_{i_j}(\beta).$$

for  $1 \leq i \leq n$  and  $1 \leq j \leq j_i$ . Then

$$r \equiv r_{1_1} ; \dots ; r_{1_{j_1}} ; \dots ; r_{n_1} ; \dots ; r_{n_{j_n}}$$

Condition 3.3 ensures that the consistent substitution rule does not bind metanotions of an unfolded alternative to metanotions in the rule in which the unfolding has taken place, unless these metanotions are already bound indirectly via the unfolded hypernotation. This condition can be satisfied by renaming metanotions. Conditions 3.4 and 3.5 ensure that each conjugation of two hypernotations does not affect the rest of a rule. These conditions can be satisfied trivially.

PROOF

$\Rightarrow$  Let  $\theta(g)$  be a production rule with  $\theta(g) \rightarrow \theta(\alpha), \theta(h), \theta(\beta) \rightarrow \theta(\alpha)\theta(\gamma_i)\theta'(\beta)$  for some  $\theta, \theta'$  with  $\theta(h) = \theta(h_i)$ . Because there exists a complete set of conjugations of  $(h, h_i)$ , we have  $\theta(h) = \rho \circ \theta_{i_j}$  and  $\theta'(h_i) = \rho \circ \theta'_{i_j}$  for some  $\rho, \theta_{i_j}, \theta'_{i_j}$  with  $1 \leq j \leq j_i$ .

Define  $\xi$  with

$$\xi(N) = \begin{cases} \theta(N) & \text{if } N \in \text{Var}(\theta_{i_j}(g, \alpha\beta)) \setminus \text{Var}(\theta'_{i_j}(\gamma_i)) \setminus \text{Var}(\theta_{i_j}(h)) \\ \theta'(N) & \text{if } N \in \text{Var}(\theta'_{i_j}(\gamma_i)) \setminus \text{Var}(\theta_{i_j}(g, \alpha\beta)) \setminus \text{Var}(\theta_{i_j}(h)) \\ \rho(N) & \text{if } N \in \text{Var}(\theta_{i_j}(g, \alpha\beta)) \cap \text{Var}(\theta'_{i_j}(\gamma_i)) \end{cases}$$

Then  $\xi(r_{i_j})$  is a production rule, because of 3.3. We claim that for all  $N \in \text{Var}(g, \alpha\beta)$

$$\xi \circ \theta_{i_j}(N) = \theta(N) \quad (3.6)$$

Furthermore, for all  $N \in \text{Var}(\gamma_i)$

$$\xi \circ \theta'_{i_j}(N) = \theta'(N) \quad (3.7)$$

Therefore,  $\xi(r_{i_j}) = \xi \circ \theta_{i_j}(g) : \xi \circ \theta_{i_j}(\alpha) \xi \circ \theta'_{i_j}(\gamma_i) \xi \circ \theta_{i_j}(\beta) = \theta(g) : \theta(\alpha) \theta'(\gamma_i) \theta(\beta)$ . Consequently,  $\theta(g) \rightarrow \theta(\alpha) \theta'(\gamma_i) \theta(\beta)$  in the new grammar.

It remains to proof 3.6 and 3.7.

3.6 Let  $N \in \text{Var}(g, \alpha\beta)$ . Case 1: suppose  $N \notin \text{Map}(\theta_{i_j})$ . Then  $N \in \text{Var}(\theta_{i_j}(g, \alpha\beta))$ .  
 Case 1.1: suppose  $N \notin \text{Var}(\theta_{i_j}(h))$ . Then, due to 3.4,  $\xi \circ \theta_{i_j}(N) = \xi(N) = \theta(N)$ .  
 Case 1.2:  $N \in \text{Var}(\theta_{i_j}(h))$ . Then  $\xi \circ \theta_{i_j}(N) = \rho \circ \theta_{i_j}(N) = \theta(N)$ . Case 2:  $N \in \text{Map}(\theta_{i_j})$ , say  $\theta_{i_j}(N) = h'$ . For all  $N' \in \text{Var}(h')$  we have  $N' \in \text{Var}(\theta_{i_j}(h))$ , because  $N \in \text{Var}(h)$  due to 3.4. Then  $\xi \circ \theta_{i_j}(N) = \rho(h') = \theta(N)$ .

3.7 Similar.

$\Leftarrow$  Similar.

□

Folding is defined to be the inverse transformation of unfolding. Both transformations are nondeterministic. For unfolding this is the case if there exists more than one complete set of conjugations.

Again, unfolding may be used to reduce ambiguity. However, an ill chosen set of conjugations may lead to ambiguity.

EXAMPLE 3.3.3 Unfolding of hypernotations. Remember that

```
PACK :: STYLE pack.
NOTETY STYLE pack :
  STYLE begin token, NOTETY, STYLE end token.
```

Now

```
SOID NEST closed clause :
  SOID NEST serial clause defining LAYER PACK.
  ≡ {Unfold SOID NEST serial clause defining LAYER PACK}
SOID NEST closed clause :
  STYLE begin token,
  SOID NEST serial clause defining LAYER,
  STYLE end token.
```

### 3.3.3 Rule Introduction and Pruning

The transformations in this subsection provide means for introducing new rules to a grammar, or remove superfluous ones. It is clear that a rule which is not used at all, i.e. no instance of its left hand side occurs within any right hand side, may be introduced to or removed from the grammar at will. In addition, a hyper-rule may be introduced or removed, if every useful production rule which can be derived from it can be derived from another rule as well. Remember that a rule which is not used, may become used by folding.

It is essential to dispose of rules which will not be needed anymore, or cannot be productive. The process of removing this dead wood is called *pruning*. Unused rules may become used after folding. Rule introduction followed by folding is known as *abstraction* [Par90].

#### Introduction and Removal of Metarules

A metarule may become useless after unfolding metanotions. A set of metarules  $R$  is called *unused* if every left hand side metanotion of a rule in  $R$  does not occur within any metarule outside of  $R$  or within any hyper-rule.

TRANSFORMATION 6 *Add/Delete unused metarule*

Let  $R$  be an unused set of metarules.

Then

$$G \equiv G + R$$

A metarule is called *superfluous* if it does not contribute to the language of the left hand side metanotion.

TRANSFORMATION 7 *Add/Delete superfluous metarule*

Let  $r_0; \dots; r_n$  be metarules, with  $r_i = m : h_i$ . for  $0 \leq i \leq n$ .

Assume that

$$\bigcup_{0 \leq i \leq n} L_M(h_i) = \bigcup_{1 \leq i \leq n} L_M(h_i)$$

Then

$$r_0 ; r_1 ; \dots ; r_n \equiv r_1 ; \dots ; r_n$$

#### Introduction and Removal of Hyper-rules

A hyper-rule may become useless after unfolding all occurrences of its left hand side. A second possibility is that only some production rules which can be derived from a hyper-rule, can be used in any derivation. And as a third possibility, all useful production rules of a hyper-rule may be derivable from another hyper-rule as well, making the rule *superfluous*.

DEFINITION 3.3.1 The set of *used production rules*  $UPR(G)$  of a grammar  $G$  is defined as follows:

1. A production rule whose left hand side notion is conjoint with the start hypernotation is used.
2. Each production rule whose left hand side is conjoint with a right hand side notion of a rule already in  $UPR(G)$  is also used.
3. No other production rule is used.

TRANSFORMATION 8 *Add/Delete hyper-rule*

Let  $R$  be a set of hyper-rules, such that

$$UPR(G + R) = UPR(G)$$

Then

$$G + R \equiv G$$

A special case of this transformation is called *specialization*. It consists of the introduction of a variant of a hyper-rule, which is specialized to a dedicated task. If a hyper-rule has a specialization for every possible application, the generally formulated rule can be removed from the grammar. Therefore, specialization provides a second mean for eliminating certain rules. Specialization may have the advantage of enabling simplifications which are not possible in the general case.

EXAMPLE 3.3.4 Specialization of a rule.

```

where THING1 or THING2 : where THING1 ; where THING2.
                                                                    ≡ {Specialization}

where THING1 or THING2 : where THING1 ; where THING2.
where (TAB) is (bold TAG) or (NEST) is (new LAYER) :
  where (TAB) is (bold TAG) ;
  where (NEST) is (new LAYER).

```

Here, specialization has the advantage over unfolding `where THING1 or THING2` of preventing prefix sharing in the rule `where (TAB) is (bold TAG) or (NEST) is (new LAYER)` is used.

Now suppose that every right hand side occurrence of `where THING1 or THING2` is covered by `where (TAB) is (bold TAG) or (NEST) is (new LAYER)`. Then

```

where THING1 or THING2 : where THING1 ; where THING2.
where (TAB) is (bold TAG) or (NEST) is (new LAYER) :
  where (TAB) is (bold TAG) ;
  where (NEST) is (new LAYER).

```

$\equiv \{\text{Delete superfluous hyper-rule}\}$

where (TAB) is (bold TAG) or (NEST) is (new LAYER) :  
 where (TAB) is (bold TAG) ;  
 where (NEST) is (new LAYER).

Notice that by specialization all unused production rules can be weeded out, at the cost of sacrificing the generality of the description.

### Introduction and Removal of Blind Alleys

Sometimes it is possible to establish that a hyper-rule contains a member which yields blind alleys only, typically after case introduction and subsequent simplifications. For example, a rule always producing a member **where false** will never be productive and may be deleted. Introduction of a blind alley may be useful for enabling folding.

TRANSFORMATION 9 *Add/Delete blind alley*

Let  $r = g : \alpha$ . be a hyper-rule with  $L(\alpha) = \emptyset$ . Then

$$G \equiv G + \{r\}$$

EXAMPLE 3.3.5 In developing a predicate for uniting, one may derive

where MOOD1 unites to MOOD2 :  
 unless MOOD1 equivalent MOOD2,  
 where safe MOOD1 subset of safe MOOD2.  
 $\equiv \{\text{Unfold where safe MOOD1 subset of safe MOOD2}\}$   
 where MOOD1 unites to MOOD2 :  
 unless MOOD1 equivalent MOOD2,  
 where MOOD1 equivalent MOOD2.

which yields a blind alley. Therefore, this rule may be deleted from the grammar.

### 3.3.4 Predicate Introduction and Removal

The terms ‘predicate introduction’ and ‘removal’ refer to the insertion and elision of a predicate testing for equality in an alternative. The idea behind these transformations is that identity of hypernotions is interchangeable with a differentiation which is compensated for by an equality test. For performing an equality test, the existence, or else introduction, of a predicate **where THING1 is THING2** defining the equality relation is assumed.

Predicate introduction and removal have numerous applications. For example, by predicate introduction different hypernotions can be unified which may be needed for enabling left-factorization or left-recursion removal. Furthermore, by introducing predicates stepwise decomposing metanotions consistency between the definition of metanotions and their applications may be reached.

One may notice the great similarity between guards [Kos94] and equality tests. However, equality tests may compare any two affix expressions, while guards compare a metanotion with a synonymous metanotion or a conforming affix expression. Examples of how affix expressions are made conforming are given in §4.2.4.

TRANSFORMATION 10 *Predicate introduction/removal*

Let  $N \in M$  and  $h \in (V_T \cup M)^*$ , such that  $L_M(h) \subseteq L_M(N)$ , and  $g : \alpha.[N := h]$  is a hyper-rule.

Assume that

$$N \notin \text{Var}(h) \tag{3.8}$$

Then

$$g : \alpha.[N := h] \equiv g : \alpha, \text{where } (N) \text{ is } (h).$$

Condition 3.8 prevents introduction of a blind alley.

If a rule contains a predicate `where (N) is (h)` then one or more occurrences of  $N$  in other hypernotations may be replaced with  $h$ . This is called *apply property*.

EXAMPLE 3.3.6 Predicate introduction and removal.

$$\begin{array}{l} \text{SOID NEST series with PROPSETY :} \\ \quad \text{where (PROPSETY) is (DECS DECSETY LABSETY),} \\ \quad \dots \\ \hspace{20em} \equiv \{\text{Predicate removal}\} \\ \text{SOID NEST series with DECS DECSETY LABSETY :} \\ \quad \dots \\ \hspace{20em} \equiv \{\text{Predicate introduction}\} \\ \text{SOID NEST series with DECSETY1 LABSETY :} \\ \quad \text{where (DECSETY1) is (DECS DECSETY),} \\ \quad \dots \end{array}$$

### 3.3.5 Left-factorization

Left-factorization is used for removing prefix sharing and sharing computations in different branches of a predicate.

There are some problems in left-factoring two level grammars. First of all, the hyper-rules which are intended to serve as alternatives for a hypernotation, may have different heads. Secondly, these rules may not have exactly the same prefixes. Both problems can be solved by using predicate introduction as a preparation for left-factoring. As a third problem, a shared prefix may be preceded by a predicate. For this reason, a transformation *change order* is introduced (Transformation 3.3.11).

The transformation presented here, is a straightforward generalization of the left factorization transformation which is known from context free grammars. In this case, there is only a slight complication in preserving the bindings between metanotions, caused by the consistent substitution rule.

DEFINITION 3.3.2 A hypernotation  $h = \alpha_0 X_1 \alpha_1 \dots X_n \alpha_n$  is *uniquely assignable* [Weg80], if for each notion  $h'$  derived from  $h$  there is a unique affix assignment  $\theta$  with  $\theta(h) = h'$ .

Notice that a transparent grammar is uniquely assignable, because its metagrammar is unambiguous.

TRANSFORMATION 11 *Left-factorization*

Let  $g(X, Y)$  be an uniquely assignable hypernotation. Then

$$\begin{aligned} h(X) &: \alpha(Y)\beta_1; \\ &\quad \vdots \\ &\quad \alpha(Y)\beta_n. \\ &\equiv \\ h(X) &: \alpha(Y), g(X, Y). \\ g(X, Y) &: \beta_1; \\ &\quad \vdots \\ &\quad \beta_n. \end{aligned}$$

Left-factorization is performed by rule introduction and folding.

EXAMPLE 3.3.7

```

SOID NEST series with PROPSETY :
  strong void NEST unit,
  go on token,
  SOID NEST series with PROPSETY ;
  where (PROPSETY) is (EMPTY),
  SOID NEST unit.
                                     ≡ {Change order (in second alternative)}

SOID NEST series with PROPSETY :
  strong void NEST unit,
  go on token,
  SOID NEST series with PROPSETY ;
  SOID NEST unit,
  where (PROPSETY) is (EMPTY).
                                     ≡ {Predicate introduction (2×)}

SOID NEST series with PROPSETY :
  SOID1 NEST unit,
  go on token,
  SOID NEST series with PROPSETY,

```

```

    where (SOID1) is (strong void) ;
SOID1 NEST unit,
    where (PROPSETY) is (EMPTY),
    where (SOID1) is (SOID).

```

≡ {Left-factoring}

```

SOID NEST series with PROPSETY :
SOID1 NEST unit,
    SOID1 SOID NEST rest series with PROPSETY.
SOID1 SOID NEST rest series with PROPSETY :
go on token,
    SOID NEST series with PROPSETY,
    where (SOID1) is (strong void) ;
where (PROPSETY) is (EMPTY),
    where (SOID1) is (SOID).

```

Notice that `SOID1 SOID NEST rest series with PROPSETY` is uniquely assignable, while e.g. `SOID1 SOID NEST PROPSETY rest series` is not. It would be inappropriate for threading the metanotions of `series` to the alternatives of `rest series`, because of the unintended inference between `NEST` and `PROPSETY` in juxtaposition.

### 3.3.6 Left-recursion Removal

In [Sar95], a transformation is defined for removing left-recursion from attribute grammars. The same transformation can be applied to (Extended) Affix Grammars.

Evaluation of left-recursive rules may give rise to termination problems when a top-down parsing scheme is to be used, unless termination is guaranteed by application of affix directed parsing. These problems can be solved by either using a more powerful parsing scheme or by eliminating the left-recursion.

The transformation presented here is only applicable to direct left-recursive rules. A generalization to indirect left-recursion is possible [Sar95], but in combination with the transformations in the rest of this chapter removal of direct left-recursion turns out to be sufficient for obtaining a top-down parsable grammar for Algol 68.

In 2VWG it may be a matter of taste whether some hypernotation is called left-recursive or not. Therefore, a two level grammar which is to be treated against left-recursion is assumed to be such that it has an underlying context free grammar.

First, recall the transformation for eliminating direct left-recursion from context free grammars:

TRANSFORMATION 12 *Left-recursion removal for context free grammars*

Let  $C$  produce all non left-recursive alternatives for  $A$ . Then

$$\begin{aligned} A &:: A, B; \\ &C. \\ &\equiv \\ A &:: C, B'. \\ B' &:: B, B'; \\ &\varepsilon. \end{aligned}$$

Left-recursion is removed from a two level grammar in an analogous way:

TRANSFORMATION 13 *Left-recursion removal*

Let  $c(X)$  produce all non left-recursive alternatives of  $a(X)$ . Furthermore, assume that  $b'(X, Y)$  is uniquely assignable. Then

$$\begin{aligned} a(X) &:: a(Y), b(X, Y); \\ &c(X). \\ &\equiv \\ a(X) &:: c(Y), b'(X, Y); \\ b'(X, Y) &:: b(Z, Y), b'(X, Z); \\ b'(X, X) &:: \varepsilon. \end{aligned}$$

It is clear that with predicate introduction, change order and abstraction, a directly left-recursive grammar can be brought into the above form. Many indirect left-recursive predicates can be made directly left-recursive with unfolding.

PROOF See [Sar95]. □

EXAMPLE 3.3.8 Left recursion removal from hyper-rules in EAG.

```
(NEST, COMMON) joined definition of (PROPS1) :
  (NEST, COMMON) joined definition of (PROPS),
  (and also) token,
  (NEST, COMMON) definition of (PROP),
  where (PROPS1) is (PROPS * PROP) ;
  (NEST, COMMON) definition of (PROPS1).
                                                                 ≡ {Remove left-recursion}
(NEST, COMMON) joined definition of (PROPS1) :
  (NEST, COMMON) definition of (PROPS),
  (NEST, COMMON) rest joined definition of (PROPS1, PROPS).
(NEST, COMMON) rest joined definition of (PROPS1, PROPS1) : .
(NEST, COMMON) rest joined definition of (PROPS1, PROPS) :
  (and also) token,
  (NEST, COMMON) definition of (PROP),
  where (PROPS2) is (PROPS * PROP),
  (NEST, COMMON) rest joined definition of (PROPS1, PROPS2).
```

Notice that left-recursion removal may cause local ambiguity [Kos74]. In the grammar for Algol 68, all local ambiguities resulting from left recursion removal can be solved with a small lookahead.

### 3.3.7 Restriction

Within a hyper-rule a metanotion is often restricted by the consistent substitution rule to a subset of its language. For example, the predicate `where (PROPSETY) is (LAB LABSETY)` restricts every occurrence of `PROPSETY` in a rule to those productions that are of the form `LAB LABSETY`.

In a similar fashion metanotions may be restricted to patterns occurring outside the rule. Consider the following hyper-rule

```

WHETHER PROP identified in NEST new PROPSETY :
  where PROP resides in PROPSETY, WHETHER true.

```

It is clear that `where` is the only productive choice for `WHETHER`. One might discover this fact by unfolding `WHETHER` and concluding that `unless true` is a blind alley. Another possibility is to examine every call of `identified`. This will lead to the conclusion that these are all of the form `where PROP identified in NEST`. Therefore, `WHETHER` can be *restricted* to `where` without losing any productions. Obviously, restriction provides another mean of pruning the set of production rules of a grammar.

**DEFINITION 3.3.3** A hypernotation  $h$  *matches* a hypernotation  $h'$ , if  $h$  and  $h'$  are conjoint and  $h$  is a left hand side hypernotation and  $h'$  is a right hand side hypernotation. Furthermore, if  $h$  matches  $h'$ , then  $h'$  matches  $h$ .

When matching `WHETHER PROP identified in NEST new PROPSETY` against its call `where PROP identified in NEST` one may also conclude that `PROPSETY` must necessarily be of the form `DECSETY LABSETY`, because a `NEST` is of the form `LAYER` or `NEST LAYER` and a `LAYER` is of the form `new DECSETY LABSETY`. Therefore, it is justified to restrict `PROPSETY` to `DECSETY LABSETY`. Conversely, it would also be possible to substitute `PROPSETY` with `NOTION`.

**TRANSFORMATION 14** *Restriction*

Let  $r = h_0 : h_1, \dots, h_n$  be a hyper-rule.

Let  $N \in M$  and  $h \in (V_T \cup M)^*$ .

Assume that

$$\text{Var}(h) \cap \text{Var}(r) \subseteq \{N\} \tag{3.9}$$

Assume that there exists a hypernotation  $h_i$  with  $0 \leq i \leq n$  and  $N \in \text{Var}(h_i)$ , such that for every hypernotation  $g$  matching  $h_i$

$$P(g) \cap P(h_i) = P(g) \cap P(h_i[N := h])$$

Then

$$r \equiv r[N := h]$$

Condition 3.9 prevents unintended bindings caused by the consistent substitution rule. This condition can be satisfied by renaming metanotions.

### 3.3.8 Invariant Introduction and Removal

In two level grammars, predicates provide the means which may be needed for formulating assertions which may be needed for the preservation of certain context information. For this purpose, the following rule is introduced.

TRANSFORMATION 15 *Invariant Introduction/Removal*

Let  $h$  be a predicate and let  $g : \alpha\beta.$  be a hyper-rule with

$$\alpha\beta \equiv \alpha, h, \beta$$

Then

$$g : \alpha\beta. \equiv g : \alpha, h, \beta.$$

### 3.3.9 Lifting and Sinking

Sometimes a predicate or a left-corner can be lifted to another rule.

TRANSFORMATION 16 *Lifting/Sinking*

Let  $h(X, Y)$  be uniquely assignable. Then

$$\begin{aligned} g & : h(X, Y), \beta. \\ h(X, Y) & : \alpha(X)\gamma_1; \dots; \alpha(X)\gamma_n. \\ & \equiv \\ g & : \alpha(X), h(X, Y), \beta. \\ h(X, Y) & : \gamma_1; \dots; \gamma_n. \end{aligned}$$

Lifting is simulated by unfolding, rule introduction and folding. A generalization to ‘deeper’ left-corners is possible.

### 3.3.10 Embedding

Sometimes it is useful to endow hypernotations with extra parameters or remove useless ones.

TRANSFORMATION 17 *Embedding*

Suppose that  $\alpha, h', \beta \equiv \alpha, h'(X), \beta$ . Then

$$\begin{aligned} g & : \alpha, h, \beta. \\ & \equiv \\ g & : \alpha, h'(X), \beta. \end{aligned}$$

Because of the consistent substitution rule, embedding cannot be simulated by rule introduction and folding. Special cases are *removal of a constant parameter* and *removal of a superfluous parameter*.

### 3.3.11 Change Order

The position of a predicate in an alternative is in principle irrelevant, because predicates produce the empty string. Consequently, a predicate may be moved from one to any other position in an alternative. This transformation will be called *change order*. Changing the order of hypernotations in an alternative may be needed for unifying sequences of hypernotations, enabling subsequent transformations like left-factorization, left-recursion removal or folding.

TRANSFORMATION 18 *Change order*.

Assume for sequences of hypernotations  $\zeta$  and  $\eta$  that  $\zeta \equiv \eta$ . Then

$$\begin{aligned} h & : \alpha\zeta\beta\eta\gamma. \\ & \equiv \\ h & : \alpha\eta\beta\zeta\gamma. \end{aligned}$$

### 3.3.12 Symbolic Evaluation

Sometimes it is possible to evaluate predicates partially or completely at ‘specification time’. For example,

`unless (union of MOODS mode) equivalent (MOOD)  $\equiv$  where true`

Actually, this can be seen an application of the rule for equivalent hypernotations, but it will be referred to as symbolic evaluation. Furthermore, instantiation of a metanotion (by restriction) is also called symbolic evaluation.

### 3.3.13 Renaming

Sometimes, readability of a grammar can be improved by altering the appearance of hypernotations. *Renaming* a hypernotation  $h$  into  $g$  means the collateral replacement of all instances  $h'$  of  $h$  into instances  $g'$  of  $g$ , such that  $h' \equiv g'$  for each  $h'$  and  $g'$ . Renaming can be simulated by rule introduction and folding. Metanotions can also be renamed by rule introduction and folding.

In order to satisfy the applicability conditions of a transformation, it may be necessary to rename metanotions in a hyper-rule, i.e. the consistent replacement of a metanotion with a synonym. By convention, these renamings are performed implicitly when demanded.

### 3.3.14 Transformation of the Metagrammar

Any language-preserving transformation may be applied to the metagrammar. However, the language produced by the metagrammar is in principle irrelevant. The metanotions of a two level grammar can be viewed as variables of a data type designated by the variable. A type  $A$  can be implemented by some other type  $B$ , if the operations on  $A$  can be replaced with operations on  $B$  expressing the same behavior. However, if one is interested in one particular grammar, it is not necessary that  $B$  completely models  $A$ .

In order to limit the scope of this thesis, we will not investigate into transformations on data-types, but only remark that there are several problems in defining type transformations on two level grammars. First, in some kinds of two level grammars, e.g. 2VWG and EAG, the operations on data types are not explicit but hidden within conjugations. Secondly, it may be convenient to enlarge a domain with values which are never used in any production. Conversely, it is clear that unused values may be removed from a domain. Therefore, operations may be replaced with operations which are equivalent for only some subset of their domains. Thirdly, variables may occur within terminal symbols. Therefore, altering the domain of a type may invoke changes to the representation of the symbols. This problem can be circumvented by replacing the representation of symbols with a *lexical grammar* (see §4.2.8) whose terminal symbols do not contain metanotions.

It is a matter of taste which predicates are considered primitive operations. It is clear that it is sufficient to model the splitting, concatenation and equality test on strings.

#### TRANSFORMATION 19 *Transformation of metagrammar*

The metagrammar may be replaced with any other metagrammar, as long as each hypernotation is replaced with an equivalent hypernotation.

EXAMPLE 3.3.9 We will implement LAYER with LAYER', where

LAYER :: new DECSETY LABSETY.	LAYER' :: new * DECSETY' * LABSETY'.
DECSETY :: DECS ; EMPTY.	DECSETY' :: DECS ; empty.
LABSETY :: LABS ; EMPTY.	LABSETY' :: LABS ; empty.
	empty :: nil.

Now all hypernotations are replaced with their images, e.g.

MODE1 NEST enquiry clause defining new DECSETY2	is replaced with
MODE1 NEST enquiry clause defining new * DECSETY2'	* empty.

Notice that the type transformation is not possible without prior transformation of the grammar. For example, the hypernotation where (PROPSETY) is (DECS DECSETY LABSETY) is

not equivalent to `where (PROPSETY) is (DECS DECSETY' LABSETY')`. Clearly, in a transparent grammar splitting and building of trees does model the splitting and concatenation of text values affixes. In §3.5 and §4.2.4 we will continue this subject.

### 3.4 Further Transformation Rules

Due to the syntactic simplicity of grammatical formalisms, it is not possible to formulate general rules from predicate calculus, set theoretic rules or rules derived from the algebraic specification of data types. Transformation rules of these kinds are available in richer formalism such as CIP-L [Par90]. In some two level grammars, such rules can be simulated by proving equivalences between certain predicates. As an example, some rules using equality tests, some 'logic' rules and some rules operating on lists which may be useful for many specifications are given.

#### 3.4.1 Transformations using Equality Tests

Assume the existence of a predicate `where THING1 is THING2` implementing the equality test. Obviously, the following properties hold:

```

where THING1 is THING2 ≡
    where THING2 is THING1 (symmetry)
where THING is THING ≡
    where true (reflexivity)
where THING1 is THING2, where THING2 is THING3 ⇒
    where THING1 is THING3 (transitivity)
where THING1 is THING2 ≡
    where THING THING1 is THING THING2

```

#### 3.4.2 Transformations on Lists

Suppose that `LIST :: LIST ITEM ; ITEM.` and  $G_{LIST}$  is unambiguous. Then

```

where (LIST1 ITEM1) is (LIST2 ITEM2) ≡
    where (LIST1) is (LIST2), where (ITEM1) is (ITEM2)
where (ITEM) is (LISTETY ITEM1 LISTETY1) ≡
    where (ITEM) is (ITEM1)

```

Application of such rules will be referred to with the general term *recomposition*.

**Logic Rules**

Any number of logic rules can be formulated. For example:

$$\begin{aligned} \text{where } \text{THING}, \text{ unless } \text{THING} &\equiv \text{where } \text{false} \\ (\text{where } \text{THING}; \text{ unless } \text{THING}) &\equiv \text{where } \text{true} \end{aligned}$$

(Let the round brackets denote implicit abstraction.)

Suppose that  $\text{where } \text{THING}_1 \Rightarrow \text{where } \text{THING}_2$ . By invariant introduction:

$$\text{where } \text{THING}_1 \equiv \text{where } \text{THING}_1, \text{ where } \text{THING}_2$$

In ordered grammars:

$$\begin{aligned} \text{where } \text{THING} &\equiv (\text{unless } \text{THING}, \text{ where } \text{false} ; \text{ where } \text{true}) \\ \text{unless } \text{THING} &\equiv (\text{where } \text{THING}, \text{ where } \text{false} ; \text{ where } \text{true}) \end{aligned}$$

Furthermore, if in an ordered grammar

$$\text{where } \text{THING}_1 \Leftrightarrow \neg \text{where } \text{THING}_2$$

then

$$\text{where } \text{THING}_1, \text{ alpha}; \text{ where } \text{THING}_2, \text{ beta} \equiv \text{where } \text{THING}_1, \text{ alpha}; \text{ beta}$$
**3.5 Sample Developments**

In the first two examples (§3.5.1, §3.5.2), general predicates for list concatenation and splitting are derived. Due to a different direction of flow, it is necessary to have two different predicates. These predicates allow lists to be implemented by tree structures instead of strings. Nondeterministic splitting of strings is simulated by ambiguity on the first level. In the resulting predicates, affix values are split only in accordance with the metagrammar. Therefore, these predicates are useful in making a grammar transparent.

In the third example (§3.5.3), the application of both predicates is demonstrated.

**3.5.1 Concatenation of Lists**

After rule introduction it is possible to derive

$$\begin{aligned} \text{where } (\text{LISTETY}_1) \text{ concatenated with } (\text{LISTETY}_2) \text{ is } (\text{LISTETY}): \\ \text{where } (\text{LISTETY}_1 \text{ LISTETY}_2) \text{ is } (\text{LISTETY}). \\ \equiv \{\text{Case introduction (by unfolding LISTETY}_2 \text{ and then LIST)}\} \end{aligned}$$



where (EMPTY) concatenated with (LIST ITEM) is (LIST ITEM) : .  
 $\equiv \{\text{Fold ITEM and LIST ITEM to LIST}\}$   
 where (EMPTY) concatenated with (LIST) is (LIST) : .

The final grammar reads, after transformation to tree structures and insertion of flow symbols:

where (>LISTETY) concatenated with (>EMPTY) is (LISTETY>) : .  
 where (>LIST) concatenated with (>ITEM) is (LIST \* ITEM>) : .  
 where (>LIST2) concatenated with (>LIST \* ITEM) is (LIST1 \* ITEM>) :  
   where (LIST2) concatenated with (LIST) is (LIST1).  
 where (>EMPTY) concatenated with (>LIST) is (LIST>) : .

### 3.5.2 Splitting of Lists

Again, after rule introduction, it is possible to derive

where (LIST) splits to (LISTETY1 ITEM1 LISTETY2) :  
   where (LIST) is (LISTETY1 ITEM1 LISTETY2).  
 $\equiv \{\text{Unfold LIST}\}$

(1) where (ITEM) splits to (LISTETY1 ITEM1 LISTETY2) :  
   where (ITEM) is (LISTETY1 ITEM1 LISTETY2).

(2) where (LIST ITEM) splits to (LISTETY1 ITEM1 LISTETY2) :  
   where (LIST ITEM) is (LISTETY1 ITEM1 LISTETY2).

Focus on rule 1:

where (ITEM) splits to (LISTETY1 ITEM1 LISTETY2) :  
   where (ITEM) is (LISTETY1 ITEM1 LISTETY2).  
 $\equiv \{\text{Restrict LISTETY1 and LISTETY2 to EMPTY}\}$

where (ITEM) splits to (EMPTY ITEM1 EMPTY):  
   where (ITEM) is (EMPTY ITEM1 EMPTY).  
 $\equiv \{\text{Simplify by unfolding EMPTY and predicate removal}\}$

where (ITEM) splits to (EMPTY ITEM EMPTY) : .

Next, focus on rule 2:

where (LIST ITEM) splits to (LISTETY1 ITEM1 LISTETY2) :  
   where (LIST ITEM) is (LISTETY1 ITEM1 LISTETY2).  
 $\equiv \{\text{Unfold LISTETY2, Unfold EMPTY, Predicate removal}\}$

where (LIST ITEM) splits to (LIST ITEM EMPTY) : .

(3) where (LIST ITEM) splits to (LISTETY1 ITEM1 ITEM2) :  
   where (LIST ITEM) is (LISTETY1 ITEM1 ITEM2).

(4) where (LIST ITEM) splits to (LISTETY1 ITEM1 LIST2 ITEM2) :  
   where (LIST ITEM) is (LISTETY1 ITEM1 LIST2 ITEM2).

Now focus on rule 3:

```

where (LIST ITEM) splits to (LISTETY1 ITEM1 ITEM2) :
  where (LIST ITEM) is (LISTETY1 ITEM1 ITEM2).
                                                    ≡ {Distribution of equality test}
where (LIST ITEM) splits to (LISTETY1 ITEM1 ITEM2) :
  where (LIST) is (LISTETY1 ITEM1),
  where (ITEM) is (ITEM2).
                                                    ≡ {Predicate removal, Fold EMPTY}
where (LIST ITEM) splits to (LISTETY1 ITEM1 ITEM) :
  where (LIST) is (LISTETY1 ITEM1 EMPTY).
                                                    ≡ {Fold hypernotation}
where (LIST ITEM) splits to (LISTETY1 ITEM1 ITEM) :
  where (LIST) splits to (LISTETY1 ITEM1 EMPTY).

```

Finally, rule 4 is treated:

```

where (LIST ITEM) splits to (LISTETY1 ITEM1 LIST2 ITEM2) :
  where (LIST ITEM) is (LISTETY1 ITEM1 LIST2 ITEM2).
                                                    ≡ {Distribution of equality test, Predicate removal}
where (LIST ITEM) splits to (LISTETY1 ITEM1 LIST2 ITEM) :
  where (LIST) is (LISTETY1 ITEM1 LIST2).
                                                    ≡ {Fold hypernotation}
where (LIST ITEM) splits to (LISTETY1 ITEM1 LIST2 ITEM) :
  where (LIST) splits to (LISTETY1 ITEM1 LIST2).

```

Transformation to tree structures and insertion of commas and flow symbols yields the following grammar:

```

where (>ITEM) splits to (EMPTY>, ITEM>, EMPTY>) : .
where (>LIST * ITEM) splits to (LIST>, ITEM>, EMPTY>) : .
where (>LIST * ITEM) splits to (LISTETY1>, ITEM1>, ITEM>) :
  where (LIST) splits to (LISTETY1, ITEM1, EMPTY).
where (>LIST * ITEM) splits to (LISTETY1>, ITEM1>, LIST2 * ITEM>) :
  where (LIST) splits to (LISTETY1, ITEM1, LIST2).

```

### 3.5.3 Ravelling of Moods

In [Wij76] the following predicate is defined:

```

where MOIDS ravels to MOODS :
  where (MOIDS) is (MOODS).
where MOIDS ravels to MOODS :
  where (MOIDS) is (MOODSETY union of MOODS1 mode MOIDSETY),
  where MOODSETY MOODS1 MOIDSETY ravels to MOODS.

```

With predicate introduction, this is transformed into

```

where MOIDS ravel to MOODS :
  where (MOIDS) is (MOODS).
where MOIDS ravel to MOODS :
  where (MOIDS) is (MOODSETY union of MOODS1 mode MOIDSETY),
  where (MOODSETY MOODS1) is (MOODS2),
  where (MOODS2 MOIDSETY) is (MOIDS2),
  where MOIDS2 ravel to MOODS.

```

Notice the splitting and concatenation of lists. After folding and imaging into EAG (see §4.1), the following syntax is obtained:

```

where (>MOIDS) ravel to (MOODS>) :
  where (MOIDS) is (MOODS) ;
  where (MOIDS) splits to (MOODSETY, union of * MOODS1 * mode, MOIDSETY),
  where (MOODSETY) concatenated with (MOODS1) is (MOODS2),
  where (MOODS2) concatenated with (MOIDSETY) is (MOIDS2),
  where (MOIDS2) ravel to (MOODS).

```

Notice that efficiency of ravelling can be improved in two ways. First of all, MOIDS is split in all possible ways, but due to meta-parsing only one split will succeed. Secondly, MOODSETY is concatenated to the parameter of the recursive call, which is unnecessary. Clearly, the following (deterministic) syntax is equivalent:

```

where (>MOID) ravel to (MOOD>) :
  where (MOID) is (MOOD).
where (>MOID) ravel to (MOODS>) :
  where (MOID) is (union of * MOODS * mode).
where (>MOIDS * MOID) ravel to (MOODS>) :
  where (MOIDS) ravel to (MOODS1),
  where (MOID) ravel to (MOODS2),
  where (MOODS1) concatenated with (MOODS2) is (MOODS).

```

Until now, the author has not succeeded in deriving this syntax using an unfold/fold strategy.

## 3.6 Conclusions

Left-recursion removal, substitution and left-factorization are well known techniques for making a context free grammar ‘more’  $LL(1)$ . It has been shown that these transformations can be generalized to two level grammars.

The transformations introduced in this chapter are sufficient for applying a simple unfold/fold-strategy to the hyper-rules of a two level grammar. It is interesting to investigate what more transformations known from formal program development can be applied successfully to two level grammars. However, two level grammars suffer from the disadvantage of their notation not being akin to traditional mathematics.

One may observe that due to the referencing problem, applicability of most of the transformations defined in this chapter is undecidable. However, in a transparent grammar applicability of most transformations is decidable. We have not worked out this question.

We have not investigated which transformations preserve well-formedness of a two level grammar. We remark that the development process is steered by the developers intentions concerning affix flow (see §3.5).

## Chapter 4

# An Extended Affix Grammar for Algol 68

In [Kos69] Koster proposed a method of transcribing a 2VWG literally into some kind of affix grammar and explores the application of this technique to Algol 68. In this chapter it will be shown how this path can be followed to its final extent, resulting in an EAG for Algol 68 which is derived from the description in 2VWG in a straightforward way until a point is met where a fundamental difference between the EAG and 2VWG formalisms becomes apparent.

In the first section, a method of imaging a 2VWG into EAG is defined. In general, imaging must be preceded by transformations yielding some equivalent grammar whose image has a better operational behavior. In §4.2, an overview of the derivation process of an EAG for Algol 68 is given. Experiences with the EAG-compiler are described in §4.3. The last section contains the conclusions which can be drawn from the work described in this chapter.

### 4.1 Imaging 2VWG into EAG

DEFINITION 4.1.1 Two level grammars  $G$  and  $G'$  are *strongly equivalent*, if they generate the same language and there exists a bijection between their parse trees.

Every 2VWG can be transformed into a strongly equivalent EAG by the construction of an EAG whose underlying context free grammar is a *pattern grammar* [Mał84] or *skeleton grammar* [Weg80].

DEFINITION 4.1.2 The *cross reference relation* [Mał84]  $\sim$  on  $H$  is defined as follows:  $h \sim h'$ , if  $h$  matches  $h'$ . The symmetric and transitive closure of  $\sim$  is an equivalence relation on  $H$ . Let  $[h]$ , the *class* of  $h$ , denote the equivalence class of  $h$ . Notice that  $[h]$  is closed under complaint substitution.

THEOREM 4.1.1 Let  $G = (M, V, T, P_M, P_H, S)$  be a 2VWG with basic hypernotations  $B$ . Assume that the representation of symbols has been replaced with a lexical grammar (see §4.2.8).

Construct an EAG  $G' = (A_N, A_T, V_N, V_T, P_A, P'_H, C, S')$  with

$$\begin{aligned}
A_N &= M \\
A_T &= V \\
V_N &= \{[h](h) \mid h \in B \setminus T\} \\
V_T &= T \\
P_A &= P_M \\
P'_H &= \{h'_0 : h'_1, \dots, h'_n \mid h_0 : h_1, \dots, h_n \in P_H\} \text{ where } h'_i = \begin{cases} [h_i](h_i) & \text{if } h_i \in B \setminus T \\ h_i & \text{if } h_i \in T \end{cases} \\
S' &= [S](S)
\end{aligned}$$

Then  $G$  and  $G'$  are strongly equivalent.

However, the construction above does not follow the intuition expressed in [Kos69]. For that, another construction is defined. Our intention is to split up hypernotations in a first level part and a second level part. First, we will characterize the possible underlying context free grammars for a given 2VWG.

**DEFINITION 4.1.3** A string  $\alpha_0 \dots \alpha_n \in V^+$  is a *separation* of  $[h]$  if for all basic hypernotations  $g, g' \in [h]$  there exist  $\beta_1, \dots, \beta_n \in (V \cup M)^*$  with

$$\begin{aligned}
g &= \alpha_0 \beta_1 \alpha_1 \dots \beta_n \alpha_n \\
g' &= \alpha_0 \beta'_1 \alpha_1 \dots \beta'_n \alpha_n
\end{aligned}$$

and for all conjugations  $(\theta, \theta')$  of  $(g, g')$  it holds that  $\theta(\beta_i) = \theta'(\beta'_i)$  for  $1 \leq i \leq n$ .

Separations are to be considered potential nonterminals of an underlying context free grammar, while each  $\beta_i$  is a potential affix expressions occurring on the  $i$ -th affix position. A 2VWG  $G$  is called *separated* if for each  $[h]$  there exists separation.

**CONJECTURE 4.1.1** A transparent grammar is separated. Unfold metanotions in the alternatives of the main metanotion until each non-empty alternative of contains at least one terminal symbol. Then the terminals in each alternative of the main metanotion are a separation for each class of basic hypernotations. When necessary, metanotions in the basic hypernotations are also unfolded.

**THEOREM 4.1.2** Let  $G$  be a separated 2VWG. Construct an EAG  $G'$  as in theorem 4.1.1, except for  $P'_H$ . Construct  $P'_H$  as follows:

$$P'_H = \{h'_0 : h'_1, \dots, h'_n \mid h_0 : h_1, \dots, h_n \in P_H\}$$

where each  $h'_i = h_i$  if  $h_i \in T$  and

$$h'_i = \alpha_0 \dots \alpha_n(\beta_1, \dots, \beta_n)$$

if

$$h_i = \alpha_0\beta_1\alpha_1\dots\beta_n\alpha_n$$

and  $\alpha_0\dots\alpha_n$  is a separation of  $[h_i]$ . Then  $G$  and  $G'$  are strongly equivalent.

#### 4.1.1 Strategy

The following strategy is followed in transforming a 2VWG into EAG:

1. Unfold hypernotations and metanotations until the grammar is well separated.
2. Develop executable versions of all predicates. For this, unused predicates are pruned and predicates defining relations between metanotations are transformed into *functions* defining a mapping between intended inherited and derived affixes. Here, the developer intuition is called upon.
3. Image the grammar into EAG in the way indicated by theorem 4.1.2. The separations should be chosen in such a way that each display consists of a sufficient number of affix positions. Otherwise flow is obstructed.
4. Define the control and check whether the resulting grammar is well-formed.

The strategy requires intuition about which metanotations are superfluous in the sense of [Kos69] and which metanotations are used for enforcing context conditions.

## 4.2 Deriving an EAG for Algol68

As reported in this thesis, an EAG for Algol 68 has been derived using the transformational methods of the previous chapter.

### 4.2.1 Overview of the process

The first concern is to make the grammar separated and to split up every hypernotation into a head and a display. In turn, the display is split up into distinct affix positions. The predicates are simplified and turned into functions where possible. Ambiguity is removed by unfolding and pruning. The next step is to impose tree structured domains onto the metagrammar, requiring the affix expressions to be more consistent with the metagrammar. All indirect left-recursion is transformed into direct left-recursion which is subsequently removed from the grammar, enabling the generation of top-down parsers. Finally, optimizations like left-factoring are applied in order to prevent needless backtracking. However, the resulting grammar contains one trouble spot, where nondeterministic affix assignment still hinders executability (see §4.2.9).

## 4.2.2 Separation of the First and Second Level

In 2VWG hypernotations are conjoint if and only if they have a commonly derived protonotion. The question whether two hypernotations are conjoint is in general undecidable, but for Algol 68 this question has been resolved. In the Revised Report all intersections are indicated by so called *pragmatic remarks*. This information is incorporated into the definition by separating all hypernotations into a head and a display.

There is great liberty in splitting up hypernotations into a head and a display, resulting in many possible underlying context free grammars which can be obtained from any given 2VWG. For example, a **strong void new closed clause** might be imaged onto an instance of (SOID, NEST, ENCLOSED) **clause** or alternatively **strong (MOID, NEST) closed clause**. Nevertheless, (**strong, void, new \* EMPTY \* EMPTY**) **closed clause** which is obtained after folding EMPTY twice has been preferred. This decision is of a purely pragmatic nature. Semantically it is only required that all conjoint hypernotations obtain the same head and the same number of affix positions. Let the example above illustrate our intention and characterize the notions of ‘intended image’, ‘intended head-word’ and ‘intended affix positions’.

Before a 2VWG can be transcribed into EAG in the intended way, some problems need to be resolved. A first problem to be attacked is *punning*. Punning is caused by non-recursive metanotions which are used as abbreviations. Punning has to be avoided for two reasons. First, in many cases intended head-words do not occur on the first level, but are produced by some metanotion. Secondly, affix expressions intended to occur on different affix positions may be abbreviated by a single metanotion.

By unfolding, all non-recursive metanotions can be removed from the grammar, when necessary. In this way, punning can be avoided. Consider for example the hypernotation **SOME UNIT**, not having any first level part at all, in which UNIT is used as an abbreviation for a finite set of constructs.

### EXAMPLE 4.2.1

```

SOME unit : SOME UNIT.
                                                    ≡ {Unfold UNIT}

SOME unit :
  SOME assignation coercee ;
  SOME identity relation coercee ;
  SOME routine text coercee ;
  SOME jump ;
  SOME skip ;
  SOME TERTIARY.
                                                    ≡ {Rule introduction, Fold}

SOME unit :
  SOME assignation coercee ;
  SOME identity relation coercee ;
  SOME routine text coercee ;
  SOME jump ;
  SOME skip ;
  SOME tertiary.

```

SOME tertiary :  
 SOME TERTIARY.

Now all other occurrences of SOME TERTIARY are folded to SOME tertiary. By rule introduction, folding of hyper-rules and unfolding of metanotions all occurrences of ENCLOSED, SECONDARY and PRIMARY are also removed from the grammar in a similar way.

In section 4.2.7, some syntax is derived showing that

$$\begin{array}{c} \text{SORT MOID MORF coercee} \\ \equiv \\ \text{MOID1 NEST MORF, can coerce (MOID1, morf) to (MOID) in (SORT) position} \end{array}$$

and

$$\begin{array}{c} \text{SORT MOID COMORF coercee} \\ \equiv \\ \text{MOID1 NEST COMORF, can coerce (MOID1, comorf) to (MOID) in (SORT) position} \end{array}$$

After unfolding SOME and SOID and application of these properties, the syntax for unit is imaged into EAG as

```
(SORT, MOID, NEST) unit :
  (SORT, MOID1, NEST) assignation,
    can coerce (MOID1, comorf) to (MOID) in (SORT) position ;
  (SORT, MOID1, NEST) identity relation,
    can coerce (MOID1, comorf) to (MOID) in (SORT) position ;
  (SORT, MOID1, NEST) routine text,
    can coerce (MOID1, morf) to (MOID) in (SORT) position ;
  (SORT, MOID, NEST) jump ;
  (SORT, MOID, NEST) skip ;
  (SORT, MOID, NEST) tertiary.
```

Notice that due to unfolding SOME and SOID, the affixes SORT, MOID and NEST are enabled to occur on distinct affix positions. This is necessary because a NEST is always inherited, while MOID may be derived. Furthermore, the predicate for coercion requires MOID to be inherited in strong and firm positions and derived in meek, weak and soft positions. Therefore, the flow of MOID in a unit depends dynamically on the value of the affix SORT. Here, the grammar does take advantage of free affix flow. Statically fixed flow requires unfolding of SORT, resulting in a much bigger grammar, or in major changes in the handling of modes (see next chapter).

A second problem in deriving an EAG is posed by the 'syntax for general constructions' and 'syntax of general predicates', where the metanotions NOTION and THING simulate the lifting of intended first level parts of hypernotions to the second level. Therefore these rules are removed. This is made possible by introducing specialized rules and by unfolding hyper-rules.

EXAMPLE 4.2.2

SOID NEST series with PROPSETY :  
 where (PROPSETY) is (LAB LABSETY) and SOID balances SOID1 and SOID2,  
 ....  
≡ {Unfold and}

SOID NEST series with PROPSETY :  
 where (PROPSETY) is (LAB LABSETY),  
 where SOID balances SOID1 and SOID2, ...;  
 where (PROPSETY) is (LAB LABSETY) and SOID balances SOID1, where SOID2,  
 ....  
≡ {Delete blind alley}

SOID NEST series with PROPSETY :  
 where (PROPSETY) is (LAB LABSETY),  
 where SOID balances SOID1 and SOID2, ... .

Notice that `and` is unfolded using two conjugations. In this case, a most general unifier does not exist. The blind alley caused by an unintended conjugation has been eliminated.

By unfolding hypernotations and pruning, ambiguity can be removed, making predicates deterministic.

EXAMPLE 4.2.3 After unfolding `WHETHER` one obtains

where MOID1 is firm MOID2 :  
 where MOID1 equivalent MOID2 or MOID1 unites to MOID2 or  
 MOID1 deprefs to firm MOID2.  
≡ {Unfold or}

where MOID1 is firm MOID2 :  
 where MOID1 equivalent MOID2 ;  
 where MOID1 unites to MOID2 or MOID1 deprefs to firm MOID2 ;  
 where MOID1 equivalent MOID2 or MOID1 unites to MOID2 ;  
 where MOID1 deprefs to firm MOID2.  
≡ {Unfold or twice}

where MOID1 is firm MOID2 :  
 where MOID1 equivalent MOID2 ;  
 where MOID1 unites to MOID2 ;  
 where MOID1 deprefs to firm MOID2 ;  
 where MOID1 equivalent MOID2 ;  
 where MOID1 unites to MOID2 ;  
 where MOID1 deprefs to firm MOID2.  
≡ {Delete superfluous rules}

where MOID1 is firm MOID2 :  
 where MOID1 equivalent MOID2 ;  
 where MOID1 unites to MOID2 ;  
 where MOID1 deprefs to firm MOID2.

Now straightforward imaging into EAG has become possible. Furthermore, the predicate `is firm` is no longer ambiguous and therefore deterministic.

After imaging into EAG, it may be necessary to rename some hypernotations. Consider for example the hypernotation `actual-NEST-PARAMETERS` which is imaged onto `actual (NEST, PARAMETERS)`. For pragmatic reasons this hypernotation is renamed into `actual parameters (NEST, PARAMETERS)`.

### 4.2.3 From Relations to Functions

For reasons of generality of description, every definition of a predicate in [Wij76] begins with an affix `WHETHER`, producing `where` or `unless`, allowing some relation and its negation to be defined by a single predicate. It is necessary to specialize these predicates to both forms. Since many predicates are used in one form only in any actual production, many unused forms can then be pruned from the grammar.

Now the grammar contains two special predicates `where THING1 is THING2` and `unless THING1 is THING2`, defining the equality and inequality relation on strings. The inequality relation is implemented in EAG by the primitive predicate `not equal`. The equality relation can be implemented by the primitive predicate `equal`, but in order to obtain an executable specification one must let the consistent substitution rule implement the equality test:

```
where THING1 is THING2 : .
```

This version of `is` is a *function* yielding an affix value as soon as the other affix is known. This is an essential step, because in this way most predicates in the `where`-form become functions calculating derived affixes from inherited ones.

In order to guarantee the termination of most predicates, critical affix positions must be marked as inherited. If too many affix positions are marked critical, predicates cannot act like functions, resulting in a specification which is not well-formed. Flow should be chosen in such a way that each predicate relates a finite number of affix values to each combination of critical affix values.

### 4.2.4 From Flat Domains to Tree Structures

The run-time behavior of an EAG can be made more efficient by making the splitting of affix values deterministic. This requires all splits to follow direct metaproductions and the metagrammar to be unambiguous. If this is the case, the metagrammar can be implemented with tree structures and typing can be performed statically.

Notice that ambiguity in the metagrammar is redundant, since it is not used to model properties of a language. The metagrammar for Algol 68 has been made unambiguous by unfolding and pruning. Furthermore, the metagrammar has been implemented in EAG with tuples (tree structures).

Implementing the metagrammar with tuples (a type transformation) requires all matching affix expressions to be of the same type `n-tuple`. This can be accomplished by using predicate introduction or the predicates developed in §4.2.7 for the stepwise composition or decomposition of affixes. Furthermore, all metanotions like `REFETY` and `REFLEXETY` have to be unfolded.

In general, where  $(M_1)$  is  $(M_2 + M_3)$  is not equivalent to where  $(M_1)$  is  $(M_2 * M_3)$ , unless  $M_1$  produces  $M_2 * M_3$  in the metagrammar. If this is not the case, a predicate where  $(M_1)$  is  $(M_2, M_3)$  (a specialization of the equality test) is developed, using a strategy of case introduction, predicate introduction and possibly folding.

EXAMPLE 4.2.4 Define

```
MODE :: INTREAL.
INTREAL :: SIZETY * integral ;
          SIZETY * real.
```

Note that  $SIZE * INTREAL$  [Wij76, 810a] is not a mode. Therefore, the following predicate is developed:

```
where (MODE) is (SIZE, INTREAL) :
  where (MODE) is (SIZE INTREAL).
                                                                    ≡ {Unfold INTREAL}

where (MODE) is (SIZE, SIZETY integral) :
  where (MODE) is (SIZE SIZETY integral).

where (MODE) is (SIZE, SIZETY real) :
  where (MODE) is (SIZE SIZETY real).
                                                                    ≡ {Predicate introduction}

where (MODE) is (SIZE, SIZETY integral) :
  where (MODE) is (SIZETY1 integral),
  where (SIZETY1) is (SIZE SIZETY).

where (MODE) is (SIZE, SIZETY real) :
  where (MODE) is (SIZETY1 real),
  where (SIZETY1) is (SIZE SIZETY).
```

Because  $SIZE * SIZETY$  is not a  $SIZETY$ , a predicate for adding sizes is developed:

```
where (SIZETY1) is (SIZE, SIZETY) :
  where (SIZETY1) is (SIZE SIZETY).
                                                                    ≡ {Unfold SIZETY, Unfold SIZE, Delete blind alleys}

where (SIZETY1) is (long, long LONGSETY) :
  where (SIZETY1) is (long long LONGSETY).

where (SIZETY1) is (short, short SHORTETY) :
  where (SIZETY1) is (short short SHORTETY).
                                                                    ≡ {Predicate introduction}

where (SIZETY1) is (long, long LONGSETY) :
  where (SIZETY1) is (long LONGSETY1),
  where (LONGSETY1) is (long LONGSETY).

where (SIZETY1) is (short, short SHORTETY) :
  where (SIZETY1) is (short SHORTETY1),
  where (SHORTETY1) is (short SHORTETY).
```

Now a type transformation to tuples is possible, yielding:

```

where (MODE) is (SIZE, SIZETY * integral) :
  where (MODE) is (SIZETY1 * integral),
    where (SIZETY1) is (SIZE, SIZETY).
where (MODE) is (SIZE, SIZETY * real) :
  where (MODE) is (SIZETY1 * real),
    where (SIZETY1) is (SIZE, SIZETY).
where (SIZETY1) is (long, long * LONGSETY) :
  where (SIZETY1) is (long * LONGSETY1),
    where (LONGSETY1) is (long * LONGSETY).
where (SIZETY1) is (short, short * SHORTETY) :
  where (SIZETY1) is (short * SHORTETY1),
    where (SHORTETY1) is (short * SHORTETY).

```

Furthermore, one must beware of metanotions producing `EMPTY`. For example, `integral` is not a mode, but `EMPTY * integral` which obtained by folding `EMPTY` is.

Notice that the syntax has been made transparent. Furthermore, notice that in this case the predicates developed can be simulated by conjugation rules in a conjugation grammar. However, no finite set of conjugation rules for the hypernotions `where-MOIDS-is-MOODS` and the (specialized) equality test `where-MOIDS-is-MOIDS` does exist. But using a strategy of case introduction and folding, we can derive

```

is moody (MOIDS, MOODS) :
  where (MOIDS) is (MOODS).
                                                    ≡ {Unfold MOIDS, Unfold MOODS and Pruning}

is moody (MOID, MOOD) :
  where (MOID) is (MOOD).

is moody (MOIDS MOID, MOODS MOOD) :
  where (MOIDS MOID) is (MOODS MOOD).
                                                    ≡ {Distribution of equality test}

is moody (MOID, MOOD) :
  where (MOID) is (MOOD).

is moody (MOIDS MOID, MOODS MOOD) :
  where (MOIDS) is (MOODS), where (MOID) is (MOOD).
                                                    ≡ {Folding twice}

is moody (MOID, MOOD) :
  where (MOID) is (MOOD).

is moody (MOIDS MOID, MOODS MOOD) :
  is moody (MOIDS, MOODS), is moody (MOID, MOOD).

```

A transparent version of `is moody (MOID, MOOD)` is derived by unfolding `MOID`, then `MODE`, and pruning.

### Strategy

It has been exemplified how a two level grammar can be made transparent (a conjugation grammar is obtained by the construction in conjecture 2.1.1). The following strategy is

formulated for making two level grammars transparent:

1. Make the metagrammar unambiguous, using a strategy of unfolding and pruning.
2. Unify hypernotations in the same class by predicate introduction.
3. Make the predicates transparent using a strategy of case introduction and folding.
4. Extend the the metagrammar with a main metanotion producing  $h$  for each  $[h]$ .

If all equality tests are transparent and flow is defined, the equality tests can be replaced with guards.

### 4.2.5 Left-recursion Removal

Left-recursion removal allows the generation of top-down parsers. Top-down parsers are small and can be obtained easily from a grammar [Kos74].

Left-recursion is removed from the grammar for Algol 68 by transforming the left-recursion into direct left-recursion, which is easily removed. Transforming all indirect left-recursion into direct left-recursion is not trivial, because repeated of unfolding of `coercees` loops. Therefore, predicates for coercion are derived. The process is illustrated by the examples in §4.2.7.

### 4.2.6 Optimizations

The efficiency of parsers generated from Extended Affix Grammars can be improved by using a tree structured metagrammar (§4.2.4), rather than the original string-structured metagrammar. Other optimizations are to make predicates deterministic (example 4.2.3), removal of superfluous parameters and left-factorization. In the absence of affix directed parsing, hypernotations which are not conjoint are given different head-words.

### Left-factoring Two Level Grammars

Not every context free grammar can be made  $LL(1)$  by repeated unfolding and left-factoring, because the process may loop. In the grammar for Algol 68, repeated unfolding and left-factoring loops when it is tried to remove the prefix sharing of `portrait` and `joined-portrait` in a `collateral-clause`. A more general method for removing prefix sharing is introduced. Consider the following grammar (due to Alblas, cited in [Kos92]):

```

a : b ; c.
b : f, b, g ; d.
c : f, c, h ; e.

```

Here, the process of repeated substitution and left-factorization loops, although an equivalent  $LL(1)$  syntax does exist. By viewing this grammar as a two level grammar, the syntax can be left-factored, using a strategy of rule introduction and folding.

EXAMPLE 4.2.5 Introduce the following rules :

```
BC :: b ; c.
bc (b) : b.
bc (c) : c.
```

It is not essential that BC produces *b* and *c*. It is only required that both alternatives are disjunct. Now

```
a : b ; c.
a : bc (BC).
```

≡ {Fold hyper-rules, fold metanotation}

and

```
bc (b) : b.
bc (c) : c.
```

≡ {Unfold hypernotations *b* and *c*}

```
bc (b) : f, b, g ; d.
bc (c) : f, c, h ; e.
```

≡ {Fold hypernotation twice}

```
bc (b) : f, bc (b), g ; d.
bc (c) : f, bc (c), h ; e.
```

≡ {Predicate introduction}

```
bc (BC) :
  f, bc (BC), g, where (BC) is (b) ;
  d, where (BC) is (b) ;
  f, bc (BC), h, where (BC) is (c) ;
  e, where (BC) is (c).
```

≡ {Left-factoring}

```
bc (BC) :
  f, bc (BC), rest bc (BC) ;
  d, where (BC) is (b) ;
  e, where (BC) is (c).
```

```
rest bc (BC) :
  g, where (BC) is (b) ;
  h, where (BC) is (c).
```

If  $g = h = \varepsilon$ , as in the original example, then `rest bc (BC)` can be eliminated by symbolic evaluation.

The same technique is applicable to two level grammars. After left-recursion removal, the syntax for `portrait` and `joined-portrait` in [Wij76, 331] can be left-factored in a similar way, after introducing the metarule `JOINETY :: joined ; empty..` The EAG for Algol 68 has not been left-factored completely, because this would severely mutilate the structure of the grammar.

### 4.2.7 Sample Derivations

The derivation of an EAG from a 2VWG is illustrated by two representative examples. First, predicates for performing coercion are derived. The second example shows the derivation of a top down parsable syntax for formulas.

#### Coercion

The syntax for coercion as given in the Revised Report poses several problems. First of all, there are many hypernotations which do not have a first level head. Secondly, the grammar hides left-recursion which is to be removed. A third problem is how to guarantee termination of the recognizers generated from the grammar. In the syntax as it is, parsers will get lost in infinite derivations, because the underlying context free grammar is cyclic and termination cannot be enforced by affix directed parsing.

In this section, a top down parsable syntax specifying the coercions of Algol 68 is derived. This results in a predicate, testing whether the a priori mode of a construct in a certain syntactic position can be coerced to some a posteriori mode. In meak, weak and soft positions, the predicate generates a finite number of possible a posteriori modes. Flow is bidirectional, which is implemented by taking advantage of the feature of free affix flow.

The derivation is not given in full detail. Instead, the method followed is exemplified by deriving a new syntax for coercions in soft positions. The rest of the syntax is obtained analogously.

First, some new rules are introduced

```
strong MOID FORM : STRONG MOID FORM.
firm MOID FORM : FIRM MOID FORM.
meek MOID FORM : MEEK MOID FORM.
soft MOID FORM : SOFT MOID FORM.
```

which are used for folding. For example

```
softly deprocedured to MODE FORM :
  SOFT procedure yielding MODE FORM.
                                     ≡ {Fold soft MOID FORM}

softly deprocedured to MODE FORM :
  soft procedure yielding MODE FORM.
```

By unfolding SOFT in soft MOID FORM one obtains

```
soft MOID FORM :
  unchanged from MOID FORM ;
  softly deprocedured to MOID FORM.
≡ {Unfold unchanged-from, Unfold softly-deprocedured-to, Predicate introduction}
soft MOID FORM :
  MOID FORM ;
```

```

soft MOID1 FORM,
  where (MOID1) is (procedure yielding MODE),
  where (MOID) is (MODE).

```

≡ {Remove left-recursion}

```

soft MOID FORM :
  MOID1 FORM,
  rest soft FORM (MOID, MOID1).
rest soft FORM (MOID, MOID) : .
rest soft FORM (MOID, MOID1) :
  where (MOID1) is (procedure yielding MODE),
  where (MOID2) is (MODE),
  rest soft FORM (MOID, MOID2).

```

Notice that left-recursion removal is possible, although the rule is cyclic. The original rule has a non left-recursive kernel MOID-FORM, followed by  $\varepsilon$ -productions. However, the number of  $\varepsilon$ -productions is determined exactly by the affix values chosen.

We continue our derivation. Renaming, predicate introduction and elimination yields

```

soft MOID FORM :
  MOID1 FORM,
  can coerce (MOID1) to (MOID) in soft position.
can coerce (MOID1) to (MOID) in soft position :
  where (MOID1) is (MOID) ;
  where (MOID1) is (procedure yielding MODE),
  can coerce (MODE) to (MOID) in soft position.

```

≡ {Rule introduction, Fold}

```

soft MOID FORM :
  MOID1 FORM,
  can coerce (MOID1) to (MOID) in soft position.
can coerce (MOID1) to (MOID) in soft position :
  where (MOID1) is (MOID) ;
  where (MOID1) softly deprocedures to (MOID).
where (MOID1) softly deprocedures to (MOID) :
  where (MOID1) is (procedure yielding MODE),
  can coerce (MODE) to (MOID) in soft position.

```

In a similar way, the syntax for weak, meak, firm and strong MOID FORM is treated.

Now focus attention on the syntax for soft-MODE-FORM-coercee:

```

soft MODE FORM coercee :
  SOFT MODE FORM.

```

≡ {Fold to soft MODE FORM}

```

soft MODE FORM coercee :
  soft MODE FORM.

```

≡ {Unfold soft MODE FORM}

```

soft MODE FORM coercee :

```

```

MOID1 FORM,
  can coerce (MOID1) to (MODE) in soft position.
                                                    ≡ {Rule introduction, Fold}

soft MODE FORM coercee :
  MOID1 FORM,
    can coerce (MOID1, FORM) to (MODE) in (soft) position.
can coerce (MOID1, FORM) to (MODE) in (soft) position :
  can coerce (MOID1) to (MODE) in soft position.
                                                    ≡ {Replace MODE with MOID (inverse of restriction)}

soft MOID FORM coercee :
  MOID1 FORM,
    can coerce (MOID1, FORM) to (MOID) in (soft) position.
can coerce (MOID1, FORM) to (MODE) in (soft) position :
  can coerce (MOID1) to (MODE) in soft position.

```

The syntax for weak-, meak-, firm- and strong-MOID-FORM-coercee is also treated in a similar way. Finally, folding SORT results in

```

          SORT MOID FORM coercee
                ≡
MOID1 FORM, can coerce (MOID1, FORM) to (MOID) in (SORT) position

```

Now that coercion is performed by predicates, critical affix positions must be indicated in order to guarantee termination. It is easy to verify that a terminating syntax can be found by letting soft, weak and meek coercions derive a posteriori modes from some critical a priori mode, while in firm and strong positions both a priori and a posteriori modes must be marked critical. This results in the subtle intertwinement of for example

```

can coerce (>MOID) to (>UNITED) in firm position :
  can coerce (>MOID) to (MOID1>) in meek position,
  where (>MOID1) unites to (>UNITED).

```

Sometimes the directionality of flow contradicts the interpretation which is suggested by the name of a predicate, as for example in

```

where (MODE>) is rowed to (>EMPTY * row * of * MODE) : .

```

In can coerce (MOID1, FORM) to (MOID) in (SORT) position the flow of MOID is left unspecified. Here, flow is bidirectional instead of undetermined until runtime. The affix FORM is needed to decide whether deproceduring is permitted before voiding in strong positions. The metarule for FORM can be replaced with FORM :: morf ; comorf..

## Formulas

The derivation of the new syntax for formulas is particularly interesting, because of the non-deterministic nature of the original syntax. The derivation involves techniques for transforming indirect left-recursion into direct left-recursion, left-recursion removal, simplification

of conjugations and preventing the non-deterministic choice of affix values by suggesting a correct choice.

First, in `operand`, `SECONDARY` is folded to `secondary`, and the coercion transformation of the previous example is applied to `formula-coercee`. Now it is possible to make `formula` directly left-recursive by unfolding its first operand. However, rash unfolding of `MODE1-NEST-DYADIC-TALLETY-operand` in the rule for `DYADIC-formula` (see [Wij76, 542a]) will result in 90 rules (45 conjugations by two alternatives). Furthermore, the affix `DYADIC` of `DYADIC-formula` contains the minimal priority of the operators in a `formula`. Consequently, the priority of each `ADIC-operand` is chosen non-deterministically, causing implementations to backtrack. Notice that the rule for `DYADIC-formula` is of the form `generate (X), test (X)`, which explains its inefficiency.

The main problem in the derivation is to *invert* the imaginary flow of `ADIC` in `ADIC-operand`. In this way, each `operand` is dictated its minimal priority, as opposed to generating all possible choices and rejecting unsuccessful ones. For clarity's sake, in our presentation the affixes `SORT` and `MODE` and the predicates for coercion and identification are elided. It is trivial to perform the same derivation while preserving all context conditions, but in the following abstract derivation it is easier to focus on the main problem. The derivation is given in detail, because it exemplifies the application of many transformations.

Let us calculate:

```

DYADIC formula :
  DYADIC TALLETY operand,
  DYADIC operator,
  DYADIC TALLY operand.
                                                    ≡ {Invariant introduction of (1) and (2)}

DYADIC formula :
  DYADIC TALLETY operand,
(1)  where (ADIC1) is (DYADIC TALLETY),
      DYADIC operator,
      DYADIC TALLY operand,
(2)  where (ADIC2) is (DYADIC TALLY).
                                                    ≡ {Apply properties (1) and (2)}

DYADIC formula :
  ADIC1 operand ,
  where (ADIC1) is (DYADIC TALLETY),
  DYADIC operator,
  ADIC2 operand,
  where (ADIC2) is (DYADIC TALLY).
                                                    ≡ {where (ADIC2) is (DYADIC TALLY) ≡
where (ADIC2) is (DYADIC i TALLETY2)}

DYADIC formula :
  ADIC1 operand ,
  where (ADIC1) is (DYADIC TALLETY),
  DYADIC operator,
  ADIC2 operand,
  where (ADIC2) is (DYADIC i TALLETY2).
                                                    ≡ {Rule introduction of (3), Fold twice}

```

DYADIC formula :  
 DYADIC new operand,  
 DYADIC operator,  
 DYADIC i new operand.

- (3) ADIC new operand :  
 ADIC1 operand, where (ADIC1) is (ADIC TALLETY).

In the same way, MONADIC-formula is treated:

MONADIC formula :  
 monadic operator,  
 MONADIC operand.

≡ {Invariant introduction, Apply invariant}

MONADIC formula :  
 monadic operator,  
 ADIC1 operand, where (ADIC1) is (MONADIC TALLETY).

≡ {Fold new operand}

MONADIC formula :  
 monadic operator,  
 MONADIC new operand.

Now focus on new-operand (3):

ADIC new operand :  
 ADIC1 operand, where (ADIC1) is (ADIC TALLETY).

≡ {Unfold operand}

ADIC new operand :  
 ADIC1 formula, where (ADIC1) is (ADIC TALLETY) ;  
 where (ADIC1) is (MONADIC), secondary, where (ADIC1) is (ADIC TALLETY).

≡ {Symbolic evaluation}

ADIC new operand :  
 ADIC1 formula, where (ADIC1) is (ADIC TALLETY) ;  
 secondary.

≡ {Rule introduction of (4), Fold}

ADIC new operand :  
 ADIC new formula ;  
 secondary.

- (4) ADIC new formula :  
 ADIC1 formula, where (ADIC1) is (ADIC TALLETY).

After repeated unfolding, we remove the left-recursion from new-formula (4):

ADIC new formula :  
 ADIC1 formula, where (ADIC1) is (ADIC TALLETY).

≡ {Unfold ADIC1}

ADIC new formula :  
 DYADIC formula, where (DYADIC) is (ADIC TALLETY) ;  
 MONADIC formula, where (MONADIC) is (ADIC TALLETY).

≡ {Symbolic evaluation}

ADIC new formula :  
 DYADIC formula, where (DYADIC) is (ADIC TALLETY) ;  
 MONADIC formula.

≡ {Unfold DYADIC-formula, Unfold MONADIC-formula}

ADIC new formula :  
 DYADIC new operand,  
 DYADIC operator,  
 DYADIC i new operand,  
 where (DYADIC) is (ADIC TALLETY) ;  
 monadic operator,  
 MONADIC new operand.

≡ {Unfold DYADIC-new operand}

ADIC new formula :  
 DYADIC new formula,  
 DYADIC operator,  
 DYADIC i new operand,  
 where (DYADIC) is (ADIC TALLETY) ;  
 secondary,  
 DYADIC1 operator,  
 DYADIC1 i new operand,  
 where (DYADIC1) is (ADIC TALLETY) ;  
 monadic operator,  
 MONADIC new operand.

≡ {Left-recursion removal}

ADIC new formula :  
 secondary,  
 DYADIC1 operator,  
 DYADIC1 i new operand,  
 where (DYADIC1) is (ADIC1 TALLETY),  
 ADIC ADIC1 rest formula ;  
 monadic operator,  
 MONADIC new operand,  
 ADIC ADIC1 rest formula.

ADIC ADIC rest formula : .

ADIC DYADIC rest formula :  
 DYADIC operator,  
 DYADIC i new operand,  
 where (DYADIC) is (ADIC2 TALLETY),  
 ADIC ADIC2 rest formula.

≡ {Rule introduction of (5), Fold}

ADIC new formula :  
 secondary,  
 DYADIC1 operator,  
 DYADIC1 i new operand,  
 ADIC DYADIC1 rest new formula ;  
 monadic operator,  
 MONADIC new operand,  
 ADIC MONADIC rest new formula.

- (5) ADIC ADIC2 rest new formula :  
 where (ADIC2) is (ADIC1 TALLETY),  
 ADIC ADIC1 rest formula.

$\equiv$  {Unfold rest-formula}

ADIC new formula :  
 secondary,  
 DYADIC1 operator,  
 DYADIC1 i new operand,  
 ADIC DYADIC1 rest new formula ;  
 monadic operator,  
 MONADIC new operand,  
 ADIC MONADIC rest new formula.

ADIC ADIC1 rest new formula :  
 where (ADIC1) is (ADIC TALLETY) ;  
 where (ADIC1) is (DYADIC TALLETY),  
 DYADIC operator,  
 DYADIC i new operand,  
 ADIC DYADIC rest new formula.

$\equiv$  {Invariant introduction of (6) and (8) (due to Mark-Jan Nederhof)  
 ADIC ADIC1 formula  $\Rightarrow$  where (ADIC1) is (ADIC TALLETY)  
 (with induction to the number of calls of rest-new-formula,  
 using transitivity of being a prefix.)}

ADIC new formula :  
 secondary,  
 DYADIC operator,  
 DYADIC i new operand,  
 ADIC DYADIC rest new formula,  
 (6) where (DYADIC) is (ADIC TALLETY) ;  
 monadic operator,  
 MONADIC new operand,  
 ADIC MONADIC rest new formula.

- ADIC ADIC1 rest new formula :  
 (7) where (ADIC1) is (ADIC TALLETY) ;  
 where (ADIC1) is (DYADIC TALLETY),  
 DYADIC operator,  
 DYADIC i new operand,  
 ADIC DYADIC rest new formula,  
 (8) where (DYADIC) is (ADIC TALLETY).

$\equiv$  {Symbolic evaluation of (7) (due to Mark-Jan Nederhof)}

ADIC new formula :  
 secondary,  
 DYADIC operator,  
 DYADIC i new operand,  
 ADIC DYADIC rest new formula,  
 where (DYADIC) is (ADIC TALLETY) ;  
 monadic operator,  
 MONADIC new operand,  
 ADIC MONADIC rest new formula.

ADIC ADIC1 rest new formula :  
 where (ADIC1) is (DYADIC TALLETY),

```

DYADIC operator,
DYADIC i new operand,
ADIC DYADIC rest new formula,
where (DYADIC) is (ADIC TALLETY) ;
.

```

Finally, focus on tertiary:

```

tertiary :
  ADIC formula.
                                                    ≡ {Invariant introduction}

tertiary :
  ADIC1 formula, where (ADIC1) is (priority i TALLETY).
                                                    ≡ {Fold new formula}

tertiary :
  priority i new formula.

```

The rules for `formula`, `rest-formula` and `operand` have become superfluous. These are deleted from the grammar, after which `new-formula` is renamed into `formula`, `rest-new-formula` is renamed into `rest-formula` and `new-operand` is renamed into `operand`. We will give a resumé of our syntax:

```

tertiary :
  priority i formula ;
  secondary.

ADIC formula :
  secondary,
  DYADIC operator,
  DYADIC i operand,
  ADIC DYADIC rest formula,
  where (DYADIC) is (ADIC TALLETY) ;
monadic operator,
  MONADIC operand,
  ADIC MONADIC rest formula.

ADIC ADIC1 rest formula :
  where (ADIC1) is (DYADIC TALLETY),
  DYADIC operator,
  DYADIC i operand,
  ADIC DYADIC rest formula,
  where (DYADIC) is (ADIC TALLETY) ;
.

ADIC operand :
  ADIC formula ;
  secondary.

```

Notice that a solitary `secondary` is not a `formula`, because each `secondary` in a `formula` must be firm. After left-factoring `tertiary` and `operand`, the syntax for `tertiary` and `formula` is  $LL(1)$  when affix directed parsing is used and operator priorities are known beforehand. Therefore, it is worthwhile to consider a previous scan over the input text for collecting priority declarations.

### 4.2.8 On the Terminal Objects of Algol 68

A language is defined over some set of terminal symbols  $T$ . In an implementation a set of *concrete objects*  $\Sigma$  is used to depict *abstract objects* in  $T$  by some *representation* in  $\Sigma^*$  [Stie75]. A terminal symbol may have multiple representations, and a sequence of objects in  $\Sigma^*$  may be the representation of more than one terminal symbol. The representation of symbols should be chosen in such a way that no ambiguity may arise.

#### Lexical Grammars

DEFINITION 4.2.1 A *representation* is a mapping

$$\text{Repr} : T \rightarrow \Sigma^* \rightarrow \mathbb{B}$$

where  $\Sigma$  is a finite set of concrete objects, and  $T$  is a (possibly infinite) set of terminal symbols of some grammar  $G$ .

Define  $\text{Repr} : T^* \rightarrow \Sigma^* \rightarrow \mathbb{B}$  as the homomorphic extension of  $\text{Repr} : T \rightarrow \Sigma^* \rightarrow \mathbb{B}$  to sequences of symbols. A representation is termed a *proper representation* if for all  $w \in L(G)$

$$\text{Repr}(w, v) \wedge \text{Repr}(w', v) \Rightarrow w = w'$$

A representation may be specified by some *lexical grammar*  $G'$  with start symbols  $T$  and terminal symbols  $\Sigma$ , satisfying for all  $t \in T$  and  $s \in \Sigma^*$

$$\text{Repr}(t, s) \Leftrightarrow t \rightarrow^* s$$

The language which is defined by the composite grammar of  $G$  and  $G'$  is called a *representation language* of the *strict language*  $L(G)$ .

#### Formalizing the Representation of Algol 68

In [Wij76], the representation of many symbols is defined in a precise, but not entirely formal way. The role of typographical display features and the representation of all symbols is formalized in a lexical grammar in the 2VWG-formalism. Care has been taken to define the representation of symbols in such a way that by composing the grammar for the strict language with the lexical grammar a grammar for the reference language is obtained. This composed grammar has served as the starting point for deriving an implementation of the *reference language* [Wij76].

In order to distinguish between the terminal symbols of the strict language and of the reference language, the latter ones are called *marks*. Marks are indivisible symbols with a one-to-one hardware representation.

The syntax for Algol 68 has an infinite number of terminal symbols, while only a finite number of marks is available for any representation. Consequently, it is unavoidable that certain

symbols obtain the same representation, and that many symbols in the strict language are represented by a *composite representation*, i.e. a sequence of two or more marks. As a result, the lexical grammar is quite ambiguous.

The role of typographical display features (*layout* in short) needs clarification [Stie75]. It is stated in [Wij76, 94d] that layout is of no significance, while layout does play the *syntactic* role of separator between two or more adjacent **bold-TAG-symbols**. Furthermore, a blank may figure as the representation of the **space-symbol** in **string-** or **character-denotations**. The lexical grammar describes in a precise way where layout may or may not occur:

NOTION symbol : NOTION marks, layout option.

The rules for NOTION marks produce the possible representations of the symbols in [Wij76, 941]. It is clear that in this new terminology NOTION-marks and not NOTION-symbols are intended within **string-** or **character-denotations** and within composite representations. The only place where the lexical grammar is incapable of dealing with typographical display features is in prescribing the obligatory layout separating adjacent **bold-TAG-symbols**. This problem is not easily solved in a nondeterministic grammar without annotations for indicating determinism.

The verbal directions of [Wij76, 9422] are formalized in so called *peeling rules* [Stie75], which break down the representations of the infinite set of **TAX-symbols**, yielding a composite representation. For example:

DYAD cum NOMAD BECOMSETY symbol :  
DYAD marks, NOMAD marks, BECOMSETY marks, layout option.

### Lexical analysis in EAG

The lexical grammar for Algol 68 is imaged into EAG using the same techniques which were applied to the rest of the grammar, and is subjected to the same transformational methods. For efficiency reasons, the rules for NOTION token, NOTION symbol and NOTION marks have been specialized for each possible occurrence.

The EAG-compiler supports a mechanism for recognizing the longest possible sequence of symbols composed of some alphabet (*strictness annotations* in regular expressions). This feature can be used in disambiguating most of the lexical analysis.

A *bold representation* is a representation of any symbol whose representation is composed of bold faced letters and digits. The symbols of [Wij76, 941] are called *reserved symbols*. Obviously, some reserved symbols and all **bold-TAG-symbols** have a bold representation. Following the verbal directions of [Wij76, 9422b], it is clear that the principle of the longest match must be applied to bold representations. For this reason a rule

**bold marks (bold letter + bold letgits) :**  
A-Z (bold letter), A-Z0-9\*! (bold letgits).

is used in the lexical grammar for recognizing bold representations.

From the EAG an underlying context free grammar, the head grammar, can be extracted. Using a tool like the Grammar Workbench (GWB) [Dek92], formal properties of the head grammar, e.g. follow sets, can be calculated. In this way it is easy to prove that a `letter-ABC-symbol` cannot follow a `TAG-symbol`, which implies that the principle of the longest match is also applicable to `TAG-symbols`.

### 4.2.9 Implementation Status

The EAG for Algol 68 consists of approximately 1600 rules and is considerably bigger than the 2VWG. The increase in size is mainly caused by the lexical grammar and by introducing specialized rules, such as the rules created by unfolding metanotions which are used for abbreviating similar rules.

The EAG has not been completed. Lack of a modular structure and static semantic checks makes large grammars hard to manage. Furthermore, compilation time of large EAGs is excessive (see §4.3). For these reasons, it was decided not to finish the EAG, but to try to transform the EAG into CDL3 (see next chapter).

The standard library and format texts have not been implemented yet. Furthermore, the EAG still contains one trouble spot in the treatment of declarers specifying modes. Look at the syntax for `applied-mode-indications`:

```
(MOID * TALLY>, >NEST) applied mode indication with (TAB>) :
  where (MOID * TALLY, TAB) identified in (NEST),
        tab token (TAB).
```

In 2VWG, correct values for `MOID` and `TALLY` are chosen nondeterministically from an infinite domain. A value for `TALLY` is chosen in such a way that no circular chains of `mode-definitions` are possible, a very brief way of specifying a test for cycles. `MOID` is chosen to be equivalent to the `MOID` of `TAB`. Furthermore, the spellings of modes are chosen in such a way that equivalent spellings are also *equal* if the `particular-program` demands so. In this way, the equivalence test can remain buried in the predicate `identified`. Furthermore, each `MODE` is checked implicitly for well-formedness by the equivalence test. Notice that the nondeterministic choice of `MOID` and `TALLY` allows a `particular-program` to have an infinite number of parses. Here, the 2VWG is under-specified.

In EAG on the other hand, the values for `MOID` and `TALLY` must be *constructed*. Of course, it is not yet possible to identify `MOID` and `TALLY` in the `NEST`, because `MOID` and `TALLY` must be known first in order to construct the `NEST`. A possible solution is the following:

1. Replace `MOID` and `TALLY1` in `actual-MOID-TALLY1-NEST-declarer` [Wij76, 42c] with a *place-holder* referring to `TAB` (see also §5.2.2).
2. After all declarations have been gathered, just before a `LAYER` is added to a `NEST` [Wij76, 32a, 35b, 35e], the following actions are undertaken:
  - (a) Check the `LAYER` for cyclic mode declarations.
  - (b) In each `DEC`, replace the place-holders for each `MOID` with an actual spelling for a `MOID`.

- (c) In each DEC, replace equivalent modes with equal modes.

However, it is not clear how this or another solution is to be derived transformationally from the 2VWG. The problem is that the possible values of MOID and TALLY are not specified by a predicate or a conjugation. Because this problem has not been solved yet, our EAG is not well-formed, while it does have the same generative meaning as the 2VWG.

### 4.3 Practical Usability of eag-compile

Experimentation with `eag-compile` has revealed some limitations of the compiler.

#### 4.3.1 Compiling and Debugging

Compilation of the EAG for Algol 68 may last more than 1 hour, resulting in executables of about 1.5 megabytes if a top-down parser is generated. Left-corner parser are even bigger. It is clear that in this way experimentation with large grammars is not possible and that testing and debugging a grammar becomes a time consuming process. A module structure supporting separate compilation is suggested for `eag-compile`.

Most errors made turned out to be mismatches between affix nonterminals and affix expressions. A static typing system would have prevented these errors.

Debugging is supported by the possibility of letting parsers produce trace output. However, this output generally consists of thousands of lines, making it a challenging task to find an error.

#### 4.3.2 The Typing System

Formulating meta-grammars in EAG using tuples may lead to unaesthetic definitions. This will be illustrated with some examples from Algol 68. Consider for example the rules

```
MODE :: procedure PARAMETY yielding MOID.
DUO :: procedure with PARAMETER1 PARAMETER2 yielding MOID.
```

Due to the typing system, this must be written in EAG as

```
MODE :: procedure * PARAMETY * yielding * MOID.
DUO :: procedure * with PARAMETER1 PARAMETER2 * yielding * MOID.
with PARAMETER1 PARAMETER2 :: with * PARAMETER1 PARAMETER2.
PARAMETER1 PARAMETER2 :: PARAMETER * PARAMETER.
procedure :: "procedure".
with :: "with".
yielding :: "yielding".
```

Otherwise, DUO is not a MODE. Furthermore, affix expressions in displays and meta-alternatives do not match affix nonterminals, unless they are obtainable by one single metaproduction.

As a consequence, ground tree structures are not denotable. Both problems must be solved with predicate introduction.

Another thorn in the flesh is the `nil` in e.g.

```
MOID :: MODE ; void * nil.
void :: "void"
```

which is obligatory because `MODE` is of type `tuple`.

Due to the implicit concatenation and the convention that small letters denote terminals, the metagrammar of `CDL3` is much cleaner. Forbidding ambiguity in the metagrammar does not seem to be restrictive in practice.

Versions 1.3 and lower of `eag-compile` do not support polymorphism, obliging the user to define for example

```
where (string) is string (string) : .
where (tuple) is tuple (tuple) : .
```

when she wants to write

```
where (string) is (string) : .
where (tuple) is (tuple) : .
```

For this reason, union types have been introduced in version 1.4 of `eag-compile`.

`eag-compile` does not support synonyms. For large grammars, the list of synonym declarations can be quite large. Therefore, it is worthwhile to consider the future implementation of synonyms in `eag-compile`.

### 4.3.3 An Experiment

Experimentation with parts of the grammar of Algol 68 has shown that the run-time behavior of parsers generated by `eag-compile` can be rather bad, but that improvements are possible. A first improvement can be realized by removing left-recursion. Furthermore, it was discovered that a parser may spend quite some time on meta-parsing and that checking whether some `bold-TAG-symbol` is reserved, i.e. does not have the same representation as any other symbol, is quite expensive.

In an experiment, the time and memory consumption of five parsers generated from grammars for priority declarations in Algol 68 has been measured. In all grammars, the independence test has been elided, and the grammars have been pruned as far as possible.

The first grammar contains the left-recursive rule

```
(NEST) joined priority definition of (DECS * DEC) :
(NEST) joined priority definition of (DECS),
and also token,
(NEST) joined priority definition of (DEC).
```

```
(NEST) joined priority definition of (DEC) :
  (NEST) priority definition of (DEC).
```

Lexical analysis contains the rules

```
priority token :
  pragment sequence option,
  priority symbol.
priority symbol :
  bold symbol ("PRIO"), layout.
...
bold TAG :: bold letter + bold letgits.
bold letter :: A-Z.
bold letgits :: A-Z0-9*.
tao symbol (bold TAG) :
  bold tag symbol (bold TAG).
bold tag symbol (bold TAG) :
  bold symbol (bold TAG),
  not predefined (bold TAG).
bold symbol (string) :
  bold marks (string), layout.
not predefined (bold TAG) :
  unless (bold TAG) is ("AND"),
  unless (bold TAG) is ("AT"),
  ...,
  unless (bold TAG) is ("WHILE").
```

In the other four grammars, the left-recursion has been removed, resulting in the *LL*(1) rules

```
(NEST) joined priority definition of (DECS) :
  (NEST) priority definition of (DEC),
  (NEST) rest joined priority definition of (DECS, DEC).
(NEST) rest joined priority definition of (DECS, DECS) : .
(NEST) rest joined priority definition of (DECS1, DECS) :
  and also token,
  (NEST) priority definition of (DEC),
  (NEST) rest joined priority definition of (DECS1, DECS * DEC).
```

In two grammars, the test for reserved symbols has been replaced with a primitive predicate. Also in two grammars, the metarules for *NEST*, *LAYER*, *DECSETY*, *DECS*, *DEC*, and *bold TAG* have been deleted. Here, meta-parsing is not necessary, because it can be verified that the affix values assigned to these affixes are always in their domain. However, there are cases when the metagrammar is not dispensable.

The parsers have been tested with a list of 40 priority declarations as input. Execution time has been measured using the Unix command *rusage*, yielding the system time and user time consumed by the process. Memory consumption has been measured using the Unix command *top*, indicating the total size of the process in the *SIZE*-field.

## Results

Left-corner parser:

Meta-Parsing	Reservedness Check	Bin. Size	Proc. Size	Exec. Time
yes	by predicate	360 448	20 560	aborted after 300 sec.

Top-down parser:

Meta-Parsing	Reservedness Check	Bin. Size	Proc. Size	Exec. Time
yes	by predicate	286 720	18 556	15.8
yes	primitive	262 144	9 280	10.8
no	by predicate	278 528	11 924	12.9
no	primitive	253 952	2 652	2.8

Legend:

- Bin. Size : Size of executable (bytes)
- Proc. Size : Maximum size of text+data+stack-segment (kilobytes)
- Exec. Time : Average system+user-time (seconds)

## Discussion

The experiment shows that the time and space requirements of parsers generated by `eag-compile` are quite immoderate, even for  $LL(1)$  grammars. The left-corner parser does not terminate within reasonable time. The large time consumption is probably caused by swapping due to the high memory requirements.

Although the input sentence from the experiment is not likely to occur in practice, it is very well possible that an Algol 68 program does contain more than 40 declarations which have to be collected. The parser from the experiment did not have to backtrack, while a parser for Algol 68 will have to backtrack over pieces of text containing units (and therefore complete programs). This will be the case even if the grammar is left-factored completely, because the priority of an operator may be declared after its application in a formula. Furthermore, no time was spent on checking context conditions. For these reasons, it is to be expected that a parser for Algol 68 in EAG is of no practical value at all.

The experiment reveals that at least three major optimizations to an EAG are possible. The first one is left-recursion removal, resulting in much smaller and more efficient parsers. Secondly, it turns out that checking whether some symbol is reserved may be responsible for half the memory consumption of a parser generated by `eag-compile`. Because this check may need to be performed by many applications, `eag-compile` was adapted to support a primitive predicate

```
not in reserved word list (>string, >reserved)
```

where `reserved` is a tuple containing text denotations. The gain of efficiency of this optimization was unexpectedly high. A third optimization is realized by preventing meta-parsing. If it can be determined statically that the value assigned to an affix is in its domain, the affix can be made free, i.e. it can be replaced with an affix which is not meta-defined. Freeing

affix nonterminals should be a task of the compiler. Static typing is another way of making meta-parsing obsolete. The metagrammar can be restricted to be unambiguous: in §3.5 it was shown how nondeterministic splitting of affix values can be simulated by nondeterminism on the first level.

It remains to be investigated how the efficiency of parsers generated by `eag-compile` is influenced by backtracking and the insertion of cut-operators.

It is concluded that present implementations of unrestricted Extended Affix Grammars, at least `eag-compile`, are of limited use for real life grammars. More research is needed in order to establish what optimization techniques can lead to acceptable results. It might be worthwhile to investigate into the properties of an EAG-compiler using static typing. This would make affix evaluation much more efficient. Furthermore, static typing could help the user prevent many errors.

#### 4.3.4 Reliability of `eag-compile`

Experimentation with parts of the EAG for Algol 68 has revealed a number of bugs in `eag-compile`. Version 1.1 does not handle metarules of type `string` correctly. Version 1.2 contains a bug in the memory management of some metarules.

`eag-compile` has been thoroughly tested with large grammars performing lexical analysis for Algol 68, grammars parsing formulas and grammars gathering lists of declarations in Algol 68. Until today, no bugs have been found in `eag-compile` version 1.4, which was released on April 30 1994.

Parsers generated by `eag-compile` do not notice affix nonterminals remaining unground in a parse. Due to the absence of flow annotations, it cannot be determined statically whether a parse tree can be fully decorated.

## 4.4 Conclusions

### 4.4.1 Conclusions about Transformation of Two Level Grammars

Two level grammars are well suited for performing program transformations. A two level van Wijngaarden Grammar can be transformed into an Extended Affix Grammar which is rather similar to the original grammar. However, it is necessary to remove certain abbreviations, resulting in a longer and less general specification. Under-specification in the 2VWG, allowing an infinite number of parses, may lead to EAGs which are not well-formed.

We have derived an EAG for Algol 68 transformationally. The grammar is top-down parsable. Only at one point, we have failed to derive a solution. The problem is caused by the nondeterministic choice of affix values from an infinite domain. The values chosen are not prescribed by the specification.

Using our transformations, a two level grammar can be made transparent mainly by developing predicates which compose and decompose affix values in accordance with the metagrammar.

### 4.4.2 Conclusions about the EAG Formalism

The syntax for Algol 68 shows that the possibility of applying an affix before its value is known, is a useful feature in describing programming languages.

Left-recursion is not essential for describing programming languages, at least not for describing Algol 68. This allows the generation of top-down parsers. Predicates are indispensable in describing the context conditions of Algol 68.

The grammar for Algol 68 makes extensive use of free affix flow. In this case, flow can be fixed statically after unfolding metanotions, resulting in a longer description. Therefore, free affix flow can be a useful feature for describing programming languages. However, not specifying the direction of flow makes it harder to verify that no affixes can remain unground in a parse. In particular, it is necessary to specify well-formedness conditions for EAGs with free affix flow.

The syntax for Algol 68 shows that many computations can be specified rather compactly by nondeterministic predicates (e.g. coercion).

Dynamic typing is not essential for implementing Extended Affix Grammars. Static typing is more efficient and helps the user prevent errors. Using our transformations, the metagrammar can be restricted to be tree-structured and statically type-able. Nondeterministic splitting of affix values can be simulated by nondeterminism on the first level.

The suitability of recursive backup parsing [Kos74] for implementing nondeterministic Extended Affix Grammars remains to be investigated. It is interesting to compare the efficiency of optimized recursive backup parsing with tabular methods. Results in the related AGFL formalism [Kos91<sub>2</sub>] are promising. In [Gro92], some optimizations for recursive backup parsing are described. These have not yet been implemented in EAG-COMPILE.

### 4.4.3 Conclusions about `eag-compile`

Specifications in EAG are executable, but large compilation time make `eag-compile` unsuitable for experimentation with large grammars. Debugging of grammars is complicated by the impossibility of printing tracing and error messages, while programming errors are not caught by static semantic checks. We conclude that `eag-compile` is not suitable for most practical purposes. Therefore, we will try to transform our grammar into CDL3.

The tuple system of `eag-compile` enforces contrived definitions. Many affix expressions cannot be denoted directly.

Optimizations are necessary for obtaining parsers with a reasonable run-time behavior. It is worthwhile to consider an implementation with static typing. Dynamic typing without optimizations turns out to be quite expensive.

Left-corner parsers are implemented quite inefficiently. Therefore, it is advisable to remove all left-recursion.

Version 1.4 of `eag-compile` is reliable. Thorough testing has not revealed any bugs.

## Chapter 5

# Towards a Parser for Algol 68 in CDL3

In this chapter we will describe our attempt of transforming the EAG for Algol 68 into a deterministic parser in CDL3. Care has been taken to preserve the pragmatic value of the original grammar as much as possible. The parser is intended to be an implementation of Algol 68 rather than a specification. Therefore, the grammar has been extended with alternatives handling incorrect input.

In §5.1, a method of imaging EAG into CDL3 is defined. It is assumed that the grammar is first made deterministic and that all affixes are made to flow from left to right in two passes. For these reasons, some additional transformations are defined in §5.2. The next section describes the process of making our grammar deterministic, following the classical scheme of lexical analysis, context free analysis and then context sensitive analysis. We have not yet succeeded in making context sensitive analysis entirely deterministic. Finally, suggestions for future work are given, and conclusions are drawn.

### 5.1 Imaging EAG into CDL3

As remarked in §2.3.2, a CDL3 grammar can be seen as a transparent and deterministic version of an EAG. Whether a grammar is deterministic depends on the parsing scheme used. For our purpose, we will use the following definition.

**DEFINITION 5.1.1** A two level grammar  $G$  is called *deterministic*, if  $G$  is transparent (ensuring unique split-ability of each affix value) and if all alternatives for each hypernotation are mutually disjoint.

Two alternatives are *disjoint* if their director-sets are disjoint (in which case a unique choice for an alternative can be made using look-ahead), or if a unique choice for one alternative can be made on basis of predicates (using affix directed parsing). In the latter, case we will say that the alternatives start with disjoint predicates.

### 5.1.1 Strategy

Because ambiguity of context free grammars is undecidable, transparency of two level grammars is also undecidable. Furthermore, disjointness of predicates is in general undecidable. Therefore, there cannot exist an automatic method of transforming EAG into CDL3. We will formulate a strategy instead.

Let  $G$  be an EAG. Our case study shows that it is often possible to image  $G$  into CDL3 using the following strategy:

1. Replace the meta-grammar with a tree-structured metagrammar (§4.2.4). Furthermore, if the metagrammar contains empty productions, replace these alternatives with alternatives producing an affix terminal (§3.3.14).
2. Make  $G$  transparent, using the transformations in chapter 3 (see §4.2.4 for examples).
3. Make context free parsing deterministic (§5.3.1 and §5.3.3).
4. Make all predicates deterministic (§5.3.5).
5. Construct *type definitions* for each hyper-rule (§5.1.2). Using predicate introduction, all left hand side hyper-notions of the same type are unified.
6. Define the affix flow and impose two pass affix evaluation (§5.3.4).
7. Replace all equality tests with guards. The type of guard depends on the affix flow.
8. Order the rules, classify their effect (§2.3.1) and repair all defects, if any.

### 5.1.2 Typing the Hyper-rules

Assume that  $G$  is a transparent EAG and let  $K$  be its imaginary main metanotion. For this purpose, we will imagine that ‘(’ and ‘)’ are terminal symbols of the metagrammar only occurring in alternatives for  $K$ . For each alternative  $K :: h$ . we can assume that  $h$  is of the form  $\alpha_0(N_1)\alpha_1 \dots (N_n)\alpha_n$  with  $\alpha_i \in A_T^*$  and  $N_i \in A_N^*$  for  $1 \leq i \leq n$ . Otherwise, alter the division between the first and second level. This is always possible, because each basic hypernotation  $h'$  is derivable from  $K$  in a unique way.

DEFINITION 5.1.2 Let the rules for  $K$  be constructed as above. If  $K :: h$ . and  $h' \in [h]$ , then  $h$  is called the *type* of  $h'$ . Now the function *Type* specifying the domain of each affix position of each basic hypernotation is defined in the obvious way.

### 5.1.3 Rule Ordering and Algorithm Classification

Let  $G$  be a grammar which is  $LL(1)$  with the affix level. Thus, if the director-sets of two alternatives are not disjoint, then for every affix assignment one of the alternatives must be shielded by the forbidden symbol. The rules of the grammar can be ordered and their effect can be classified using the process described in [Kos94]:

1. Non-predicate rules which have a non-failing alternative are specified as ACTION, and as PRED otherwise.
2. Predicate rules which cannot fail are specified as FUNCTION, and as TEST otherwise.
3. Non-failing alternatives should come last in the ordering of rules.
4. All defects are repaired by adding error alternatives generating a suitable error message [Kos72<sub>2</sub>] and possibly mending the error. We have not investigated into the automatic repairment of erroneous Algol 68 programs.

#### 5.1.4 Example

Our CDL3 grammar contains the following syntax for `unit`. The affix expressions which are joined to `SOID` result from imposing two pass affix evaluation. Their origin will be described later. Notice that the original rule for `unit` is not *LL*(1). However, this problem can be solved by look-ahead and by application of information which is collected in a pre-scan over the input text. This results in a grammar which is well-structured and remarkably similar to its original specification in 2VWG.

EXAMPLE 5.1.1 Syntax for `unit`.

```

ACTION forced unit (/SOID>, >NEST) :
  unit (/ SOID, NEST) ;
  this position (POS),
  syntax error ("Unit expected", POS),
  make erroneous (SOID).
PRED unit (/ SOID>, >NEST) :
  this position (POS),
  (routine text (/ MOID, NEST),
   / [anisort MOID routine text at POS -> SOID] ;
  jump (/ SOID, NEST) ;
  skip (/ SOID, NEST) ;
  tertiary (/ SOID, NEST),
  (rest assignation (/ SOID, MODE, NEST),
   / make moid (MODE, MOID),
   [anisort MOID assignation at POS -> SOID] ;
  rest identity relation (/ SOID, MOID, NEST),
   / [anisort MOID identity relation at POS -> SOID] ;
  +)).

```

The rule for `forced-unit` has been introduced for repairing defects corresponding to erroneous input sentences. Yielding an erroneous `SOID` matching any other `SOID` provides a possibility of continuing parsing, while preventing cascading of errors.

## 5.2 Additional Transformations

We will describe two more transformations on grammars. The first transformation can be used for splitting up a grammar into two simpler grammars. The second transformation can be useful in imposing two pass affix evaluation.

### 5.2.1 Decomposition of Grammars

Often, it is desirable to decompose a grammar into sub-grammars, or to alter an existing decomposition, as is motivated and exemplified in §5.3.1. When decomposing a grammar into sub-grammars, we will view the terminal symbols of one grammar as the start symbols of another grammar. For this reason, we will assume that a grammar can have a set of start symbols.

**DEFINITION 5.2.1** A grammar  $G$  can be *decomposed* into grammars  $G'$  and  $G''$ , notation  $G = G'' \circ G'$ , if  $P_M = P'_M \cup P''_M$ ,  $P_H = P'_H \cup P''_H$ ,  $P'_H \cap P''_H = \emptyset$  and  $V'_T = S''$ .

Obviously, any grammar  $G$  can be viewed as the composition  $G'' \circ G'$  of grammars  $G'$  and  $G''$  by taking  $G' = G$  and  $G''$  a possibly trivial lexical grammar (§4.2.8) describing the representation of the terminal symbols of  $G$ .

**DEFINITION 5.2.2** A grammar  $G$  is *less granular* than  $G'$  if  $w \in L(G') \Rightarrow w \in L(G)$ .

Assume that  $G = G'' \circ G'$ . Notice that every grammar describing the language  $(S'')^*$  is less granular than  $G$ . Consequently, in order to check whether a sentence  $w$  belongs to  $L(G)$ , we can first check whether there exists a  $v \in (S'')^*$  producing  $w$ , and then check whether  $v$  belongs to  $L(G')$ . This property will be used in §5.3.2.

**TRANSFORMATION 20** *Decomposition*

If  $G = G'' \circ G'$  and  $G = F'' \circ F'$ , then the representation languages (§4.2.8) of  $G'$  and  $F'$  are the same. Therefore,  $G'$  and  $G''$  may be replaced with  $F'$  and  $F''$ .

### 5.2.2 Place-Holders

In two level grammars, it is customary to apply affixes before their ground instances are defined. For this reason, implementations usually support a mechanism of delayed evaluation such as the second pass in CDL3. However, such a mechanism may turn out to be insufficiently powerful for certain particular purposes. Presupposing left to right affix evaluation and a fixed number of passes, problems arise when an affix value should flow from right to left and this value cannot be calculated until the last pass. Consequently, there is no subsequent pass in which to inherit this value after its calculation.

The purpose of the transformation defined below is to invert affix flow, or to fix it to one direction in case of dynamic flow. The essence of this technique is to replace an inherited

affix  $X$  with a derived affix whose (ground) value is understood to represent the identity of  $X$  as well as the restrictions (or context conditions) imposed on  $X$ . Furthermore, this representant, which we will call a *place-holder*, is to be unified with the original value of  $X$  later. For this reason, the usual grammatical unification mechanism is made explicit (with predicate introduction) and replaced with a more general mechanism, when needed.

One may notice that free affix flow in fact simulates first order logic variables. Using place-holders, it is also possible to simulate higher order logic variables, because we will simulate the assignment of a predicate to a variable.

We will formulate the transformation of replacing an affix with a place-holder for its simplest case, where flow is inverted in one hypernotation only. Application of the transformation to chains of calls of hypernotations should be obvious.

#### TRANSFORMATION 21 *Place-Holders*

Let

$$p = h(>X) : \alpha, f(>X, >Y), \beta.$$

be a production rule with calls

$$q = g : \gamma, h(>X), \delta.$$

Then an equivalent grammar is obtained by introducing a new metarule  $X' :: fY$ ., replacing  $p$  with

$$p' = h(X'>) : \alpha, [fY \rightarrow X'], \beta.$$

and replacing each rule  $q$  with

$$q' = g : \gamma, h(X'>), \delta, \text{unify}(X, X').$$

where

$$\text{unify}(X, fY) : f(X, Y).$$

When  $X$  or  $Y$  denotes multiple variables, it is assumed that `unify` enforces the consistent substitutions required, if any. We will call  $fY$  a place-holder for  $X$ . Notice that  $f$  on the meta-level is not a hypernotation, but merely a representant of the metanotion  $f$  *carried* with  $Y$ .

When necessary, a place-holder may be endowed with redundant information, such as the position in the input text in order to enable sensible error reporting when `unify` should happen to fail.

### 5.3 Deriving a Parser for Algol 68 in CDL3

In the previous chapter it is described how a two level grammar can be made transparent. In this section we will describe our attempt of making the grammar for Algol 68 deterministic. First, we will give an overview of our parsing algorithm. The lexical analyzer (§5.3.1) converts an input text to a unique sequence of tokens. The lexical analyzer is driven by a pre-scanner (§5.3.2) which reads the complete input text once before actual parsing commences. The pre-scan is necessary for collecting e.g. `priority-declarations`. Using the information collected by the pre-scanner and look-ahead, context free parsing can be made deterministic (§5.3.3). Context free parsing takes place in the first pass of the analysis module of our grammar, while context sensitive analysis (§5.3.5) takes place in its second pass. Notice that with this approach it is not necessary to build an explicit parse tree. However, it is possible to endow the hyper-rules with second pass affixes deriving an abstract syntax tree or some intermediate code representing the program.

#### 5.3.1 Lexical Analysis

A lexical analyzer divides the input text (a sequence of marks) into a sequence of symbols. For enabling look-ahead of symbols or a pre-scan over the input text, each input text must preferably be map-able onto an *unique* sequence of symbols. Without guidance from the parsing process, this is not possible for the `symbols` as defined in [Wij76]. Therefore, the composition of the main grammar and the lexical grammar is altered in such a way that unambiguous lexical analysis is possible.

DEFINITION 5.3.1 A *lexical analyzer* implements a mapping

$$Lex : \Sigma^* \rightarrow T^* \rightarrow \mathbb{B}$$

with

$$Lex(v, w) \Rightarrow Repr(w, v) \tag{5.1}$$

$$Repr(w, v) \wedge \neg Lex(v, w) \Rightarrow w \notin L(G) \tag{5.2}$$

Lexical analysis is termed *unambiguous* if

$$Lex(v, w) \wedge Lex(v, w') \Rightarrow w = w'$$

In general, *Lex* will not be unambiguous. In this case, an input sentence  $v \in \Sigma^*$  may represent multiple sequences of symbols in  $T^*$ . In principle, representations yielding unambiguous lexical analyzers are used for representing a (representation) language in a *hardware language*.

Let *Repr* be the representation of a language. Clearly, *Repr* is proper if and only if there exists a function *Lex*, defining a lexical analyzer for the language, which is unambiguous.

The second condition imposed on *Lex* implies that all lexical analyses yielding non-syntactic sequences of symbols may be rejected. This property can be used for making lexical analysis

unambiguous. A well known technique of disambiguating lexical analysis is application of the *principle of the longest match* [Wai85].

According to [Wai85] the higher organizational costs involved in separating lexical analysis from parsing, i.e. decomposing a grammar into sub-grammars, can only be justified by realizing greater savings in in other areas, such as efficiency (e.g. using finite state automata for recognizing regular languages). In the authors opinion however, mere software engineering considerations provide a quite valid justification. First, decomposition separates the machine dependent part from the machine independent part. Secondly, decomposition separates different layers of abstraction, which is of importance for the pragmatic value of a grammar. For example, left-factoring `SIZE-symbols` in the original grammar for Algol 68 is undesirable, because it would lead to a contrived grammar. In the lexical grammar on the other hand, its is trivial and natural.

### Decomposition into Sub-Grammars

We have decomposed 5.2.1 the grammar for Algol 68 into a *main grammar* derived from [Wij76, 3–7, 911a–e] and a *lexical grammar* derived from [Wij76, 8,911f–94].

The main grammar has terminal symbols `was token` (SYMBOL) and `was symbol` (SYMBOL). A `token` is either a `symbol`, or a `pragment-sequence` followed by a `symbol`. It is convenient to view a `token` as a terminal symbol, because left-factoring `pragment-sequence-option` is again undesirable in the main grammar, but trivial in the lexical grammar.

The lexical grammar is decomposed into a small *token grammar* [Wij76, 911f] and a *symbol grammar*. The token grammar constructs `tokens` from a possibly empty sequence of `pragments` and a `symbol`. The symbol grammar constructs `symbols`, `comments` and `pragmats` from sequences of `marks` in the input text.

A symbol fits into one of the following categories:

```

SYMBOL :: TAG ;
        bold TAG ;
        OPERATOR ;          # DYAD BECOMSETY ; DYAD cum NOMAD BECOMSETY
        SIZETY STANDARD ;
        MOID DENOTATION ;
        PREDEFINED ;        # 94d--g
        unknown with ID.
```

These categories are not entirely disjunct (e.g. a tilde may be the representation of the pre-defined `skip-symbol` or the operator `tilde-symbol`), but this problem is easily circumvented.

### Lexical Analysis

During a pre-scan, guidance from the parsing process is not available, making lexical analysis quite ambiguous. It has been possible to disambiguate lexical analysis completely, using the following techniques:

1. Application of the principle of the longest match, which is a special case of the next point.
  2. Application of context information (see below).
  3. Unification of symbols with equal representations into intermediate ‘unknown’ symbols (see below).
  4. Choosing a more convenient composition of the main grammar and the lexical grammar.
- Ad 2. Using the GWB [Dek92], it is possible to collect many facts about ungrammatical sequences of symbols which may therefore be rejected by the lexical analyzer (*application of context information*). For example:

```

    NOMAD marks  ∉ Follow(TAD symbol)
    equals symbol ∉ Follow(colon symbol)
    colon symbol ∉ Follow(becomes symbol)
    NOMAD marks  ∉ Follow(skip symbol)

```

These facts replace the guidance from the parsing process, and justify application of the principle of the longest match to all symbols with a non-bold composite representation.

- Ad 3. Sequences of marks which represent multiple symbols can be unified into new intermediate symbols. For example, the `brief-else-symbol` and the `brief-ouse-symbol` have been folded to the newly introduced `again-symbol` (producing ‘|:’).
- Ad 4. It is beneficial to consider `MOID-denotation` and `TAX-symbol` as terminal symbols for the following reasons. First, in our lexical grammar `SIZE-symbols` (and in fact all other `bold-marks`) can be left-factored without disturbing the structure of the main grammar. Secondly, `SYMBOL` defines a regular language. Therefore, it is possible to apply a faster parsing scheme to lexical analysis. This is the classical argument for separating lexical analysis from syntactical analysis. However, we have not pursued this possibility. Thirdly, our lexical grammar is not ambiguous. Using the original representation, it is not possible to distinguish e.g. `letter-ABC-symbols` in a `denotation` and in a `TAG-symbol`.

### 5.3.2 The Pre-Scan

For efficient application of top down parsing techniques, it is important that the grammar does not necessitate backtracking over unlimited amounts of input. Because the syntax of Algol 68 contains structures which may contain a whole program, backtracking is not a feasible approach [Kos72<sub>1</sub>]. Due to loss of similarity to the original grammar and the different semantic actions which have to be undertaken upon recognizing certain locally ambiguous constructs, it is not desirable to merge these constructs into one syntactic category [Kos72<sub>1</sub>, Hun77].

Several authors have pointed out the possibility of preventing backtracking by performing a pre-scan over the input text, inserting *markers* containing knowledge about locally ambiguous constructs [Kos72<sub>1</sub>, Koc77]. We will apply a similar technique. Instead of inserting markers

in the input and in the grammar, the pre-scanner defines predicates for looking beyond the next symbol of the input.

The pre-scanner parses the input according to a grammar which is less granular than the grammar for Algol 68. The author has not derived the pre-scanner transformationally. A possible way of doing so might be to derive a less granular grammar for the pre-scanner using ‘language-enlarging’ transformations.

### Look-ahead

The pre-scanner defines predicates for testing whether a `bold-TAG` is an operator or a `mode-indication`, whether a `TAG` is a `label` or an `identifier` and whether an operator has a `priority-definition`. This information is stored in a nested environment which is built during the pre-scan.

The input text is stored in a tree structure containing a subtree for each block structure. This allows one to look beyond a `NOTION-PACK` as if it were one token. Because we have made lexical analysis deterministic, any input sentence is represented by a unique tree. Each block structure is marked with its *middler*, if any. A *middler* is either a `go-on-symbol`, an `and-also-symbol`, an `again-symbol` (e.g. ‘|:’) or an `inout-symbol` (e.g. ‘|’).

The pre-scanner also defines predicates for finite look-ahead, up to three symbols, and unlimited look-ahead, up to two tokens beyond a `declarer`, `PACKs` and beyond both. Unlimited look-ahead is implemented efficiently by skipping whole blocks in one leap. Although discouraged in [Kos72<sub>1</sub>], we will look beyond `declarers`, because the structure of the grammar greatly benefits from not left-factoring them.

### 5.3.3 Context Free Analysis

We have made the underlying context free grammar for Algol 68 deterministically parsable. First, the grammar has been left-factored as much as possible, while leaving the grammar reasonably structured. Secondly, using the `GWB` [Dek92], all rules have been checked for *LL*(1)-ness. It appeared that in the rules which are not *LL*(1), it is always possible to make a deterministic choice for one alternative by using look-ahead.

We will enumerate the shortcomings of the *LL*(1) parsing method (see [Kos74]) and exemplify their treatment in our particular case.

### Left-recursion

The treatment of left-recursion has been described in the previous chapter.

### Multiple Empty Alternatives

Multiple empty alternatives can be assumed to occur within predicates only (use abstraction otherwise). Therefore, their treatment is not considered in this section. Notice that super-

fluos empty alternatives can always be removed, because we have assumed that ambiguity plays no semantic role.

### Ambiguity

The treatment of grammars describing inherently ambiguous languages falls outside the scope of this thesis. Although Algol 68 is not ambiguous, the underlying context free syntax of our grammar is. Ambiguity originates from the following causes:

1. During the first pass, the MOID of a construct is not known. This results in the following ambiguities:
  - (a) The underlying context free syntax for `slice-or-call` (an artifact resulting from left-recursion removal) is ambiguous. We have not solved this problem. Instead, the `style-i-sub-symbol` is not supported, causing `slice` and `call` to start with symbols with a distinguishable representation.
  - (b) The underlying context free syntax for `collateral-clause` is ambiguous. It is possible to merge the syntax for `portrait` and `joined-portrait` by left-factoring (§4.2.6), but the result is contrived and not cannot be evaluated in two passes. We have not yet solved this problem in a transformational way.
2. The underlying context free syntax for `formulas` is ambiguous. This problem can be solved by using affix directed parsing and the `priority-definitions` collected by the pre-scanner.
3. On the context free level it is not possible to distinguish an `applied-operator` from an `applied-mode-indication` or to distinguish the `applied-identifier` of a `primary` from the `applied-identifier` of a `jump` with an empty `go-to-option`. This problem can be solved by letting the pre-scanner notice `defining-mode-indications`, `defining-operators` and `label-definitions`.

We observe that it is sometimes unavoidable to merge different rules with the same underlying context free syntax. The obvious strategy to follow is to let each rule resulting from merging other rules derive its context conditions for postponed checking in a predicate following the merged rule. We have not worked out transformations for merging rules in this way. It might be worthwhile to explore the combination of place-holders with the strategy applied in §4.2.6.

The treatment of ambiguity on the semantic level, i.e. within predicates, is postponed until the next section.

### Prefix Sharing

We have applied the following techniques for resolving prefix-sharing:

1. Common prefixes can be left-factored, possibly after unfolding hypernotations hiding these prefixes, resulting in rules as in e.g. example 5.1.1. The possibility of left-factoring

‘deeper’ common prefixes is strongly limited by our desire to preserve the structure of the original specification as much as possible. Furthermore, left-factoring may interfere with affix flow. For example, left-factoring the `declarer` of an `identity-declaration` and an `identifier-declaration` implies that the affix `VICTAL` of `declarer` cannot be inherited in the first pass. For these reasons, we have also applied the following technique.

2. Prefix-sharing can be solved by looking beyond nonterminal symbols (see example 5.3.1). In this way, it is possible to preserve the structure of the grammar and to prevent merging of constructs with a different semantics.
3. Sometimes, the right alternative can be chosen using affix directed parsing. The underlying context free syntax for `choice-clause` has been made mode independent by look-ahead and by embedding the syntax with an affix ‘`CHOICE :: bold choice using boolean ; bold case ; brief choice.`’ enforcing the middlers and closer of a `choice-clause` to be compliant with its opener (the original affix `CHOICE` has been removed by unfolding).

EXAMPLE 5.3.1 Syntax for routine-text.

```

PRED routine text (/ MODE>, >NEST) :
  is routine text without parameters,
  declarer (formal, MOID / NEST),
  should be token (routine symbol),
  forced unit (/ SOID, NEST),
    / unify (strong MOID, SOID) in (NEST),
      make procedure (MOID, MODE) ;
  is routine text with parameters,
  is token (brief begin symbol),
  add (/ NEST, DECS, NEST1),
  declarative defining (DECS / NEST1),
  should be token (brief end symbol),
  forced declarer (formal, MOID / NEST),
  should be token (routine symbol),
  forced unit (/ SOID, NEST1),
    / unify (strong MOID, SOID) in (NEST),
      where (DECS) like (PARAMETERS),
      make procedure (PARAMETERS, MOID, MODE).

FUNCTION add (/ >NEST, >DECS, NEST1>) :
  + / make decsety (DECS, DECSETY),
  empty (LABSETY),
  make layer (DECSETY, LABSETY, LAYER),
  add (NEST, LAYER, NEST1).

TEST is routine text with parameters :
  is token beyond pack and declarer (routine symbol).

TEST is routine text without parameters :
  is token beyond declarer (routine symbol).

```

Notice the rule for `add` with an empty first pass, which was introduced for imposing two pass affix evaluation. The rules for `unify` are explained in the next section.

### Local Ambiguity

The `style-i-sub-symbol` causes many local ambiguities. As mentioned before, we have chosen not to support this symbol.

The `begin-symbol` is used to announce a variety of constructs, such as the e.g. `ENCLOSED-clauses` and `declarative-defining`. For disambiguating between different instances of the `ENCLOSED-clause`, we have chosen the classical solution of letting the pre-scanner mark the middler (§5.3.2) of the construct.

Other cases of local ambiguity can be found in the syntax for `declaration`. Here, the `and-also-symbol` is overloaded in order to separate both `definitions` and `declarations`. This problem has been solved by using three symbols of look-ahead.

EXAMPLE 5.3.2 Syntax for `joined-mode-definition`.

```

ACTION joined mode definition (>DECSETY> / >NEST) :
  mode definition (DECSETY / NEST),
  rest joined mode definition (DECSETY / NEST).
ACTION rest joined mode definition (>DECSETY> / >NEST) :
  is another mode definition,
  was token (and also symbol),
  mode definition (DECSETY / NEST),
  rest joined mode definition (DECSETY / NEST)
+.
TEST is another mode definition :
  is third token (is defined as symbol).

```

The origin of the transient parameter `DECSETY` is be described later.

### 5.3.4 Imposing Two Pass Affix Evaluation

We have not experienced fundamental problems in imposing a two pass structure on affix evaluation. In our scheme, declarations are collected in the first pass in an derived affix, and inherited in the second pass for identification. In this way, application before declaration is implemented elegantly (§5.3.5). However, this approach forces us to let `declarer` derive its mode in the first pass, at which time the mode of each `applied-mode-indication` is still unknown. The classical solution for this problem is to derive a reference to an entry in the mode-table instead (see §5.3.5). Furthermore, we have made the `SOID` of a construct always derived in the second pass. Although in accordance with our intuition, this choice is also forced upon us by the fact that in general the mode of a construct cannot be known until after identification.

### Fixing Flow

The flow of `SORTs` and `MOIDs` in `EAG` can not be modeled in `CDL3`, because of the following reasons. First, flow may be inherited when it should be derived in order to fit into a two pass

affix evaluation scheme. Secondly, flow may be bidirectional, while it should be statically fixed. Consider for example the identity-relation in EAG

```
(boolean * nil, NEST) identity relation :
  (SORT1, reference * to * MODE, NEST) tertiary,
  identity relator,
  (SORT2, reference * to * MODE, NEST) tertiary,
  where sort (soft) balances sort (SORT1) and sort (SORT2).
```

and verify that due to balancing SORT1 is derived when SORT2 is inherited and vice versa. It is not a priori possible to determine a statically fixed direction of flow for SORT. Clearly, most constructs do not confine SORT to a unique value. Therefore, in a deterministic grammar a construct cannot derive SORT. Conversely, it is clear that inheriting SORT in a construct presumes balance to possess angelic foresight. Similar problems arise when trying to establish a direction of flow for modes.

We haven chosen to make the flow of SORT and MOID, which are folded back to SOID, derived for every construct. However, as mentioned above, most constructs can derive a number of syntactic positions in SORT and a possibly infinite number of modes in MOID. For example, `jump`, `skip` and `nihil` can derive an infinite number of MOIDS (in fact, these constructs are polymorphic). Therefore, all valid choices of an affix are represented with a place-holder. In this way, evaluation of context conditions is postponed until sufficient information is available for making a deterministic choice.

#### EXAMPLE 5.3.3

```
PRED skip (/ SOID>, >NEST) :
  this position (POS),
  was token (skip symbol),
  [strong animoid skip at POS -> SOID].
```

The affix `animoid` is a MOID which is willing to be unified to any MOID.

### 5.3.5 Context Sensitive Analysis

We will briefly describe the treatment of context dependence in our parser.

#### Declarations and Independence of Properties

Collecting declarations and verifying their independence can be performed more efficiently than in the original syntax.

Our purpose is to collect declarations in the first pass, and identify them in the second pass. For clarity's sake, we will show a somewhat simplified syntax. Embedding `series` with a second LAYER which is initially `new EMPTY EMPTY` and packing both LAYERS in a transient parameter yields:

```

PRED serial clause (/ SOID>, >NEST) :
  empty (LAYER),
  series (LAYER / SOID, NEST LAYER).

```

We have embedded the syntax for `declaration-of-DECS` and its descendants with all declarations collected in the range until now. New declarations are added to the declarations collected until now and returned in the same transient parameter.

```

PRED series (>LAYER> / SOID>, >NEST) :
  split (LAYER, DECSETY, LABSETY),
  declaration (DECSETY / NEST),
  make layer (DECSETY, LABSETY, LAYER),
  should be token (go on symbol),
  forced series (LAYER / SOID, NEST) ;
...

```

The syntax shows that application before definition can be implemented in CDL3 in quite an elegant way.

Notice that in the original syntax the independence check of two properties is evaluated  $2^n$  times for each `LAYER` containing  $n$  properties. Because the predicate `where-PROP1-independent-PROP2` is symmetric and each property can occur in a `LAYER` only once, we have

```

where PROP independent PROPSETY1 PROPSETY2 ≡
  where PROP independent PROPSETY1

```

Application of this equivalence reduces the number of checks for mutual independence to  $2^{n-1}$ .

For `joined-fields-definition` more optimizations are possible. After application of the above equivalence and the equivalence

```

where MODE1 field TAG1 independent MODE2 field TAG2 ≡ unless TAG1 is TAG2

```

the independence check for `FIELDS` can be specialized to a version which can be evaluated in the first pass. We have embedded `joined-fields-definition` with an initially `EMPTY` transient parameter `FIELDSETY` containing the `FIELDS` currently collected. Consider the following syntax:

```

ACTION rest joined fields definition (>MODE, >FIELDSETY> / >FIELDS,
>NEST):
  is another field definition,
  was token (and also token),
  field definition (MODE, FIELD / FIELDS),
  add (FIELDSETY, FIELD, FIELDSETY),
  rest joined fields definition (MODE, FIELDSETY / FIELDS, NEST) ;
+.

```

The transient parameter `FIELDSETY` is an abbreviation for one inherited parameter `FIELDSETY1` and one derived parameter `FIELDSETY2`. Clearly, the invariant

```

where (FIELDS) is (FIELDSETY1 FIELD FIELDSETY3)

```

holds for some FIELDSETY3. This invariant tells us that

where (FIELD) independent (FIELDS)  $\equiv$  where (FIELD) independent (FIELDSETY1)

Therefore, the second pass parameter FIELDS can be eliminated. Embedding field-definition with FIELDSETY and sinking of add into field-definition yields

```
ACTION rest joined fields definition (>MODE, >FIELDSETY> / >NEST) :
  is another field definition,
  was token (and also symbol),
  field definition (MODE, FIELDSETY),
  rest joined fields definition (MODE, FIELDSETY / NEST) ;
+.
```

After lifting the independence test in defining-field-selector, the syntax for field-definition becomes

```
ACTION field definition (>MODE, >FIELDSETY>) :
  this position (POS),
  (defining field selector (TAG),
   make field (MODE, TAG, FIELD) ;
   (where (FIELD) independent (FIELDSETY),
    add (FIELDSETY, FIELD, FIELDSETY) ;
    declaration error ("Field not independent", POS)) ;
   syntax error ("Tag token expected", POS))
```

Notice that MODE is the mode of a declarer. Therefore, it can be inherited in the first pass. Independence of other properties can only be verified in the second pass.

### Identification of Properties

The syntax for identification searches a NEST for the QUALITY belonging to a TAX. We have added a rule for identifying properties in the standard environment:

```
TEST where (QUALITY>, >TAX) identified in (>NEST) :
  split (NEST, NEST1, LAYER1),
  (where (QUALITY, TAX) resides in (LAYER1) ;
   where (QUALITY, TAX) identified in (NEST1)) ;
is primal (NEST),
is standard (QUALITY, TAX).
```

Notice that the predicates `resides` and `independent` are mutually exclusive. Therefore, the test for independence in [Wij76, 721a] always succeeds in an ordered grammar.

The original syntax for operator identification is not deterministic, because operators may be overloaded. We have solved this problem by specializing a copy of `identified` towards operator identification. This copy has been embedded with the a priori modes of the operands of an operator. This means that identifying an operator invokes the coercion mechanism.

## Coercion

In the EAG (and 2VWG), the mode of each construct is coerced inside the construct to its a posteriori mode, while it may be the case that this mode is not yet known. For example, the first operand of a formula yields an a posteriori mode which is determined by the operator *following* the operand. In the EAG this implies that affixes flow from right to left in strong and firm positions. In CDL3 on the other hand, modes cannot be known until the second pass, while in the second pass affixes must flow from left to right.

In CDL3, it is not possible to coerce a construct before its a posteriori mode is known. Otherwise, coercion cannot be made deterministic. A solution to this problem is to let each construct yield its a priori mode instead of its a posteriori mode. For this reason, the placeholders of example 5.1.1 have been introduced. In this way, the nondeterministic choice for an a posteriori mode is delayed until enough information is available for making a deterministic choice. Still, in a `slice` and a `call` only the form of the a posteriori mode is available, and not its actual value. However, one can verify that the syntax for Algol 68 has been formulated in such a way that in a `slice` and a `call` this form of the a priori mode and the syntactic position provide sufficient information for calculating a unique a posteriori mode. For example, let `MODE` be the a priori mode of the primary of a `call`. Then there exists a unique `MODE1` of the form `procedure-with-PARAMETERS-yielding-MOID` such that

can coerce (`MODE`, `morf`) to (`MODE1`) in (`meek`) position

Although we have verified that the syntax for coercion can indeed be made deterministic, we have not yet succeeded in doing so. However, there do not seem to be any fundamental obstacles ahead.

Most nondeterministic predicates are of the form

generate (`X`), test (`X`)

A general strategy for making such constructions deterministic, is to unfold `generate`, enabling subsequent simplification. Alternatively, the predicate `test` can be sunk into `generate`. However, both solutions suffer from the practical problem that it may be necessary to specialize `generate` for many cases. A better solution might be to parameterize `generate` with `test`, possibly by replacing `X` with a place-holder `test X`.

## Balancing

Because we have chosen the `SOID` of a construct to be derived, balancing becomes nondeterministic. For this reason, balancing is delayed by introducing a place-holder `SOID :: balance` of `SOID1` and `SOID2`.

## The Mode-Table

In §4.2.9 it is mentioned that in the original syntax it is unavoidable to make a nondeterministic choice for `MODE` in `applied-mode-indication`. The classical solution for this problem

is to replace each mode with an entry in a mode-table. In this light, the MU-application of a mode can be seen as a reference to an entry in the mode-table, while each MU-definition corresponds to an entry. In our framework, a reference to an entry in the mode-table can be seen as a place-holder for the mode defined in that entry. Whenever a LAYER is finished, the modes defined in it should be checked for missing definitions and cycles and be equivalenced. However, we have not succeeded in deriving this solution transformationally.

## 5.4 Overview of the Parser

### 5.4.1 Module Structure

The parser consists of the following modules:

Module	Description
<code>algo168</code>	Driver
<code>analysis</code>	Non-predicate syntax rules
<code>modes</code>	Construction and access operators on MODE Syntax for mode equivalencing Mode-table Relationships between modes
<code>coercion</code>	Syntax for coercion
<code>nests</code>	Construction and access operators on NEST Syntax for independence and identification
<code>standard</code>	Implementation of standard library
<code>symboltable</code>	Symbol-table
<code>prescan</code>	Pre-scanner and predicates for look-ahead
<code>scanner</code>	Lexical analysis
<code>inout</code>	Interface to OS
<code>errors</code>	Error reporting

The implementation of MOID en NEST has been hidden in the modules `modes` resp. `nests`. The symbol-table is introduced for efficiency reasons.

### 5.4.2 Implementation Status

The `style-i-sub-symbol` and the `style-i-bus-symbol` are not supported. The syntax for `format-text` has not been implemented.

Due to unjustified optimism and unrealistic planning, the author has had insufficient time for completing the pre-scanner, implementing the mode-table and unification of equivalent modes and making the syntax for coercion deterministic.

### 5.4.3 Size of the Parser

Because our implementation has not been completely finished, it is not possible to compare the size of our implementation with the size of the original specification. Moreover, the CDL3

program contains a considerable amount of code for io, lexical analysis, the pre-scan, interfaces and abstraction, which is not present in the 2VWG. However, it is possible to compare the number of hyper-rules of the syntax defined in [Wij76, 2–5] with the number of corresponding hyper-rules in our EAG and CDL3 grammar:

[Wij76, 2–5]	2VWG	EAG	CDL3
no. of rules	249	359	450

In EAG, most extra rules originate from unfolding metanotions which are used as abbreviations and from left-factoring and left-recursion removal. The additional rules in the CDL3 grammar consist mainly of error alternatives.

## 5.5 Practical Usability of CDL3

We will briefly enumerate the restrictions which CDL3 imposes on a two level grammar, as well as the features offered by CDL3, and discuss their implications on the practical usability of CDL3.

- The metagrammar is required to be in near GNF and the use of the metagrammar is to be consistent with its definition. Using our transformations, it is easy to comply to these restrictions.
- CDL3 requires all affixes in the same pass to flow from left to right. Furthermore, affix evaluation is restricted to two passes. This has not turned out to be too restrictive.
- Grammars are required to be free of left-recursion. We have circumvented this restriction without notable problems.
- Grammars are required to be free of local ambiguity. For Algol 68, quite some pain has to be taken in order to satisfy this condition.
- Predicates are required to be deterministic. We have not yet succeeded in making all predicates deterministic, but there do not seem to be fundamental obstacles.
- The static semantic checks of CDL3 have proven to be invaluable in catching many errors at compile-time.
- The module structure of CDL3 provides valuable means of abstraction.

## 5.6 Conclusions

We have presented two more transformations for making grammars deterministic, which are the insertion of place-holders and the decomposition into suitable sub-grammars. The place-holder transformation is useful for simulating delayed evaluation of context conditions. In this way, it has been possible to fix the direction of flow statically. Furthermore, place-holders can be used for inverting affix flow.

In making the underlying context free grammar of Algol 68 deterministic, we have mostly resorted to classical techniques. The parsing properties of the underlying context free grammar can be improved further, but this would conflict with our desire of viewing a grammar both as an implementation and a specification. Furthermore, further left-factoring would complicate two pass affix evaluation.

A strategy has been formulated for transforming specifications in EAG into efficient parsers in CDL3. It is possible to derive parsers whose structure closely resembles the original grammar. Therefore, in our opinion the grammatical programming paradigm has proven its applicability to complex real-life grammars.

It is possible to derive recursive descent parsers for complex languages. The structure of such a parser closely resembles the original language specification. The restrictions imposed by CDL3 have turned out to be no fundamental obstacles. However, some effort may have to be undertaken in order to deal with local ambiguity. Therefore, it may be worthwhile to consider EAG implementations supporting the second level of CDL3 and a nondeterministic context free level.

Unfortunately, we have not succeeded in deriving the whole parser transformationally.

### 5.6.1 Future Work

Our parser for Algol 68 remains to be finished. The original grammar still contains some interesting puzzles to be attacked with transformational methods.

The parser can be extended to a parser for full Algol 68 by implementing `format-text` and implementing the standard environment. Furthermore, the challenge of supporting the `style-i-symbol` remains to be undertaken.

The parser can be embedded with affixes deriving an abstract parse tree which may serve as input for an optimizer and code generator, in order to obtain an implementation of Algol 68.

Many transformations on a CDL3 grammar can be supported automatically. It might be worthwhile to consider a grammar lab giving support to the transformational manipulation of grammars.



# Chapter 6

## Conclusions

### 6.1 Two Level Grammars

We have described three types of two level grammars. In 2VWG there is no strict border between the first and second level. An EAG may be viewed as a 2VWG in which the first level is separated from the second level. CDL3 can be seen as a transparent and deterministic version of EAG.

In a transparent grammar, all substitutions in a conjugation unifying two hypernotations can be obtained by applying direct metaproductions. This allows the second level to be implemented by tree-structured domains. Furthermore, in a transparent grammar it is decidable whether hypernotations are conjoint, and if so, there exists exactly one unifier which can be computed. In unrestricted two level grammars, the set of possible unifiers of two hypernotations may be infinite.

### 6.2 Transformation of Two level Grammars

Left-recursion removal, substitution and left-factorization are well known techniques for making a context free grammar ‘more’  $LL(1)$ . It has been shown that these transformations can be generalized to two level grammars.

We have defined transformations which enable the application of a simple unfold/fold-strategy to the hyper-rules of a two level grammar. Applicability of most transformations is undecidable. However, in a transparent grammar applicability of most transformations is decidable, allowing automatic support. This subject remains to be investigated.

Using our transformations, a two level grammar can be made transparent. Furthermore, it is possible to improve the run-time behavior of a two level grammar in a transformational way.

We have formulated a method for imaging specifications in 2VWG into EAG. However, under-specification in the 2VWG, allowing an infinite number of parses, may lead to EAGs which are not well-formed. This is also the case for Algol 68. Furthermore, we have formulated a strategy for transforming specifications in EAG into efficient parsers in CDL3. We have defined

a transformation for simulating delayed evaluation of context conditions when the two pass evaluation mechanism of CDL3 is too restrictive.

As a case study, we have derived an EAG for Algol 68 transformationally. The grammar is free of left-recursion. However, we have completely not succeeded in making the grammar well-formed. Furthermore, we have tried to transform the EAG into CDL3. We have not been able to finish our parser and to solve all problems transformationally. However, we hope that the framework established in this thesis turns out to be valuable in approaching similar problems.

The main problem in deriving an executable grammar for Algol 68 is to prevent the nondeterministic choice for the spelling of a mode. The original grammar allows many programs to have an infinite number of parses.

### 6.2.1 Extended Affix Grammars

Free affix flow is a useful feature for describing programming languages. However, free affix flow makes it harder to verify whether a grammar is well-formed.

Dynamic typing is not essential for implementing Extended Affix Grammars. Static typing is more efficient and helps the user prevent errors. Using our transformations, the metagrammar can be restricted to be tree-structured and statically type-able. Nondeterministic splitting of affix values can be simulated by nondeterminism on the first level.

Version 1.4 of `eag-compile` is reliable. However, its efficiency should be improved. We have not had the opportunity of investigating the suitability of optimized recursive backup parsing for implementing EAG.

### 6.2.2 CDL3

Left-recursion is not essential for describing programming languages, at least not for describing Algol 68. Left-recursion removal allows the generation of top-down parsers.

The grammatical programming paradigm can be applied to complex real-life grammars. It is possible to derive reliable recursive descent parsers whose structure closely resembles the original grammar. The restrictions imposed by CDL3 turn out to be no fundamental obstacles.

### 6.2.3 Future Work

The grammar for Algol 68 still contains a number of puzzles on which to try transformational methods.

It remains to be investigated to what extent our transformations can be supported automatically. We suggest the inclusion of our transformations in a grammar work lab.

# Bibliography

- [Alb91] H. Albas, B. Melichar (Editors), *Attribute Grammars, Applications and Systems*, LNCS 545, Springer-Verlag, 1991.
- [Bra92] M.G.J. v.d. Brand, *PREGMATIC : a generator for incremental programming environments* PhD thesis, Catholic University of Nijmegen, 1992.
- [Dek92] C. Dekkers, C.H.A. Koster, M.-J. Nederhof, A. van Zwol, *Manual for the Grammar Work Bench Version 1.0*, Department of Informatics, Catholic University of Nijmegen, 1992.
- [Der90] P. Deransart and M. Jourdan (Editors), *Attribute Grammars and their Applications*, Springer-Verlag, 1990.
- [Deu74] P. Deussen, *A Decidability Criterion for van Wijngaarden Grammars*, In: *Acta Informatica* 5, Springer Verlag, 1975.
- [Feu78] H. Feuerhahn, C.H.A. Koster, *Static Semantic Checks in an Open Ended Language*, In: P.G. Hibbard and S.A. Schuman (Editors), *Constructing Quality Software*, North Holland, 1978.
- [Gro92] F. Grootjen, *Efficient Recursive Backup Parsing*, Masters Thesis, Catholic University of Nijmegen, 1992
- [Hed75] G.E. Hedrick, *Proceedings of the 1975 International Conference on Algol 68*, Oklahoma State University, Stillwater, 1975.
- [Hun77] R.B. Hunter, A.D. McGettrick, R. Patel, *LL versus LR Parsing with Illustrations from Algol 68*, In [Sch77].
- [Koc77] W. Koch, C. Oeters, *The Berlin Algol 68 Implementation*, In [Sch77].
- [Kos69] C.H.A. Koster, *Syntax-directed parsing of Algol 68 Programs*, In [Pec69].
- [Kos70] C.H.A. Koster, *Affix Grammars*, In [Pec71].
- [Kos72<sub>1</sub>] C.H.A. Koster, *Towards a Machine Independent Algol 68 Translator*, Stichting Mathematisch Centrum, Amsterdam, 1972.
- [Kos72<sub>2</sub>] C.H.A. Koster, *Error reporting, error treatment and error correction in ALGOL translation*, In: *GI – Jahrestagung*, Springer-Verlag, 1972.

- [Kos74] C.H.A. Koster, *A Technique for Parsing Ambiguous Languages*, In: D. Siefkes (Editor), *GI – 4. Jahrestagung*, LNCS 26, Springer-Verlag, 1975.
- [Kos<sub>1</sub>] C.H.A. Koster, *Affix Grammars for Programming Languages*. In: [Alb91]
- [Kos91<sub>2</sub>] C.H.A. Koster, *Affix Grammars for Natural Languages*, In: [Alb91].
- [Kos91<sub>3</sub>] C.H.A. Koster, J.G. Beney, *On the Borderline Between Grammars and Programs*, In [Mał91].
- [Kos92] C.H.A. Koster, *Lecture notes of a course in compiler construction*, Handouts, Department of Informatics, Catholic University of Nijmegen, 1992.
- [Kos93] C.H.A. Koster, *Informatics and Syntax*, Akten des 26. Linguistischen Kolloquiums, Poznań 1991, In: *Sprache – Kommunikation – Informatik*, Max Niemeyer Verlag, Tübingen, 1993.
- [Kos94] C.H.A. Koster, J.G. Beney, *Draft CDL3 Manual*, Department of Informatics, Catholic University of Nijmegen, 1994.
- [Krä78] B. Krämer, H.W. Schmidt, *On the Implementation of van Wijngaarden Grammars. Part I: Informal Introduction of Conjugation Grammars*, IST-Report No. 44, Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn, June 1978.
- [Kül87] P. Küling, *Affix-Grammatiken zur Beschreibung von Programmiersprachen*, Technische Universität Berlin, Fachbereich Informatik, 1987.
- [Mai68] B.J. Mailloux, *On the Implementation of Algol 68*, PhD Thesis, University of Amsterdam, 1968.
- [Mał82] J. Małuszyński, *Grammatical Unification*, In: *Information Processing Letters*, Vol. 15, No. 4, North-Holland, Amsterdam, 1982.
- [Mał84] J. Małuszyński, *Towards a Programming Language Based on the Notion of Two-level Grammar*, In: *Theoretical Computer Science*, Vol. 28, North-Holland, Amsterdam, 1984.
- [Mał91] J. Małuszyński, M. Wirsing (Editors), *Programming Language Implementation and Logic Programming*, LNCS 528, Springer-Verlag, 1991.
- [Mei86] H. Meijer, *Programmer: A Translator Generator*, PhD thesis, Catholic University of Nijmegen, 1986.
- [Mei90] H. Meijer, *The project on Extended Affix Grammars at Nijmegen*, In [Der90].
- [Par90] H.A. Partsch, *Specification and Transformation of Programs, A Formal Approach to Software Development*, Springer-Verlag, 1990.
- [Pec69] J.E.L. Peck (Editor), *Proceedings of an informal conference on Algol 68 Implementation*, University of British Columbia, Vancouver, 1969.
- [Pec71] J.E.L. Peck (Editor), *Algol 68 Implementation*, North-Holland, Amsterdam, 1971.

- [Sch77] V.B. Schneider (Editor), *Proceedings of the Strathclyde Algol 68 Conference*, University of Strathclyde, Glasgow, 1977.
- [Sar95] J. Sarbo, *Lecture Notes VB2: Unification based formalisms and their implementation*, Lecture notes of an advanced course in compiler construction, Department of Informatics, Catholic University of Nijmegen, 1995.
- [Seu93] M. Seutter, *Informal introduction to the Extended Affix Grammar formalism and its compiler*, Technical Report No. 93-19, Catholic University of Nijmegen, 1993.
- [Stie75] J. Stier, E.A. Akkoyunlu, *On the Terminal Objects of Algol 68*, In [Hed75].
- [Wai85] W.M. Waite, G. Goos, *Compiler Construction*, Springer-Verlag, 1985.
- [Wat74] D.A. Watt, *Analysis-Oriented Two-Level Grammars*, PhD thesis, University of Glasgow, January 1974.
- [Weg80] L.M. Wegner, *On Parsing Two-Level Grammars*, In: Acta Informatica 14, Springer-Verlag, 1980.
- [Wij76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker (Editors), *Revised Report on the Algorithmic Language Algol 68*, Springer-Verlag, 1976.
- [Wil92] R. Willems, C.H.A. Koster, E. Boiten, M. van den Brand, *Parsing Methods for Natural Languages*, Syllabus, Department of Informatics, Catholic University of Nijmegen, 1992.



# Index

- 2VWG, 4
- abstract algorithm, 24
- abstract object, 72
- abstraction, 6, 35
- TEST, 23
- add
  - blind alley, 37
  - hyper-rule, 36
  - superfluous metarule, 35
  - unused metarule, 35
- additional metarule, 5
- admissible parse tree, 17
- affix, 12, 18
  - applying
    - occurrence, 23
    - position, 23
  - applying only, 17
  - assignment, 14, 22
  - critical position, 59
  - defining
    - occurrence, 23
    - position, 23
  - delayed evaluation, 16
  - directed parsing, 16
  - expression, 12, 19, 21
  - flow, 23
  - language, 14, 19
  - meta-defined, 14
  - nonterminal, 12, 18
  - position, 12
    - applying, 17
    - critical, 17
    - defining, 17
    - derived, 12, 19
    - inherited, 12, 19
    - number of, 19
  - production rule, 12, 19
  - term, 14
    - terminal, 12, 19
- affix flow, 16
- algorithm
  - abstract, 24
  - classification, 19
- algorithm classification, 82
- alternative, 4, 13, 19
- ambiguity, 90
  - local, 92
- application of context information, 88
- apply property, 38
- applying
  - affix
    - occurrence, 23
    - position, 17, 23
- applying only, 17
- assignment guard, 21
- associated function, 19
  
- balancing, 96
- basic
  - hyper-rule, 5, 14, 19, 22
  - hypernotation, 5, 14, 22
- blind alley, 7, 37
  
- CDL3, 18
- change order, 44
- checks
  - static semantic, 23
- class, 53
- classification of algorithms, 19
- coercion, 64, 96
- compiler correctness, 1
- complete set of conjugations, 9
- compliant substitution, 6
- composition operator, 17
- concrete object, 72
- conjoint, 9
- conjugation, 9

- grammar, 10
  - rule, 10
- conjugation grammar, 61
- consistent substitution rule, 14, 22
- consistent substitution rule, 5
- constant parameter
  - removal, 44
- context information, 88
- control, 12, 19
- correctness
  - compiler, 1
- critical affix position, 17, 59
- cross reference relation, 53
- current meta-derivation, 10
- currying, 85
  
- declarations, 93
- decomposition, 84
- decomposition of grammars, 87
- defect, 24
- defining
  - affix
    - occurrence, 23
    - position, 17, 23
- delayed affix evaluation, 16
- delete
  - blind alley, 37
  - hyper-rule, 36
  - superfluous metarule, 35
  - unused metarule, 35
- dependency graph, 17
- derived
  - affix position, 12, 19
  - hyper-rule, 5
  - hypernotation, 5
- deterministic
  - grammar, 81
- deterministic grammar, 22
- direct
  - metaproduction, 5
  - production relation, 7, 15, 22
- director set, 22
- disjoint, 81
- display, 14, 21
- domain, 5, 14, 19
  
- EAG, 12
  
- effect, 23
- embedding, 44
- enclosed group, 20
- equal guard, 21
- equality test
  - transformation of, 46
- equivalence
  - strong, 53
  - weak, 28
- equivalent
  - grammars, 28
  - hypernotations, 28, 29
  - rules, 28, 30
- evaluation
  - symbolic, 44
- expansion, 30
- expression
  - syntax of, 24
- extended affix grammar, 12
  
- faithful substitution, 6
- first pass parameter, 19
- fixed flow, 92
- flow, 47
  - affix, 16
  - fixed, 92
  - inversion, 67
- flow inversion, 84
- flow symbol, 13
- fold
  - hypernotation, 33
  - metanotation in hyper-rule, 31
  - metanotation in metarule, 30
- forbidden symbol, 18
- formulas, 66
- foundation
  - sematical, 27
- free affix flow, 16
- function, 55
- FUNCTION, 23
  
- global variable, 24
- granularity, 84
- guard, 21
  - assignment, 21
  - equal, 21
  - implicit, 21

- join, 21
  - split, 21
  - sub, 21
- head, 14, 21
- head grammar, 16
- hyper-derivation, 9
- hyper-replacement, 11
- hyper-rule, 4, 12, 19
  - basic, 5, 14, 22
  - derived, 5
- hypernotation, 4, 12, 19
  - basic, 5, 14, 22
  - derived, 5
  - start, 4, 12, 19
- identification, 95
- implementation grammar, 29
- implicit guard, 21
- inconsistent
  - use of metagrammar, 7
- independence, 93
- inherited affix position, 12, 19
- interlude, 24
- invariant
  - introduction, 43
  - removal, 43
- inversion of flow, 84
- invert flow, 67
- join guard, 21
- junction, 10
- language, 7, 15, 22
- layout, 73
- left hand side, 4, 13, 19
- left-corner, 43
- left-factoring, 62
- left-factorization, 39
- left-recursion, 62
- left-recursion removal, 41
- lexical
  - analyzer, 86
  - grammar, 87
- lexical analysis, 73, 87
- lexical grammar, 72
- lifting, 43
- list
  - transformation of, 46
- local ambiguity, 92
- logic rule, 47
- longest match, 87
- look-ahead, 89
- main grammar, 87
- main metanotation, 11
- mark, 72
- marker, 88
- matching hypernotations, 42
- member, 4, 13, 19
- meta-defined affix, 14
- meta-nonterminal, 4
- meta-variable, 4
- metagrammar, 5, 14, 19, 21
  - transformation, 45
- metanotation, 4, 12, 18
  - main, 11
- metaproduction
  - direct, 5
  - relation, 5
  - terminal, 5
- metarule, 4, 12, 19
  - additional, 5
- middler, 89
- modes, 96
- module, 24
- more general unifier, 11
- most general unifier, 11
- nonterminal
  - affix, 12, 18
  - symbol, 12, 18
- notion, 7
- ordered grammar, 19, 47
- pass, 19
- pattern grammar, 53
- peeling rule, 73
- place-holder, 85
- postlude, 24
- pre-scan, 88
- PRED, 23
- predicate, 4, 13, 16, 19
  - introduction, 38
  - lifting, 43

- primitive, 18
- removal, 38
- sinking, 43
- prefix sharing, 90
- prelude, 24
- primitive predicate, 18
- principle of longest match, 87
- production
  - direct, 7
  - relation, 15
  - rule, 6, 14, 15, 22
    - affix, 12
- productive, 7
- prompt evaluation, 16
- proper representation, 72
- protonotion, 4, 14, 22
- pruning, 35, 55
- punning, 56
  
- recognition problem, 8
- recomposition, 46
- reference language, 72
- referencing problem, 8
- removal
  - of constant parameter, 44
  - of superfluous parameter, 44
- renaming, 44
- representation, 7, 72
  - composite, 73
  - language, 72
- reserved symbol, 73
- restriction, 42
- right hand side, 4, 13, 19
- rule schemata, 6
  
- schemata
  - rule, 6
- second pass parameter, 19
- sematical foundation, 27
- semi-terminal, 13
- separated grammar, 54
- separation, 54
- separation of levels, 56
- sinking, 43
- skeleton grammar, 53
- specialization, 36
- specification grammar, 29
  
- split guard, 21
- start hypernotation, 4, 12, 19
- static flow, 17
- static semantic checks, 23
- strict language, 7, 72
- strictness annotation, 73
- strongly equivalent, 53
- sub-guard, 21
- substitution, 6, 7
  - compliant, 6
  - consistent, 5, 14, 22
  - faithful, 6
  - uncompliant, 6
- superfluous
  - hyper-rule, 35
  - metarule, 35
- superfluous parameter
  - removal, 44
- symbol grammar, 87
- symbolic evaluation, 44
- synonym, 5, 25
- syntactic mark, 4
- syntax directed programming, 18
  
- terminal
  - affix, 12
  - metaproduction, 5
  - symbol, 4, 12, 18
- termination, 18
- TEST, 23
- token grammar, 87
- transformation
  - of metagrammar, 45
  - on lists, 46
  - rule, 29
  - using equality test, 46
- transient parameter, 22
- transparent, 25, 61
- transparent grammar, 11
- tuple, 17, 75
- two level van Wijngaarden grammar, 4
- type, 17, 19, 75, 82
  
- unambiguous
  - lexical analysis, 86
- uncompliant substitution, 6
- underlying context free grammar, 16

unfold

  hypernotation, 33

  metanotation in hyper-rule, 31

  metanotation in metarule, 30

unifier, 11

  more general, 11

  most general, 11

uniquely assignable, 39

unused metarule, 35

used production rule, 36

variable

  global, 24

weak equivalence, 28

well-formedness conditions, 15, 17