Internal Use Only

TR-I-0054

Pragmatic Extensions to Unification-Based Formalisms.

単一化に基づく形式化の実用的な拡張

Yves NICOLAS
イブ・ニコラ

November, 1988

Abstract

A type system for feature structure has been developed in terms of initial algebra theory of abstract data types. In this system, there are essentially two kinds of feature structure types: atomic and complex types. Usual simple types such as boolean, numbers or strings are used as atomic types. Complex types are defined recursively by means of the set of feature/value pairs. Operations defined on atomic types enable us to avoid unification when it is unnecessary and hence to reduce computational cost. As an application of this type system to grammatical formalisms, an Earley-like parser using this system has been developed and integrated inside a small environment including trace and step functions.

ATR Interpreting Telephony Research Laboratories
ATR 自動翻訳電話研究所

Pragmatic Extensions to Unification-Based Formalisms

Yves NICOLAS*

ATR Interpreting Telephony Research Laboratories
Twin 21 MID Tower
2-1-61 Shiromi
Higashi-ku
Osaka 540
Japan

ABSTRACT. A type system for feature structures has been developed in terms of initial algebra theory of abstract data types. In this system, there are essentially two kinds of feature structure types: atomic and complex types. Usual simple types such as boolean, numbers or strings are used as atomic types. Complex types are defined recursively by means of the set of feature/value pairs. Operations defined on atomic types enable us to avoid unification when it is unnecessary and hence to reduce computational cost. As an application of this type system to grammatical formalisms, an Earley-like parser using this system has been developed and integrated inside a small environment including trace and step functions.

RESUME Nous avons développé un système de feature structures typées dans l'optique de la théorie algébrique des types abstraits. Dans ce système coexistent deux sortes de types de feature structures: les types atomiques et les types complexes. Comme types atomiques, nous utilisons des types simples usuels, tels que booléen, nombre ou chaîne de caractères. Les types complexes sont définis récursivement par leurs attributs. Les opérations définies sur les types atomiques permettent d'éviter l'unification quand elle n'est pas nécessaire et de réduire ainsi le temps de calcul. Comme application de ce système à la linguistique computationnelle, nous avons implanté un analyseur inspiré de celui d'Earley et qui construit une feature structure typée représentant la phrase au cours de l'analyse. Cet analyseur a été intégré au sein d'un petit environnement comprenant des fonctions de trace et permettant le débogage de grammaires.

* Intern student from the ENST (Ecole Nationale Supérieure des Télécommunications), Paris, France.

Table of contents

INTRODUCTION 4

F-STRUCTURES 5

I. Presentation 5

II.Classical definitions 8

III The f-structure type system 13

THE PARSER 28

I. Earley's algorithm 28

II. The interface evaluators 32

III The grammar formalism 34

IV. The algorithm 36

THE PAULINE SYSTEM 45

I. Introduction 45

II. The top level menu 45

III. Structures 48

IV. Evaluators 51

V. Grammar rules and parsing 53

VI. the tracer 55

CONCLUSION 61

APPENDIX: A PURE UNIFICATION GRAMMAR 62

REFERENCES 65

ACKNOWLEDGEMENTS

I would like to thank Dr. Akira Kurematsu, President of ATR Interpreting Telephony Research Laboratories, for inviting me to his laboratory.

I would also like to thank Mr. Teruaki Aizawa, Head of the Natural Language Understanding Department, who welcomed me in his department and supported my research.

I am also very grateful to Kyoshi Kogure, who directed my research and gave me constant support and advice, and to Rémi Zajac, visiting researcher from the Groupe d'Etudes pour la Traduction Automatique (Grenoble, France), with whom I had many fruitful discussions.

I would also like to thank Dr. Hisao Kuwabara for taking care of all administrative formalities before and during our stay in Japan, and also Mr Isao Ito, who took care of all our material problems.

I would also like to thank Frank Giacobbi who corrected the English text. I am responsible for all remaining mistakes, due to last minute changes.

I would also like to extend this acknowledgment to all the employees of ATR who, through their constant kindness, helped me to rapidly acquaint me with the Japanese way of life.

INTRODUCTION

Unification-grammars are now widespread in natural-language processing and computational linguistics. Their success is partly due to a simple intuitive encoding of lexical and grammatical information inside feature structures and to a very powerful mechanism used to combine this information, that is unification. Unification is used in several domains of computer science such as theorem proving, logic programming, and artificial intelligence. It is one of the basic operations of the PROLOG language. Its application to computational linguistics enables us to handle complex phenomena, either semantic or syntactic, in a simple homogeneous way. However, one can wonder why there is not at this time any real-size natural language processing system using pure unification as the basic operation. In fact, unification-based formalisms raise two major problems:

- Unification algorithms are expensive. [BAR 87] showed for example that the recognition problem for GPSG is NP-complete, solvable in exponential time.
- If unification can solve simply complex problems, it often fails in performing simple operations, or requires encoding them using complex, non intuitive mechanisms. Performing arithmetic operations with pure unification can be done only by redefining integers from the constant zero and the predecessor and successor operations.

Therefore, real-size natural language processing systems should include both unification and some less powerful mechanisms to handle simple operations in the most efficient way.

These facts were our basic assumptions while beginning this work. The solution we chose to integrate operations other than unification was to develop a type system for feature structures in which two kinds of types occur:

- Atomic types. They are the usual simple types such as numbers, boolean or strings of characters. Their specification is imported in the formalism and thus we are able to perform on them their specific operations such as arithmetic, concatenation, etc. Toward unification, objects of these atomic types behave just like atomic values do in the usual approaches to feature structures and unification.
- Complex types. They handle feature/value pairs. Such types are defined by a set of valid features, considered operations on the type.

The type system has been developed using the algebra approach to data types as it can be found in [GOG 78]. The first chapter of this work is devoted to its description.

After designing this type system, we had to integrate it inside a parsing algorithm. We modified Earley's parsing algorithm so that it can build representation structures for the parsed sentence. This algorithm has been developed with the feature structure type system in mind but it is in fact independent of the particular chosen representation structures. Thus, it can handle pure unification grammars, but also grammars written to represent sentences by trees, forests or other kinds of structures. The communication between the structures module and the parsing algorithm is made through three well specified interface evaluators. The second chapter of this work presents the specification of these evaluators, the underlying grammar formalism and a description and proof of our algorithm.

We finally went to an implementation work and integrated both the feature structure type system and the parsing algorithm in an environment which includes some trace functions. The program has been called PAULINE, which stands for Parsing Augmented Unification-grammars: a Linguistic INteractive Environment. The third and last chapter of this work is devoted to the reference manual of this program.

Chapter I
F-STRUCTURES

I. PRESENTATION

I.1 UNIFICATION GRAMMARS

Unification grammars come from parallel research in theoretical linguistics, computational linguistics and logic programming. Among the major works in the realm, are the *Lexical Functional Grammar* (LFG) from Bresnan and Kaplan, [KAP 83], and the *Functional Unification Grammar* from Kay, [KAY 83]. Later, Gazdar originated the *Generalized Phrase Structure Grammar* (GPSG), [GAZ 85]. Implementation works on GPSG led to the *Head-driven Phrase Structure Grammar* (HPSG), [POL 87]. Other such works led to prototypes like D-PATR. Independently, Colmerauer's work on Q-systems, metamorphosis grammars and PROLOG led to other formalisms such as the *Definite Clause Grammars* (DCG), [PER 80].

The common characteristic of all those formalisms is a model for representing and combining information, which is encoded inside feature structures, or f-structures, on which a unique operation is performed, namely unification, which corresponds roughly to the merging of information found in the two arguments.

I.2 F-STRUCTURES

F-structures are a generalization of record structures usual in computer languages (*structure* types in C or COMMON-LISP, *record* types in Pascal or ADA). They can be defined as a set of feature/value pairs. For example, the agreement information for a noun can be encoded in the following structure:

$$s_1 \begin{bmatrix} \text{gender} & \text{masculine} \\ \text{number} & \text{plural} \end{bmatrix}$$

Two points make f-structures different from the classical attribute structures used in computational linguistics:

• Composition

The value of a feature in an f-structure can itself be an f-structure. For example, the lexical information for *John* in a dictionary could be encoded in the following structure:

$$s_2 \begin{bmatrix} \text{cat} & \text{N} \\ \text{agreement} & \begin{bmatrix} \text{gender} & \text{masculine} \\ \text{number} & \text{plural} \end{bmatrix} \end{bmatrix}$$

This leads to the notion of *path* as a list of features. For example, the value of the path <agreement number> in s_2 is *plural*. Since it is not a composed f-structure, we speak in this case of an *atomic* f-structure. Composed f-structures will be referred to as *complex* f-structures. We will clarify these ideas later.

• Reentrancy

Two paths can share a same value inside a f-structure. This phenomenon, called reentrancy, is noted by boxed indices. The subject-verb agreement rule can, for example, be encoded in the following structure[1]:

[1] The linguistic examples are derived from [SHI 86] and HPSG.

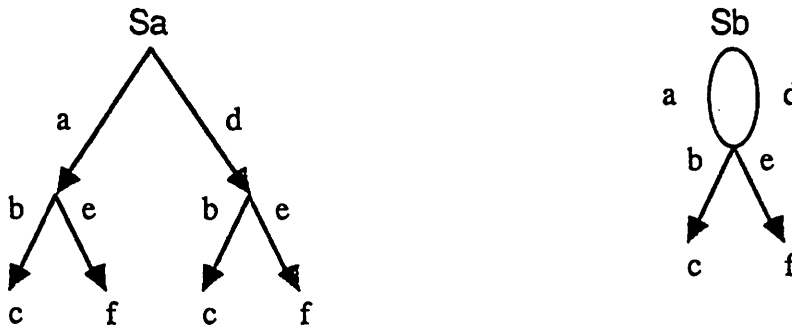
$$\begin{bmatrix} \text{cat} & \text{S} \\ \text{head} & \begin{bmatrix} \text{cat} & \text{V} \\ \text{subject} & \boxed{1} \end{bmatrix} \\ \text{subject} & \begin{bmatrix} \text{cat} & \text{NP} \\ \text{head} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

It is necessary to distinguish between reentrant equality, or identity, and simple structure equality. Let us consider:

$$s_a \begin{bmatrix} \text{a} & \begin{bmatrix} \text{b} & \text{c} \\ \text{e} & \text{f} \end{bmatrix} \\ \text{d} & \begin{bmatrix} \text{b} & \text{c} \\ \text{e} & \text{f} \end{bmatrix} \end{bmatrix} \quad \text{and} \quad s_b \begin{bmatrix} \text{a} & \boxed{1} \begin{bmatrix} \text{b} & \text{c} \\ \text{e} & \text{f} \end{bmatrix} \\ \text{d} & \boxed{1} \end{bmatrix}$$

The values of paths <a> and <d> are identical in s_b but not in s_a. In both cases, they are equal[1].

We have always represented f-structures as matrices. Another representation is perhaps more adequate for reentrancy: an f-structure can be seen as a node of a directed acyclic graph (DAG), with each feature/value pair corresponding to an outcoming edge of this node. For example, the above s_a and s_b can be represented by the following graphs:



Two structures are then identical if and only if they are represented by the same node in the graph, and equal if the labels of the outcoming edges are the same, their ends being also equal. The graph leaves (nodes without any outcoming edge) are atomic f-structures. They are called variables if they have no value. If the structure does not include any reentrancy, its graph representation is a tree.

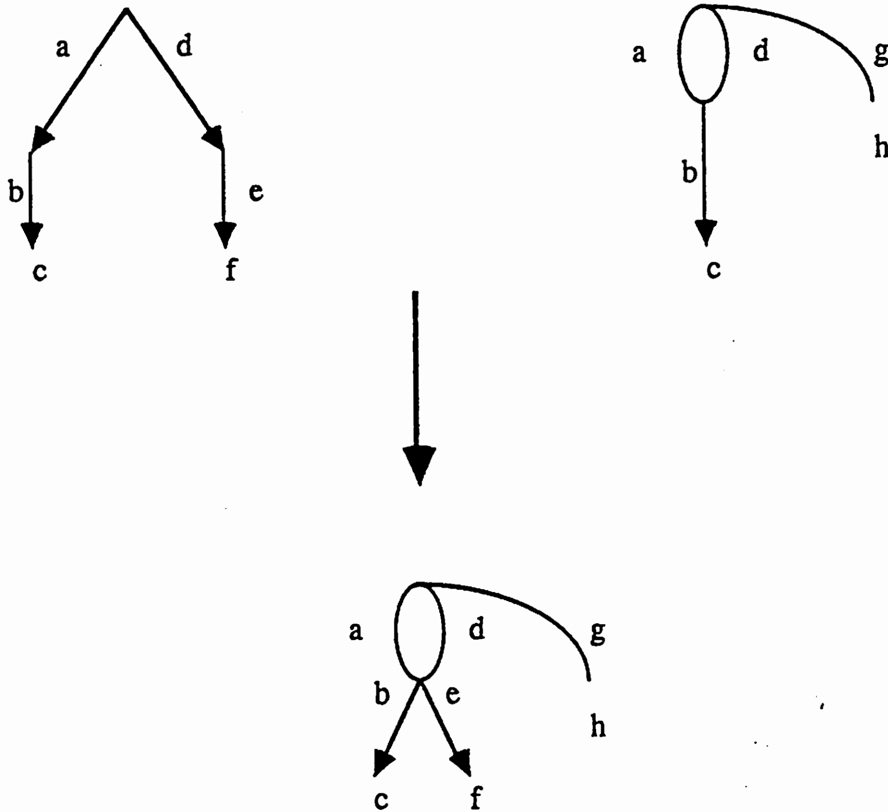
I.3 UNIFICATION

Unification allows combining information found in two structures while checking its consistency. We merely present examples here, before giving a precise definition in section II.

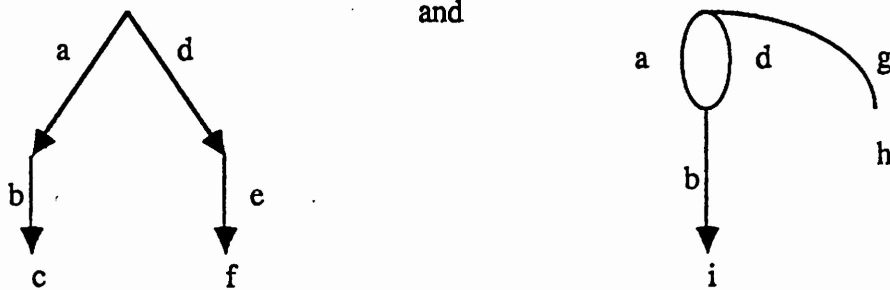
Roughly, unification is defined as follows: a variable f-structure can be unified with any, the result of the operation being the non-variable f-structure. Two atomic f-structures unify if and only if they are equal, the result being either of the two arguments. In the unification of two complex f-structures, the features set of the result is the union of the two features sets, the value of a feature in the result being the unification of the values of this feature in the two argument structures.

[1] The difference between identity and equality can be compared to the one between *eql* and *equal* functions in LISP.

EXAMPLE 1



EXAMPLE 2



can not unify since the values of path <a b> in the two structures are not compatible.

Finally, we present a sample unification grammar which handles subject-verb agreement in sentences. The example is derived from [SHI 86].

We consider the following grammar rule:

(R) S → NP VP
<S head> = <VP head>
<S head subject> = <NP head>
<S subject> = <NP>

where equal signs stand for unification. Let us first parse the French sentence *Les chiens aboient*. (*The dogs bark*). We assume that other grammar rules build the following structures:

$$\textit{les chiens}\begin{bmatrix} \text{cat} & \text{NP} \\ \text{head} & \begin{bmatrix}\text{gender} & \text{masculine}\\ \text{number} & \text{plural}\end{bmatrix}\end{bmatrix}$$

$$\textit{aboient}\begin{bmatrix} \text{cat} & \text{VP} \\ \text{head} & \begin{bmatrix}\text{tense} & \text{present indicative}\\ \text{subject} & \begin{bmatrix}\text{number} & \text{plural}\\ \text{person} & 3\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

At the reduction of rule R, the structure for S is void (variable). The first unification leads to

$$\begin{bmatrix} \text{head} & \begin{bmatrix}\text{tense} & \text{present indicative}\\ \text{subject} & \begin{bmatrix}\text{number} & \text{plural}\\ \text{person} & 3\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

the second to

$$\begin{bmatrix} \text{head} & \begin{bmatrix}\text{tense} & \text{present indicative}\\ \text{subject} & \begin{bmatrix}\text{number} & \text{plural}\\ \text{person} & 3\\ \text{gender} & \text{masculine}\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

and the last to

$$\begin{bmatrix} \text{head} & \boxed{1}\begin{bmatrix}\text{tense} & \text{present indicative}\\ \text{subject} & \begin{bmatrix}\text{number} & \text{plural}\\ \text{person} & 3\\ \text{gender} & \text{masculine}\end{bmatrix}\end{bmatrix} \\ \text{subject} & \boxed{1}\end{bmatrix}$$

While parsing the incorrect sentence *Les chiens aboit* (*The dogs barks*), we would have had:

$$\textit{aboit}\begin{bmatrix} \text{cat} & \text{VP} \\ \text{head} & \begin{bmatrix}\text{tense} & \text{present indicative}\\ \text{subject} & \begin{bmatrix}\text{number} & \text{singular}\\ \text{person} & 3\end{bmatrix}\end{bmatrix}\end{bmatrix}$$

and the second unification would have failed. The sentence would not have been accepted.

II. CLASSICAL DEFINITIONS

The definitions and the algorithm presented in this section are derived from works such as [SHI 86], [PER 86] and [POL 87].

II.1 FORMAL DEFINITIONS

Let A and F be two possibly infinite sets.

A is the atomic values set.
F is the feature set.

Let $\perp$ and T be two objects, both of them neither in A, nor in F.

The f-structures set is defined by

$$S = \bigcup_{k=0}^{\infty} S_k$$

where sets S_k are defined recursively as follows:

$S_0 = A \cup \{T\}$

if $k \geq 1$, S_k is the set of mappings s from F to S_{k-1} such that $\{f \in F / s(f) \neq T\}$ is finite.

Elements of A are called *atomic* f-structures. Elements of S_k for $k \geq 1$ are called *complex* f-structures. The constant Top (T) is also called *variable* f-structure.

We define a path as a string of features, that is, an element of F*. For the sake of simplicity, if $p = < f_1, \dots, f_n >$ is a path, and for $s \in S_k$ with $k \geq n$, we shall note:

$$s(p) = (\dots(s(f_1))(f_2))\dots)(f_n)$$

We can now define unification as the commutative operation $\sqcup$ from $S \times S$ to $S \cup \{\perp\}$ such that

$\forall s \in S \quad s \sqcup T = s$

$\forall s_1 \in A \quad \forall s_2 \in A \setminus \{T, s_1\}$

$s_1 \sqcup s_2 = \perp$

$\forall s \in A \quad s \sqcup s = s$

$\forall s_1, s_2 \in S \setminus S_0$

if $\exists f \in F / s_1(f) \sqcup s_2(f) = \perp$

then $s_1 \sqcup s_2 = \perp$

else $\forall f \in F \quad (s_1 \sqcup s_2)(f) = s_1(f) \sqcup s_2(f)$

II.2 A UNIFICATION ALGORITHM

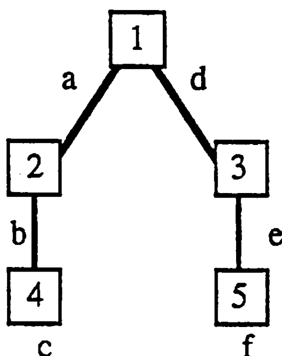
We assume here that f-structures are represented as directed acyclic graphs (DAG, see section I). To take reentrancy into account, we need to define *forwarding* of nodes. If a node n_1 is forwarded to a node n_2, the first one becomes invisible and each access to n_1 is actually an access to n_2. Because of possible successive forwardings, there can be some sequences of invisible nodes. If one wants to consult a node, one has to find the last non-forwarded node of such a sequence. This operation is called dereferenciation.

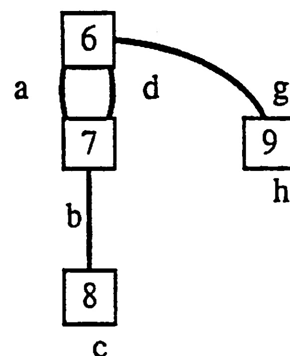
Let G be the graph of the two structures to be unified. Its edges are represented by 3-tuples (n_d, f, n_a). Such an edge goes from node n_d to node n_a, f being its label. If the node has no outcoming edge, it can have a value v. In that case, it represents the atomic f-structure v. If the node has no value, it represents the variable f-structure T.

We can now present the unification algorithm, taken from [PER 85]

```
UNIFY
    INPUT:    2 nodes n1 and n2
    OUTPUT:   a node or ⊥

    dereference n1 and n2

    if      n1 and n2 are identic
            return n1
    else
        if  n1 = T
            forward n1 to n2
    else
        if  n2 = T
            forward n2 to n1
    else
        if  n1 and n2 are atomic and equal
            forward n2 to n1
    else
        if  n1 and n2 are complex
            SHARED = {(n2,f,nf) ∈ G / ∃ (n1,f,n'f) ∈ G}
            COMPLEMENTS = {(n2,f,nf) ∈ G} \ SHARED
            forward n2 to n1
            for each edge(n1,f,nf) ∈ G
                if ∃ (n2,f,n'f) ∈ SHARED
                    n = UNIFY(nf,n'f)
                    if n=⊥      return ⊥
                    else        G = (G\{(n1,f,nf)}) ∪ {(n1,f,n)}
            for each edge (n2,f,nf) ∈ COMPLEMENTS
                G = G ∪ {(n1,f,nf)}
    else
        return ⊥

    return n1
```

EXAMPLE. As an example, we apply this algorithm to example 1 of paragraph I.3. In the following diagrams, forwardings are noted by means of arrows. The two structures to be unified are





The algorithm acts as follows:

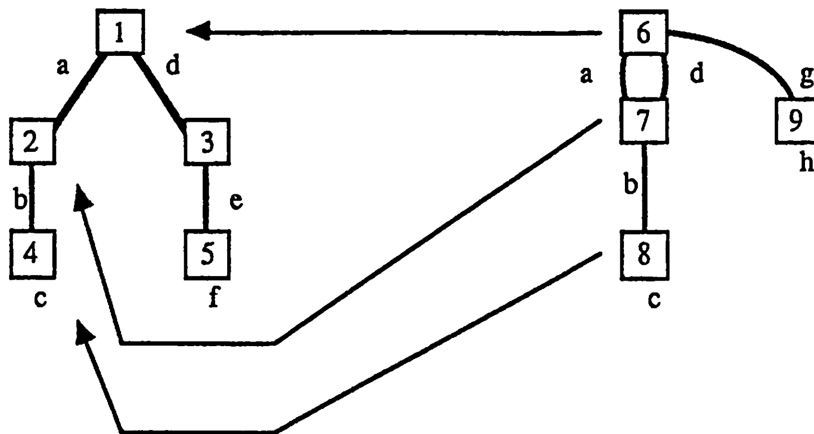
UNIFY([1],[6])

SHARED = { ([6],a,[7]) , ([6],d,[7]) }

COMPLEMENTS = { ([6],g,[9]) }

→ UNIFY ([2],[7]) = [2]

and the graph becomes:



→ UNIFY ([3],[7])

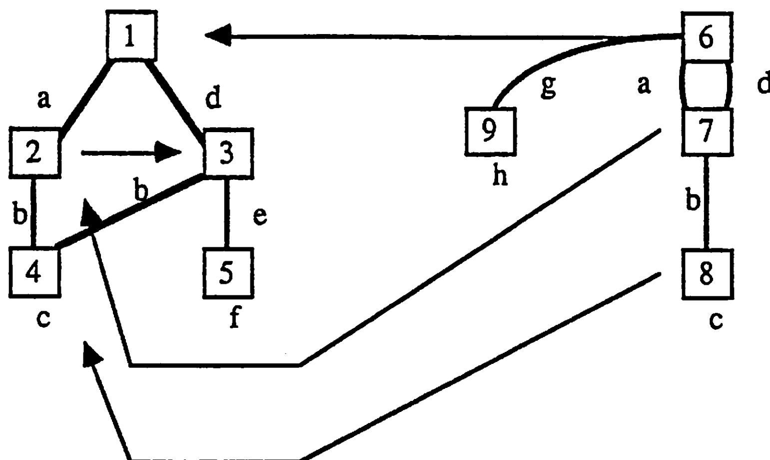
dereference [7] → [2]

SHARED = ∅

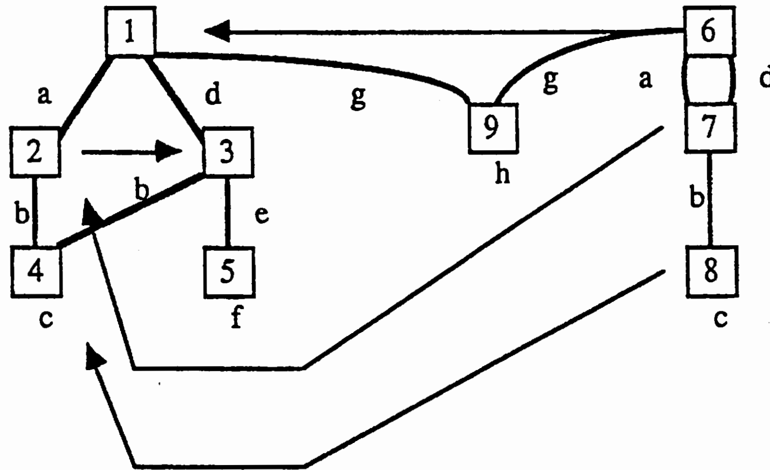
COMPLEMENTS = { ([2],b,[4]) }

= [3]

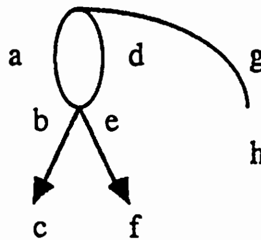
and the graph becomes:



In the unification of [1] and [6], there only remains adding the edge from COMPLEMENTS. The final graph is then:



The returned result is [1]. If we consider forwardings, this is equivalent to:



which is the result we are looking for.

II.3 COMMENTS

This approach to f-structures and unification lacks precision. First of all, the formal definitions given in paragraph II.1 do not take reentrancy into account. Representing f-structures as graphs allows a proper definition of reentrancy but, in that case, unification is defined by means of an algorithm which operates on a particular data-structure, that is, graphs. Besides, the algorithm described above is destructive. It irreversibly alters its argument, and it is thus necessary to work on copies. Doing these copies can be costly and redundant. Many solutions have been proposed to solve such problems: include in each node an updatings slot, which records the modifications to the initial structure, which is not modified itself (see [PER 85]); include in each node a copy slot so that a structure is altered if and only if it is already a copy (see [WRO 87]). The algorithm that will be presented in section III and where reentrancy is described by means of node indexing, is similar to Wroblewski's algorithm. It thus performs a non destructive unification[1].

On the other hand, this classical point of view on f-structures is rather simple and very homogeneous, since there is only one data type, namely f-structures and one operation, namely unification. This is, in fact, a very powerful operation which enables it to do complex things in a very simple and elegant way. But it is costly and it often

[1] Non destructive unification is sometimes called *conjunction*. The definition of conjunction given in [ZAJ 88] is formally equivalent to unification, the difference being that it is considered an operation and not a procedure with side effects on its arguments.

fails in performing simple operations. For example, it is not easy, nor obviously intuitive, to describe arithmetic operations on integers by means of f-structures and unification. In any event, it is much more complex than the usual implementations of integer data type. One of our purposes was to study how to design a framework where both unification and classical implementation of simple operations could take place. This was done by means of the f-structure type system that will be described below.

There are many other approaches to f-structures and unification. [KAS 86] presents, for example, an equational formalism. It is mathematically very elegant but looses the intuitive description of f-structures as feature/value pairs, which is much more convenient for a grammar writer. [ZAJ 88] presents f-structures as hierarchical sets and define a complete framework of operations on complex f-structures. [AIT 86] presents a good approach to reentrancy by means of index function. We will describe it later.

III THE F-STRUCTURE TYPE SYSTEM

In this section, we present a specification for an f-structure type system. An abstract data type is defined by its objects, the operations that can be performed on them and the equations binding these operations. We define two kinds of f-structure types: atomic types and complex types. The first are the usual simple types such as boolean, number or string. Their specification is incorporated in the formalism so that we can use their operations. The others are defined recursively by the set of their valid features, considered as operations on the type. Reentrancy is described by means of index functions as in [AIT 86]. On these complex types, we define the subsumption ordering. The unification of two f-structures will then be their greatest lower bound according to this order.

After outlining the algebraic theory of abstract data types, we described first the modifications to the imported specification for atomic types, then the way to define complex types and finally present a unification algorithm.

III.1 THE ABSTRACT DATA TYPES THEORY

This paragraph recalls the notion of abstract data type specification as it is presented in [GOG 78]. Its purpose is to give the definition of a specification and to give the major points of the reasoning that shows that a specification can define a type in a unique way. The notations are taken from [GOG 78] and will be used throughout all of section III.

III.1.1 Presentation

An abstract data type possibly involves many *sorts* of objects. Let S be the set of these sorts. For s in S, the set of objects of sort s will be denoted A_s and called the *carrier* of sort s.

We define an *S-signature* Σ or *operator domain* on S as a family of sets $\Sigma_{w,s}$ where w is a string of S* and s is in S. An element of $\Sigma_{w,s}$ is called the *operation symbol* of *rank* (w,s), of *arity* w and of *sort* s.

A *Σ-algebra* A contains:

- the family $(A_s)_{s \in S}$
- $\forall\ s_1,\ldots,s_n \in S,\ \forall\ s \in S,\ \forall\ \sigma \in \Sigma_{s_1 \ldots s_n,s}$

  a function $\sigma_A: A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ (σ_A is the operation of A named σ; if n=0, it is the constant of A named σ).

The definition of Σ-homomorphisms between two algebras A and B as functions from A to B that preserve the operations of Σ enables us to speak of algebra categories and to define an initial algebra in category C as an element of C such that:

$$\forall B \in C, \exists! \ h: A \rightarrow B, \text{ h homorphism.}$$

It is shown that, if two algebras are initial in the same category, they are isomorphic. An abstract data type is then defined as the isomorphic class of an initial algebra in a Σ-algebra category.

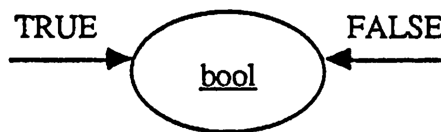
Equations on Σ-algebras are then defined as pairs (L,R), or L=R, where each of the two constituents L and R are expressions composed of variables from $\cup A_s$ and operations from Σ. One shows that, if E is a set of equations, there exists an initial algebra in the category of all Σ-algebras satisfying the equations of E.

It is possible to conclude at this point that an abstract data type is defined uniquely by its specification, the 3-tuple (S,Σ,E).

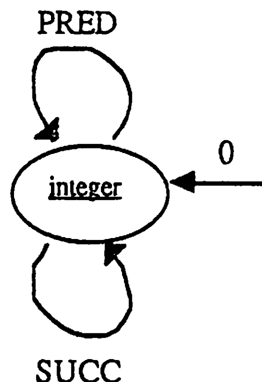
III.1.2 Examples

In these examples, S and Σ are represented graphically. The ovals are objects sorts and the arrows the operations of Σ.

EXAMPLE 1: THE BOOLEAN TYPE. It is the simplest of all classical data types. It contains only two constants:

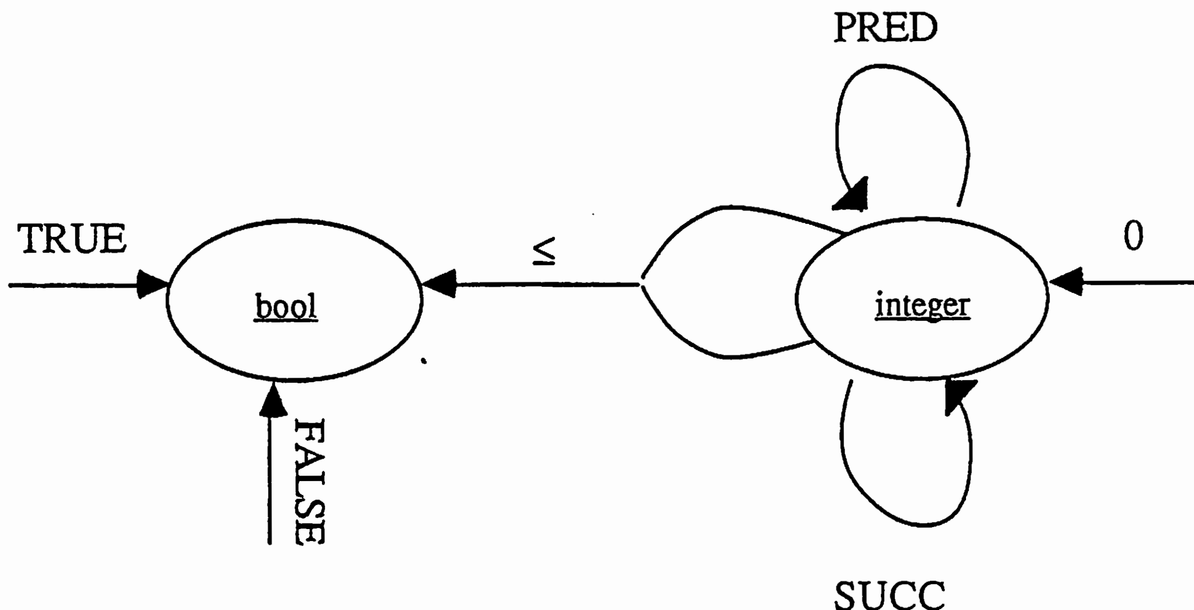


EXAMPLE 2: THE INTEGER TYPE. It can be specified by means of the constant 0 and the two operations, predecessor (PRED) and successor (SUCC), and the reciprocity axioms that bind them:



PRED (SUCC (X)) = X
SUCC (PRED (X)) = X

EXAMPLE 3: THE ORDERED INTEGER TYPE. We add the predicate ≤ to the above defined *integer* type. This specification involves two object sorts. The equations on PRED and SUCC have not been rewritten.



$\leq(X,X) = TRUE$

$\leq(X,SUCC(X)) = TRUE$

$\leq(X,Y) = TRUE$ et $\leq(Y,Z) = TRUE \Rightarrow \leq(X,Z) = TRUE$

$\leq(Y,X) = TRUE$ et $X \neq Y \Rightarrow \leq(X,Y) = FALSE$

New specifications can be built from existing ones. For example, from the *integer* type we defined above, we could build a specification for the *stack of integers* type. The preexisting type will be said to be protected in the new specification if it is not modified by the new constants and operations. A classical example of non-protection would be the equality between TRUE and FALSE in an extension of type *boolean.*

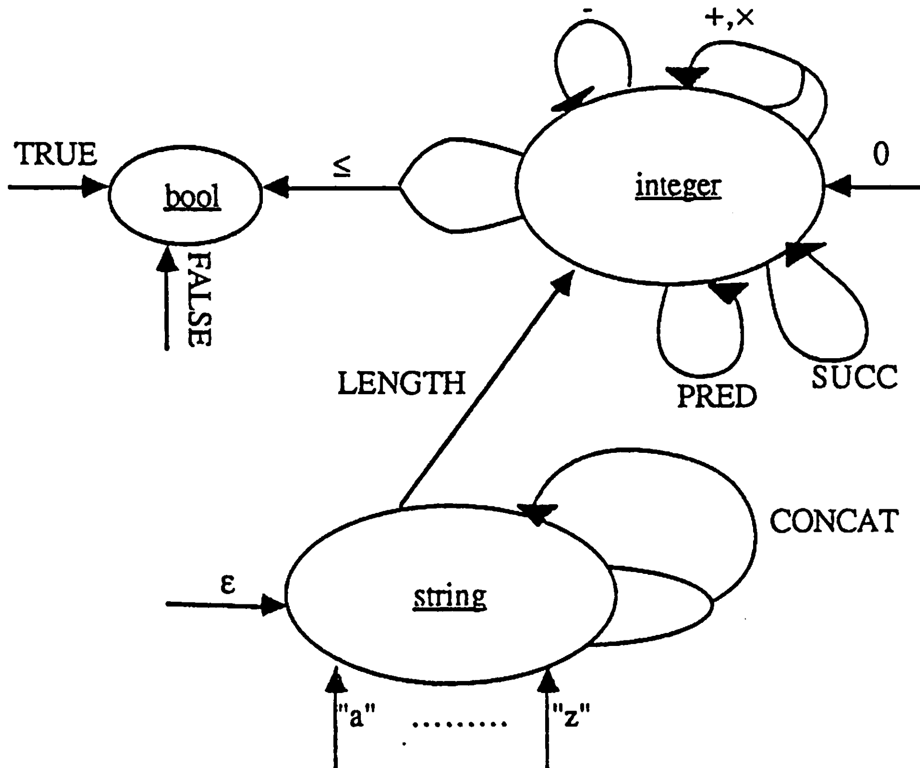
The notation s of a sort will now also design the carrier A_s of sort s.

III.2 ATOMIC TYPES

Atomic f-structures are objects of preexisting types. The specification of these types is imported in the formalism and is used as a ground on which we build the whole f-structure type system. We shall denote it as (S_a,Σ_a,E_a). However, we have to make some slight modifications to this existing specification. This is the purpose of this section.

EXAMPLE. Typical atomic types would be *boolean, integer* and *string*. In that case, the specification (S_a,Σ_a,E_a) would be the following[1]:

[1] Operations take their usual meaning. The equations have not been written.



To each sort s in S_a, we add two constants, that is we put two new elements in $\Sigma_{a\,\varepsilon,s}$. These constants are:

- A top or null value T_s. Its purpose is to be the value of all paths which result should be of type s but have not yet been instantiated inside complex f-structures.
- A unification fail value $\perp_s$. This constant is also meant to play the role of an error value[1].

These two constants are meant to be used only inside complex f-structures. Hence, we do not need to perform existing operations on them. Therefore, we add to E_a the following equations:

$$\forall\ \sigma \in \Sigma_{a_{s_1 \ldots s_n, s}}$$

$$\forall\ 1 \le i \le n \qquad \sigma(X_1,\ldots,X_{i-1},\perp_{s_i},X_{i+1},\ldots X_n) = \perp_s$$

$$\sigma(X_1,\ldots,X_{i-1},T_{s_i},X_{i+1},\ldots X_n) = \perp_s$$

We could have chosen another behavior of the existing operations on the top value. But, in the parser we will describe below, evaluations occur only once for each structure, and so there was no need to define a more complex behavior. Nevertheless, enabling evaluations on the top value would be very dependent on the particular specification used for atomic types and these evaluations should be described according to this specification.

We must finally define unification of atomic f-structures, that is, for each s in S_a we add its symbol $\sqcup_s$ to $\Sigma_{a_{ss,s}}$ and we add to E_a the following equations:

---

[1] In fact, the mechanism used to add these two constants to the existing specification is very close to the one used by [GOG 78] to put error values in his formalism.

$$\sqcup_s (X,X) = X$$
$$\sqcup_s(X,\bot) = \bot$$
$$X \neq \bot \Rightarrow \sqcup_s(X,T) = X$$
$$Y \neq X \text{ et } Y \neq T \Rightarrow \sqcup_s(X,Y) = \bot$$
$$\sqcup_s(X,Y) = \sqcup_s(Y,X)$$

The specification we obtain at this point, which is a modification of (S_a,Σ_a,E_a), shall be denoted as (S_0,Σ_0,E_0).

Unification has been defined by means of equations added to the specification. In fact, it could also be defined as for complex f-structures by means of subsumption ordering, as seen in the next paragraph.

III.3 COMPLEX TYPES

III.3.1 Attribute structures

Let F be a set of features, possibly infinite.

From the atomic f-structures type specification (S_0,Σ_0,E_0) that we have just described, we define recursively a specification including n sorts of attribute structures (S_n,Σ_n,E_n) by the following inductive mechanism:

- S_{k+1} is the union of S_k and the singleton made of the new sort s_{k+1}.
- We define the two constants T_{k+1} and $\bot_{k+1}$, respectively top or null value and unification fail value.
- $\Sigma_{k+1} = \Sigma_k \cup \{p, I_p, p \in F_{k+1}\}$, where F_{k+1} is a finite subset of F with

  $p: s_{k+1} \rightarrow s_p$, feature operation

  $I_p: s_{k+1} \times s_p \rightarrow s_{k+1}$, corresponding instantiation.

- $E_{k+1} = E_k \cup E_{I_{k+1}} \cup E_{\bot_{k+1}}$

  where

  - the equations in $E_{I_{k+1}}$ describe the relationship between a feature operation and its corresponding instantiation. They are:

    $$\forall p \in F_{k+1}, \forall a \in s_{k+1}, \forall v \in s_p, \forall p' \in F_{k+1} \setminus \{p\}$$
    $$p(I_p(a,v))=v$$
    $$p'(I_p(a,v))=p'(a).$$

  - the equations in $E_{\bot_{k+1}}$ describe the behavior of feature and instantiation operations on the constants T_{k+1} and $\bot_{k+1}$. They are:

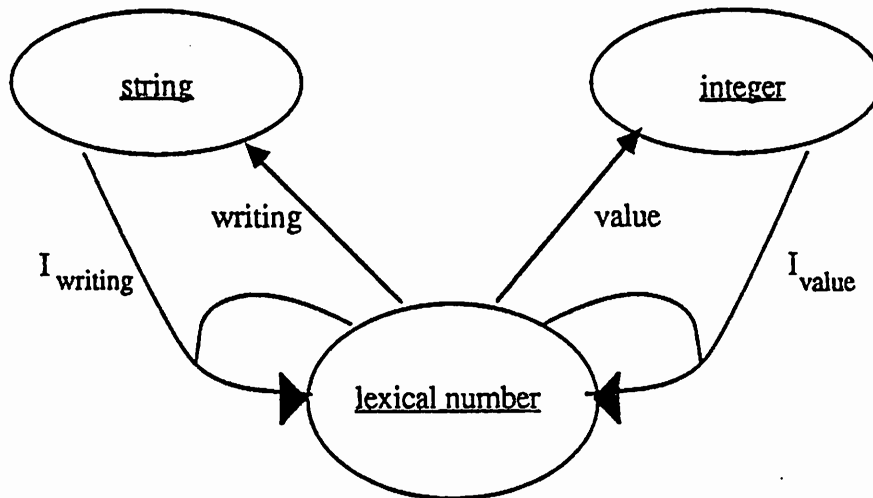
    $$\forall p \in F_{k+1}, \forall a \in s_{k+1}, \forall v \in s_p,$$
    $$p(\bot_{k+1})=\bot_{s_p}$$
    $$p(T_{k+1})=T_{s_p}$$
    $$I_p(\bot_{k+1},v)=\bot_{s_p}$$

The structures we get at this time are called attribute structures since reentrancy is not taken into account. The major point is that the attribute structure types are defined by means of their features, which are considered operations.

EXAMPLE. In a dictionary, we can represent the lexical information associated with numbers by such f-structures:

$$\begin{bmatrix} \text{writing} & \text{"thousand"} \\ \text{value} & 1000 \end{bmatrix}$$

To define this attribute structure type, we use the atomic sorts *integer* and *string*. The new sort, *lexical number*, is then specified by its two features, *value* and *writing*. The complete specification is[1]:



The above mentioned structure is then:

$$I_{writing}(I_{value}(\top,1000),\text{"thousand"}).$$

III.3.2 Reentrancy

Reentrancy can be described as an equivalence relation on paths (see [ZAJ 88]). We will treat it by means of index functions as in [AIT 86]. The two points of view are exactly the same, and if f is an index function, the equivalence relation on paths is exactly the R_f relation that will be defined below.

We define a path as a string of features, that is an element of F*. The null path is denoted as ε. A valid path in an attribute structure type s is defined recursively as follows:

- ε is valid in s
- $p_1...p_n$ is valid in s ⇔
  i) p_n is a feature of type s.
  ii) $p_1...p_{n-1}$ is a valid path in the result type of operation p_n.

Each valid path in a type s can be considered an operation on this type if we associate string concatenation on path names and composition on path operations. We will assume that:

$$\forall a \in s_k \qquad \varepsilon(a) = a.$$

For each attribute structure type s_k, let Δ_k be the set of all valid paths in this

[1] In the following picture, and for the sake of simplicity, we did not represent the operations on atomic sorts.

type. Δ_k corresponds roughly to the term domain defined in [AIT 86] and can be infinite. For each a in s_k, we define a function f_a from Δ_k to N, the set of natural integers. This function is a way to index the paths of the structure, and is called the *index function* of a. Two paths of Δ_k are said to be identical if and only if f_a takes the same value on both. We will now consider pairs (a,f_a). Such pairs are called *indexed structures*. The index function f_a induces an equivalence relation R_f on Δ_k, which is defined by:

$$p \, R_f \, p' \Leftrightarrow f(p) = f(p').$$

Two indexed structures (a,f) and (a',f') can be equivalent providing the indices are renamed. This is formally described by the equivalence relation $\equiv_N$ defined by:

$$(a,f) \equiv_N (a',f') \Leftrightarrow a = a' \text{ et } R_f = R_{f'}$$

The f-structure sort corresponding to s_k is then:

$$s_k^* = s_k^{\sim}/\equiv_N$$

where $s_k^{\sim} = \{(a,f),\ a \in s_k,\ f:\Delta_k \rightarrow N \ / \ \forall p,p' \in \Delta_k,$
$f(p)=f(p') \Rightarrow$
$p(a)=p'(a)$ et
$\forall w/pw \in \Delta_k \ f(pw)=f(p'w)\}.$

The conditions on the elements of $s_k^{\sim}$ ensure that inside a complex f-structure, the underlying attribute structure and the index function are consistent. This is to keep only well-formed terms in the meaning of [AIT 86].

On s_k^*, for each p in F_k, we define the new feature operation p^* by:

$p^*: s_k^* \rightarrow s_{k'}^*$
$p^*((a,f)) = (p(a),f')$
where $\forall \ q \in \Delta_{k'},\ f'(q) = f(pq).$

We also redefine the corresponding instantiation operation I_{p^*} by:

$I_{p^*}((a,f),(v,g)) = (a',f')$
where $a' = I_p(a,v)$
and
$\forall \ q,\ f'(pq) = g(q)$
$\forall \ w \in \Delta_k,\ f'(w) = f(w)$ if p is not a prefix of w.

This construction of complex f-structure sorts does not modify atomic sorts because, in their case, the only valid path is ε and the index function is always degenerate (that is $s^* = s$).

Such complex f-structures can be cyclic. We shall only consider those that are acyclic, but without formally defining the conditions that would eliminate those that are cyclic.

On s_k^*, we now define the subsumption ordering, denoted », by:
if s_k^* is atomic:
$(a_1,f_1) \gg (a_2,f_2) \Leftrightarrow$
$a_1 = \top$ or $a_2 = \bot$ or $a_1 = a_2$

if s_k^* is complex:

$$(a_1,f_1) \gg (a_2,f_2) \Leftrightarrow$$
$$a_1 = \top \text{ or } a_2 = \bot \text{ or}$$
$$R_{f_2} \supseteq R_{f_1} \text{ and}$$
$$\forall\, p \in \Delta_k,$$
$$p(a_1,f_1) \text{ atomic} \Rightarrow$$
$$p(a_1,f_1) \gg p(a_2,f_2)\,.$$

This definition is almost the same as that of [AIT 86], the only difference being the handling of atomic types. For this order, s_k^* is a lattice. The unification of two structures is then defined as their greatest lower bound (glb) in this lattice. For atomic types, this definition is consistent with the equations given at the end of paragraph III.2.

In this type system, the type lattice defined in [AIT 86] is a flat one. The type notion defined here is a bit different than the one introduced in [AIT 86] since reentrancy patterns are not part of the type definition. For example, the two structures:

are of the same type in our system but not in Aït-Kaci's. The proper handling of atomic types and the possibility of using already existing operations on them is one of the major interesting points of the type system described above. It is then possible to define operations other than unification on complex f-structures, as in [ZAJ 88].

III.3.3 A Unification Algorithm

The algorithm that we present here is non-destructive: it does not alter its arguments but creates a new f-structure while unifying them. In that way, it is similar to Wroblewski's algorithm (see [WRO 87]). This algorithm has been implemented and is used in the PAULINE system that will be described in chapter III.

F-structures are represented by record structures of the following type:

```
node = record
    type
    index
    value
end
```

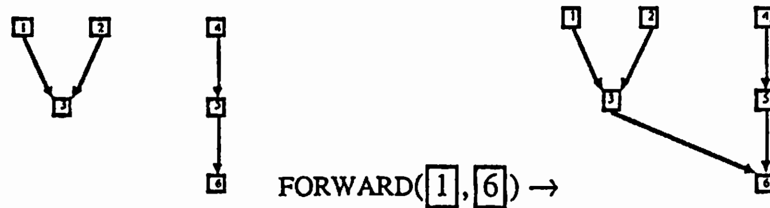
The *index* slot is the value of the index function on the void path. The *type* slot is the type of the f-structure (that is its sort in the specification). If the f-structure is atomic, the *value* slot contains its value, NIL representing the null value of the type. If the f-structure is complex, the *value* slot is a set of feature/value pairs, each value being itself a *node* as defined above.

Forwardings between nodes must not affect the original structures which are to be unified. Therefore, they are made inside a *unification environment*. This one can be seen as a set of inverted trees so that having the same dereferenciation is an equivalence relation on the set of indices. We assume the existence of the two functions:

FORWARD (i_1,i_2)
DEREFERENCE (i)

To keep the unification environment coherent, the two operations FORWARD (i_1,i_2) and FORWARD (DEREFERENCE(i_1), i_2) must be equivalent. The two functions FORWARD and DEREFERENCE are very close to the UNION and FIND functions described in [NAS 86].

UNIFICATION ENVIRONMENT EXAMPLE.



For an index i, there is at most one node structure which index slot is equal to i. The function NODE(i) returns this structure. The function CREATE-NODE returns a node where the *index* slot has been initialized to an integer which has not yet been used as an index. The constant ⊥ will be used as a generic unification failure value. We assume also the existence of the global variable NEW-INDICES, which is a set containing all indices of nodes created during unification. This variable is necessary because indices created during unification can be irreversibly altered which is not the case for indices found in the initial arguments of the UNIFY function. We assume also that the sets of indices of structures to unify are disjoint.

The UNIFY function is the top level one. It initializes the unification environment, calls UNIFY-INDICES which applies recursively the algorithm and finally performs a garbage collection, that is in the result of the unification replaces all forwarded indices by their dereferenciation. This is done by means of the PURGE function which will be described below.

```
UNIFY
        INPUT:    2 nodes n1 and n2
        OUTPUT:   a node or ⊥

        Initialize the unification environment

        result= UNIFY-INDICES (n1.index,n2.index)

        if result = ⊥
            return ⊥
        otherwise return PURGE(result)
```

The UNIFY-INDICES function first checks the equality of the types of the structures to unify, then calls TARGET which computes the index of the result and finally makes the values unification by calling UNIFY-VALUES.

```
UNIFY-INDICES

        INPUT:    2 indices i1 and i2
        OUTPUT:   an index or ⊥

        i'1 = DEREFERENCE(i1)
        i'2 = DEREFERENCE(i2)
```

```
if i'1 = i'2
    return i1

else

    if NODE(i'1).type ≠ NODE(i'2).type
        return ⊥

    else
        v1 = NODE(i'1).value
        v2 = NODE(i'2).value
        result = TARGET(i'1,i'2)
        v = UNIFIER-VALUES(v1,v2,NODE(i'1).type)
        if v = ⊥
            return ⊥
        else
            result.type = NODE(i'1).type
            result.value = v
            return result.index
```

The UNIFY-VALUES function checks the equality of the structures if they are atomic and performs the unification of all features if they are complex. The two functions UNIFY-INDICES and UNIFY-VALUES, by calling themselves each-other, perform the recursive unification.

```
UNIFY-VALUES

    INPUT:    2 values v1 and v2
              a type t
    OUTPUT:   a value or ⊥

    if t is atomic

        if v1 = NIL return v2
        if v2 = NIL or v1 = v2 return v1
        else return ⊥

    else

        result = ∅
        for each feature p of t
            n = UNIFY-INDICES(p(v1),p(v2))
            if n=⊥ return ⊥
            else
                result = result ∪ {(p,n.index)}
        return result.
```

The TARGET function computes the index of the result of a unification. If its arguments are not in NEW-INDICES, it creates a new index. As a side effect, it performs all necessary forwardings. Previously, its arguments have always been dereferenced.

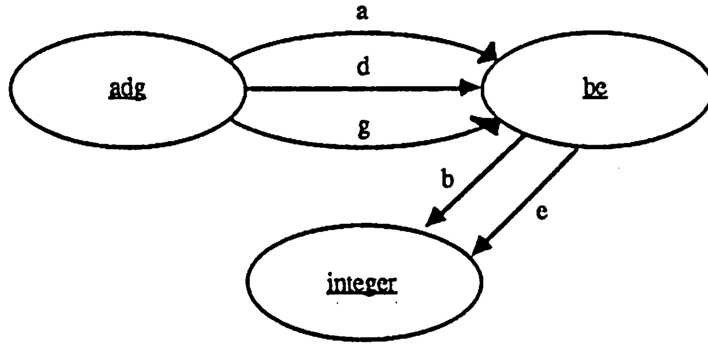
```
TARGET

    INPUT:      2 indices i1 and i2
    OUTPUT:     a node

    if {i1,i2} ∩ NEW-INDICES = ∅
        result = CREATE-NODE
        NEW-INDICES = NEW-INDICES ∪ {result.index}
        FORWARD(i1,result.index)
        FORWARD(i2,result.index)
        return result

    else
        if i1 ∉ NEW-INDICES
            return TARGET(i2,i1)
        else
            FORWARD(i2,i1)
            return NODE(i1)
```

PURGE functions as a garbage collector: it dereferences in the result structures all indices that have been created during unification.

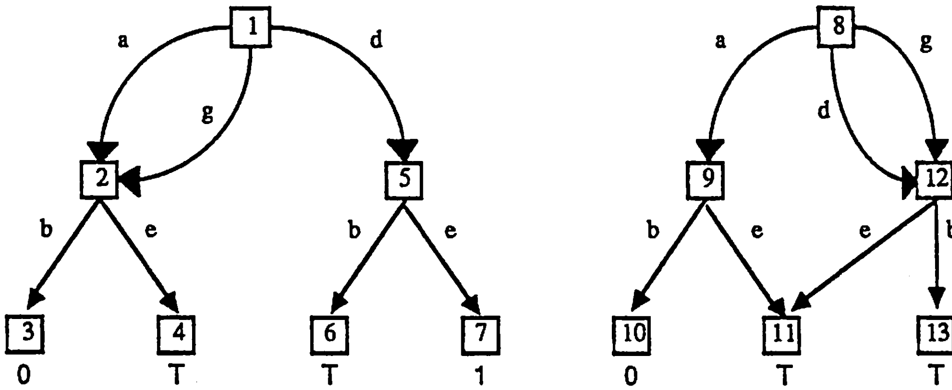
```
PURGE

    INPUT:   an index i
    OUTPUT:  an index

    i = DEREFERENCE(i)

    if NODE(i).type is complex
        v = ∅
        for each (p,i') in NODE(i).value
            v = v∪{(p,NODE(PURGE(i')))}
        NODE(i).value =v

    return i
```

EXAMPLE.

In the following example, f-structures are represented as graphs. The value of the index function is put on each node inside a box. We use the *integer* atomic type and two complex types, *adg* and *be*, according to the following specification where only feature operations have been mentioned:



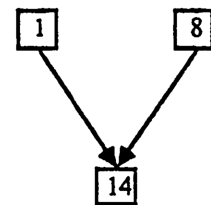
We want to unify the two following structures:



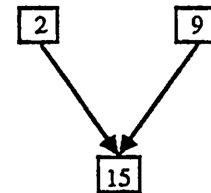
The unification performs itself as follows:

FUNCTION CALLS	GRAPH MODIFICATIONS	ENVIRONMENT MODIFICATIONS

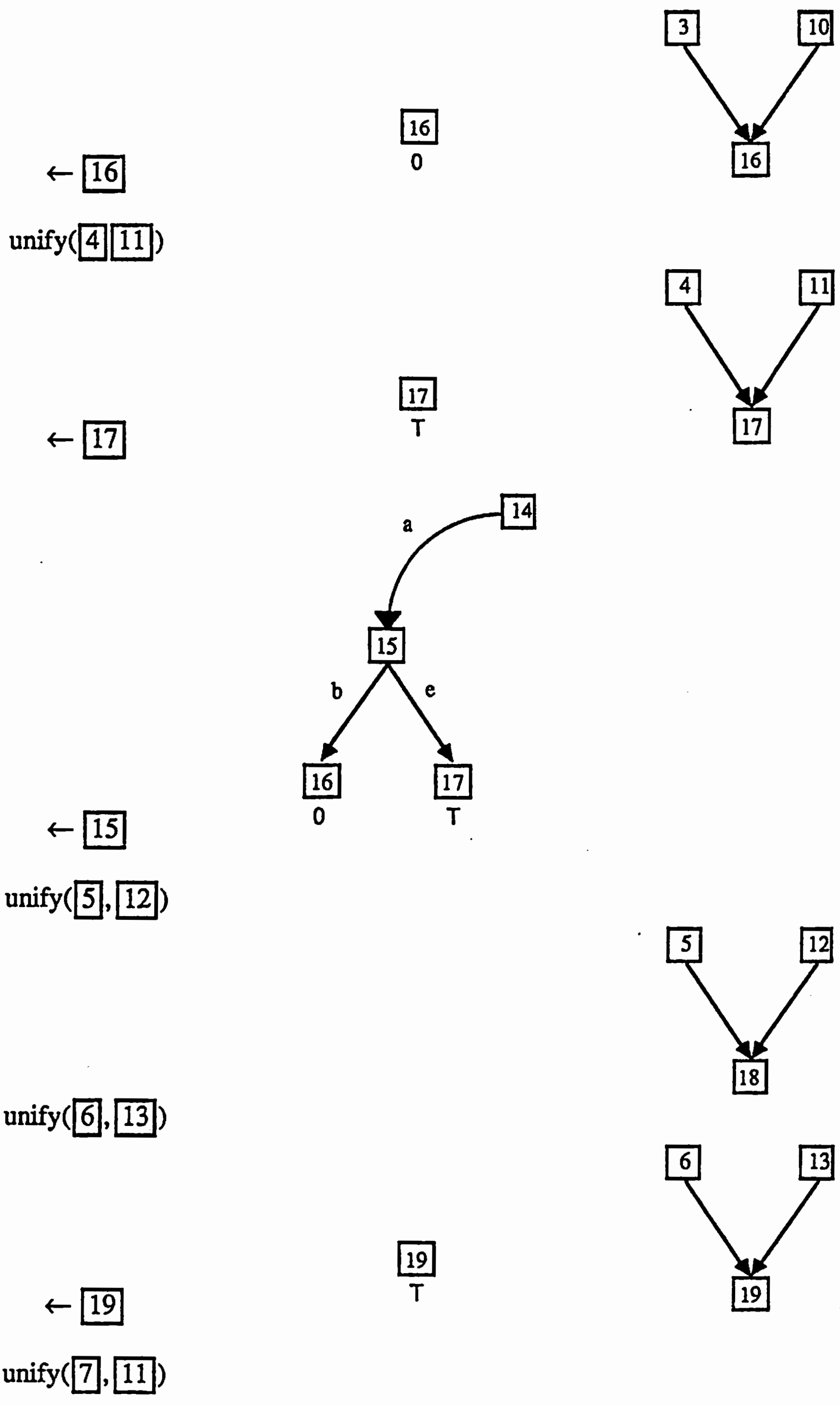
unify(1, 8)



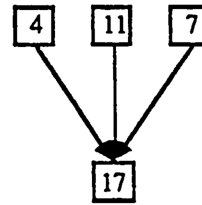
unify(2, 9)

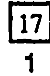


unify(3, 10)

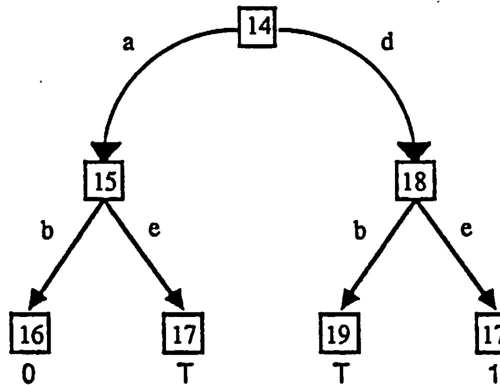

FUNCTION CALLS
GRAPH MODIFICATIONS
ENVIRONMENT MODIFICATIONS
3
10
16
0
16
← 16
unify(4 11)
4
11
17
T
17
← 17
14
a
15
b
e
16
0
17
T
← 15
unify(5, 12)
5
12
18
unify(6, 13)
6
13
19
T
19
← 19
unify(7, 11)

FUNCTION CALLS	GRAPH MODIFICATIONS	ENVIRONMENT MODIFICATIONS



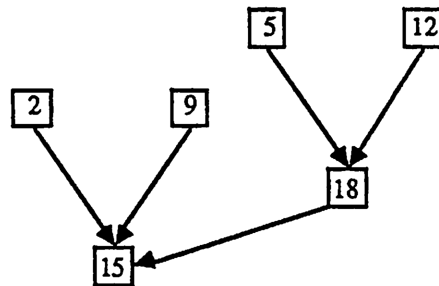


← [17]

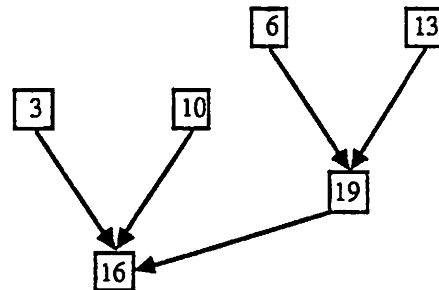


← [18]

unify([2], [12])



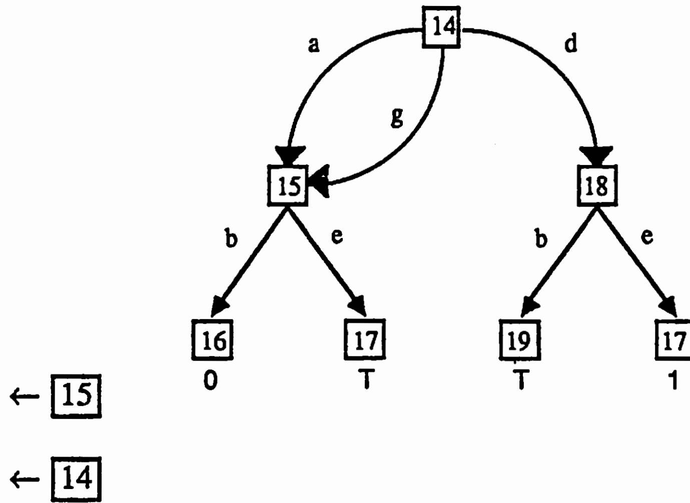
unify([16], [19])



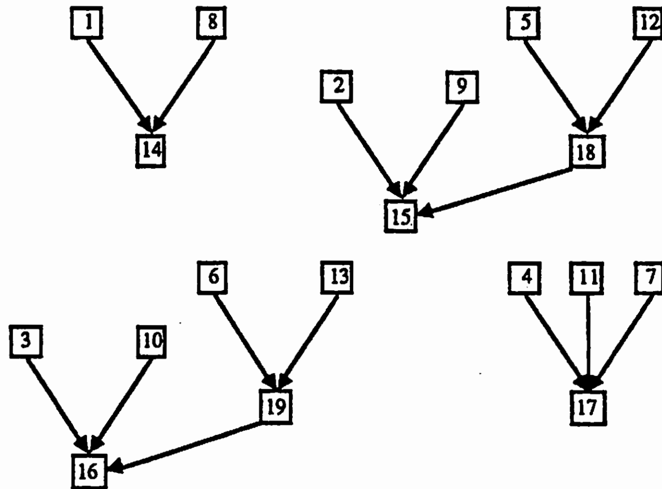
← [16]

unify([17], [17])
← [17]

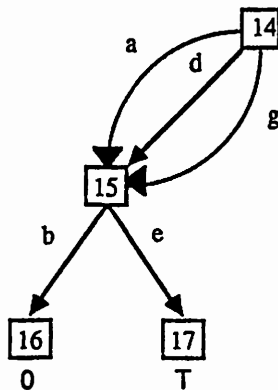
FUNCTION CALLS	GRAPH MODIFICATIONS	ENVIRONMENT MODIFICATIONS



At the end of the algorithm, the unification environment is as follows:



The PURGE function gets rid of the forwarded nodes. Hence, the final graph is:



Chapter II
THE PARSER

The purpose of this chapter is to describe the parsing algorithm we have implemented. We modified Earley's algorithm[1] so that it can build representation structures during the parsing stage. We made it independent of the particular structures chosen to represent information. In the PAULINE system, we used typed f-structures as described in chapter I. Therefore, most of the examples found in this work use such structures. However, the parser can work with other kinds of structures such as trees, forests or graphs, assuming the existence of some interface evaluators on these structures. In fact, besides its use in the PAULINE system, this parser has also been combined with the structures and evaluators described in [ZAJ 88].

In the first section of this chapter, we present the usual version of Earley's algorithm. In the second section, we expose the specification of the interface evaluators used by the parser to manipulate structures. In the third section, we describe the underlying grammar formalism. The last section is devoted to a description and proof of the algorithm.

I. EARLEY'S ALGORITHM

I.1 NOTATIONS

The notations defined in this paragraph will be used throughout the chapter.

We consider context-free grammars, that is 4-tuples $G=(\Sigma, N, P, S)$ where

• Σ is an *alphabet*, that is, a finite set of objects called *words*[2].

The language described by G is a subset of Σ^* that will be defined later.

We shall call *sentence* each element of Σ^*.

• N is a finite set whose intersection with Σ is empty. Its elements are the *non-terminal symbols* of the grammar.

• P is the set of grammar *productions* or *rules*, that is objects

$$A \rightarrow X_1 \dots X_k$$

where $A \in N$ and $X_1,\dots,X_k \in \Sigma \cup N$.

• S is an element of N called *initial symbol* of the grammar.

Elements of $\Sigma \cup N$ are called *grammar symbols*.

We define *derivation* as a relation $\rightarrow$ on $(\Sigma \cup N)^*$ by:

$$u \rightarrow v \iff \exists\, x,w,z \in (\Sigma \cup N)^* \;\; \exists\, A \in N \,/\; u = w\,A\,z,\; v = w\,x\,z,\; A \rightarrow x \in P.$$

[1] See [EAR 68].

[2] We will not make any assumption of the particular form of words, which will be considered more or less as generic objects. The only formal definition that could be given for words in our formalism is "elements of Σ".

The reflexive and transitive closure of this relation is noted $\rightarrow^*$.

The language described by such a grammar is finally defined as the subset of Σ^* such that:

$$x \in L_G \Leftrightarrow S \rightarrow^* x$$

One can also define a grammar with preterminal symbols. It is a 5-tuple (Σ, P, N, T, S, D) where T is a set disjoint with both Σ and N, D is a subset of $\Sigma \times T$, and the other constituents have the same meaning as above except that, in the definition of productions and derivation, Σ should be replaced by T. The language described by such a grammar is defined by:

$$a_1 \ldots a_n \in L_G \Leftrightarrow \exists\, t_1,\ldots,t_n \in T \,/\, S \rightarrow^* t_1 \ldots t_n \text{ and } \forall i, (a_i,t_i) \in D.$$

We shall not use preterminal symbols in the description of the usual Earley's algorithm that follows. However, we use them in the algorithm we implemented, and the relation defined between words and preterminals by the dictionary D is comparable to the lexical evaluator that will be defined later and is one of the interface evaluators used by our parser.

I.2 DESCRIPTION OF THE ALGORITHM

In this paragraph, we present the usual version of Earley's algorithm as it can be found in [AHO 72]. This algorithm takes a sentence $a_1 \ldots a_n$ as argument and builds a *parse list* $I_0,\ldots,I_n$ where each I_i is a set of *items*. An item is a pair $(A \rightarrow \alpha \bullet \beta, i)$ where

$A \rightarrow \alpha\beta$ is a production of P
i is an integer between 0 and n-1.

At the end of the execution of the algorithm, if such an item is in I_j, we have the following derivations:

$$S \rightarrow^* \gamma A \delta$$
$$\alpha \rightarrow^* a_{i+1} \ldots a_j$$
$$\gamma \rightarrow^* a_1 \ldots a_i$$

γ and δ being strings of grammar symbols.

In an item $(A \rightarrow \alpha \bullet \beta, i)$, α is the recognized part of the rule, β the unrecognized one. The parsed sentence is in the language described by the grammar if and only if there is at least one item $(S \rightarrow \alpha \bullet , 0)$ in I_n after the execution of the algorithm.

The inner loop of the algorithm processes items one after another, which leads to the addition of new items in the parse list. The top-level function, PARSE, initializes the parse list and, for each item to process, calls the proper procedure according to the unrecognized part of the rule.

```
PARSE
    INPUT:    a sentence a1...an
    OUTPUT:   the parse list

    For i = 0 to n
        Ii = Ø
    For each production S → α
        I0 = I0 ∪ {(S → • α , 0)}

    For i = 0 to n
        For each item X = (A → α • β,j) in Ii
            if β = ε                    COMPLETER(X,i)
            if β = B γ with B ∈ N       PREDICTOR (X,i)
            if β = a γ with a ∈ Σ       SCANNER (X,i)

    return I0,...,In
```

Items are processed according to their unrecognized part by one of the three procedures PREDICTOR, SCANNER and COMPLETER. As side effects, these procedures can add new items to the parse list.

The PREDICTOR processes items whose first unrecognized symbol is a non terminal B. It creates new items with an empty recognized part, one for each rule $B \rightarrow \gamma$ of P.

```
PREDICTOR
    INPUT:    an item X = (A → α • B β, j) where B ∈ N
              an integer i such that X ∈ Ii

    For each production B → γ
        Ii = Ii ∪ {( B → • γ , i) }
```

The SCANNER processes items $(A \rightarrow \alpha \bullet a\, \beta, j)$ in I_i where a is in Σ. It checks whether $a = a_{i+1}$. If yes, a is recognized and the procedure adds the item $(A \rightarrow \alpha a \bullet \beta, j)$ to I_{i+1}.

```
SCANNER
    INPUT:    an item X = (A → α • a β, j) where a ∈ Σ
              an integer i such that X ∈ Ii

    if a = ai+1
        Ii+1 = Ii+1 ∪ { (A → α a • β, j) }
```

The COMPLETER processes items $(A \rightarrow \alpha \bullet , j)$ in I_i. Such items mean that rule $A \rightarrow \alpha$ has been recognized between a_{j+1} and a_i. Hence, for each item $(B \rightarrow \beta \bullet A\, \gamma , k)$ in I_j, the procedure adds item $(B \rightarrow \beta\, A \bullet \gamma , k)$ to I_i.

COMPLETER

INPUT: an item $X = (A \to \alpha \bullet, j)$
an integer i such that $X \in I_i$

For each item $(B \to \beta \bullet A \gamma, k)$ in I_j
$I_i = I_i \cup \{ (B \to \beta A \bullet \gamma, k) \}$

I.3 COMMENTS

Earley's algorithm is interesting because its complexity is $O(G^2n^3)$, where G is the grammar size, that is the number of productions in P, and n is the length of the sentence to parse. However, the only structure we get for the parsed sentence is the derivation tree that can be built from the parse list. Derivation trees are highly dependent on the particular grammar chosen to describe the language. One will often prefer to build other representation structures.

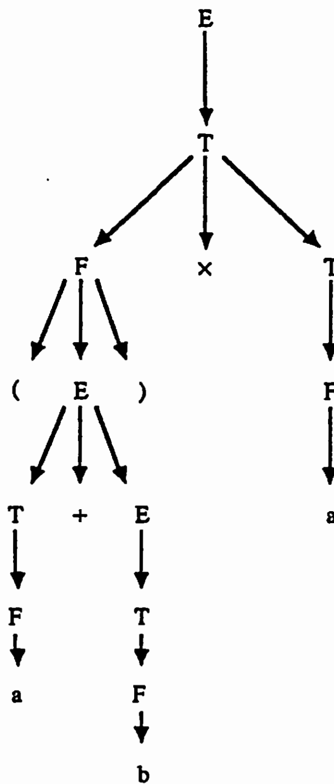
EXAMPLE. Let us consider the grammar:

$$G = (\{a,b,\times,+,(,)\},P,\{E,T,F\},E)$$

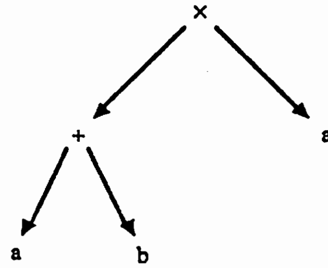
where P is the set of following productions:

$$E \to T \qquad T \to F \qquad F \to a$$
$$E \to T+E \qquad T \to F \times T \qquad F \to b$$
$$F \to (E)$$

This grammar describes multiplicative and additive expressions on variables *a* and *b*. The derivation tree built from the parse list after the execution of Earley's algorithm on sentence $(a + b) \times a$ is the following:



To this rather complex tree, one will prefer the representation tree:



which has to be computed from the derivation tree by some transduction.

The grammar formalism and the parsing algorithm that we will now present enable us to build a representation structure for the sentence during the execution of Earley's algorithm.

II. THE INTERFACE EVALUATORS

The algorithm we implemented is independent of the kind of representation structures we want to associate with sentences. It only assumes the existence of some interface evaluators that enable it to manipulate structures. The basic idea is the following one: in a production $A \rightarrow X_1 \ldots X_k$, we want to assign a representation structure to each grammar symbol. When recognizing the right-hand side of the rule, we will progressively assign structures to X_i, provided that these structures verify some constraints specific to X_i in the rule. Once the whole right-hand side of the rule has been recognized, we will assign a structure to A from the ones associated to X_i and an expression specific to the grammar rule. Therefore, the parser must be able to perform three kinds of operations on structures:

- associate representation structures with words. This can be compared to morphological analysis, that is, the computation of the lexical information needed by the parser.
- evaluate constraints, so that the assignment of structures to rhs-symbols of a rule can be controlled by the grammar writer.
- build a new structure from a list of existing structures and some expression. This enables us to assign a structure to the left-hand side of a rule once the right-hand side has been fully recognized.

Each of these three tasks will be performed by an interface evaluator.

$\mathcal{S}$ will denote the set of all representation structures.

II.1 THE LEXICAL EVALUATOR

The grammar formalism underlying our algorithm uses preterminal symbols. When such a symbol is found in a grammar rule, the algorithm has to check whether the corresponding word in the sentence matches this symbol or not. In addition, it must assign a structure to the symbol. These two tasks are preformed by the lexical evaluator. Formally, it is a function

$$E_l : \Sigma \times T \rightarrow \{A \,/\, A \text{ finite subset of } \mathcal{S}\}\ .$$

If $a \in \Sigma$, $t \in T$. $E_l(a,t)$ is the set of all representation structures for word a that match preterminal t. The result is a set allowing us to handle lexical ambiguities.

EXAMPLE. We suppose that preterminals are grammatical categories. Applications of

the lexical evaluator to the french word *programmes* and various preterminals would be for example:

$$E_l(\text{"programmes"},\text{noun}) = \left\{ \begin{bmatrix} \text{cat} & \text{noun} \\ \text{gender} & \text{masculine} \\ \text{number} & \text{plural} \end{bmatrix} \right\}$$

$$E_l(\text{"programmes"},\text{verb}) = \left\{ \begin{bmatrix} \text{cat} & \text{verb} \\ \text{tense} & \text{present indicative} \\ \text{person} & 2 \end{bmatrix}, \begin{bmatrix} \text{cat} & \text{verb} \\ \text{tense} & \text{present subjunctive} \\ \text{person} & 2 \end{bmatrix} \right\}$$

$$E_l(\text{"programmes"},\text{adjective}) = \varnothing.$$

The parser is independent of the exact operations performed by the lexical evaluator. The latter can simply find structures associated with the word in some word/structures dictionary, or perform a real morphological analysis, that is, compute the structures from the word and a low-level dictionary. It can also look in a structures table built by a previous morphological analysis.

II.2 THE CONTEXTUAL EVALUATOR

This evaluator builds new structures from a *context*, that is, a list of already existing structures, and a *contextual expression*. The parser is independent of the particular syntax chosen for those contextual expressions. They will typically consult the structures of the context and perform some operations on them. Formally, the contextual evaluator is a function

$$E_c : E \times C \rightarrow \{A \,/\, A \text{ finite subset of } S\}$$

where E is the set of contextual expressions and C the set of contexts. The result of the contextual evaluator is a set of structures, once again so that we can handle ambiguities and disjunctions.

EXAMPLE. We consider the complex type *lexical number* specified in chapter I, paragraph III.3.1, and the three following structures:

$$A \quad \begin{bmatrix} \text{writing} & \text{"thousand"} \\ \text{value} & 1000 \end{bmatrix}$$

$$B \quad \begin{bmatrix} \text{writing} & \text{"two"} \\ \text{value} & 2 \end{bmatrix}$$

$$C \quad \begin{bmatrix} \text{writing} & \text{"hundred"} \\ \text{value} & 100 \end{bmatrix}$$

If we want to be able to add the values of two numbers, we could use such a contextual expression as

$$e = (+\ (0\ \text{value})\ (1\ \text{value}))$$

and we would have

$$E_c(e\,,\,(A,B)) = \{1002\}$$
$$E_c(e\,,\,(A,C)) = \{1100\}.$$

There is no disjunction here and the result of these contextual evaluations is thus a singleton.

II.3 THE PREDICATIVE EVALUATOR

The last evaluator handles constraints on structures. This allows the selective assignment of structures to right-hand side symbols of grammar rules. Let P be a set of predicates. The predicative evaluator is a function

$$E_p : P \times S \rightarrow \{\text{TRUE}\,,\,\text{FALSE}\}$$

As for the contextual evaluator, the parser is independent of the syntax chosen for predicates. In fact, these can be particular contextual expressions (they must refer to

only one structure). In that case, the predicative evaluator is defined from the contextual evaluator by:

$$E_p(p,s) = \text{TRUE} \Leftrightarrow E_c(p,(s)) \neq \emptyset$$

III THE GRAMMAR FORMALISM

III.1 PRESENTATION

We use an augmented context-free grammar with preterminal symbols, defined as a 5-tuple $G = (\Sigma,P,N,T,S)$, where Σ,N,T and S have the same meaning as in section I. P is the set of grammar rules whose form is the following:

$$(A,e) \rightarrow (X_1,p_1) \dots (X_k,p_k)$$

where

$A \in N$

$X_1, \dots , X_k \in N \cup T$

e is a contextual expression

$p_1, \dots , p_k$ are predicates.

We define the context-free grammar underlying G as $G' = (\Sigma,P',N,T,S)$ where

for $A \in N,\ X_1, \dots , X_k \in N \cup T$,

$$A \rightarrow X_1 \dots X_k \in P'$$
$$\Leftrightarrow \exists\ (A,e) \rightarrow (X_1,p_1) \dots (X_k,p_k) \in P.$$

An *instantiation* of length r of the grammar rule $(A,e) \rightarrow (X_1,p_1) \dots (X_k,p_k)$ is a r-tuple $(s_1,\dots,s_r)$ of S^r such that

$$\forall\ 1 \leq i \leq r\ \ E_p(p_i,s_i) = \text{TRUE}$$

We define two distinct derivation relations:

- The context-free one $\rightarrow$ defined from G' as in section I.
- The augmented derivation $\Rightarrow$ defined on strings of $((N \cup T) \times S)^*$ by:

$u \Rightarrow v$

$\Leftrightarrow$ $\exists\ (A,s) \in N \times S$

$\exists\ (X_1,s_1)\dots(X_k,s_k) \in ((N \cup T) \times S)^*$

$\exists\ w,z \in ((N \cup T) \times S)^*$

$\exists\ R = (A,e) \rightarrow (X_1,p_1) \dots (X_k,p_k) \in P$

such that

$u = w\ (A,s)\ z$

$v = w\ (X_1,s_1) \dots (X_k,s_k)\ z$

$(s_1,\dots,s_k)$ is an instantiation of R

$s \in E_c\ (e, (s_1,\dots,s_k))$

The reflexive and transitive closures of these two relations will be respectively denoted as $\rightarrow^*$ and $\Rightarrow^*$.

We finally define the language L_G described by such a grammar:

Let $a_1 \dots a_n \in \Sigma^*$

$a_1 \dots a_n \in L_G$

$\Leftrightarrow \exists\ (t_1,\dots,t_n) \in T^n$

$$\exists\ (s,s_1,\dots,s_n) \in S^{n+1}$$
such that
$$(S,s) \Rightarrow^* (t_1,s_1) \dots (t_n,s_n)$$
$$\forall\ 1 \leq i \leq n\ s_i \in E_l(a_i,t_i)$$

In this last derivation, the structure s will be called a *parse structure* of sentence $a_1 \dots a_n$. There can be many such structures.

III.2 EXAMPLE

We present a grammar for the formal language $a^n b^n$ in the formalism that has just been described.

We use an f-structure type system including two atomic types: *integer* and the scalar type of extension {a, b}. This type system includes a unique complex type specified by the two features *letter*, whose value is in {a, b}, and *counter* whose value is an integer. F-structures of this type handle sequences of letters. For example, the sequence *aaa* will be represented by the following structure:

$$\begin{bmatrix} \text{letter} & a \\ \text{counter} & 3 \end{bmatrix}$$

Let $\Sigma = \{a,b\}$
$N = \{S,A\}$
$T = \{lex\}$

The lexical evaluator is defined on $\Sigma \times T$ by:
$E_l(a,lex) = \{a\}$
$E_l(b,lex) = \{b\}$

In the syntax used for contextual expressions and predicates, accesses to structures in the context are represented by LISP lists whose CAR is the rank of the structure in the context and whose CDR is the path in the structure. For example, the list (2 letter) refers to the value of feature *letter* in the second structure of the context. Lists with equal signs as second element are path assignments inside structures. *If* and functional expressions use the same syntax as in LISP. *SEQ* evaluates expressions in sequence.

Grammar rules are then the following:

R_1 $(S,e_S) \rightarrow (A,p_a)\ (A,p_b)$
with
e_S : (if (= (1 counter) (2 counter))
(1 counter))
p_a : (= (1 letter) a)
p_b : (= (2 letter) b)

R_2 $(A,e_A) \rightarrow (lex,true)$
with
e_A: (seq ((0 counter) = 1))
((0 letter) = (1))
(0))

R_3 $(A,e'_A) \rightarrow (lex, true)\ (A, true)$
with
e'_A: (if (= (1) (2 letter))
(seq ((0 counter) = (+ 1 (2 counter)))

((0 letter) = (1))
(0)))

The sentence *aabb* is in the language since we have the following derivations:

(D_0) $(S,2) \Rightarrow (A, \begin{bmatrix} \text{letter} & a \\ \text{counter} & 2 \end{bmatrix}) (A, \begin{bmatrix} \text{letter} & b \\ \text{counter} & 2 \end{bmatrix})$

(D_1) $(A, \begin{bmatrix} \text{letter} & a \\ \text{counter} & 2 \end{bmatrix}) \Rightarrow (\text{lex},a) (A, \begin{bmatrix} \text{letter} & a \\ \text{counter} & 1 \end{bmatrix})$

(D_2) $(A, \begin{bmatrix} \text{letter} & a \\ \text{counter} & 1 \end{bmatrix}) \Rightarrow (\text{lex},a)$

and the derivations equivalent to the two last for sequence *bb*.

We can see in this example that, if predicates associated to right-hand side symbols of grammar rules encode constraints specific to these symbols, crossed constraints involving many right-hand side symbols can be encoded inside contextual expressions, provided an adequate syntax has been defined. This is used here to check the equality of numbers of letters in the two sequences (*counter* features). If these are not equal, the contextual evaluation of e_S does not return any value.

IV. THE ALGORITHM

Two characteristics make our algorithm different from the standard Earley's parser that has been described in section I.

- Each item includes a *lookahead symbol*. It is a preterminal that must be found after the whole recognition of the rule to which the item corresponds. It is a classical extension to Earley's algorithm. It can be found as an exercise in [AHO 72] and in the version presented in [TOM 86]. Formally, items become 3-tuple $(A \rightarrow \alpha \bullet \beta, i, t)$ where the two first constituents keep the same meaning as in section I and t is an element of $T \cup \{ \$ \}$. An item $(A \rightarrow \alpha \bullet, j, t)$ in I_j will be completed if and only if

$$E_l(a_{j+1},t) \neq \varnothing$$

The symbol \$ stands for the end of the sentence. If j=n, the above condition is replaced by t=\$.

- The second characteristic is the building of structures during parsing. For this purpose, we need two operations: when a rule is completed, the parser must evaluate the associated contextual expression on the structures that have been assigned to the right-hand side symbols of the rule; in each item, when the dot is moving right, we must assign a structure to the corresponding symbol and thus evaluate the corresponding predicate on all candidate structures which come from the lexical evaluator if the symbol is a preterminal, or from the contextual evaluator if it is a non terminal.

IV.1 DESCRIPTION OF THE ALGORITHM

The items we use are 4-tuples

$$((A,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k),(s_1, \ldots, s_r),\ i,\ t)$$

where $(A,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k)$ is a grammar rule.
i is the position of the item in the sentence.

$t \in T \cup \{ \$ \}$ is the lookahead symbol.

$r \leq k$ and $(s_1, ..., s_r)$ is an instantiation of the rule.

In items, the production is not dotted. However, the length of the grammar rule instantiation has exactly the same meaning as the position of the dot in items described in section I.

To determine the lookahead symbol of items created by rule predictions, we need a function FIRST. Its argument is a string of grammar symbols, and its result the set of all preterminal symbols that can begin rewritings of this string, according to the underlying context-free grammar. This function is as follows:

```
FIRST
    INPUT:   a string A1 ... An of (N ∪ T)*
    OUTPUT:  a set of preterminal symbols

    if n = 0
        return { $ }
    else if A1 ∈ T
        return {A1}
    else
        result = ∅
        For each rule A1 → X1 ... Xk
            result = result ∪ FIRST (X1)
        return result
```

The basic principles of the algorithm are the same as in section I. The SCANNER and the COMPLETER, while recognizing one more symbol inside grammar rules, call an auxiliary function, MOVE-DOT, which evaluates the corresponding predicate on candidate structures and, if possible, builds the new item.

```
PARSE
    INPUT:   a sentence a1...an
    OUTPUT:  the parse structures of this sentence.

    For each rule which lhs is S
        I0 = I0 ∪ {(R,(),0,$)}
    For i = 0 to n
        For each item Y = ((A,e) → (X1,p1)...(Xk,pk) , (s1,...,sr) , j, t)
        in Ii
            if r = k --> COMPLETER (Y,i)
            if Xr+1 ∈ N --> PREDICTOR (Y,i)
            else --> SCANNER (Y,i)
    For each item ((S,e) → (X1,p1)...(Xk,pk) , (s1,...,sk) , 0 , $)
    in In
        compute Ec(e,(s1,...,sk))
    Return the union of these evaluations.
```

In this algorithm, the PREDICTOR has to take into account the lookahead symbol that must be put inside items.

PREDICTOR

INPUT: an item $((A,e) \rightarrow (X_1,p_1)...(X_k,p_k)\ ,\ (s_1,...,s_r)\ ,j\ ,\ t)$
where $X_{r+1} \in N$
a position i

For each rule R' which lhs is X_{r+1}
if $r+1 = k$
$I_i = I_i \cup \{\ (R',(),i,t)\ \}$
else
for each t' in FIRST(X_{r+2})
$I_i = I_i \cup \{\ (R',(),i,t')\ \}$

The COMPLETER first evaluates the contextual expression of the rule on the structures that have been assigned to right-hand side symbols of the rule, then finds the items which could be completed and calls MOVE-DOT which tries to integrate the results of the contextual evaluation inside the items to complete. These operations occur if and only if the next word in the sentence is compatible with the lookahead symbol in the item, that is if the result of the lexical evaluator on both of them is not empty.

COMPLETER

INPUT: an item $((A,e) \rightarrow (X_1,p_1)...(X_k,p_k)\ ,\ (s_1,...,s_k)\ ,\ j\ ,\ t)$
a position i

If $(E_l(a_{i+1},t) \neq \emptyset)$ or $(i = n$ et $t = \$)$
For each s in $E_c(e,(s_1,...,s_k))$
For each item
$Y'=((B,e') \rightarrow (X'_1,p'_1)...(X'_k,p'_k)\ ,\ (s'_1,...,s'_{r'})\ ,\ j'\ ,\ t')$
such that $X'_{r'+1} = A$
$I_i = I_i \cup$ MOVE-DOT(Y',s)

The SCANNER calls the lexical evaluator on the next word in the sentence and the first non recognized symbol. Its result is the set of candidate structures. These are sent to MOVE-DOT to see whether they can be associated to the symbol or not.

SCANNER

INPUT: an item $Y=((A,e) \rightarrow (X_1,p_1)...(X_k,p_k)\ ,\ (s_1,...,s_r)\ ,\ j\ ,\ t)$
where $X_{r+1} \in T$
a position i

For each s in $E_l(a_{i+1},X_{r+1})$
$I_{i+1} = I_{i+1} \cup$ MOVE-DOT(Y,s)

MOVE-DOT takes two arguments, a structure and an item. It evaluates the predicate corresponding to the first non-recognized symbol of the item on the argument structure. If it is false, the function returns the empty set. Otherwise, it builds a new item which is a copy of the argument item except that the structure has been added to the rule instantiation, and returns a singleton made of this new item.

MOVE-DOT

INPUT: an item $((A,e) \rightarrow (X_1,p_1)...(X_k,p_k)\ ,\ (s_1,...,s_r)\ ,\ j\ ,\ t)$
a structure s
OUTPUT: a set of items.

if $E_p(p_{r+1},s)$ = TRUE
 return $\{\ ((A,e) \rightarrow (X_1,p_1)...(X_k,p_k)\ ,\ (s_1,...,s_r,s)\ ,\ j\ ,\ t)\ \}$
else
 return $\varnothing$

IV.2 PROOF OF THE ALGORITHM

This proof is derived from [AHO 72]. We first show that each element in the result of the PARSE function is a parse structure of the sentence, according to the definition given at the end of paragraph III.1. We then show that every parse structure of the sentence is in the result of the PARSE function.

lemma 1. If an item

$$((A,e) \rightarrow (X_1,p_1)\ ...\ (X_k,p_k),(s_1,\ ...,\ s_r),\ i,\ t)$$

is in I_j after parsing sentence $a_1...a_n$, then

i. $\exists\ \gamma,\ \delta /\quad S \rightarrow^* \gamma\ A\ \delta$ and $t \in$ FIRST(δ)

ii. $\gamma \Rightarrow^* (t_1,\sigma_1)\ ...\ (t_i,\sigma_i)$

iii. $(X_1,s_1)\ ...\ (X_r,s_r) \Rightarrow^* (t_{i+1},\sigma_{i+1})\ ...\ (t_j,\sigma_j)$

with

$\forall\ 1 \leq h \leq j\ \ \sigma_h \in E_l(a_h,t_h)$

Proof. The proof is an induction on the number of items in the parse list. Let n_0 be the number of items put in I_0 by the first step of the PARSE function. These items are all of the following form:

$$((S,e) \rightarrow (X_1,p_1)\ ...\ (X_k,p_k),(),\ 0,\ \$)$$

They clearly verify the three conditions of the lemma. Let us now assume that the parse list contains m items, such that $n_0 \leq m$, each of them verifying the conditions of the lemma. Let us consider the (m+1)th item built by the algorithm. Three cases are possible:

- The item is put in I_i by the PREDICTOR. This item is:

$$\text{I} = ((A,e) \rightarrow (X_1,p_1)\ ...\ (X_k,p_k),(),\ i,\ t)$$

The PREDICTOR builds it while processing item

$$\text{I}' = (B \rightarrow Y_1...Y_{r-1}AY_{r+1}...Y_{k'},(s_1,...,s_{r-1}),\ j,\ t')$$

which is already in I_j. Thus, by inductive hypothesis:

$\exists\ \gamma,\ \delta /\quad S \rightarrow^* \gamma\ B\ \delta$ and $t \in$ FIRST(δ)

$\gamma \Rightarrow^* (t_1,\sigma_1)\ ...\ (t_j,\sigma_j)$

$(Y_1,s_1)\ ...\ (Y_{r-1},s_{r-1}) \Rightarrow^* (t_{j+1},\sigma_{j+1})\ ...\ (t_i,\sigma_i)$

The first condition of the lemma is hence true for item I, because, if

$\gamma' = \gamma Y_1...Y_{r-1}$

$\delta = Y_{r+1}...Y_{k'}\delta$

we have $\quad S \rightarrow^* \gamma'\ A\ \delta'$

and $\quad t \in$ FIRST (δ'), because that is the way the PREDICTOR finds the new lookahead symbol.

By combining conditions **ii.** and **iii.** of the lemma on item `I'`, we prove condition **ii.** on item `I`. Condition **iii.** is then clearly verified on item `I` since its recognized part is empty.

• The item is put in I_{j+1} by the SCANNER. This item is

$$I = ((A,e) \rightarrow (X_1,p_1) \dots (X_k,p_k),(s_1, \dots, s_r),\ i,\ t)$$

where X_r is a preterminal. The scanner is processing item

$$I' = ((A,e) \rightarrow (X_1,p_1) \dots (X_k,p_k),(s_1, \dots, s_{r-1}),\ i,\ t)$$

which is already in I_j. Conditions **i.** and **ii.** are true on `I` since they are the same as on `I'`. By inductive hypothesis (condition **iii.**) we have the following derivation:

$$(X_1,s_1) \dots (X_{r-1},s_{r-1}) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \dots (t_j,\sigma_j)$$

Since X_r is a preterminal,

$$(X_1,s_1) \dots (X_r,s_r) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \dots (t_j,\sigma_j)(X_r,s_r)$$

Since item `I` has been put in I_{j+1}, necessarily

$$s_r \in E_l(a_{j+1},X_r)$$

which, combined with the previous derivation, shows condition **iii.** on item `I`. Let us notice that we have besides

$$E_p(p_r,s_r) = \text{true}$$

since the result of MOVE-DOT is not empty. This shows that in all items put in the parse list, the list of structures (second constituent) remains an instantiation of the grammar rule[1].

• The item is put in I_j by the COMPLETER. This item is

$$I = ((A,e) \rightarrow (X_1,p_1) \dots (X_k,p_k),(s_1, \dots, s_r),\ h,\ t)$$

where $X_r \in N$. The COMPLETER builds it while processing item

$$I' = ((X_r,e') \rightarrow (Y_1,p'_1) \dots (Y_{k'},p'_{k'}),(s'_1, \dots, s'_{k'}),\ i,\ t)$$

which is already in I_j. We know also that there is an item

$$I'' = ((A,e) \rightarrow (X_1,p_1) \dots (X_k,p_k),(s_1, \dots, s_{r-1}),\ h,\ t)$$

in I_i. Conditions **i.** et **ii.** are true on `I` since they are the same as on `I"`.
We have also:

$$(D_1) \qquad (X_1,s_1) \dots (X_{r-1},s_{r-1}) \Rightarrow^* (t_{h+1},\sigma_{h+1}) \dots (t_i,\sigma_i)$$

$$(D_2) \qquad (Y_1,s'_1) \dots (Y_{k'},s'_{k'}) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \dots (t_j,\sigma_j)$$

Since `I` has been put in the parse list, necessarily:

$$s_r \in E_c(e',(s'_1, \dots, s'_{k'}))$$

and hence

$$(D_3) \qquad (X_r,s_r) \Rightarrow (Y_1,s'_1) \dots (Y_{k'},s'_{k'})$$

---

[1] We could have put it in the inductive hypothesis but did not, for the sake of simplicity.

Combining derivations D_1, D_2 and D_3, one shows that condition **iii** is true for item I, which finally ends the proof of the lemma. ♦

Theorem 1. If the result of function PARSE is not empty, the sentence is in the language L_G. In that case, each element in the result is a parse structure of the sentence.

Proof. Let s be an element of the result of function PARSE. Then there is an item

$$I = ((S,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k),(s_1, \ldots, s_k),\ 0,\ \$)$$

in I_n such that

$$s \in E_c(e,(s_1, \ldots, s_k)).$$

From condition **i.** of lemma 1, we have:

$$\exists\, \gamma, \delta /\ \ S \rightarrow^* \gamma\, S\, \delta \text{ et } \$ \in \text{FIRST}(\delta)$$

And hence $\delta = \varepsilon$. Moreover, since the position of the item is 0, from condition **ii.** of the lemma, we have also $\gamma = \varepsilon$. Because of condition **iii.** we have the following derivation:

$$(X_1,s_1) \ldots (X_k,s_k) \Rightarrow^* (t_1,\sigma_1) \ldots (t_n,\sigma_n)$$

where

$$\forall\ 1 \leq h \leq n\ \sigma_h \in E_l(a_h,t_h)$$

Then, since $s \in E_c(e,(s_1, \ldots, s_k))$,

$$(S,s) \Rightarrow (X_1,s_1) \ldots (X_k,s_k)$$

and thus

$$(S,s) \Rightarrow^* (t_1,\sigma_1) \ldots (t_n,\sigma_n)$$

$$\text{where } \forall\ 1 \leq h \leq n\ \sigma_h \in E_l(a_h,t_h)$$

which finally proves the theorem. ♦

We now have to show that, if s is a parse structure of the sentence, it is in the output of the PARSE function.

If s is a parse structure of the sentence, there is a derivation

(D) $\quad (S,s) \Rightarrow^* (t_1,\sigma_1) \ldots (t_n,\sigma_n)$

$$\text{where } \forall\ 1 \leq h \leq n\ \sigma_h \in E_l(a_h,t_h)$$

Let $(A,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k)$ be a rule which applies at some point in this derivation. Then

$$(S,s) \Rightarrow^* \quad \gamma\,(A,s_A)\,\delta \quad \Rightarrow \gamma\,(X_1,s_1) \ldots (X_k,s_k)\,\delta$$
$$\Rightarrow^* (t_1,\sigma_1) \ldots (t_n,\sigma_n)$$

and, splitting these derivations

$$\exists\, i /\ \gamma \Rightarrow^* (t_1,\sigma_1) \ldots (t_i,\sigma_i)$$

and $\quad \forall\, r \leq k\ \ \exists j / (X_1,s_1) \ldots (X_r,s_r) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \ldots (t_j,\sigma_j)$

The grammar rule, its instantiation, strings γ and δ, and integer i are uniquely defined by the subderivation of (D)

$$(S,s) \Rightarrow^* \quad \gamma\,(A,s_A)\,\delta$$

Integer j is uniquely defined once we know r. For the demonstration, we shall consider instances

$$\mathcal{I} = ((S,s) \Rightarrow^* \gamma\,(A,s_A)\,\delta\ ,\ r\ )$$

The first constituent is a subderivation of (D).It defines an instantiated grammar rule $(A,s_A) \Rightarrow (X_1,s_1) \ldots (X_k,s_k)$ and an integer i such that $\gamma \Rightarrow^* (t_1,\sigma_1) \ldots (t_i,\sigma_i)$.The second constituent r is an integer r lower than k. It defines an integer j such that

$$(X_1,s_1) \ldots (X_r,s_r) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \ldots (t_j,\sigma_j).$$

Since integer i and j are defined by the instance, we shall denote them as $i(\mathcal{I})$ and $j(\mathcal{I})$ any time confusion might be possible.

---

lemma 2. According to the notations defined above, for each instance

$$((S,s) \Rightarrow^* \gamma\,(A,s_A)\,\delta\,,\,r\,)$$

and for each t in FIRST(δ), the item

$$(\,(A,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k),(s_1,\ \ldots,\ s_r),\ i,\ t)$$

is in I_j at the end of the execution of the algorithm.

Proof. Let $\mathcal{I} = ((S,s) \Rightarrow^* \gamma\,(A,s_A)\,\delta\,,\,r\,)$ be an instance. We define the following quantities:

$\tau_1(\mathcal{I})$ is the length of derivation $(S,s) \Rightarrow^* \gamma\,(A,s_A)\,\delta$.

$\tau_2(\mathcal{I})$ is the length of the shortest derivation

$$\gamma \Rightarrow^* (t_1,\sigma_1) \ldots (t_i,\sigma_i)$$

$\tau_3(\mathcal{I})$ is the length of the shortest derivation

$$(X_1,s_1) \ldots (X_r,s_r) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \ldots (t_j,\sigma_j)$$

$$\tau(\mathcal{I}) = \tau_1(\mathcal{I}) + 2(\,\tau_2(\mathcal{I}) + \tau_3(\mathcal{I}) + j(\mathcal{I}))$$

The proof of the lemma is an induction on $\tau(\mathcal{I})$, called rank of *instance* $\mathcal{I}$.

If $\tau(\mathcal{I}) = 0$ then necessarily $\tau_1(\mathcal{I}) = \tau_2(\mathcal{I}) = \tau_3(\mathcal{I}) = j = 0$, and the items described in the lemma must have the following form:

$$(\,(S,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k),\ (),\ 0,\ \$)$$

These items are put in I_0 by the first step of the PARSE function.

Let $\mathcal{I} = ((S,s) \Rightarrow^* \gamma\,(A,s_A)\,\delta\,,\,r\,)$ be an instance. We assume that the lemma is true for each instance whose rank is less than $\tau(\mathcal{I})$ - 1.

• if r = 0. Then necessarily $\tau_3(\mathcal{I}) = 0$ and j = i. Decomposing the last derivation of $\mathcal{I}$, we get:

$$(S,s) \Rightarrow^* \gamma'\,(B,s_B)\,\delta' \Rightarrow \gamma'\,Y_1 \ldots Y_{q-1} A Y_{q+1} \ldots Y_{k'}\delta'$$

with $\gamma = \gamma'\,Y_1 \ldots Y_{q-1}$ et $\delta = Y_{q+1} \ldots Y_{k'}\delta'$. Let us then consider instance

$$\mathcal{I}' = ((S,s) \Rightarrow^* \gamma'\,(B,s_B)\,\delta'\,,\,q\text{-}1)$$

We have:

$\tau_1(\mathcal{I}') = \tau_1(\mathcal{I}) - 1$

$j(\mathcal{I}') = i(\mathcal{I}) = j(\mathcal{I})$

$\tau_2(\mathcal{I}') + \tau_3(\mathcal{I}')$ is the length of the shortest derivation from $\gamma' Y_1 \ldots Y_{q-1}$ to $(t_1,\sigma_1) \ldots (t_j,\sigma_j)$.

Since $\gamma = \gamma'\,Y_1 \ldots Y_{q-1}$,

$$\tau_2(\mathcal{I}') + \tau_3(\mathcal{I}') = \tau_2(\mathcal{I}) + \tau_3(\mathcal{I}) = \tau_2(\mathcal{I})$$

and finally

$\tau(\mathcal{I}') = \tau(\mathcal{I}) - 1.$

By inductive hypothesis, items described by the lemma and corresponding to $\mathcal{I}'$ are in I_j. They have the following form:

$$(B \rightarrow Y_1 \ldots Y_{q-1} A Y_{q+1} \ldots Y_{k'}, (s'_1, \ldots, s'_{q-1}), i(\mathcal{I}'), t')$$

These items are processed by the PREDICTOR which put in I_j all items corresponding to instance $\mathcal{I}$ required by the lemma.

• if $r \neq 0$ and $X_r \in T$. Let us consider instance

$$\mathcal{I}' = ((S,s) \Rightarrow^* \gamma (A,s_A) \delta , r - 1)$$

We have

$\tau_1(\mathcal{I}') = \tau_1(\mathcal{I})$

$\tau_2(\mathcal{I}') = \tau_2(\mathcal{I})$

In addition,

$j(\mathcal{I}') = j(\mathcal{I}) - 1$ since X_r is a preterminal.

$\tau_3(\mathcal{I}') = \tau_3(\mathcal{I})$

That is because $\tau_3(\mathcal{I}')$ is the length of the shortest derivation

$$(X_1,s_1) \ldots (X_{r-1},s_{r-1}) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \ldots (t_{j-1},\sigma_{j-1})$$

and $\tau_3(\mathcal{I})$ the one of derivation

$$(X_1,s_1) \ldots (X_r,s_r) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \ldots (t_j,\sigma_j)$$

Since X_r is a preterminal,

$t_j = X_r$ and $\sigma_j = s_r$

and the two derivations have the same length. Finally:

$\tau(\mathcal{I}') = \tau(\mathcal{I}) - 2.$

By inductive hypothesis, for each t in FIRST(δ), the item

$$((A,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k),(s_1, \ldots, s_{r-1}), i, t)$$

is in I_{j-1}. Since X_r is a preterminal, the SCANNER processed this item. Since

$$(A,s_A) \Rightarrow (X_1,s_1) \ldots (X_k,s_k)$$

necessarily

$s_r \in E_l(a_j,X_r)$

and $p_r(s_r) = \text{TRUE}$

and the SCANNER put in I_j the required items.

• if $r \neq 0$ and $X_r \in N$. Provided a change in the order of the derivations composing (D), there is a rule

$$(X_r, e_r) \rightarrow (Y_1,p'_1) \ldots (Y_{k'},p'_{k'})$$

such that

$$\begin{aligned}(S,s) &\Rightarrow^* \gamma (A,s_A) \delta \\ &\Rightarrow \gamma (X_1,s_1) \ldots (X_k,s_k) \delta \\ &\Rightarrow \gamma (X_1,s_1) \ldots (X_{r-1},s_{r-1})(Y_1,s'_1) \ldots (Y_{k'},s'_{k'})(X_{r+1},s_{r+1}) \ldots (X_k,s_k) \delta\end{aligned}$$

In the same way as above, for each t in FIRST(δ), the item

$$((A,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k), (s_1, \ldots, s_{r-1}), i, t)$$

is in $I_{j'}$, for some $j' \leq j - 1$ and for each t in FIRST(δ). Let us write

$\gamma' = \gamma (X_1,s_1) \ldots (X_{r-1},s_{r-1})$

$\delta' = (X_{r+1},s_{r+1}) \ldots (X_k,s_k) \delta$

and consider instance

$$\mathcal{I}' = ((S,s) \Rightarrow^* \gamma'(X_r,s_r) \delta' , k')$$

We have

$$\tau_1(\mathcal{L}') = \tau_1(\mathcal{L}) + 1$$

Let n_1 be the length of the shortest derivation

$$(X_1,s_1) \ldots (X_{r-1},s_{r-1}) \Rightarrow^* (t_{i+1},\sigma_{i+1}) \ldots (t_{j'},\sigma_{j'})$$

and n_2 the one of

$$(X_r,s_r) \Rightarrow^* (t_{j'+1},\sigma_{j'+1}) \ldots (t_j,\sigma_j)$$

$$\tau_3(\mathcal{L}) = n_1 + n_2$$

$$\tau_3(\mathcal{L}') = n_2 - 1$$

$$\tau_2(\mathcal{L}') = \tau_2(\mathcal{L}') + n_1$$

$$j(\mathcal{L}') = j(\mathcal{L})$$

and thus

$$\tau(\mathcal{L}') = \tau(\mathcal{L}) - 1.$$

Since the sentence is in the language, there is a preterminal t' in FIRST(X_{r+1}) such that $E_l(a_{j+1},t') \neq \emptyset$. By inductive hypothesis, the item

$$((X_r, e_r) \rightarrow (Y_1,p'_1) \ldots (Y_{k'},p'_{k'}), (s'_1, \ldots, s'_{k'}), j', t')$$

is hence in I_j. Because of the derivations

$$(S,s) \Rightarrow^* \gamma\,(A,s_A)\,\delta$$

$$\Rightarrow \gamma\,(X_1,s_1) \ldots (X_k,s_k)\,\delta$$

$$\Rightarrow \gamma\,(X_1,s_1) \ldots (X_{r-1},s_{r-1})(Y_1,s'_1) \ldots (Y_{k'},s'_{k'})(X_{r+1},s_{r+1}) \ldots (X_k,s_k)\,\delta$$

the COMPLETER, while processing the above item, put in I_j the required items from items in $I_{j'}$ mentioned above. This finally ends the proof of the lemma. ♦

Theorem 2. Let s be a parse structure of sentence $a_1 \ldots a_n$; s is in the output of the PARSE function.

Proof. Since s is a parse structure, there exists

$$(D) \qquad (S,s) \Rightarrow (X_1,s_1) \ldots (X_k,s_k) \Rightarrow^* (t_1,\sigma_1) \ldots (t_n,\sigma_n)$$

where $\forall\, 1 \leq h \leq n\ \sigma_h \in E_l(a_h,t_h)$. Let us consider the instance

$$((S,s) \Rightarrow (X_1,s_1) \ldots (X_k,s_k)\,,\ k)$$

The lemma shows that, at the end of the algorithm, there is an item

$$(\,(S,e) \rightarrow (X_1,p_1) \ldots (X_k,p_k),(s_1, \ldots, s_k),\ 0,\ \$)$$

in I_n.Moreover, from derivation (D), we get

$$s \in E_c(e,(s_1, \ldots, s_k))$$

s is hence in the result returned by PARSE, which proves the theorem. ♦

From theorem 1 and 2, we finally get

Theorem 3. A sentence $a_1 \ldots a_n$ is in language L_G if and only if the execution of PARSE function with this sentence as input returns a non empty result. In that case, the output of the function is the set of all parse structures for this sentence.

Chapter III
THE PAULINE SYSTEM

I. INTRODUCTION

This chapter is the reference manual of the PAULINE system which has been developed as an application of the f-structure type system and the parsing algorithm that have been described in the two first chapters of this work. The acronym PAULINE stands for *Parsing Augmented Unification-grammars: a Linguistic INteractive Environment.*

The PAULINE system enables the user to

- use and define atomic and complex f-structures types.
- write grammars and dictionaries[1].
- parse sentences.
- trace and step the analysis of a sentence.

All these tasks are performed through a limited user interface. Except the functions related to this user interface, that is, some input and output functions, the whole system has been written in pure COMMON-LISP[2].

II. THE TOP LEVEL MENU

The PAULINE user interface is menu oriented. Most of the times, the system queries the user by prompting him to select one of several choices. In the rest of the chapter, a menu will be a list of choices (commands, grammars, or anything else). The configuration of such a menu will be the sublist of the menu that is actually accessible to the user at some point[3].

When calling PAULINE, the user first sees on the screen the content of a welcoming file while the system is being loaded. Then, a configuration of the top level menu is displayed, and he is prompted to choose a command. Here is how the screen looks like at this point:

```
     PAULINE top level menu.

Menu:
Edit Grammar
Load Grammar
Switch Grammar
Edit Dictionary
Load Dictionary
Switch Dictionary
Change Directory
Exit Pauline

  Move among the menu by typing SPACE.
```

[1] The exact meaning given here to the word *dictionary* will be defined later.

[2] PAULINE has been developed using VAXLISP, under UNIX system.

[3] Some commands can sometimes be disabled. Hence, a command menu can have several configurations.

```
    Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? Edit Grammar
```

When asking the user to select a choice, PAULINE always displays the first possible choice (here `Edit Grammar`). Typing `SPACE` causes this first choice to be replaced by the next one, and so on.

In this section, we will describe the commands from this configuration, except `Exit Pauline`, which returns command to the UNIX system. They are the basic commands that appear in every configuration of the top level menu. The latter has two other configurations than the one previously displayed: when a grammar and a dictionary have been loaded, the parse commands (`Parse` and `Parse and Time`) are added to the menu; when a sentence has been parsed, the `Trace Menu` command is added. These commands will be described in sections V and VI, respectively.

II.1 THE CHANGE DIRECTORY COMMAND

If the user selects this command from the top level menu, the program first tell him what the current directory is at this point, then displays the directories that could become the new current one, and finally prompts the user to select one of these. The selection mechanism is the same as in the top level menu. The choices proposed to the user inside the `Change Directory` command are the following ones:

- `CANCEL` returns to PAULINE top level menu without doing anything. When the user has selected a command from the top level menu and is prompted for a choice, this possibility is always offered to him. It will not be described again in the remaining paragraphs of this section.

- `UP` sets the current directory to be the father of the previous one in the directory tree of the UNIX system.

- The other possibilities are all the subdirectories of the current directory to which the user can move by selecting them.

Here is an example of the execution of the Change Directory command:

```
    PAULINE top level menu.

Menu:
Edit Grammar
Load Grammar
Switch Grammar
Edit Dictionary
Load Dictionary
Switch Dictionary
Change Directory
Exit Pauline

  Move among the menu by typing SPACE.
   Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? Change Directory

Current directory is /usr7/yves/lisp/fstypees/PAULINE
Choose a new one from the menu.
```

```
Menu:
CANCEL
UP
COMPIL
GRAMMAIRES

   Move among the menu by typing SPACE.
    Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? CANCEL
```

II.2 GRAMMAR ORIENTED COMMANDS

• `Edit Grammar` searches for grammar files in the current directory, that is, files whose type slot in the pathname is the string `"gram"`. The program then asks the user to choose among these files the one to edit and calls the GNUEMACS editor on it. Apart from the already existing grammar files, the user can edit a new one by selecting `NEW` or get back to the top level menu by `CANCEL`. Here is an example of an execution of this command:

```
Choose the grammar to edit from the menu:

Menu:
CANCEL
NEW
anbn
anbncn
lex
lex2
shieber

   Move among the menu by typing SPACE.
    Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? CANCEL
```

• `Load Grammar` searches for grammar files in the current directory in the same way as `Edit Grammar` does, and asks the user which grammar to load in the system. After the execution of this command, the loaded grammar becomes the current one, that is, the one that will be used to parse sentences. Here is what happens after selecting this command from the top level menu:

```
Choose the grammar to load from the menu:

Menu:
CANCEL
ANBN
ANBNCN
LEX
LEX2
SHIEBER

   Move among the menu by typing SPACE.
    Once you have made up your mind, type RETURN.
```

```
Selected choice (Help = ^H) ? CANCEL
```

• `Switch Grammar` enables the user to change the current grammar. It first tells the user what the current grammar is at this point and prompts him to choose one of the previously loaded grammars. If we had previously loaded the grammars ANBN and SHIEBER, choosing this command from the top level menu would display on the screen the following:

```
Current grammar is SHIEBER
Choose new current grammar from the menu:


Menu:
CANCEL
SHIEBER
ANBN

  Move among the menu by typing SPACE.
   Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? CANCEL
```

II.3 DICTIONARY ORIENTED COMMANDS

These commands (Edit Dictionary, Load Dictionary and Switch Dictionary) behave exactly like the corresponding grammar oriented command, except that dictionary files will be presumed to be of the "`lex`" type.

N.B In the actual implementation (and it is one of its weaknesses!), dictionaries must be loaded after grammars. Further, if a grammar is reloaded after some modification, the corresponding dictionary should also be reloaded even if no modification was undertaken.

III. STRUCTURES

This section describe the representation structures used in PAULINE. These structures are typed f-structures as described in the first chapter of this work. We will first present the type system used for PAULINE, then the syntax used to read and write structures.

III.1 TYPES IN PAULINE

PAULINE provides the possibility of defining atomic and complex f-structure types.

III.1.1 Atomic Types

Some basic atomic types are predefined They are:

- `boolean`
- `integer`
- `character`
- `strings`

The last three are equivalent to the corresponding COMMON-LISP data type. The first one can be seen as a scalar type whose extension would be `{true, false}`. Usual operations on these types are also predefined. We will describe them in the next section.

Besides these predefined atomic types, the user can define scalar types, which are considered atomic types. This is done from their extension by means of the `define-scalar-type` LISP function. To define such a type, one has to put a call to this function in the grammar file, according to the following syntax:

```
(define-scalar-type <name> <extension>)
```

<name> should be a LISP symbol which becomes the name of the new type.

<extension> is a list of symbols which become the objects of the new type.

Since `define-scalar-type` is a LISP function, its arguments are evaluated before computation and hence, in every call to this function, the name and the extension of the type should be quoted.

EXAMPLE. One can define a `number` type by the following call:

```
(define-scalar-type 'number '(singular plural))
```

A call to `define-scalar-type` automatically defines some functions on the new type. They will be described in the section III.

III.1.2 Complex types

There is no predefined complex f-structure type in PAULINE. The user defines such types by means of the `define-complex-type` LISP function. As for `define-scalar-type`, a call to this function has to be put inside the grammar file to define the type. The syntax of the function is as follows:

```
(define-complex-type <name> <feature descriptions>)
```

<feature description> is a list of dotted pairs whose CAR is a feature and whose CDR is the result type of this feature in the new type.

As for `define-scalar-type`, the name and the feature descriptions have to be quoted when defining a complex type.

EXAMPLE. One can define an `agreement` type by the following call:

```
(define-complex-type
        'agreement
        '((number . number) (person . integer)))
```

In the first feature description of this call, one must be aware that the two symbols `number` are interpreted quite differently. The first one defines a feature `number` in the new type. The second one is supposed to be a type name, even if the type `number` can be defined after the type `agreement`.

III.1.3 The Top Type

PAULINE also includes a predefined `top` type. It contains only one object, that is, `t`. All other types defined in PAULINE, either predefined or user defined, are considered subtypes of the `top` type. Its only object, `t`, will unify with any other object of any type. The `top` type can be useful when one does not want to specify the result type of some feature in the definition of a complex type.

III.2 F-STRUCTURES SYNTAX

III.2.1 The Syntax

The syntax described here is used by the user to write f-structures and by PAULINE to output f-structures.

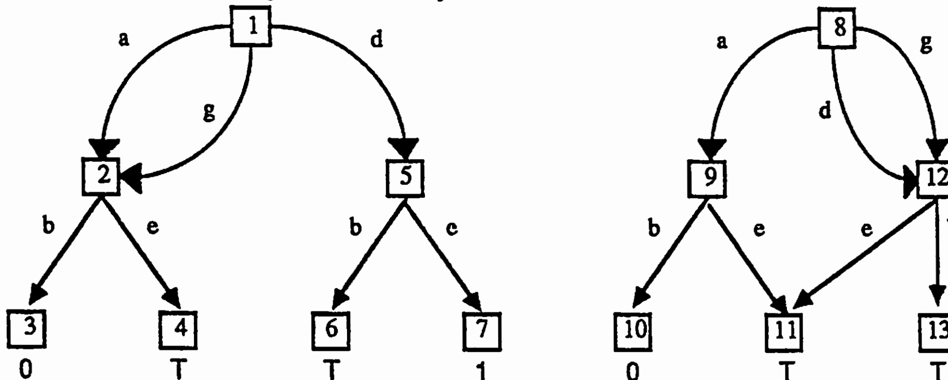
Objects of atomic types are written according to their LISP syntax. This is usually their intuitive form. Atomic null value are written as the name of their type in square brackets. It is in fact very seldom that one has to write atomic null values. PAULINE has to output them sometimes especially in case of interactive evaluation of expressions (see sections III and V).

The syntax for f-structures is as follows:

```
f-structure ::=
    atomic f-structure |
    index |
    %type-name{
            (feature-name:f-structure)*}
index ::= @itag [@= f-structure]
```

Tags for indices can be any LISP symbol; however, PAULINE always outputs numbers for these tags. If the @= part of the index is present the following f-structure is assigned to the previous index.

EXAMPLE. As an example, we show the two structures that are unified in chapter I, paragraph III.3.3 according to Pauline syntax. These two structures are as follows:



The two complex types are defined by the following calls to define-complex-type:

```
(define-complex-type
        'be
        '((b . integer) (e . integer)))
(define-complex-type
        'adg
        '((a . be) (d . be) (g . be)))
```

The syntax for the first structure is:

```
%adg{ a :   @i1 @= %be{b : 0}
      d :   %be{e : 1}
      g :   @i1}
```

The syntax for the second one is:

```
%adg{ a :   %be{ b :   0
                 e :   @i2}
      d :   @i3 @=%be{e : @i2}
      g :   @i3}
```

II.2.2 Templates

PAULINE enables the user to define and use templates. The system refers to them by means of an identification number. Templates are defined inside grammar files according to the following syntax:

```
#nt f-structure
```

n is a natural integer that becomes the identification number of the template, and f-structure is written according to the above described syntax and becomes the value of the new template. The definition of templates has to be enclosed between the two following LISP function calls:

```
(enable-template-definition)
(disable-template-definition)
```

Inside f-structures, templates are referred to by the syntax

```
#nt
```

IV. EVALUATORS

IV.1 PREDICATES AND EXPRESSIONS

Predicates and expressions are put in grammar rules to be evaluated on structures assigned to grammar symbols. In this paragraph, we describe the syntax of *simple contextual expressions*. These are used both for rule expressions and predicates attached to right-hand side symbols in a way that will be described in section V.

Simple contextual expressions, or simply expressions, are always evaluated according to a context, that is, a list of structures (see chapter II). Their syntax can be described as follows:

```
expression ::= NIL |
               atomic f-structure |
               template |
               access |
               assignment |
               control form |
               functional form
```

Atomic f-structures and templates keep the syntax defined in the previous section. NIL is the inconsistent object. It evaluates to itself. The evaluation of atomic f-structures is also the structure itself. Templates evaluate themselves to a copy of their value.

Accesses enable us to consult the context. An access is a list whose CAR is a number and CDR a path. They evaluate themselves to the value of this path in the structure which rank in the context is the CAR of the access.

Assignments enable us to modify structures of the context. Their syntax is as follows:

```
assignment ::= access (== | =) expression |
               access &= access
```

- In the == assignment, the path of the access must be void. The expression is evaluated and the structure referred to by the access is replaced by the value without any type checking.
- The = assignment is a simple path assignment. The expression is evaluated and the value is assigned to the path referred to by the access. If the type of the value does not match the type of the access, the assignment is not done and the returned result is NIL. Otherwise, the evaluation of the assignment returns the value of the expression, that is, the new value of the access.
- The &= assignment destructively unifies the two accesses; the returned value is the value of the unification. If the latter fails, the two involved structures, that is the two structures referred to by the CARs of the accesses become NIL.

Three control forms are provided:

```
control form ::=  (SEQ expression+)  |
                  (cond (test-expression expression)*)
                  (if test then-expression else-
expression)
```

• COND and IF constructions have the same syntax as in LISP. The user should remember however that tests will be verified if and only if they evaluate themselves to the boolean value true.

• SEQ evaluates in sequence the expressions that follow and returns the value of the last one. There is however one exception to this rule: if one of the expressions evaluates itself to NIL, SEQ also returns NIL.

The functional forms also have the same kind of syntax as in LISP, that is, a list whose CAR is the operation symbol and the CDR a list of expressions which are evaluated before. However, PAULINE checks the types of arguments. The operator has to be defined in the grammar file by means of the following LISP function:

```
(define-intern-function   <operation symbol>
                          <args types>
                          <result type>
                          <computation method>)
```

• <args type> is a list of types names. If one of these names is enclosed in brackets, there can be several arguments of this type.

• <computation method> is a LISP lambda expression.

• <result type> is unimportant since PAULINE does not actually use it.

EXAMPLE. Addition on integers can be defined as follows:

```
(define-intern-function '+ '((integer)) 'integer #'+)
```

Function symbols can be overloaded. PAULINE chooses the right computation method from the types of the arguments. If there is no previously defined function that matches the types of the arguments, the result of the evaluation is NIL.

Some functions are predefined as follows:

• on boolean type:

+	:	logical or
*	:	logical and
!	:	negation
=	:	equality predicate
&	:	non destructive unification

• on integer type:

+	:	addition
-	:	soustraction
*	:	multiplication
/	:	division
%	:	modulo
=	:	equality predicate
<	:	order
&	:	non destructive unification

• on type character type:

<	:	lexicographic order
=	:	weak equality predicate

== : strong equality predicate[1]
& : non destructive unification

- on `string` type:
  - < : lexicographic order
  - + : concatenation
  - = : weak equality predicate
  - == : strong equality predicate[2]
  - & : non destructive unification

In addition, the definition of a scalar type automatically defines the following functions:

= : equality predicate
< : ordered, defined as the one of the extension given as an argument to `define-scalar-type`.
& : non destructive unification

and the definition of a complex type automatically defines the non destructive unification operation & on this type.

IV.2 THE LEXICAL EVALUATOR

In this paragraph, we describe dictionaries in PAULINE and the way how the lexical evaluator (see chapter II) has been implemented.

Dictionary files are sequences of entries, where the syntax of an entry is as follows:

```
entry ::= word : f-structure
```

The word is simply written as a sequence of characters (all comparisons will be case independent). The f-structure is written according to the syntax described in paragraph III.2. There can, of course, be several entries for the same word.

The lexical evaluator called on a word w and a preterminal symbol `t` acts as follows: it first finds the list `l` of all structures associated with w in the current dictionary. If `t` is a string equal to w, it returns the whole list. That enables us to put words directly in grammar rules. Otherwise, it returns the list of all structures s in `l` verifying one of the two conditions:

- The type of s is equal to `t`
- The value of the CAT feature in s is equal to `t`.

V. GRAMMAR RULES AND PARSING

V.1 GRAMMAR RULES

Grammar rules are defined in grammar files by means of the `defrule` LISP macro, which syntax is the following:

```
(defrule <production>
         <rule expression>
         <list of predicates>)
```

---

[1] The two latter operations correspond respectively to COMMON-LISP functions `char-equal` and `char=`. Hence:

```
        a = A   is true
but     a == A  is false.
```

[2] The two latter operations correspond respectively to COMMON-LISP functions `string-equal` and `string=`. See the preceding footnote.

• The second symbol of the `<production>` is ignored by PAULINE and is meant to represent the arrow.

• `<rule expression>` is a list of simple contextual expressions as defined in previous section. The result of the contextual evaluation will be the list of these expressions where inconsistent objects (that is `NIL`) have been deleted.

• `<list of predicates>` contains in the order of production the predicates associated with right-hand side symbols of the rule, as simple contextual expressions. They will be verified if and only if they evaluate to `true`. Only the structure attached to the corresponding symbol is passed in the context. The other structures of the rule can neither be consulted, nor be modified by predicates.

EXAMPLE. The rule R_1 described in chapter II, paragraph III.2 would be defined by

```
(defrule S ==> A A
         ((if   (= (1 counter) (2 counter))
                (1 counter)))
         ((= (1 letter) a) (= (2 letter) b)))
```

V.2 PARSING

Once a grammar and a dictionary have been loaded, the top level menu enters the following configuration:

```
Menu:
Parse
Parse and Time
Edit Grammar
Load Grammar
Switch Grammar
Edit Dictionary
Load Dictionary
Switch Dictionary
Change Directory
Exit Pauline

  Move among the menu by typing SPACE.
   Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? Parse
```

Selecting the `Parse` command, the user is prompted to enter a sentence. The program assumes that this sentence ends with a dot. While parsing the sentence, it displays on the screen the words of the sentence, one by one, to show what point it has reached in the parse. After parsing, PAULINE displays the results, that is all structures that have been assigned to symbol `S` in rules recognizing the whole sentence. Besides doing the same things, the `Parse and Time` command also displays some time information.

EXAMPLE. Using the `Parse and Time` command, here is what appears on the screen while parsing sentence *aaabbb* with the grammar described in chapter II, paragraph III.2. The sentence input by the user appears in bold faced type.

```
Parser> a a a b b b.
Parsing . . .
          a a a b b b
Result(s):
3

Parsing time:          CPU: 4.89s    Elapsed: 12.28s.
Garbage collection:    CPU: 1.41s    Elapsed: 4.95s.
```

VI. THE TRACER

Once a sentence has been parsed, a last command is added to the top level menu: the `Trace Menu` command. Selecting this command brings to the screen the following menu:

```
Menu:
Display trees
Node Structure
Node Trace
Symbol stepper
Exit Tracer

  Move among the menu by typing SPACE.
   Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? Display trees
```

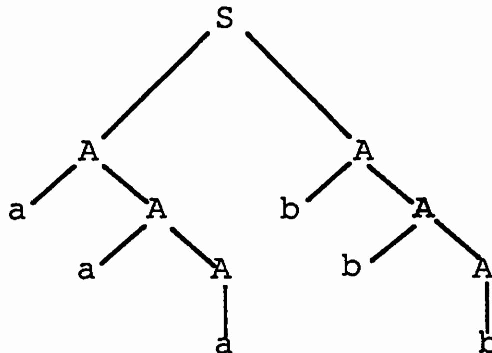
In this section, we describe the commands available in this menu.

VI.1 DERIVATION TREES

The first command of the tracer menu, `Display Trees`, displays on the screen all derivation trees for the previously parsed sentence. Leafs of these trees are labelled by the corresponding word and not by the preterminal symbol.

It is possible to look at the structure assigned to a node of these trees by means of the `Node Structure` command in the tracer menu. After selecting this command, the user is prompted to select a node. For this purpose, he should enter a list of numbers. The first is the rank of the derivation tree (beginning at 0), and then, at each level, the rank of the selected son should be put in the list.

EXAMPLE. The unique derivation tree for the sentence parsed in the previous section is:



The node corresponding to sequence *bb* in the sentence (which has been put in bold faced type in the above picture) would be selected by the list `(0 1 1)`.

Once the node has been selected, PAULINE displays the structures assigned to this node.

EXAMPLE. After parsing sentence *aaabbb*, the execution of these two commands proceeds as follows. (Once again, what has been entered by the user has been put in bold faced type.)

```
Menu:
Display trees
Node Structure
Node Trace
Symbol stepper
Exit Tracer

  Move among the menu by typing SPACE.
   Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? Display trees

 S   A    a
          A    a
               A    a
     A    b
          A    b
               A    b


Menu:
Display trees
Node Structure
Node Trace
Symbol stepper
Exit Tracer

  Move among the menu by typing SPACE.
   Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? Node Structure

Node? (0 1 1)
%SEQUENCE {LETTER:  B
           COUNTER: 2}
```

VI.2 THE TRACER

The tracer enables us to move inside the recognition of a symbol in a derivation tree. It corresponds to the `Node Trace` command of the tracer menu. After selecting this command, the user is prompted for a node, which should be entered according to the format described in previous paragraph. Then, PAULINE displays information about the rule corresponding to the node: its identification number, a dotted production, and the lookahead symbol. The command available at this point are the following:

- `LEFT` Moves the dot in the production one symbol left.

- `RIGHT` Moves the dot in the production one symbol right.
- `DOWN` Traces the recognition of the symbol preceding the dot in the production.
- `UP` Moves back where the previous `DOWN` command was executed.
- `EVAL` Calls a read-eval-print loop of simple contextual expressions. The context is set to structures assigned to symbols preceding the dot in the production.
- `EXIT` Goes back to the tracer menu.

EXAMPLE. Here is a sample session of the tracer.

```
Menu:
Display trees
Node Structure
Node Trace
Symbol stepper
Exit Tracer

  Move among the menu by typing SPACE.
   Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? Node Trace

Node? (0)
#0: rule:0   S ==> . A A   Lookahead symbol: NIL

Selected choice (Help = ^H) ? RIGHT

#0: rule:0   S ==> A . A   Lookahead symbol: NIL

Selected choice (Help = ^H) ? DOWN

--#1: rule:2   A ==> . LETTER A   Lookahead symbol:
LETTER

Selected choice (Help = ^H) ? RIGHT

--#1: rule:2   A ==> LETTER . A   Lookahead symbol:
LETTER

Selected choice (Help = ^H) ? RIGHT

--#1: rule:2   A ==> LETTER A .   Lookahead symbol:
LETTER

Selected choice (Help = ^H) ? EVAL

You enter the evaluation loop.
Edition commands of EDLIN are available.

  To exit, eval ().

Eval> (2)
%SEQUENCE {LETTER:  A
           COUNTER: 2}
```

```
Eval> ()
--#1: rule:2   A ==> LETTER A .   Lookahead symbol:
LETTER

Selected choice (Help = ^H) ? UP

#0: rule:0   S ==> A . A   Lookahead symbol: NIL

Selected choice (Help = ^H) ? LEFT

#0: rule:0   S ==> . A A    Lookahead symbol: NIL

Selected choice (Help = ^H) ? EXIT
```

VI.3 THE SYMBOL STEPPER

The symbol stepper is certainly the most interesting feature of the tracer menu. It enables the user to simulate the actual parsing of the sentence. It is not a real step-by-step analysis, since the stepper uses structures built during the analysis without actually redoing it.

After this command has been selected, the user is prompted to enter a non-terminal symbol and a position in the sentence (the beginning of the sentence is assumed to be 0). Then, PAULINE displays the predictions made on this symbol at this position. If there are several such predictions, the user is asked to make a choice. Each of the possible choices is displayed with information on the part of the sequence this choice will be able to recognize. This information will always be displayed by the stepper when the user is prompted for a choice. Afterwards, the stepper displays information on the selected rule, exactly as the tracer does. The commands that can be selected at this point are the following (some of them can be disabled according to the configuration of the rule).

- `RIGHT` Moves the dot one symbol right. In that case, there can be several possible structures to assign to the first non recognized symbol. The user is then asked to choose one of them.
- `BACK` Is equivalent to the `LEFT` command inside the tracer.
- `EVAL` Is the same as in the tracer.
- `EXIT` Goes back to the tracer menu.
- `BRANCH` Goes back to the previous prediction.
- `PROCEED` Simulates the analysis. The behavior depends on the position of the dot in the production being stepped. If the dot is at the end of the production, the stepper displays information about the completion of the rule, that is, the evaluation of the rule expression, recalls the rules in which the completed rule was predicted then goes up to this rule and moves the dot one point right if the candidate structure built from the rule expression matches the corresponding predicate. In the other cases, either the `PROCEED` command makes a prediction and goes down to the stepping of the predicted rule, or it displays information on the scanning that takes place at this point.

EXAMPLE. We show only the behavior of the `PROCEED` and `RIGHT` commands since the other ones are not very different from their equivalents in the tracer.

```
PAULINE Stepper.

Parsed sentence: a a a b b b.
```

```
Symbol to step? S
Stepping position? 0
#0: rule:0   S ==> . A A   Lookahead symbol: NIL

Selected choice (Help = ^H) ? RIGHT

CHOICE BETWEEN:
 - 1. %SEQUENCE {LETTER:  A
                 COUNTER: 3}
 leads to complete recognition of:
     a a a b b b
 leads to incomplete recognition of:
     a a a b
     a a a b b
 - 2. %SEQUENCE {LETTER:  A
                 COUNTER: 2}
 leads to incomplete recognition of:
     a a
 - 3. %SEQUENCE {LETTER:  A
                 COUNTER: 1}
 leads to incomplete recognition of:
     a
Choice? (1..3)1

 A --> %SEQUENCE {LETTER:  A
                 COUNTER: 3}
#0: rule:0   S ==> A . A   Lookahead symbol: NIL

Selected choice (Help = ^H) ? PROCEED

 PREDICTION on symbol A at position 4.
CHOICE BETWEEN:
 - 1. rule:2   A ==> . LETTER A   Lookahead symbol:
NIL
 leads to complete recognition of:
     b b
     b b b
 - 2. rule:1   A ==> . LETTER   Lookahead symbol: NIL
 leads to complete recognition of:
     b
Choice? (1..2)1

--#1: rule:2   A ==> . LETTER A   Lookahead symbol:
NIL

Selected choice (Help = ^H) ? PROCEED

  SCANNER:
  Symbol: LETTER
  Word: b
      Result(s) of the lexical evaluation:
        B
  Rhs-predicate: TRUE
  LETTER --> B
--#1: rule:2   A ==> LETTER . A   Lookahead symbol:
NIL
```

```
Selected choice (Help = ^H) ? RIGHT

CHOICE BETWEEN:
  - 1. %SEQUENCE {LETTER:  B
                  COUNTER: 2}
  leads to complete recognition of:
       b b b
  - 2. %SEQUENCE {LETTER:  B
                  COUNTER: 1}
  leads to complete recognition of:
       b b
Choice? (1..2)1

  A --> %SEQUENCE {LETTER:  B
                   COUNTER: 2}
--#1: rule:2   A ==> LETTER A .   Lookahead symbol:
NIL

Selected choice (Help = ^H) ? PROCEED

  COMPLETION of rule A ==> LETTER A.
  ---->" b b b"
  Rhs structures:
  LETTER --> B
  A --> %SEQUENCE {LETTER:  B
                   COUNTER: 2}
  Rule expression: ((IF (= (1) (2 LETTER)) (SEQ ((0)
== (@ 1)) ((0 LETTER) = (1)) ((0 COUNTER) = (+ 1 (2
COUNTER))) (0))))
  Result of the contextual evaluation:
      %SEQUENCE {LETTER:  B
                 COUNTER: 3}
 Wants to fit into:
#0: rule:0   S ==> A . A   Lookahead symbol: NIL
 Rhs-predicate: (= (2 LETTER) B)
 A --> %SEQUENCE {LETTER:  B
                  COUNTER: 3}
#0: rule:0   S ==> A A .   Lookahead symbol: NIL

Selected choice (Help = ^H) ? PROCEED

 COMPLETION of rule S ==> A A.
 ---->" a a a b b b"
 Rhs structures:
 A --> %SEQUENCE {LETTER:  A
                  COUNTER: 3}
 A --> %SEQUENCE {LETTER:  B
                  COUNTER: 3}
 Rule expression: ((IF (= (1 COUNTER) (2 COUNTER)) (1
COUNTER)))
 Result of the contextual evaluation:
3
```

CONCLUSION

In this work, we first presented a feature structure type system where two kinds of types are defined: atomic and complex types. Atomic types are usual simple types such as boolean, numbers or strings whose specification is incorporated in the formalism. Operations defined on these types such as arithmetic, concatenation, etc. enable us to avoid the unification cost for simple operations needed by any real-size Natural Language Processing system. Complex types are defined by a set of valid features considered operations, and reentrancy is treated by means of index functions.

We then described a parsing algorithm derived from Earley's algorithm which builds representation structures during parsing but assumes only the existence of three interface evaluators to handle structures, no matter what, exactly, these structures are (trees, graphs, f-structures...).

Finally, we presented the PAULINE system, which is a prototype environment including both the type system described in chapter I and the parsing algorithm described in chapter II.

Further research extending this work could be:

- Define subtypes in the type system to be able to handle the power of inheritance mechanisms (see [AIT 86]).
- Enable the parsing algorithm and the grammar formalism to follow user defined parsing strategies.
- Study the cost of this algorithm (this depends on the operations the interface evaluators perform).
- Improve the treatment of dictionaries and lexical items in PAULINE, and add a morphological analyzer to the program.

In its current implementation, the PAULINE system is only a prototype. However, since it can handle both unification and simpler operations like arithmetic, it can be a step toward practical use of unification grammars in actual Natural Language Processing systems.

APPENDIX
A Pure Unification Grammar

This appendix presents the sample grammar 1 of [SHI 86] written in PAULINE system.

GRAMMAR FILE

;**Types definition.** There are two scalar types, the first one for grammatical ;categories the second for grammatical number. There is two complex types: ;syntagm, to represent structures associated with phrases and sentences, and ;agreement, to represent agreement informations.

```
(define-scalar-type 'cat '(s np vp v))
(define-scalar-type 'number '(plural singular))

(define-complex-type 'syntagm '((cat . cat) (head .
head) (subject . head)))

(define-complex-type 'head '((agreement . agreement)
(subject .
head)))

(define-complex-type 'agreement '((number . number)
(person .
integer)))
```

; **Template definition.** Two templates are defined, to initialize the structures for ;sentences and verbal phrase. (Nominal phrases are treated here as preterminal ;symbols.).

```
(enable-template-definition)

#1t %syntagm{cat:s}
#2t %syntagm{cat:vp}

(disable-template-definition)
```

; **Grammar rules:**

```
(defrule S ==> NP VP
       ;rule expression:
       ((seq ((0) == #1t)
             ;unifications:
             ((0 head) &= (2 head))
             ((0 head subject) &= (1 head))
             ((0 subject) &= (0 head subject))
             ;result:
             (0)))
       ;rhs predicates:
       (true true))

(defrule VP ==> V
        ((seq ((0) == #2t)
              ((0 head) &= (1 head))
              (0)))
        (true))
```

DICTIONARY FILE

```
Uther:%syntagm
         {cat:np
          head:%head
                 {agreement:%agreement
                              {number:singular
                               person:3}}}

Sleeps:%syntagm
         {cat:v
          head:%head
                 {subject:%head
                           {agreement:
                             %agreement
                               {number:singular
                                person:3}}}}

Sleep:%syntagm
         {cat:v
          head:%head
                 {subject:%head
                           {agreement:
                             %agreement
                               {number:plural}}}}
```

SAMPLE PAULINE SESSION

In this section, we show the parsing of the sentence *Uther sleeps*. We assume that the grammar and the dictionnary described above have previously been loaded. Here is the execution of `Parse` command:

```
Menu:
Parse
Parse and Time
Edit Grammar
Load Grammar
Switch Grammar
Edit Dictionnary
Load Dictionnary
Switch Dictionnary
Change Directory
Exit Pauline

  Move among the menu by typing SPACE.
   Once you have made up your mind, type RETURN.

Selected choice (Help = ^H) ? Parse

Parser> Uther sleeps.
Parsing . . .
          Uther sleeps
```

```
Result(s):

%SYNTAGM {CAT:      S
          HEAD:     %HEAD
                     {SUBJECT:@i1
                              @= %HEAD
                                 {AGREEMENT:
                                   %AGREEMENT
                                     {NUMBER: SINGULAR
                                      PERSON: 3}
                                  SUBJECT: %HEAD {}}}
          SUBJECT: @i1 }
```

REFERENCES

[AHO 72] A.V. AHO, J.D. ULLMAN. *The Theory of Parsing, Translation and Compiling*. Prentice Hall 1972.

[AIT 86] H. AIT-KACI. An Algebraic Semantics to the Effective Resolution of Type Equations. In *Theoretical Computer Science* 45 p 293-351. 1986.

[BAR 87] G.E. BARTON Jr, R.C. BERWICK, E.S. RISTAD. *Computational Complexity and Natural Language*. MIT Press. 1987.

[EAR 68] J. EARLEY. *An Efficient Context-free Parsing Algorithm*. PhD Thesis, Computer Science Dpt, Carnegie Mellon University. 1968.

[GAZ 85] G. GAZDAR, G.E. KLEIN, G.K. PULLUM, I.A. SAG. *Generalized Phrase Structure Grammar*. Harvard University Press. 1985.

[GOG 78] J.A. GOGUEN, J.W. THATCHER, E.G. WAGNER. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*, Vol 4. Prentice Hall 1978.

[KAP 83] R. KAPLAN, J. BRESNAN. Lexical-functionnal Grammar: A Formal System for Grammatical Representation. In *The Mental Representation of Grammatical Relations*. MIT Press. 1983.

[KAS 86] R.T. KASPER, W.C. ROUNDS. A Logical Semantics for Feature Structures. In *The Proceedings of the 24th Annual Meeting of the ACL*. New York. 1986.

[KAY 83] M. KAY. *Unification Grammar*. Technical Report, Xerox Palo Alto Research Center. 1983.

[NAS 86] H. AIT-KACI, R. NASR. LOGIN: a Logic Programming Language with Built-in Inheritance. In *Journal of Logic Programming* 1986:3.

[PER 80] F.C.N. PEREIRA, D.H.D. WARREN. Definite Clause Grammars for Language Analysis. In *Artificial Intelligence*. 13:231-278. 1980.

[PER 85] **F.C.N. PEREIRA.** A Structure-sharing Representation for Unification-based Grammar Formalisms. In *The Proceedings of the 23rd Annual Meeting of the ACL.* Chicago. 1985.

[PER 86] **F.C.N. PEREIRA.** Grammars and Logic of Partial Information. In *The Proceedings of the International Conference of Logic Programming.* Melbourne. 1986.

[POL 87] **C. POLLARD, I.A. SAG.** *Information-based Syntax and Semantics, Volume 1: Fundamentals.* CSLI Lecture Notes 13. 1987.

[SHI 86] **S.M. SHIEBER.** *An Introduction to Unification-based Approaches to Grammar.* CSLI Lecture Notes 4. 1986.

[TOM 86] **M. TOMITA.** *Efficient Parsing for Natural Language.* Kluwer Academic Press. 1986.

[WRO 87] **D. WROBLEWSKI.** Nondestructive Graph Unification. In *The Proceedings of the 6th National Conference on Artificial Intelligence.* Seattle. 1987.

[ZAJ 88] **R. ZAJAC.** Operations on Typed Feature Structures: Motivations and Definitions. *ATR Technical Report.* TR-I-0045, 1988.