Head-Corner Parsing using Typed Feature Structures

Mark Moll

August 1995

Subdepartment Software Engineering and
Theoretical Computer Science
Computer Science Department
Universitity of Twente

Abstract

In this report a description will be given of how typed feature structures can be specified. A specification language will be presented for the specification of types, words and grammar rules. An unification algorithm for typed feature structures as well as an algorithm to compute the least upper bound relation for a type lattice will be given. Finally, a head-corner parsing schema for typed feature structures will be presented.

Samenvatting

In dit rapport zal een beschrijving worden gegeven van de manier waarop getypeerde *feature structures* kunnen worden gespecificeerd. Een specificatie taal zal worden gepresenteerd voor de specificatie van types, woorden en grammatica regels. Een unificatie algoritme voor getypeerde *feature structures* zal worden gegeven, evenals een algoritme om de kleinste bovengrens relatie voor een type tralie te berekenen. Tot slot zal er een *head-corner* ontleed schema voor getypeerde *feature structures* worden gepresenteerd.

Preface

This is my Master's thesis with which my Computer Science study at the University of Twente has come to an end. The assignment has been carried out between November 1994 and August 1995 at the subdepartment Software Engineering and Theoretical Computer Science.

During a period of time of about 9 months I have done quite some reading, writing and programming. Actually, most of the time I have spent on programming. If there's one thing I have learnt from my assignment, then it is, that it always takes about ten times longer to program something than you think it would take.

There are a number of people that I would like to thank. First of all, I am grateful to the members of my graduation committee for their time and the support they have given me. Also I would like to thank my friends and family, who had to put up my 'self-proclaimed social isolation' these last two months. Finally, I'd like to mention Risto Miikkulainen, from whom I have learnt a lot about doing research.

Mark Moll,
August 1995.

Contents

Chapter 1

Introduction

"When *I* use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."

"The question is," said Alice, "whether you *can* make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master—that's all."

Lewis Carroll, *Through the Looking Glass*

The question is what language is being used, I would say. If the language being considered is a formal system, like logic or mathematics, Humpty Dumpty would be right. In these languages things have exactly the meaning that has been assigned to them—neither more nor less. But *within* one formal system the meaning of the entities is fixed, otherwise it would not be a very useful system. With natural languages words have a meaning upon which there is some general agreement. That is, words have a more or less fixed meaning, which is what Alice is trying to say. But the same word can have different connotations for different people, based on the mental models that have been formed for each word Sowa (1984).

1.1 Assignment Description

In computational linguistics formal systems are defined and used to describe natural language utterances. Of course, these systems cannot fully capture the (intended) meaning of natural language, but for most practical situations this is not a problem. In this report I will describe (a specification language for) such a formal system: the typed feature structures. The specification language itself could also be called a formal system, but that might go a bit to far; no statements about the soundness or completeness of this language are made. But designing a formal system was not the aim of my thesis. The aim was to design a complete parsing system that could parse sentences using typed feature structures. In order to do so I could use an existing implementation of a parser for untyped feature structures. The first version of this parser was made by Verlinden (1993). An improved version of this parser, that uses a more efficient unification algorithm was made by Veldhuijzen van Zanten (1994). This parser needed to be reimplemented to make it suitable for typed feature structures. Also, a specification language and a new unification algorithm for typed feature

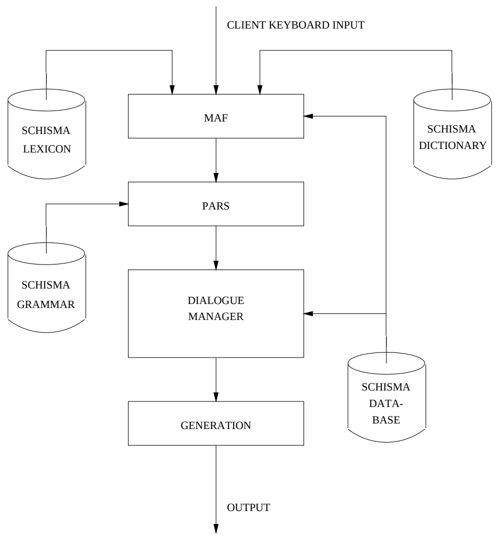


Figure 1.1: **Conceptual view of** SCHISMA.

structures needed to be defined.

1.2 SCHISMA

My assignment was carried out as part of the SCHISMA project. This is a joint research project of Parlevink (the language-engineering project of the Computer Science Department) and the Speech and Language group of KPN Research (the R&D department of the Royal PTT Nederland). Within the SCHISMA project a theater information and booking system is being developed. This dialogue system is going to have a natural language interface.

A global architecture of the Schisma dialogue system is shown in figure 1.1. This is the architecture of a first prototype of the dialogue system. The actual status of a dialogue does not dynamically influence the process of parsing, nor the preprocessing of the input in the module MAF, that handles morphological analysis and fault-correction. The system processes user input typed in on a keyboard. The parser is that part of the dialogue system that should identify the relevant semantic information communicated by the user. It outputs one (ideally) or several (in case of semantic ambiguities) analyses of the input from the MAF module. The dialogue manager then selects the most likely analysis given the status of the current dialogue. More about the dialogue manager can be found in Hoeven et al. (1995).

With my thesis project I focus on the PARS module. To allow the PARS module to work independently from the MAF module, the parser is equipped with a (very simple) scanner to read words and look them up in a lexicon. When the MAF and PARS module are integrated, the scanner can be removed.

1.3 Context-Free Grammars

In the field of computational linguistics (natural) languages are formalized by means of a grammar and a lexicon[1]. There are many classes of grammars Sudkamp (1988). An important class of grammars are the

[1]Often the lexicon is considered to be a part of the grammar. Henceforth we will use the word ‘grammar’ to refer to both the grammar and the lexicon.

context-free grammars. A context-free grammar G is a 4-tuple $\langle N,\Sigma,P,S\rangle$. N is a set of nonterminals (‘names of parts of sentences’). The set Σ is a collection of terminals. Strictly speaking, the terminals correspond to words, but often word categories, like ‘noun’ or ‘verb’, are used as terminals instead of words. These word categories are also known as pre-terminals to distinguish them from the words. The elements of the set $V = N \cup \Sigma$ are called *constituents*. The third part of a grammar, P, is a set of production rules. Finally, the symbol S is the start symbol of the grammar. S is a nonterminal ($S \in N$) and is the first symbol in a derivation of a sentence. The following example illustrates these concepts.

Let grammar G be defined as

$$\begin{aligned} G &= \langle N,\Sigma,P,S\rangle \\ N &= \{S, NP, VP\} \\ \Sigma &= \{{}^*det, {}^*noun, {}^*verb\} \\ P &= \{S \rightarrow NP\ VP, \\ & \quad VP \rightarrow {}^*verb\ NP, \\ & \quad NP \rightarrow {}^*det\ {}^*noun\} \end{aligned}$$

The nonterminals S, NP and VP are commonly used and are intended to stand for a sentence, a noun phrase and a verb phrase, respectively. The terminals ${}^*det$, ${}^*noun$ and ${}^*verb$ are used to denote a determiner, a noun and a verb, respectively. This grammar can be used for sentences like “The man bites a dog”. If we replace the words in this sentence by their word categories, we get “${}^*det\ {}^*noun\ {}^*verb\ {}^*det\ {}^*noun$”. A possible derivation of this sentence, given the grammar above, is

$$\begin{aligned} S &\Rightarrow NP\ VP \Rightarrow {}^*det\ {}^*noun\ VP \Rightarrow {}^*det\ {}^*noun\ {}^*verb\ NP \\ &\Rightarrow {}^*det\ {}^*noun\ {}^*verb\ {}^*det\ {}^*noun \end{aligned}$$

With a derivation tree all possible derivations can be seen in one figure. For our example the derivation tree

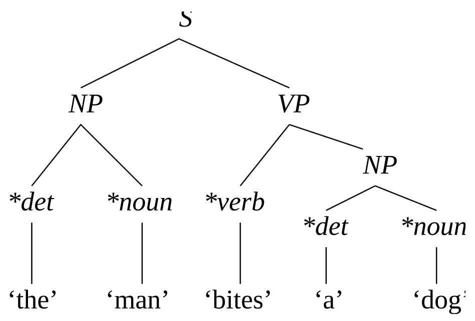


Figure 1.2: **The derivation tree for the sentence “The man bites a dog”.**

is given in figure 1.2. Note that there is also another derivation possible.

Ambiguity can now be introduced in terms of derivation trees. A sentence is considered to be ambiguous, if there is more than one valid derivation tree for it. So, a sentence that has multiple derivations, but only one derivation tree is not considered to ambiguous. A grammar from which ambiguous sentences can be derived is also called ambiguous.

1.4 Overview

In this section I shall give an overview of the rest of my thesis. In chapter 2 the general notion of types will be introduced. In this chapter the type theory is applied to the domain of feature structures. A description of a new unification algorithm for typed feature structures based on the algorithm of Veldhuijzen van Zanten will conclude this chapter.

In chapter 3 the type specification language $\mathcal{TFS}$ is introduced. How types, words and grammar rules can be specified is explained here. Also, an algorithm for the computation of the least upper bound relation of a type lattice will be given. At the end of this chapter an example specification will be used to explain the advantages of typing for writing a grammar.

Chapter 4 deals with the way that semantics can be specified with the $\mathcal{TFS}$ language. Semantics can be expressed using a quasi-logical language called QLF. Some implementation issues like the way unbound variables are represented are also described here.

In chapter 5 the head-corner parsing algorithm will be described. This algorithm will be applied to typed feature structures. Again, the most important implementation issues will be discussed. A short description will be given of what classes[2] have been defined and how they are related.

In chapter 6 different approaches and implementations of unification will be discussed. This chapter is concluded with a section on future work. The parts of the implementation that still need to be done are listed here. Also, suggestions are given how the $\mathcal{TFS}$ system could be extended at at theoretical level.

Finally, in chapter 7 some concluding remarks will be made about the most important aspects of types, feature structures and head-corner parsing with respect to the $\mathcal{TFS}$ system.

[2]The term ‘class’ is a C++ concept, used to describe a collection of similar objects.

Chapter 2

Types and Feature Structures

2.1 Types

Types can be used to categorize linguistic and domain entities. In addition to that the relations between entities can be defined using an inheritance hierarchy. For types we follow the definition of Carpenter (1992). Types can be ordered using the subsumption relation. We write $s \sqsubseteq t$ for two types s and t if s *subsumes* t, that is, s is more general than t. In that case s is called a *supertype* of t, or inversely, t is a *subtype* of s. With the subsumption relation the set of types form a lattice (see figure 2.1).

The type that subsumes all other types ("the most general type") is called bottom and is denoted by $\perp$. The most general subtype for a pair of types s and t is called the *least upper bound* and is written as $s \sqcup t$. For instance, in figure 2.1 we have $s \sqcup t = x$ and $v \sqcup w = \top$. In the latter case the two types contain conflicting information and are hence inconsistent.

There are two ways to specify a type lattice. The first way is to express each new type in terms of its subtypes. This can be seen as a set-theoretical approach: each type is a set of possible values and a new type can be constructed by taking the union of other (possibly infinite) sets. For instance, the type fruit could be defined as

$$\mathsf{fruit} \quad := \quad \mathsf{apples} \cup \mathsf{bananas},$$

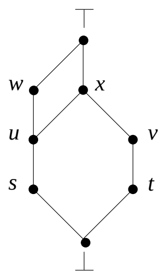


Figure 2.1: **A type lattice.** The lines represent the subsumption relation. More specific types are placed higher in the lattice. The top element '$\top$' is used to denote inconsistency between types.

where apples is a set of all apples and bananas is a set of all bananas. The bottom element $\perp$ is then the set of all entities within the domain and the top element $\top$ is the empty set $\emptyset$.

The other way to specify a type lattice is to express each type in terms of its supertypes. In this context the term ‘inheritance’ is often used; a type inherits information from its supertypes. The disadvantage of specifying types this way is that inconsistencies in the lattice are easily introduced. If a type is specified to have two supertypes that contain conflicting information, that type would be inconsistent[1]. With the set-theoretical approach this cannot happen. However, from the grammar writer’s point of view it is often easier to first introduce general concepts and later differentiate them into more specific types than to start with the most specific types and generalize over them to construct new types. Hence, in the specification language described in chapter 3 the second approach is followed.

2.2 Feature Structures

Feature structures provide a convenient way to keep track of complex relations. During parsing constraints can be checked with feature structures, and after parsing the meaning of the language utterance can (hopefully) be extracted from them. The structure of our feature structures is similar to the more traditional form of feature structures as used in the PATR-II system Shieber (1986) and those defined by Rounds and Kasper (1986).

Typed feature structures are defined as rooted DAGs (directed acyclic graphs), with labeled edges *and nodes*. More formally, we can define a typed feature structure tfs as a 2-tuple $\langle t, features\rangle$, where $t \in Types$, the set of all types, and $features$ is a (possibly empty) set of features. A feature is defined as a feature name / feature value pair. A feature value is again a typed feature structure. At first glance the labels on the nodes seem to be the only difference with the traditional feature structures, but there is more to typing than that. Every type has a fixed set of features. Such a feature value type can be seen as the appropriate value for a particular feature. It should be equal to the greatest lower bound (the most specific supertype) of all the possible values for that feature. So a typed feature structure is actually an instantiation of a type. Types are used as a sort of templates. By typing feature structures we restrict the number of ‘allowed’ (or appropriate) feature structures. Putting these restrictions on feature structures should fasten the parsing process; at an earlier stage it can be decided if a certain parse should fail.

Another advantage of typing feature structures is that it is no longer necessary to make a distinction between nodes *with* features (‘complex nodes’), nodes *without* features (‘constant nodes’) and nodes with type $\perp$ (‘variable nodes’) as is often done with traditional feature structures. In a consistent definition of the type lattice the least upper bound of a complex and a constant node should always be $\top$ (unless that constant node represents an abstract, under specified piece of information), so that two such nodes can never be unified.

2.3 Unification

The basic operation on feature structures is unification. New feature structures are created by unifying two existing ones. In figure 2.2 two basic examples show what unification means. In these feature graphs *agreement*, *singular*, *plural* and *third* are names of types, and *number* and *person* are names of features.

The unification of two feature structures fails if:

- the least upper bound of the two root nodes is $\top$, or

[1]The specification of type information is treated in chapter 3.

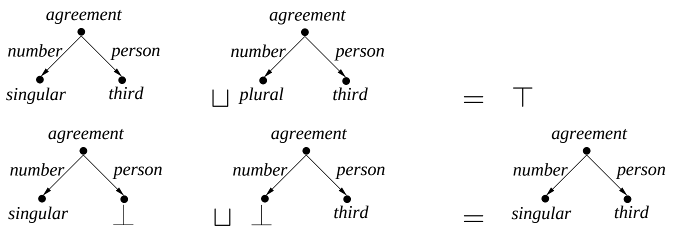


Figure 2.2: **Basic unifications.** In the first case unification fails, i.e. the feature structures contain conflicting information. The second case is self-evident.

n	parses	HCP	TOM
4	5	100	542
5	14	249	1,827
6	42	662	6,268
7	132	1,897	22,187
8	429	5,799	80,685

Table 2.1: **An untyped version of the unification algorithm compared with Tomabechi's algorithm.**

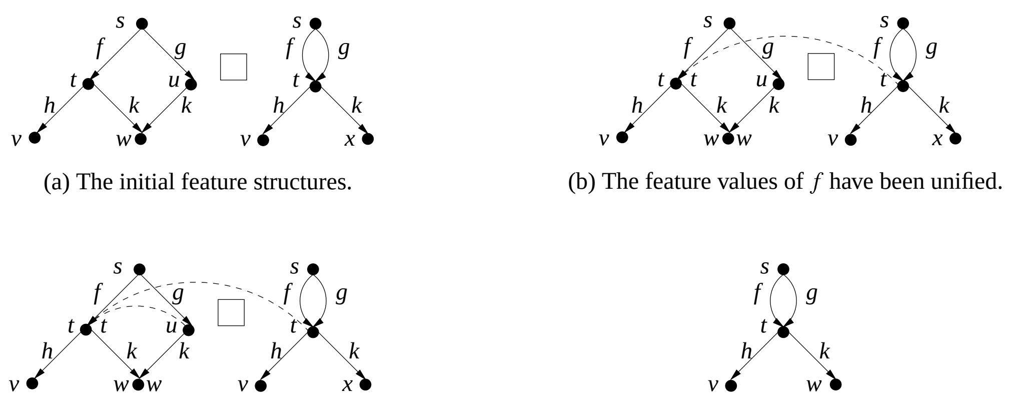
- the unification of the feature values of two features with the same name fails.

Unification is a costly operation in unification-based parse systems, because it involves a lot of copying of feature structures. In many implementations of parsing systems it takes more than 80% of the total parse time. Several algorithms have been devised to do unification efficiently Tomabechi (1991); Wroblewski (1987); Sikkel (1993). The efficiency of unification can be increased by minimizing the amount of copying in cases that unification fails, while on the other hand the overhead costs to do this should be as small as possible. Till now Tomabechi's algorithm seemed to be the fastest. With this algorithm the copying of (partial) feature structures is delayed until it has been established that unification can succeed. But Tomabechi already suggests in a footnote that the algorithm can be improved by sharing substructures. This idea has been worked out into an algorithm Veldhuijzen van Zanten and Op den Akker (1994). The copy algorithm has been implemented in a predecessor of the current parser and has proven to be very effective in experiments. Table 2.1 shows the results of one of these experiments. The algorithms were tested with the 'sentences' $Jan^n, n = 4 \ldots 8$, using the following grammar: $S \rightarrow S\ S \mid Jan$ (so the sentences are extremely ambiguous). The first column stands for the sentence length, the second column shows the number of parses and the third and fourth column show the number of nodes created during unification for Veldhuijzen van Zanten's and Tomabechi's unification algorithm.

By introducing the types, the overhead increases slightly; for every two nodes that are to be unified the least upper bound has to be looked up in a table. But still, we expect an improvement in the performance.

Before the algorithm is described in more detail, it is necessary to define the general properties of a node in a feature structure. These properties ('members' in the object-oriented programming terminology, or 'fields' in a traditional record implementation) can be divided in two kinds: (1) properties that describe the structure of a feature structure and (2) bookkeeping properties, that are used to store intermediate results. For the first kind only

- a type id, that uniquely defines the feature names and the appropriate values for the corresponding features, and



(a) The initial feature structures.

(b) The feature values of f have been unified.

(c) The feature values of g have been unified.

(d) The resulting unifact.

Figure 2.3: **An example unification.** A type id on the right of a node stands for the auxiliary type of that node. The dashed arrows indicate forward links. If a node has a forward link to another node, the feature structures starting in these nodes are unifiable. It is assumed that $x \sqsubseteq w$ and $u \sqsubseteq t$.

- a set of features, where each feature consists of a name and a value (i.e. an instance of a certain type)

are needed. To handle the bookkeeping we need the following properties:

- a forward pointer: a pointer to another node, of which the unification algorithm has established that unification with this node is possible,
- an auxiliary type id: the type id of the corresponding node in the unifact (the result of unification),
- auxiliary features: features of the node that is unified with the current node, that do not occur in the set of features of the current node,
- an unifact pointer: a pointer to the unifact that is constructed by the copy algorithm,
- a forward mark and an unifact mark: markers containing a generation number indicating whether the forward and unifact pointer can be used in the current unification process.

Unification is executed as a two-step operation: first, it is checked whether unification is possible, that is, the two feature structures to be unified contain no conflicting information. Second, the unifact is constructed using the bookkeeping information left by the first step.

Though the algorithm is implemented with object-oriented techniques in C++, the algorithm is displayed in conventional pseudo-Pascal code to enhance the readability for those not familiar with these techniques. Step one, the check if unification is possible, is shown in figure 2.4. Some auxiliary procedures for the unification algorithm are displayed in figure 2.5. Finally, figure 2.6 shows how step two, the creation of the unifact, is implemented.

The example in figure 2.3 shows how the unification algorithm works. First the generation counter is increased to make any old intermediate results obsolete. The procedure *unifiable* is then called with the two s nodes as arguments. Now subsequent calls are made to *unifiable* for each feature value pair of the two s nodes. First the feature structures starting in the two t nodes are unified. They differ only in the feature value for the k feature. It is assumed that $x \sqsubseteq w$, so that the two nodes are unifiable. The auxiliary type will then be w. Now the two t nodes are unifiable and a forward link from one t node to the other one can be made (see figure 2.3b). Now the feature values for the g feature can be unified. Because of the forward link

```
proc unify(tfs_1, tfs_2)
  nextGeneration();
  if unifiable(tfs_1, tfs_2)
    then
        return copyUnifact(tfs_1)
    else
        return ⊤
  fi
.

proc unifiable(tfs_1, tfs_2)
  tfs_1 := dereference(tfs_1);
  tfs_2 := dereference(tfs_2);
  if (tfs_1 = tfs_2) then return true fi;
  tfs_1→auxType := lub(tfs_1, tfs_2);
  if (tfs_1→auxType = ⊤)
    then
        return false
  fi;
  stillUnifies := true;
  while stillUnifies do
        foreach f ∈ tfs_2→features
          if (f ∈ tfs_1→features)
            then
                stillUnifies := unifiable(tfs_1→f, tfs_2→f)
            else
                add feature tfs_2→f to tfs_1→auxFeatures
          fi
        od
  od;
  if (stillUnifies = true)
    then
        forward(tfs_2, tfs_1);
        return true
    else
        return false
  fi
.
```

Figure 2.4: **The unification algorithm.** An improved version of Tomabechi's quasi-destructive unification algorithm.

```
proc nextGeneration ()
   $currentGeneration := currentGeneration + 1$
.

proc dereference ($tfs_1$)
   if $tfs \rightarrow forwardMark = currentGeneration \wedge tfs \rightarrow forward \neq$ nil
     then
         return $tfs \rightarrow forward$
      else
         return $tfs$
   fi
.

proc forward ($tfs_1, tfs_2$)
   $tfs_1 \rightarrow forward := tfs_2$;
   $tfs_1 \rightarrow forwardMark := currentGeneration$;
.
```

Figure 2.5: **Auxiliary procedures for the unification algorithm**

of the previous step, the feature values of the f and g feature of the left feature structure are now unified. So for unification to succeed we have to assume that $u \sqsubseteq t$. Under this assumption a forward link from the u node to the left t node can be made and the initial call to *unifiable* returns **true** (see figure 2.3c). The final step is then a call to *copyUnifact* to create the unifact from the intermediate results (see figure 2.3d). Note that this unification is non-destructive; both operands remain intact.

The procedure *lub* (called from *unifiable*) determines the least upper bound of two types. This least upper bound can be looked up in the type lattice as will be explained in chapter 3. If two types have $\top$ as least upper bound, they are not unifiable and it is not necessary to look at the feature values of the types.

The procedure *copyUnifact* (figure 2.6) only creates a new node if it is not possible to share that node with an existing feature structure. A new node is created by *createTFS*, which makes a node of the right type and initializes the features with appropriate values. The variable *needToCopy* is used to check whether a new node has to be created. Only if one of the following two situations occurs it is necessary to make a new node:

- the unifact has more features than the feature structure from which it constructed, that is, the number of auxiliary features is greater than 0,
- the unifact differs from the feature structure from which it constructed in at least one feature value.

Otherwise the node will be shared with the current node of the typed feature structure from which the unifact is constructed.

```
proc copyUnifact($tfs$)
  $tfs$ := dereference($tfs$);
  if ($tfs \rightarrow unifactMark = currentGeneration$)
    then
        return $tfs \rightarrow unifact$
  fi;
  $needToCopy := (\#tfs \rightarrow auxFeatures > 0)$;
  $i := 0$;
  foreach $f \in tfs \rightarrow (features \cup auxFeatures)$ do
    $copies[i]$ := copyUnifact($f$);
    $needToCopy := needToCopy \lor (copies[i] \neq f)$;
    $i := i + 1$
  od
  if $needToCopy$
    then
        if $tfs \rightarrow unifact = $ nil
          then
              $tfs \rightarrow unifact$ := createTFS($tfs \rightarrow auxType$)
        fi;
        for $j := 0 \ldots i - 1$ do
            add feature $copies[j]$ to $tfs \rightarrow unifact$
        od;
        $tfs \rightarrow unifactMark := currentGeneration$;
        return $tfs \rightarrow unifact$
    else
        return $tfs$
  fi
.
```

Figure 2.6: **The copy algorithm.** This procedure generates the unifact after a successful call to *unifiable*(tfs_1, tfs_2).

Chapter 3

Specification

To specify a language it is necessary to have a metalanguage. Almost always the usage of a specification language is limited to only one grammar formalism. This is not necessarily a drawback, as such a specification language can be better tailored towards the peculiarities of the formalism. For example, ALE Carpenter and Penn (1994) is a very powerful (type) specification language for the domain of unification-based grammar formalisms. But apart from expressiveness of the specification language, the ease with which the intended information about a language can be encoded is also important. An example of a language that combines expressiveness with ease of use is the Core Language Engine Alshawi (1992). Unfortunately the Core Language Engine (CLE) does not support typing. Within my thesis' subject a type specification language has been developed that can be positioned somewhere between ALE and CLE. This specification language (called $\mathcal{TFS}$) can be used to specify a type lattice, a lexicon and a unification grammar for a head-corner parser. The notation is loosely based on CLE, though far less extensive. For instance, the usage of lambda calculus is not supported.

The specification is compiled in two steps. In the first step the specification is read and stored in a symbol table. If no error has occured while reading the specification, the contents of the symbol table is mapped to C++ code. During the first step also the type lattice and the transitive and reflexive closure of the head-corner relation[1] are determined Yellin (1988). In the second compilation step the C++ code is compiled and then linked with grammar-independent code: code that implements the parsing algorithm and code used for the user interface and the 'linguistic workbench' (see figure 3.1). The 'linguistic workbench' is an environment for the grammar writer to test the specification. This workbench is not yet implemented, but in chapter 6 some suggestions are given what could be done with such a workbench. Note that since $\mathcal{TFS}$ is a specification language, the generated C++ code contains only declarations, so no procedure calls have to be executed at run-time to read or process the specification.

In the following sections the syntax and semantics of $\mathcal{TFS}$ are treated as well as the most important aspects of the compilation of the specification, such as the determination of the least upper bound relation. A complete description of the syntax of $\mathcal{TFS}$ can be found in appendix A.

[1]The head-corner relation is explained in section 5.2.

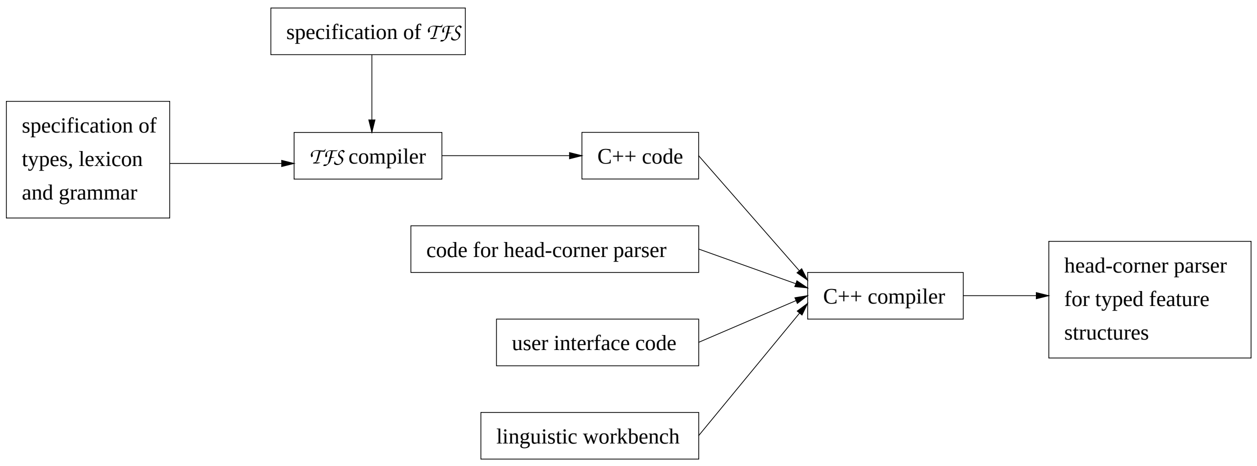

specification of $\mathcal{TFS}$
specification of types, lexicon and grammar
$\mathcal{TFS}$ compiler
C++ code
code for head-corner parser
user interface code
linguistic workbench
C++ compiler
head-corner parser for typed feature structures

Figure 3.1: **The $\mathcal{TFS}$ system.** The design of the head-corner parser for typed feature structures.

3.1 Specification of Types

In $\mathcal{TFS}$ types are specified in a certain order. There are two factors that affect the order of the type specification: (1) types can be used as supertypes in the specification of another type and (2) types should be specified before they can be used. This means that first general types are specified, followed by more specific types. A nice side-effect of specifying types this way is that no cyclic definitions (like 'A is a B and B is an A') can be created. To keep track of the least upper bound relation a lower triangle matrix with the least upper bound of every two types is updated after each addition of a type to the symbol table. The type lattice is restricted to be a *finite bounded complete partial order* Carpenter (1992). To keep the type lattice consistent the following two conditions should hold for every triplet $\langle a,b,c\rangle$ of types.

$$a \sqcup b = b \ \wedge \ b \sqcup c = c \quad \Rightarrow \quad a \sqcup c = c \tag{3.1}$$

$$a \sqcup b = a \ \wedge \ a \sqcup c = a \quad \Rightarrow \quad b \sqcup c \sqsubseteq a \tag{3.2}$$

Before we treat an algorithm that ensures that these conditions hold after every type that is added, we will give an example showing what the desired effect on the least upper bound matrix would be. The same lattice as in chapter 2, page 5 is used. In figure 3.2 the least upper bound matrix is shown just before and after the addition of type x. Initially, all the entries in the row for type x are made equal to $\top$, except the entries for its supertypes, which are made equal to x. For the computation of the new least upper bound matrix it is not necessary to check all possible triplets $\langle a,b,c\rangle$ of types to see if the lattice is still consistent. First of all, only triplets containing x have to checked, assuming that the least upper bound matrix was consistent before the addition of type x. Second, only those entries have to checked which are not equal to $\top$. And finally, it is not necessary to check triplets in which $\top$ and $\bot$ occur as arguments for the $\sqcup$ operator. This means that the first two columns of the least upper bound matrix are never inspected by the algorithm. We can apply equation 3.1 on $x \sqcup v$ and $v \sqcup t$, and $x \sqcup u$ and $u \sqcup s$. This results in new values for $x \sqcup t$ and $x \sqcup s$, respectively. The other encircled entries are found by applications of equation 3.2. In general, equation 3.1 can be applied to an entry in the last row not equal to $\top$ and suitable entries of the other rows. After no new values can be generated with this rule, equation 3.2 should be applied to all pairs of entries in the last row with values equal to the new type. The following algorithm shows how the new least upper bound can be computed, given that the matrix is correct for the first $n-1$ types and given the initial value for row n.

The implementation of rule 1 is quite straightforward: $lub[n,i]$ and $lub[i,j]$ correspond to $b \sqcup c$ and $a \sqcup b$ in equation 3.1, respectively. In the second part of the algorithm equation 3.2 is verified. We have $lub[n,i]$

$$\begin{array}{c|ccccccc}
\bot & \bot & & & & & & \\
\top & \top & \top & & & & & \\
s & s & \top & s & & & & \\
t & t & \top & \top & t & & & \\
u & u & \top & u & \top & u & & \\
v & v & \top & \top & v & \top & v & \\
w & w & \top & w & \top & w & \top & w \\
\hline
 & \bot & \top & s & t & u & v & w
\end{array}$$

(a) Before the specification of type x.

$$\begin{array}{c|cccccccc}
\bot & \bot & & & & & & & \\
\top & \top & \top & & & & & & \\
s & s & \top & s & & & & & \\
t & t & \top & \textcircled{x} & t & & & & \\
u & u & \top & u & \textcircled{x} & u & & & \\
v & v & \top & \textcircled{x} & v & \textcircled{x} & v & & \\
w & w & \top & w & \top & w & \top & w & \\
x & x & \top & \textcircled{x} & \textcircled{x} & x & x & \top & x \\
\hline
 & \bot & \top & s & t & u & v & w & x
\end{array}$$

(b) After the specification of type x.

Figure 3.2: **Computation of the least upper bound matrix.** The entries that follow from application of equations 3.1 and 3.2 are accentuated with circles.

```
proc ComputeNewLub ()
  $lub[n,\bot] := n$;                                                        $(x \sqcup \bot = x)$
  $lub[n,n] := n$;                                                           $(x \sqcup x = x)$
  for $i := n-1$ to 4 do
      if $(lub[n,i] = n)$ then
                      for $j := i-1$ to 3 do
                          if $(lub[i,j] = i)$ then $lub[n,j] := n$ fi                   (rule 1)
                      od
      fi
  od;
  for $i := 4$ to $n$ do
      if $(lub[n,i] = n)$ then
                      for $j := 3$ to $i-1$ do
                          if $(lub[n,j] = n) \wedge (lub[i,j] = \top)$ then $lub[i,j] := n$ fi    (rule 2)
                      od
      fi
  od
.
```

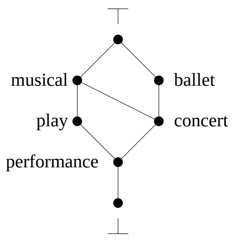
Figure 3.3: **Algorithm to compute the new least upper bound matrix**

corresponding to $a \sqcup b$ and $lub[n,j]$ corresponding to $a \sqcup c$ (or vice versa). Note that we can only set $lub[i,j]$ to n if its value was equal to $\top$. Otherwise the least upper bound of i and j would already subsume n, since n is the most specific type.

The computation of the new least upper bound matrix can be done in quadratic time. So, the computation of the least upper bound matrix starting from scratch can be done in cubic time.

A type specification consists of four parts: a type id for the type to be specified, a list of supertypes, a list of constraints and a formula expressing the semantics for the new type. The following example shows how a type lattice can be specified.

```
TYPE(performance; bottom; <constraints>; <QLF>)
TYPE(play; performance; <constraints>; <QLF>)
TYPE(concert; performance; <constraints>; <QLF>)
TYPE(musical; play, concert; <constraints>; <QLF>)
TYPE(ballet; concert; <constraints>; <QLF>)
```



For each type `<constraints>` should be replaced with PATR-II-like path equations. Path equations can have the following two forms:

$$\begin{aligned}\langle f_1\ f_2 \cdots f_n\rangle &= \langle g_1\ g_2 \cdots g_n\rangle \\ \langle f_1\ f_2 \cdots f_n\rangle &:= \mathit{node}\end{aligned}$$

The first form says that two paths (i.e., sequences of features) in a feature structure should be joined. With the second form the type of a node in a feature structure can be specified. The right-hand side should be equal to a type identifier or a constant (a string or a number). During the parsing of the specification a minimal satisfying feature structure is constructed for each path equation. So all the nodes in a feature structure have type $\perp$, unless specified otherwise. Subsequent path equations are unified to generate a new feature structure satisfying both constraints. Finally the resulting feature structure for all the constraints is unified with constraints inherited from the supertypes.

`<QLF>` should be replaced with the semantics in a quasi-logical formula. How semantics can be expressed using quasi-logical formulas is explained in chapter 4. The idea is that the constraints are only necessary *during* parsing and the semantics are passed on to be used *after* parsing.

A type inherits information from its supertypes in the following way: the constraints for the type are *unified* with the constraints of the supertypes, and the quasi-logical formula for the type is *concatenated* with a list of quasi-logical formulas for the supertypes. The QLF expressions are not evaluated, but are just translated to internal representations.

3.2 Specification of Words

Lexical entries can be specified in the same way as types. This is not surprising, since words can also be seen as types. There is, however, one restriction: a word cannot be used as supertype in the specification of other types (including words). Ambiguous words can be specified by simply defining multiple entries for the same lexeme:

```
LEX("flies", verb, <constraints>, <QLF>)
LEX("flies", noun, <constraints>, <QLF>)
```

The type identifier that is given to a word is the type identifier of the first type of the list of supertypes. In the previous example the words only had one supertype, so there is a feature structure for “flies” with a `verb` type identifier and a feature structure with a `noun` type identifier. In the lexicon every word is associated with a list of feature structures, one for each meaning.

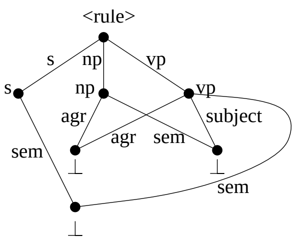
After all the words have been read, the lexicon can be generated. The lexicon is organized as a binary search tree, allowing fast access to each word. To be more precise, the search time for a word increases logarithmically with the size of the lexicon.

3.3 Specification of Grammar Rules

Grammar rules are internally also represented as typed feature structures. They have a special rule type identifier. It is not necessary to represent rules as feature structures, but such structures happen to be a very practical representation mechanism for grammar rules. Hence, in the C++ implementation the class *Rule* (the class for grammar rules) is a derived class from the class *FeatureStruc* (the class for feature structures).

In general, a grammar rule specification looks like this:

```
RULE(s --> np *vp*,
    <np agr> = <vp agr>,
    <vp subject> = <np sem>,
    <s sem> = <vp sem> )
```



The asterisks mark the head[2] in the grammar rule. Next to the specification the resulting typed feature structure is shown. Note that grammar symbols are in fact types.

Using a feature structure as the data structure for grammar rules brings along two problems, which fortunately can be solved easily. The first problem is that with feature structures the order of features is irrelevant (since the features form a set), but the order of grammar symbols in a rule is very important. This problem can be solved by implementing the set of features as an ordered list, so that the original order of the grammar symbols remains intact. In the current implementation the set of features is implemented as simple as possible: it is just an (dynamically allocated) array. Another solution would be to number the features of the root node of a rule. So, the feature structure for a rule $S \rightarrow NP\ VP$ would then have three features with names f_0, f_1 and f_2. To the values for these features are the feature structures for S, NP and VP.

The second problem occurs with grammar rules were the same symbol occurs on the left-hand and right-hand side, like in $VP \rightarrow *VP*\ PP$. The two *VP*'s have the same type symbol, but refer to different parts of a sentence. If we use the same representation as in the previous example, two features could be created with the same name, or—even worse—only one feature could be created for both *VP*'s. This problem can be solved by using indices: $VP1 \rightarrow *VP2*\ PP$. The $\mathcal{TFS}$ parser detects that indices are used (*VP*1 is not a type name) and creates a feature structure with features vp1, vp2 and pp, respectively. The nodes attached to these features have the correct types: vp, vp and np, respectively.

The next example shows how typing can make some grammar rules with only one symbol on the right-hand side superfluous.

```
TYPE(perfphrase; nounphrase; ; )
RULE(nounphrase --> *perfphrase*;
  <nounphrase kind> = <perfphrase kind>,
  <nounphrase sem> = <perfphrase sem>;
)
```

Both the type and rule specify that a performance phrase is a kind of noun phrase. So with the type specification the rule becomes superfluous.

3.4 Comparison of a Typed and Untyped Grammar

In this section the usage of the specification language $\mathcal{TFS}$ is illustrated by means of a small grammar. The differences between this specification and a specification that uses no typing will be explained.

[2] See also section 5.1.

With the small grammar shown below only the sentence “Jan lives in Amsterdam” can be parsed, due to the small size of the lexicon. The full specification of the type lattice is not shown, but most of the type specifications are straightforward: just like `s` is specified to be a subtype of `constituent`, `np` and `vp` are specified as subtypes of `constituent`. For the complete specification, see appendix D.2.

```
// the type lattice:
// (most type specifications for linguistic entities are omitted)
TYPE(constituent;;
    <agr> := agreementtype; )
TYPE(thirdsing;constituent;
    <agr num> := singular,
    <agr pers> := third; )
TYPE(s;constituent;;)
TYPE(person;noun;;
    EXISTS Person (personname(Person,Name)))

// lexical entries:
LEX("Jan"; propernoun, person, thirdsing;
    <unbound name> := "Jan"; )
LEX("lives"; transitive, thirdsing;
    <agr> := location;
    livesin(Subject,Object))
LEX("in"; prep; <agr> := inlocation;)
LEX("Amsterdam"; propernoun, thirdsing;
    <agr> := location;
    EXISTS Location (locationname(Location,"Amsterdam")))

// grammar rules:
RULE(s --> np *vp*;                     // Jan lives in Amsterdam
    <np agr> = <vp agr>,
    <s agr> = <vp agr>,
    <vp unbound subject> = <np qlf>,
    <s qlf> = <vp qlf>)
RULE(vp --> *verb* pp;                  // lives in Amsterdam
    <verb agr> = <pp agr>,
    <vp agr> = <verb agr>,
    <vp unbound subject> = <verb unbound subject>,
    <vp unbound object> = <verb unbound object>,
    <vp unbound object> = <pp qlf>,
    <vp qlf> = <verb qlf>)
RULE(pp --> *prep* np;                  // in Amsterdam
    <pp agr> = <prep agr>,
    <pp agr> = <np agr>,
    <pp qlf> = <np qlf>)
```

In this example we see how multiple inheritance can be used to give short specifications for words. The word “Jan” can be characterized by giving it three supertypes. The only extra thing that has to be done is instantiate the unbound variable in the QLF expression that is inherited from the `person` type. See also section 4.2 for the implementation of unbound variables. As said before, in the case that a word has more than one supertype, the word will get the type identifier of the first type. So, “Jan” is specified as a sort of proper noun (which is a subtype of the noun type). In an untyped grammar all this information should be written out, making the specification far less readable. But what is more important, is that this has to be done for every word that has about the same properties as “Jan”. A lot of redundancy is introduced and it is not unlikely (especially for larger grammars) that specifications for similar words are not always specified in the same way.

The specification of grammar rules would not really be that different for the untyped case. Paths are equated to check constraint and to pass on information to other constituents.

For small grammars the advantages of using typing count for little compared to the extra ‘overhead’ that is introduced by the type lattice. In the example used above the type specification was as long as the lexicon and grammar specification together. But for larger grammars typing can actually shorten the total length of the specification.

Chapter 4

Expressing Semantics

The meaning of language utterances is most conveniently described in some kind of logical language[1]. First order predicate calculus (FOPC) and lambda calculus are the most well-known logical systems. Actual implementations of natural language systems are often built on top of a Prolog compiler or a Lisp interpreter. As a result the notation for semantic expressions is often very Prolog- or Lisp-like. With the $\mathcal{TFS}$ language we have tried to abstract from such implementation details and devise a language which is as close as possible to a more conventional notation of logic. So, instead of the prefix/list notation, the infix notation is used. The starting point of the quasi-logical language QLF is FOPC. Logical operators like the quantors ($\forall, \exists, \exists!$) and connectives ($\neg, \vee, \wedge, \Rightarrow$) are replaced by keywords (`FORALL`, `EXISTS`, `EXISTS1` and `NOT`, `OR`, `AND`, `IMPL`, respectively).

4.1 Extensions

QLF is extended to make it more useful in the context of natural language interfaces. With natural language interfaces a simple yes ('true') or no ('false') is often not the desired information. To say something about the mood of an utterance four mood operators are introduced:

`DECLARATIVE` This is the mood of a normal sentence that is not a question or an command. Within a natural language interface it is most likely used for an answer to an question.

`WHQUESTION` The mood used for questions starting with 'Where…', 'What…', 'How…', etc.

`YNQUESTION` The mood used for questions that can be answered with 'yes' or 'no'. In most cases, however, these questions should not be answered with just 'yes' or 'no'. Often somewhat more cooperation is desired from a natural language interface. Consider for instance the following question:

"Are there still tickets available for Herman Finkers' performances?"

[1]Though it is questionable whether logic is as language independent as often is assumed and whether logic can describe the true and complete meaning of an utterance.

A simple question 'yes' would not suffice in such a case. Information should be given about dates, time and price of available tickets.

`IMPERATIVE` This mood can be used for commands like

"Give me more information about this performance."

These mood operators can only be used at the top level; they can not be nested in other formulas.

To say something about (the cardinality) of sets, two new operators are used:

`SET` Used as in `SET X (p(X))`, meant to denote the set of entities that satisfy predicate `p`. This can be used for phrases like "The sisters of Jan". The corresponding QLF expression could then be:

```
SET Y (EXISTS X (name(X,"Jan") AND sister(Y,X)))
```

`COUNT` Can be used in the same way: `COUNT X (p(X))`, which stands for the number of entities that satisfy predicate `p`. So with the `COUNT` operator the cardinality of sets can be denoted. Sentences like "Does Jan have three sisters?" can then be written in QLF as:

```
YNQUESTION(COUNT Y (EXISTS X (name(X,"Jan") AND sister(Y,X))) = 3)
```

Note that I have used the '=' symbol to test for equality of two terms. Other supported term relations are: $<, >, <=, >=$ and $! =$. The last one can be used to express inequality.

4.2 Scope of Variables

The scope of a variable that is bound by a quantor is limited to the formula between parentheses that follows it. Note that QLF differs here slightly from FOPC. In FOPC it is not necessary to explicitly indicate the scope of a bound variable. But in QLF the scope has to be indicated with parentheses. So the FOPC formula $\forall x \mathrm{P}(x) \wedge \mathrm{Q}(x,y)$ should be written as `FORALL X (p(X) AND q(X,Y))`. Note also that predicate names are written in lowercase and variable names in uppercase.

The scope of an unbound variable is the entire formula in which it occurs. Unbound variables that occur in QLF expressions of types can be bound in the specification of a subtype, a word or a grammar rule. The following example illustrates this:

```
TYPE(person; ; ; EXISTS X (personname(X,Y)))
LEX("Jan"; person; <unbound y> := "Jan"; )
```

The typed feature structure that is created as a result of the first line is shown in figure 4.1.

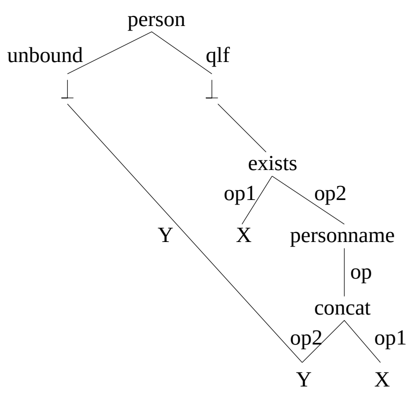


Figure 4.1: **Unbound variables.** Unbound variables are implemented in such a way that they can be bound later.

4.3 Future Extensions of QLF

The syntax of QLF can easily be extended. If one wants to add another operator to QLF, the following things have to be done:

- add a keyword to the $\mathcal{TFS}$ scanner (currently defined in the file `TFSScan.l`),
- add a token and a production rule for this token to the $\mathcal{TFS}$ parser (currently defined in the file `TFSParse.y`), and
- create a basic type for the new operator, which will give the operator a type identifier and a feature for every argument of the operator. It is also possible to define the new operator as a subtype of other basic types. The basic types are defined in the files `TFSBasicTypes.h` and `TFSBasicTypes.c`. Using the code for other basic types, it should not be too hard to define a new operator.

One possible extension of QLF could be the addition of the set membership relation. This might be useful for sentences like “Is Marie one of the sisters of Jan?”. Yet another, more complex extension could be the addition of some kind of lambda calculus. Most semantics oriented natural language systems nowadays use lambda calculus Alshawi (1992); Groenink (1992).

Chapter 5

The Parser

In this chapter the head-corner parsing algorithm will be discussed that is used to parse sentences. The first two sections are based on the chapters 10 and 11 of Sikkel (1993). Parts of these chapters can also be found in Sikkel and Op den Akker (1993). In section 5.3 an extension of the formalism with typed feature structures is introduced. Finally, in section 5.4 the most important implementation aspects of the parser are explained.

5.1 Context-Free Head Grammars

The general idea behind head-corner parsing is that the most important words and nonterminals should be recognized first. For instance, if we recognize the main verb of the sentence first, we would already know what the case (number and person) should be for the subject. Also information about whether there could/should be an object can often be derived from the main verb.

With context-free head grammars it is possible to enforce derivations, where the most important (non)terminals are derived first. Context-free head grammars can be formally defined as a 5-tuple $\langle N, \Sigma, P, S, h \rangle$, where the first four parts have the same meaning as in ordinary context-free grammars. The fifth part, h, is a function that assigns a natural number to each production rule in P. Let p be an element of P and let $|p|$ denote the length of the right-hand side of p. Then h is constrained to the following values:

- $h(p) = 0$ for $|p| = 0$,
- $1 \leq h(p) \leq |p|$ for $|p| > 0$.

The *head* of a production rule p is now defined as the $h(p)$-th symbol on the right-hand side; empty productions have head ε. Usually the function h is only defined implicitly by underlining the head of each production rule. So, if we take for example the grammar from chapter 1, the assignment of the heads might be defined as:

$$\begin{array}{rcl} S & \rightarrow & NP\ \underline{VP}, \\ VP & \rightarrow & \underline{{}^*v}\ NP, \\ NP & \rightarrow & {}^*det\ \underline{{}^*n}. \end{array}$$

Given a certain grammar there are many different definitions of h possible. The actual allocation of heads is done by the grammar writer. The definition of h could be motivated by linguistic theories or by the effect on the efficiency of the parser.

With the function h it is possible to define the head-corner relation $>_h$ on $N \times (N \cup \Sigma \cup \{\varepsilon\})$:

$$A >_h B \text{ if there is production } p = A \rightarrow \alpha \in P \text{ with } B \text{ the head of } p.$$

The transitive and reflexive closure of $>_h$ is denoted $>_h^*$.

5.2 Head-Corner Parsing

A head-corner parser belongs to the class of chart parsers. The notion of a chart parser was introduced by Kay (1980). Before the chart parsing algorithm is explained, some notational conventions will be given that will be used throughout this chapter. We write $A, B, \ldots$ for nonterminal symbols; $a, b, \ldots$ for terminal symbols; $X, Y, \ldots$ for arbitrary symbols; and $\alpha, \beta, \ldots$ for arbitrary strings of symbols. Positions in the string $a_1 \ldots a_n$ are denoted by $i, j, k, \ldots$ and l, r.

A *parsing system* for some grammar G and string $a_1 \ldots a_n$ is a triple $\mathbb{P} = \langle I, H, D \rangle$, where I is a set of items, H an initial set of items (also called *hypotheses*) and D a set of deduction steps. The deduction steps can be used to derive new items from already known items. The hypotheses are defined as

$$H \quad = \quad \{[a_1, 0, 1], \ldots, [a_n, n-1, n], [\$, n, n+1]\},$$

where \$ is the end-of-sentence token. The deduction steps in D have the following form:

$$\eta_1, \ldots, \eta_k \quad \vdash \quad \xi.$$

The items $\eta_1, \ldots, \eta_k$ are called the antecedents and are taken from $H \cup I$. The item $\xi \in I$ is called the consequent.

A chart parser uses two data structures to derive items: the *chart* and the *agenda*. If a sentence is to be parsed by a chart parser, the parsers places the initial items on the chart and places a goal on the agenda. The initial goal is the item that predicts a recognized sentence between positions 0 and n. A sentence is now parsed in the following way: An item is taken from the agenda and moved to the chart. This item is now the current item. The current item is combined with the other items on the chart to create new items. The new items that do not already occur on the chart or agenda are then added to the agenda. This process continues as long as the agenda is not empty. Figure 5.1 shows a general schema for a chart parser.

The head-corner parser that is described here corresponds to the **sHC** parsing schema as described in chapter 11 of Sikkel (1993). This schema can parse sentences from arbitrary context-free head grammars in cubic time.

A head-corner chart parser uses different kinds of items. They can be divided into the following categories:

$[l, r, A]$ **— predict items or goals** These items indicate what kind of constituents is looked for at a certain place in the sentence. An item $[l, r, A]$ is recognized if constituent A must be looked for somewhere between position l and r.

$[X, i, j]$ **— CYK items** These items are used to denote terminal items $[a, i, j]$, in the case of X being a terminal, and they are used to denote that an *arbitrary* production rule with left-hand side X has been recognized between positions i and j, in the case of X being a nonterminal. The name 'CYK' refers to the Cocke-Younger-Kasami parsing algorithm Nijholt (1990); Harrison (1978), where the same kind of items are used.

```
proc chartParser
 create initial chart and agenda;
 while agenda not is empty do
        delete (arbitrarily chosen) current item from agenda;
        foreach item that can be recognized by current
           in combination with other items on chart do
           if item is neither on chart nor on agenda then
                                                    add item to agenda
           fi
        od
 od
.
```

Figure 5.1: **General schema for a chart parser**

$[B \to \alpha \bullet \beta \bullet \gamma, i, j]$ **— double dotted items** For each such item that is recognized it holds that $\beta \Rightarrow^* a_{i+1} \ldots a_j$.

We can now define the head-corner parsing schema $\mathbb{P}_{sHC} = \langle I_{sHC}, H, D_{sHC} \rangle$. The set of items I_{sHC} is defined as

$$\begin{aligned}
I^{Pred} &= \{[l,r,A] \mid A \in N \wedge 0 \le l \le r\},\\
I^{HC(i)} &= \{[B \to \alpha \bullet \beta X \bullet \gamma, i, j] \mid B \to \alpha\beta\underline{X}\gamma \in P \wedge 0 \le i \le j\},\\
I^{HC(ii)} &= \{[B \to \alpha \bullet X\beta \bullet \gamma, i, j] \mid B \to \alpha\underline{X}\beta\gamma \in P \wedge 0 \le i \le j\},\\
I^{HC(iii)} &= \{[B \to \bullet\bullet, j, j] \mid B \to \varepsilon \in P \wedge j \ge 0\},\\
I^{CYK} &= \{[A,i,j] \mid A \in N \wedge 0 \le i \le j\},\\
I_{sHC} &= I^{Pred} \cup I^{HC(i)} \cup I^{HC(ii)} \cup I^{HC(iii)} \cup I^{CYK}.
\end{aligned}$$

The hypotheses are defined as usual:

$$H = \{[a_1,0,1], \ldots, [a_n,n-1,n], [\$,n,n+1]\}$$

Finally, the deduction steps of $\mathbb{P}_{sHC}$ are defined as

$$\begin{aligned}
D^{Init} &= \{[\$,n,n+1] \vdash [i,j,S] \mid 0 \le i \le j \le n\},\\
D^{HC} &= \{[i,j,A],[X,i,j] \vdash [B \to \alpha \bullet X \bullet \beta, i, j] \mid A >_h^* B\},\\
D^{HC(\varepsilon)} &= \{[j,j,A] \vdash [B \to \bullet\bullet, j, j] \mid A >_h^* B\},\\
D^{lPred} &= \{[l,r,A],[B \to \alpha C \bullet \beta \bullet \gamma, k, r] \vdash [i,j,C] \mid A >_h^* B \wedge l \le i \le j \le k\},\\
D^{rPred} &= \{[l,r,A],[B \to \bullet\beta \bullet C\gamma, l, i] \vdash [j,k,C] \mid A >_h^* B \wedge i \le j \le k \le r\},\\
D^{preCompl} &= \{A \to \bullet\beta\bullet, i, j] \vdash [A,i,j]\},\\
D^{lCompl} &= \{[i,k,A],[X,i,j],[B \to \alpha X \bullet \beta \bullet \gamma, j, k] \vdash [B \to \alpha \bullet X\beta \bullet \gamma, i, k] \mid A >_h^* B\},\\
D^{rCompl} &= \{[i,k,A],[B \to \bullet\beta \bullet X\gamma, i, j],[X,j,k] \vdash [B \to \bullet\beta X \bullet \gamma, i, k] \mid A >_h^* B\},\\
D_{sHC} &= D^{Init} \cup D^{HC} \cup D^{HC(\varepsilon)} \cup D^{lPred} \cup D^{rPred} \cup D^{preCompl} \cup D^{lCompl} \cup D^{rCompl}.
\end{aligned}$$

The antecedent in D^{Init} is often omitted, as well as the terminal item $[\$,n,n+1]$. In that case the items $[i,j,S], 0 \le i \le j \le n$ are just initial agenda items.

With the deduction step D^{HC} new head-corner items can be created. It says that if a nonterminal A is predicted between position i and j, and an X has already been recognized between i and j, then all double

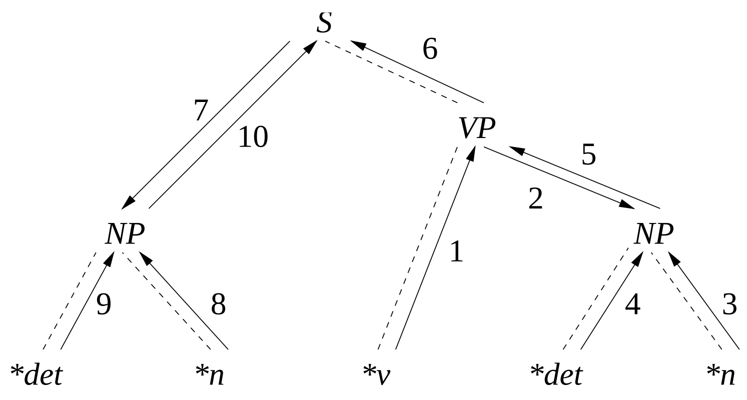


Figure 5.2: **A head-corner tree walk**

dotted items can be created that use a production rule $B \rightarrow \alpha\underline{X}\beta$ such that $A >_h^* B$. Since X is the only symbol between the two dots it must necessarily be the head of the production rule. The deduction step $D^{HC(\varepsilon)}$ can be seen as a special case of D^{HC}, where X is equal to ε.

The meaning of the predict deduction steps is rather straightforward; for every double dotted item we predict a predict item with the nonterminal that occurs just outside the double dots. The positions of the double dotted items and other predict items put constraints on the positions of the new predict items.

Finally, we have the complete deduction steps. The pre-complete deduction step just turns every completely recognized right-hand side of a production rule into a CYK item with the left-hand side of that rule as nonterminal. The other two complete deduction steps say that we can 'lift the dot' over nonterminal X, if we have recognized X at the right positions.

With the parsing system $\mathbb{P}_{sHC}$ it is now possible to parse sentences, given a context-free head grammar. We take the grammar from section 5.1 and use the sentence "The man bites a dog" again as an example. The derivation tree is (of course) the same as in figure 1.2, but the order in which the (non)terminals are recognized is now less free. As a matter of fact, the order in which items are recognized with this grammar is fixed. This is, however, not necessarily so. If we would have production rules with more than two symbols on the right-hand side, there would also be more orders to create items. In figure 5.2 the *head-corner tree walk* is given. (The terminals are not shown in this figure.) This tree shows the order in which items are recognized. The arrows upwards are due to applications of *complete* or *head-corner* deduction steps, the arrows downwards are due to applications of the *predict* deduction steps. In table 5.1 the chart is shown after the sentence has been parsed.

5.3 HC Parsing using Typed Feature Structures

In this section the head-corner parsing system is applied to typed feature structures. This will result in some extra constraints on the items and deduction steps. Before we can introduce the new parsing scheme that incorporates these constraints, we have to define yet some other notational conventions. As was already mentioned in section 3.3, every context-free grammar rule has a typed feature structure associated with it[1]. The minimal (with respect to information content) typed feature structure for a grammar rule $p = A \rightarrow \alpha \in P$ that satisfies all the specified constraints will be denoted by $\varphi_0(p)$. The notation $\varphi_0(X) = \varphi_0(p)|_X$ will be used to denote the feature structure associated with symbol X, where X is a symbol from $A\alpha$. Furthermore,

[1]Rules could also be considered to be a special class of typed feature structures. The difference, however, would be purely notational and not very relevant.

#	item	recognized by
(i)	$[^*det, 0, 1]$	initial chart
(ii)	$[^*n, 1, 2]$	initial chart
(iii)	$[^*v, 2, 3]$	initial chart
(iv)	$[^*det, 3, 4]$	initial chart
(v)	$[^*n, 4, 5]$	initial chart
(0)	$[i, j, S], 0 \leq i \leq j \leq n$	initial agenda
(1)	$[VP \rightarrow \bullet {}^*v \bullet NP, 2, 3]$	head-corner (0,iii)
(2)	$[3, 5, NP]$	right predict (0,1)
(3)	$[NP \rightarrow {}^*det\ \bullet {}^*n \bullet, 4, 5]$	head-corner (2,v)
(4)	$[NP \rightarrow \bullet {}^*det\ {}^*n \bullet, 3, 5]$	left complete (0,iv,3)
(4a)	$[NP, 3, 5]$	pre-complete (4)
(5)	$[VP \rightarrow \bullet {}^*v\ NP \bullet, 2, 5]$	right complete (0,1,4a)
(5a)	$[VP, 2, 5]$	pre-complete (5)
(6)	$[S \rightarrow NP\ \bullet VP \bullet, 2, 5]$	head-corner (0,5a)
(7)	$[0, 2, NP]$	left predict (0,6)
(8)	$[NP \rightarrow {}^*det\ \bullet {}^*n \bullet, 1, 2]$	head-corner (7,ii)
(9)	$[NP \rightarrow \bullet {}^*det\ {}^*n \bullet, 0, 2]$	left complete (7,i,8)
(9a)	$[NP, 0, 2]$	pre-complete (9)
(10)	$[S \rightarrow \bullet NP\ VP \bullet, 0, 5]$	left complete (0,9a,6)
(10a)	$[S, 0, 5]$	pre-complete (10)

Table 5.1: **A completed HC chart**

we will add indices ξ, η and ζ as subscripts to items. By writing $[B \rightarrow \alpha \bullet \beta X \bullet \gamma, i, j]_\xi$ we indicate that wherever ξ is written elsewhere in the same formula, this is an abbreviation for $[B \rightarrow \alpha \bullet \beta X \bullet \gamma, i, j]$. Finally, we will write $\varphi_0(\xi)$ for the typed feature structure that is initially associated with an item.

During the parsing process information will percolate from the terminal items to the item for a recognized start symbol of the grammar. For every item the initial feature structure needs to be specified. The way the information from these feature structures is passed on during parsing is defined in the deduction steps. The information content of initial feature structures will then increase. For these new feature structures we will use the notation $\varphi(\xi)$.

We can now define the parsing system $\mathbb{P}_{sHC(\mathcal{TFS})} = \langle I_{sHC(\mathcal{TFS})}, H, D_{sHC(\mathcal{TFS})} \rangle$, which describes a head-corner parser for typed feature structures.

$$
\begin{aligned}
I^{Pred} &= \{[l,r,A]_\xi \mid A \in N \wedge 0 \leq l \leq r \wedge \varphi_0(\xi) = \varphi_0(A)\},\\
I^{HC(i)} &= \{[B \rightarrow \alpha \bullet \beta X \bullet \gamma, i, j]\xi \mid B \rightarrow \alpha\beta\underline{X}\gamma \in P \wedge 0 \leq i \leq j \wedge\\
&\qquad \varphi_0(\xi) = \varphi_0(B \rightarrow \alpha\beta\underline{X}\gamma)\},\\
I^{HC(ii)} &= \{[B \rightarrow \alpha \bullet X\beta \bullet \gamma, i, j]_\xi \mid B \rightarrow \alpha\underline{X}\beta\gamma \in P \wedge 0 \leq i \leq j \wedge\\
&\qquad \varphi_0(\xi) = \varphi_0(B \rightarrow \alpha\underline{X}\beta\gamma)\},\\
I^{HC(iii)} &= \{[B \rightarrow \bullet\bullet, j, j]_\xi \mid B \rightarrow \varepsilon \in P \wedge j \geq 0 \wedge \varphi_0(\xi) = \varphi_0(B \rightarrow \varepsilon)\},\\
I^{CYK} &= \{[A, i, j]_\xi \mid A \in N \wedge 0 \leq i \leq j \wedge \varphi_0(\xi) = \bot\},\\
I_{sHC(\mathcal{TFS})} &= I^{Pred} \cup I^{HC(i)} \cup I^{HC(ii)} \cup I^{HC(iii)} \cup I^{CYK},
\end{aligned}
$$

$$
\begin{aligned}
H &= \{[a,i-1,i]_\xi \mid \varphi_0(\xi)=\varphi_0(a) \wedge 1 \le i \le n\} \cup \\
&\qquad \{[\$,n,n+1]_\eta \mid \varphi_0(\eta)=\bot\}
\end{aligned}
$$

$$
\begin{aligned}
D^{Init} &= \{[\$,n,n+1]_\xi \vdash [i,j,S] \mid 0 \le i \le j \le n\}, \\
D^{HC} &= \{[i,j,A],[X,i,j]_\eta \vdash [B \to \alpha \bullet X \bullet \beta, i, j]_\xi \mid A >_h^* B \wedge \\
&\qquad \varphi(X_\xi) = \varphi_0(X_\xi) \sqcup \varphi(\eta)\}, \\
D^{HC(\varepsilon)} &= \{[j,j,A] \vdash [B \to \bullet\bullet, j, j] \mid A >_h^* B\}, \\
D^{lPred} &= \{[l,r,A],[B \to \alpha C \bullet \beta \bullet \gamma, k, r]_\eta \vdash [i,j,C]_\xi \mid A >_h^* B \wedge l \le i \le j \le k \wedge \\
&\qquad \varphi(C_\xi) = \varphi_0(C_\xi) \sqcup \varphi(C_\eta)\}, \\
D^{rPred} &= \{[l,r,A],[B \to \bullet\beta\bullet C\gamma, l, i]_\eta \vdash [j,k,C]_\xi \mid A >_h^* B \wedge i \le j \le k \le r \wedge \\
&\qquad \varphi(C_\xi) = \varphi_0(C_\xi) \sqcup \varphi(C_\eta)\}, \\
D^{preCompl} &= \{A \to \bullet\beta\bullet, i, j]_\eta \vdash [A,i,j]_\xi \mid \varphi(\xi) = \varphi(\eta)\}, \\
D^{lCompl} &= \{[i,k,A],[X,i,j]_\eta,[B \to \alpha X \bullet \beta \bullet \gamma, j, k]_\zeta \vdash [B \to \alpha \bullet X\beta \bullet \gamma, i, k]_\xi \\
&\qquad \mid A >_h^* B \wedge \varphi(\xi) = \varphi(\zeta) \wedge \varphi(X_\xi) = \varphi_0(X_\xi) \sqcup \varphi(\eta)\}, \\
D^{rCompl} &= \{[i,k,A],[B \to \bullet\beta\bullet X\gamma, i, j]_\zeta,[X,j,k]_\eta \vdash [B \to \bullet\beta X \bullet \gamma, i, k]_\xi \\
&\qquad \mid A >_h^* B \wedge \varphi(\xi) = \varphi(\zeta) \wedge \varphi(X_\xi) = \varphi_0(X_\xi) \sqcup \varphi(\eta)\}, \\
D_{sHC(\mathcal{TFS})} &= D^{Init} \cup D^{HC} \cup D^{HC(\varepsilon)} \cup D^{lPred} \cup D^{rPred} \cup D^{preCompl} \cup D^{lCompl} \cup D^{rCompl}.
\end{aligned}
$$

The hypotheses are defined different than usual. The set H should be seen as a multi-set; there can be multiple occurences of $[a, i-1, i]$, if a is ambiguous. For each occurence a different feature structure is associated with it.

The assignment of initial feature structures to the items is rather straightforward, except maybe the assigment of $\bot$ to CYK items. This is because of the deduction step that creates CYK items; *preComplete* does not need an initial feature structure for a CYK item. So, in fact *any* typed feature structure could be specified as initial feature structure for CYK items, since they are not used anyway.

The constraint $\varphi(\xi) = \varphi(\zeta) \wedge \varphi(X_\xi) = \varphi_0(X_\xi) \sqcup \varphi(\eta)$ for the left- and right-complete deduction steps should be interpreted as: the feature structure $\varphi(\xi)$ for the new head-corner item is the same as the feature structure $\varphi(\zeta)$ for the old head-corner item, except that any information about X in $\varphi(\xi)$ is extended with the information of $\varphi(\eta)$.

Note that, unlike the parsing system $\mathbb{P}_{pHC(UG)}$ for untyped unification grammars as defined on pages 266–267 of Sikkel (1993), no extra constraints on features are required for the feature structures associated with the items. In our case these constraints are unnecessary, since the typing already restricts the feature structures to valid values.

5.4 Implementation

The implementation of the head-corner parser is at this moment heavily under construction. But given the source code for the head-corner parser of Veldhuijzen van Zanten (1994), the implementation should not be too hard. Due to time constraints and lack of documentation for the implementation of that parser it has not been possible to complete the $\mathcal{TFS}$ system with a working HC parser. We are now in the debugging phase of the development process (see also appendix C.3). Fortunately, most code for the HC parser is shared with the $\mathcal{TFS}$ compiler (see figure 5.3), and the $\mathcal{TFS}$ compiler already works.

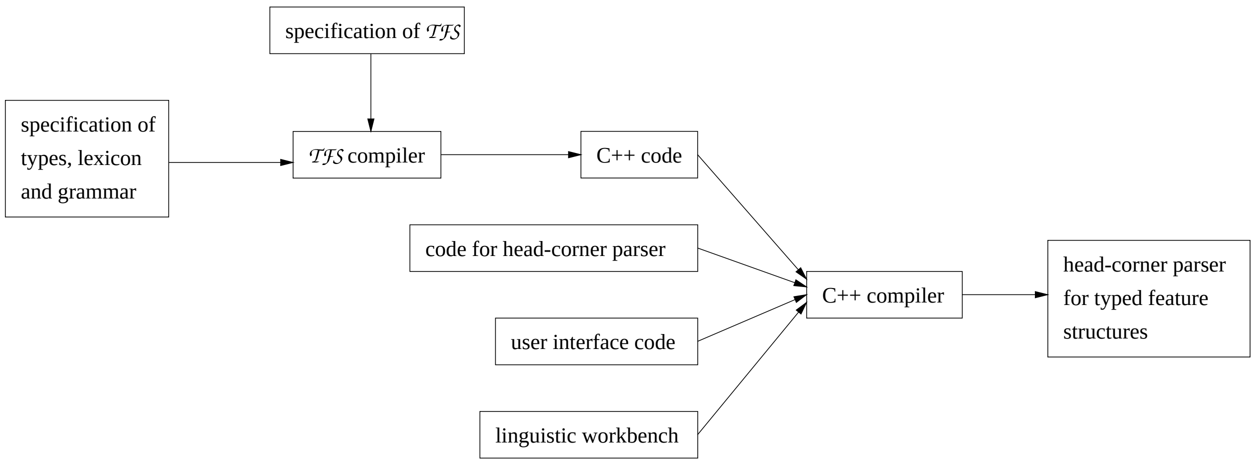

specification of 𝒯ℱ𝒮
specification of types, lexicon and grammar
𝒯ℱ𝒮 compiler
C++ code
code for head-corner parser
user interface code
linguistic workbench
C++ compiler
head-corner parser for typed feature structures

Figure 5.3: **The 𝒯ℱ𝒮 system.** The design of the head-corner parser for typed feature structures.

If the HC parser is finished, it should be possible to use the parser in 'batch-mode'. That is, a file with a number of sentences is given to the parser, and the parser prints the chart after each parsed sentence. If any debugging options have been specified (see appendix C), some additional information can be obtained. In the near future the 𝒯ℱ𝒮 system could then be extended with a more interactive window based environment. Ideally, the 𝒯ℱ𝒮 system would also have a testing environment with which (certain aspects of) a grammar can be tested. This is called the 'linguistic workbench' in figure 5.3.

The implementation has been made in the object-oriented language C++. This language is often used in the development of commercial applications because of its support of a high level of abstraction on the one hand and the abilities to still use low-level constructs on the other hand. Programming in C++ could be done in an imperative style, but then there would be no advantage in using C++ over using Modula-2 (which was used for the predecessors of the 𝒯ℱ𝒮 parser). I thought that the object-oriented design methodology Bergin (1994) might be very useful to keep a system as large as 𝒯ℱ𝒮 manageable and maintainable. Other considerations that played a role in the choice for C++ were:

- C++ can easily be interfaced with a automatic lexical analysis generator program and a compiler-compiler (see appendix C.2), and
- it would give me the chance to learn a new design methodology and programming language.

In figure 5.4 the defined classes and the relations between them are shown. A class in C++ stands for a collection of similar objects. Each objects contains some data and a number of methods that can access and modify these data. The set of data and methods are called the members of an object. Objects can communicate with other objects by sending messages through the methods of these objects. Methods and messages correspond to procedures and arguments in a imperative programming language. It is considered to be bad design, if an object directly modifies the data of another object. Usually a distinction is made between methods that are used for internal use only ('*private* methods') and methods that can be used by any object ('*public* methods'). A short overview of other object-oriented concepts is given in appendix B. In this appendix there is also a detailed description of the most important methods of each class. In the rest of this section we will focus on the overall structure of the classes and the working of the program.

The solid arrows in figure 5.4 stand for the part-of relation. So $\boxed{\text{Parser}} \longrightarrow \boxed{\text{Chart}}$ means that a *Parser* object has a *Chart* object as one of its members. In addition to the *Chart*, the *Parser* also has an *Agenda*, a *Scanner* and a *Grammar*. Both the *Chart* and the *Agenda* can be seen as a list of items. In the case of the *Agenda*, this is expressed with the inheritance relation. The only thing in which the *Agenda* differs from

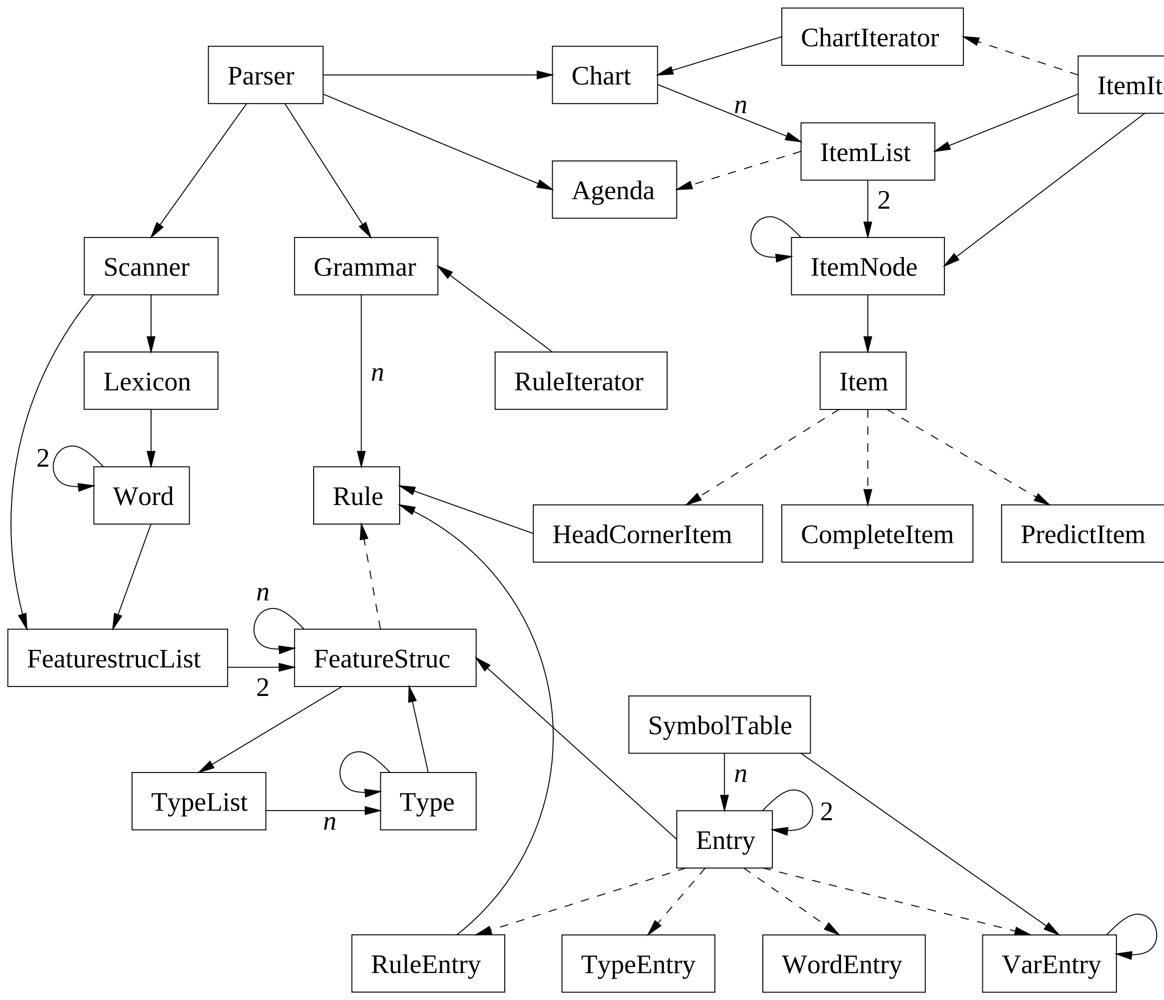


Figure 5.4: **The classes used in the $\mathcal{TFS}$ parser.** Solid arrows mean that a class has an object of another class as member. If a class has more than one object of the other class as members, then the number of objects is indicated by the label on the edge of the arrow. Dashed arrows express an inheritance relation between two classes.

an ordinary *ItemList* is in the way an *Agenda* is printed. The implementation of the *Agenda* does not need to be very complicated. Any structure that allows a simple storage and retrieval of items would do. The *ItemList* is organized as a queue; items are put on one end of the list and removed from the other end. The *Chart* is organized in a more structured way. The *Chart* is implemented as a matrix of item lists. Every entry $e_{i,j}$ in the matrix is a list of items that have position markers i and j. Since $0 \leq i \leq j \leq n$ the matrix is in fact an upper-triangular matrix. This kind of structure allows fast searching in the chart, if the position markers of the item that is searched for are already known.

An *ItemList* consists of a linked list of *ItemNode*s, which function as a sort of ‘wrappers’ for *Item* objects. In theory it is possible that an object belongs to the *Item* class, but does not belong to one the subclasses *HeadCornerItem*, *CompleteItem* or *PredictItem*. In practice items should always be of a certain kind; the parser cannot do anything with items of an unknown kind. The actual parsing is done by the items themselves. Every kind of item is equipped with a method for every deduction step, in which it occurs as antecedent. Each such method looks for the other antecedents on the chart and if they are found, the consequent is created.

The *Parser* creates the initial items (the hypotheses) by making subsequent calls to the *nextItem* method of the *Scanner* object. The *Scanner* returns one item at a time until the end of the sentence is reached. Before it can return an item, it reads a word from the input and looks it up in the *Lexicon*. If the word occurs in the *Lexicon*, it returns a list of feature structures, one feature structure for each meaning of the word. Note that the *Parser* does not have to concern itself with ambiguous words; these are handled by the *Scanner* and the *Lexicon*. Items for ambiguous and consecutive words are handled in the same way by the *Parser*.

The *Lexicon* is implemented as a binary search tree of *Word*s Aho et al. (1983). Every node in the tree contains a *Word*. In the tree attached to the left branch of a word, only words occur with a smaller lexicographical value. Likewise, in the tree at the right branch only words occur with a larger lexicographical value. A *Word* consists of a lexeme and a list of feature structures for every meaning of the word.

The *Grammar* class is nothing more than an array of grammar rules and a number of methods that use these rules. The *Rule* class is a subclass of the *FeatureStruc* class. In addition to the members of the *FeatureStruc* class it has a data member that indicates the head of the rule and methods for selecting (the feature structure assocociated with) a symbol in the rule.

The most important members of the *FeatureStruc* class were already described in section 2.3. There are a number of other methods that can inspect or modify a feature structure. There are, of course, methods that assign or retrieve a feature value given a certain feature name. Also, a variant of the *unify* method, called *build*, has been defined to create feature structures from a $\mathcal{TFS}$ specification. It differs from *unify* in the way that type checking is done; in this respect *build* is more ‘tolerant’. This method is necessary because the least upper bound of two types cannot be computed, if that least upper bound is just being defined.

The information about all types is stored in a *TypeList* object. Every feature structure shares the same *TypeList* to enforce that only one *TypeList* is used. A *TypeList* consists of an array (during the parsing of sentences) or linked list (during the parsing of a $\mathcal{TFS}$ specification) of *Type*s. Each *Type* contains a typed feature structure and a row of the least upper bound matrix. So, if the *leastUpperBound* method of the *TypeList* is called, that method will first select the type with the largest type identifier[2]. Then the right element in the row of the least upper bound matrix can be returned using the data of that type.

Finally, we have a number of classes that are used only during the parsing of a $\mathcal{TFS}$ specification. The most important one is the *SymbolTable* class. It contains many methods to facilitate the specification of semantic actions for the $\mathcal{TFS}$ grammar[3]. During the parsing of a specification all information about types, words, grammar rules and variables is stored in a so-called hash table Aho et al. (1983). A hash table is an array of entry lists. An entry is an object that contains the feature structure of a certain type, word, rule of variable and the name that is associated with it. The position of an entry in the symbol is determined by a hash function. The hash function maps a string of characters (the name of an entry) to an integer (the index in the array). Properties of a good hash function are (1) that every index in the array is approximately equally likely to be the index for a certain unknown string and (2) that similar strings are mapped to different indices. More about symbol tables, hash functions and other compiler construction topics can be found in Aho et al. (1986).

[2]Type identifiers are actually just integers.

[3]The grammar for $\mathcal{TFS}$ itself, that is, not a grammar specified with $\mathcal{TFS}$.

Chapter 6

Discussion

6.1 Unification Revisited

There has been a lot of research on unification. This research can be roughly divided into two kinds: (1) research on more efficient implementations of unification and (2) research on how unification can be applied to non-standard feature structures. The unification algorithm presented in this report could be classified as belonging to both kinds, since it uses a new method to share substructures and it is applicable to typed feature structures.

6.1.1 Other Efficient Unification Algorithms

Unification algorithms can be divided into three classes: destructive, non-destructive and quasi-destructive algorithms. With destructive unification the operands are (partially) destroyed after unification. If that is not desired, a copy has to be made of every feature structure that is going to be used as an operand for unification. Since copying takes up most of the time of unification, we can ignore the class of destructive unification algorithms, if we are looking for the most efficient algorithm.

Non-destructive unification, on the other hand, leaves the operands intact; feature structures are exactly the same before and after being used as an operand. Quasi-destructive unification is a variation on this. It leaves the feature structures intact on the outside, but it changes some administrative properties of the operands. There are three principles for efficient unification Tomabechi (1992):

- copying should be performed only for successful unifications,
- unification failures should be found as soon as possible,
- unmodified subgraphs can be shared.

The unification algorithm as it was designed by Velduijzen van Zanten, was based on an older unification algorithm of Tomabechi (1991) (which was in turn based on an algorithm by Wroblewski (1987)), where no structures where shared[1]. It appears that Tomabechi's structure-sharing unification algorithm is more or less equivalent to Veldhuijzen van Zanten's algorithm. That is, the implementation is somewhat different, but the effect on the efficiency is the same.

We see a similar situation for unification algorithms for typed feature structures. Kogure (1994) adapted Tomabechi's algorithm for typed feature structures and I adapted Veldhuijzen van Zanten's algorithm. Again, the net effect seems to be the same (although Kogure claims to have improved the structure-sharing part of Tomabechi's algorithm). It would be interesting to see how all these algorithms relate. It is, however, very difficult to make a quantative comparison between unification algorithms for typed and untyped feature structures; one has to write a grammar that is unbiased with respect to typing.

6.1.2 Extensions to the Notion of Unification

In the previous subsection the notion of unification was already extended to the domain of *typed* feature structures, but there also some other possible extensions. One way to make extensions is to take the logical approach. In the past feature structures have been formalized with an attribute-value logic Johnson (1988) or with so-called ψ-terms Aït-Kaci (1984). Based on such a logical backbone, feature structures can be extended with, for instance, disjunction and negation Hegner (1991); Eisele and Dörre (1988); Kasper (1987). To be more precise, these logical operators can be applied to the feature values of a feature structure. Many such extensions exist, complete with proofs of soundness and completeness, but often an *implementation* of an efficient unification algrithm is not given.

A more linguistic approach is taken by Bouma (1993). He introduces the so-called *default unification*, which can be used to overrule the feature values of one feature structure by another one. The following example illustrates how this can be done ('$\sqcup!$' is used as the default unification operator).

$$\begin{tikzcd} \bullet \arrow[d, "f"'] & & & \bullet \arrow[dl, "f"'] \arrow[dr, "g"] & & & & \bullet \arrow[dl, "f"'] \arrow[dr, "g"] & \\ a & \sqcup! & b & & b & = & b & & b \end{tikzcd} \qquad (6.1)$$

The left feature structure is called the default argument, the right one the non-default argument. The concept of overruling can also be applied to type inheritance. This is particularly useful, if one wants to describe conjugations with types.

6.2 Future Work

As usual there is always some work left to do. In retrospect I think that my plans were maybe a bit too ambitious, in the sense that they never could have been carried out within seven months by one person. In the next paragraphs the things that still need to be done are listed as well as some other extensions.

The implementation of the head-corner parser still needs some work. As was already said in section 5.4, most of the implementation is now finished, but the translation of Veldhuijzen van Zanten's chart parser in Modula-2 to C++ is still in progress. If this phase has been finished, the $\mathcal{TFS}$ system is able to parse files containing sentences and produce the corresponding charts.

Ideally, a grammar writer would have somewhat more flexibility. For instance, it would be nice if nonterminals could be used in a sentence and could be specified as the initial goal. In this way the grammar writer could test only those parts of the grammar she is interested in. The usage of nonterminals in sentences

[1]Tomabechi makes a difference between *feature-structure* sharing (denoting coreferences) and *data-structure* sharing (where two distinct feature structures share a substructure). In this report structure sharing is always used for the second kind of sharing.

is not that difficult to implement. To recognize nonterminals as such they need to be preceded by an ‘escape-symbol’ (like ‘\’ or any other symbol that is not used in the sentences) and followed by position markers[2]. Furthermore, there has to be a method that looks up a type given its name, and a method that creates initial items based on the type and the position markers. These methods could then also be used to create abitrary initial agenda items, if the grammar writer can specify in some way the name of the nonterminal that is to be set as goal. All this is part of what was previously called the linguistic workbench.

An extension that would turn the $\mathcal{TFS}$ system into a real application would be a graphical user interface. With such an interface it should be possible to use the $\mathcal{TFS}$ system interactively. That is, the grammar writer types in a sentence and the sentence gets parsed. Any additional information during and after parsing should be available (in other windows). A few examples of what kind of information might be shown in these windows are: a list of deduction steps that have been taken so far, a display of the current agenda and chart and a (graphical) display of the parse tree/forest for a sentence. It would also be nice if the grammar writer could edit the specifications of the types, lexicon and grammar without leaving the application.

Also from a theoretical point of view there are some extensions possible. One very valuable extension would be some kind of rating mechanism for recognized items. The ‘score’ for every item can be used in the following ways:

- in case of ambiguities the item with the highest score can be taken as the correct one,
- the chart and the agenda can be pruned during the parsing process by removing the items with a score smaller than a certain lower bound.

For such a rating mechanism it is necessary to assign an initial score to every word, type and item kind. Furthermore, functions have to be designed that compute a new score based on a number of other scores. These functions can then be used to compute the score for the left-hand side of a grammar rule, given the scores for the symbols on the right-hand side. Likewise, the score for the consequent of an item deduction step can be computed, when the scores for the antecedents are known. Possibly, initial scores can be given to rules and deduction steps as well. For this scoring mechanism to be effective it is necessary to have a sufficiently large (tagged) corpus.

[2]If the position markers are not given, an infinite number of initial items is created.

Chapter 7

Conclusions

After the design and implementation of the $\mathcal{TFS}$ system I have come to the following conclusions:

- Typing and typed feature structures in particular are useful for a natural language system. Not only hierarchies of linguistic entities can be described effectively, but also properties of domain entities and relations between them can be modeled with typed feature structures.

- Both with untyped and typed feature structures it is possible to create relations between feature structures based on the subsumption relation, but only in the case of typed feature structures a grammar writer can take full advantage of that. This is caused by the fact that typed feature structures can be *refered to*, that is, the root node of each typed feature structure has a name.

- The unification algorithm of Veldhuijzen van Zanten can succesfully be applied to typed feature structures with only a few modifications. It is likely that the extra type-checking overhead is neglible compared to the increase in performance, if a suitable lattice has been defined.

- The computation of the least upper bound relation for a set of types can be done in cubic time by an algorithm that incrementally builds a matrix encoding for this relation.

- The detachment of the specification language from the internal representation leads to more readable (and writable) specifications. Especially writing semantic expressions can be done easier in a quasi-logical language than in a PATR-II-like language.

- With a typed specification language (like $\mathcal{TFS}$) redundancies are greatly reduced, as well as the chances of inconsistencies to occur. Also, by the use of inheritance specifications tend to get shorter, an advantage that will become more apparent for larger grammars.

- The head-corner parsing system $\mathbb{P}_{sHC}$ Sikkel (1993) can also be applied to typed feature structures. This results in a similar system, where feature structures are associated with items and constraints are put on them.

- The object-oriented design methodology has proven to be very useful for a programming-in-the-large project. More than with traditional imperative programming languages one is forced to think about a good partitioning of the problem into subproblems, and—maybe even more important—about a ‘clean’ interface between the subproblems. By ‘clean’ I mean that different classes do not interfere with each other.

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley, Reading, MA.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.

Aït-Kaci, H. (1984). *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially Ordered Types*. PhD thesis, University of Pennsylvania.

Alshawi, H., editor (1992). *The Core Language Engine*. The MIT Press, Cambridge, MA.

Bergin, J. (1994). *Data Abstraction: the Object-Oriented Approach using C++*. McGraw-Hill, New York.

Bouma, G. (1993). *Nonmonotonicity and Categorial Unification Grammar*. PhD thesis, Rijksuniversiteit Groningen.

Carpenter, B. (1992). *The Logic of Typed Feature Sructures*. Cambridge University Press.

Carpenter, B. and Penn, G. (1994). Ale 2.0 user's guide. Technical report, Carnegie Mellon University Laboratory for Computational Linguistics, Pittsburgh, PA.

Carroll, L. (1871). *Through the Looking-Glass, and What Alice Found There*. Macmillan.

Carroll, L. (1960). *The Annotated Alice*. Clarkson N. Potter. Alice's Adventures in Wonderland & Through the Looking-Glass, Edited by Martin Gardner.

Eisele, A. and Dörre, J. (1988). Unification of disjunctive feature descriptions. In *Proceedings of the 26th Annual Meeting of the ACL*, Buffalo.

Groenink, A. V. (1992). Een semantische interpretator voor een prototype natuurlijke-taalinterface. Technical Report TI-SV-92-1845, PTT Research, Leidschendam, The Netherlands. In Dutch.

Grosz, B., Jones, K. S., and Webber, B., editors (1982). *Readings in Natural Language Processing*. Morgan Kaufmann, Los Altos, CA.

Harrison, M. A. (1978). *Introduction to Formal Language Theory*. Addison-Wesley, Reading, MA.

Hegner, S. J. (1991). Horn extended feature structures: Fast unification with negation and limited disjunction. In *Fifth European Chapter of the Association for Computational Linguistics (EACL'91)*, pages 33–38, Berlin.

Hoeven, G. F. v. d., Andernach, J. A., Van de Burgt, S. P., Kruijff, G.-J. M., Nijholt, A., Schaake, J., and De Jong, F. (1995). Schisma: A natural language accessible theatre information and booking system. In *Proceedings of the First International Workshop on Applications of Natural Language to Data Bases*, Versailles, France.

Johnson, M. (1988). *Attribute-Value Logic and the Theory of Grammar*. CSLI Lecture Notes Series. University of Chicago Press.

Kasper, R. T. (1987). A unification method for disjunctive feature descriptions. In *Proceedings of the 25th Annual Meeting of the ACL*.

Kay, M. (1980). Algorithm schemata and data structures in syntactic processing. Report CSL-80-12, Xerox Parc, Palo Alto, CA. Reprinted in Grosz et al. (1982).

Kogure, K. (1994). Structure sharing problem and its solution in graph unification. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 886–892.

Nijholt, A. (1990). The CYK-approach to serial and parallel parsing. Memorandum Informatica 90-13, University of Twente, Department of Computer Science.

Rounds, W. C. and Kasper, R. T. (1986). A complete logical calculus for record structures representing linguistic information. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, pages 39–43, Cambridge, MA.

Shieber, S. M. (1986). *An Introduction to Unification-Based Approaches to Grammar*. Center for the Study of Language and Information, Stanford University, Stanford, CA.

Sikkel, K. (1993). *Parsing Schemata*. PhD thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands.

Sikkel, K. and Op den Akker, R. (1993). Predictive head-corner chart parsing. In *International Workshop on Parsin Technologies*, pages 267–275, Tilburg (The Netherlands), Durbuy (Belgium).

Sowa, J. F. (1984). *Conceptual Structures*. Addison-Wesley, Reading, MA.

Stroustrup, B. (1991). *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition.

Sudkamp, T. A. (1988). *Languages and Machines*. Addison-Wesley, Reading, MA.

Tomabechi, H. (1991). Quasi-destructive graph unification. In *Proceedings of the 29th Annual Meeting of the ACL*, Berkeley, CA.

Tomabechi, H. (1992). Quasi-destructive graph unification with structure-sharing. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 440–446.

Veldhuijzen van Zanten, G. and Op den Akker, R. (1994). Developing natural language interfaces: a test case. In Boves, L. and Nijholt, A., editors, *Twente Workshop on Language Technology 8*, pages 121–135, Enschede, The Netherlands.

Verlinden, M. (1993). Ontwerp en implementatie van een head-corner ontleder voor grammatica's met feature structures. Master's thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands. In Dutch.

Wroblewski, D. (1987). Nondestructive graph unification. In *Proceedings of the Sixth National Conference on Artificial Intelligence*.

Yellin, D. (1988). A dynamic transitive closure algorithm. Research Report RC 13535, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY.

Appendix A

The syntax of $\mathcal{TFS}$

In this appendix the syntax of the specification language $\mathcal{TFS}$ is given. Words in typewriter capitals as well as text between single quotes are tokens returned by the scanner. Most tokens are identical to the keywords used in a specification. There are two exceptions: the tokens `ID` and `VARIABLE` should be replaced by any valid identifier and variable representation. Italic words stand for nonterminals.

grammar	::=	*types lexicon rules*
types	::=	*type*
	|	*types type*
type	::=	`TYPE` ’(’ *typeid* ’;’ *typeidlist* ’;’ *patheqlist* ’;’ *qlfexpr* ’)’
typeidlist	::=	ε
	|	*nonemptytypeidlist*
nonemptytypeidlist	::=	*typeid*
	|	*nonemptytypeidlist* ’,’ *typeid*
typeid	::=	`ID`
lexicon	::=	*word*
	|	*lexicon word*
word	::=	`LEX` ’(’ *lexeme* ’;’ *typeidlist* ’;’ *patheqlist* ’;’ *qlfexpr* ’)’
lexeme	::=	`STRING`
patheqlist	::=	ε
	|	*nonemptypatheqlist*

nonemptypatheqlist	::=	pathequation
	|	nonemptypatheqlist ’,’ pathequation
pathequation	::=	’<’ path ’>’ ’=’ ’<’ path ’>’
	|	’<’ path ’>’ ’:=’ node
path	::=	edge
	|	path edge
node	::=	typeid
	|	constant
edge	::=	featureid
featureid	::=	`ID`
rules	::=	rule
	|	rules rule
rule	::=	`RULE` ’(’ cfgrule ’;’ patheqlist ’)’
cfgrule	::=	symbol ’−− >’ symbollist head symbollist
symbollist	::=	ε
	|	symbollist symbol
head	::=	’*’ symbolornot ’*’
symbolornot	::=	ε
	|	symbol
symbol	::=	typeid index
index	::=	ε
	|	`NUMBER`
qlfexpr	::=	ε
	|	moodop ’(’ formula ’)’
	|	formula
moodop	::=	`DECLARATIVE`
	|	`WHQUESTION`
	|	`YNQUESTION`
	|	`IMPERATIVE`
formula	::=	qlfbool

	|	*qlfterm*
	|	*qlfset*
qlfbool	::=	’(’ *qlfbool* ’)’
	|	`NOT` *qlfbool*
	|	*qlfbool* `AND` *qlfbool*
	|	*qlfbool* `OR` *qlfbool*
	|	*qlfbool* `IMPL` *qlfbool*
	|	*qlfboolquant* `VARIABLE` ’(’ *qlfbool* ’)’
	|	*predicate* ’(’ *qlftermlist* ’)’
	|	*qlfterm qlftermrel qlfterm*
qlfboolquant	::=	`FORALL`
	|	`EXISTS`
	|	`EXISTS1`
predicate	::=	`ID`
qlftermlist	::=	*qlfterm*
	|	*qlftermlist* ’,’ *qlfterm*
qlfterm	::=	`VARIABLE`
	|	*constant*
	|	*qlfcard*
qlftermrel	::=	’$<$’ | ’$>$’ | ’$=$’ | ’$\neq$’ | ’$\leq$’ | ’$\geq$’
constant	::=	`STRING`
	|	`NUMBER`
qlfset	::=	`SET VARIABLE` ’(’ *qlfbool* ’)’
qlfcard	::=	`COUNT VARIABLE` ’(’ *qlfbool* ’)’

Appendix B

Description of Classes and Methods

This appendix is intended as a reference manual for someone who wants to modify or expand the $\mathcal{TFS}$ parser. But it might also be of interest for those who want to know more about how some structures like the lexicon are implemented and how they can be accessed. In this chapter the methods of the classes as shown in figure B.1 are described in detail. That is, for every class the most important methods, the required arguments for that method and the net effect on an object are given. The source code is available via anonymous ftp at `ftp.cs.utwente.nl/pub/doc/Parlevink/MasterThesis/Moll/TFSsources.zip`

In addition to the methods described below, for almost every class the output operator '`<<`' has been defined. This means that for instance a feature structure can be printed in the same way as a string, a number, etc.:

```
cout << "A string" << fs << 12.34;
```

In this example `fs` is a feature structure and `cout` stands for the current output file. Also, for almost every class a method called *mapToCode* is defined. This method prints an object as C++ code to the output file specified by the argument. The generated code consists of a number of declarations.

Before the classes are described I shall point out some peculiarities of object-oriented programming in general and C++ in particular (see also Bergin (1994) and Stroustrup (1991)). First, an important notion in object-oriented programming is that all methods work on an object of a certain class. Ideally, an object is only modified by its own methods (i.e., the methods of the class to which it belongs). Objects can be created by so-called *constructors*. These constructors are methods with the same name as the class name. The usage of constructors is a safe way to make sure that all data members of a class have valid values. The opposite of a constructor is called a *destructor*. Destructors are used to dispose of an object that is no longer necessary. They can take no arguments for obvious reasons. The name of a destructor is always the class name preceded by a tilde.

In most object oriented languages it is possible to define multiple methods with the same name. This is called *overloading*. The idea is that methods that have more or less the same effect on an object, but are called with different arguments, can (or should) be given the same name. Overloading is very often applied to (infix) operators. In the example above we have overloaded the '$<<$' operator.

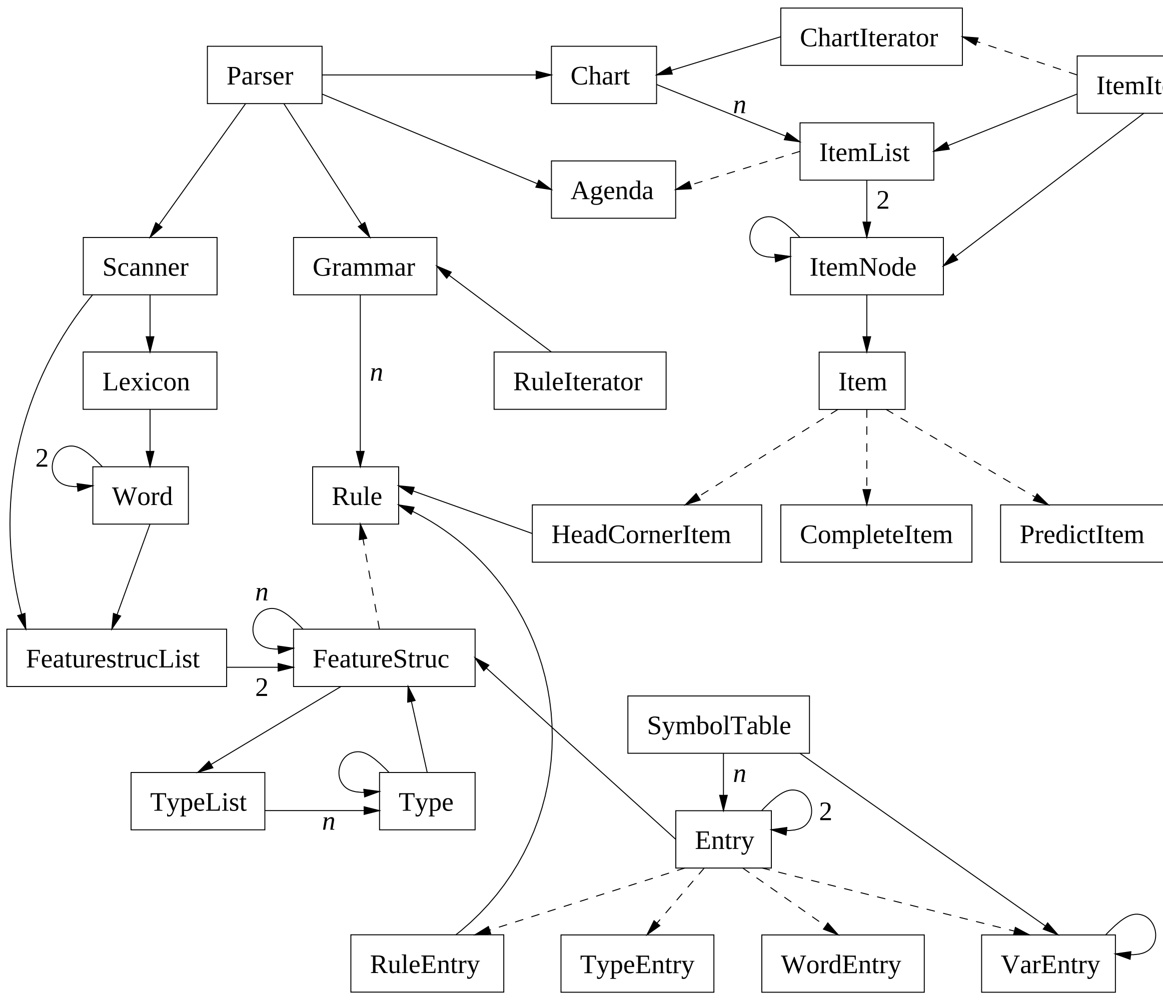


Figure B.1: **The classes used in the $\mathcal{TFS}$ parser.** Solid arrows mean that a class has an object of another class as member. If a class has more than one object of the other class as members, then the number of objects is indicated by the label on the edge of the arrow. Dashed arrows express an inheritance relation between two classes.

Finally, I want to mention the possibility of specifying *default values* for arguments. If a default value has been specified for an argument, such an argument can be omitted when the default value is the desired value.

B.1 The *FeatureStruc* Class

The *FeatureStruc* class is the most important and most extensive class. It describes the general structure of typed feature structures and operations on them. The class contains also all the information about the types and the type lattice. There is also a shared generation counter[1] to check if intermediate results can still be used. If the generation counter is increased, all intermediate results become obsolete. Note that methods that use the generation counter cannot be nested: the first thing these methods will do is increase the generation

[1]Class members with this property are called *static* members in C++.

counter, thereby destructing the intermediate results. If this becomes a problem for future extensions, it can be easily solved in the following ways: (1) make the methods 'atomic' using a 'Dijkstra-style' lock mechanism on the generation counter, or (2) make a local (with respect to the *method*, *not* with respect to the *object*) copy of the generation counter for each method that wants to use it.

FeatureStruc (2 arguments) There are three different constructors for the *FeatureStruc* class. This one creates a dummy feature structure. It is used to initialize the type list that is shared by all feature structures[1]. The type list contains all information about the types and the type lattice. The first argument to this constructor is an array of types, the second argument is an integer indicating the length of the array. This constructor should be called only once, since it is not desirable that the type lattice changes during program execution.

FeatureStruc (2 arguments) The first argument is an type identifier. The second argument is a string. The default value for the second argument is the empty string. The constructor creates an instance of the type indicated by the identifier. If the type is a string, a number or a predicate, the name of the feature structure is set to the second argument. Otherwise the second argument is ignored.

FeatureStruc (5 arguments) The last constructor is somewhat more 'low level' than the other ones, in the sense that the type list is not used to create a new feature structure, but all necessary information is given by the arguments. The first argument is the type identifier, the second argument the name of the feature structure, the third argument gives the number of features and the fourth and fifth argument specify an array of feature names and feature values, respectively.

typeId (0 arguments) This method just returns the type identifier of an feature structure. In this way other objects can read the type identifier, but can not change its value. The actual value is hidden and can only be accessed directly by feature structures.

deleteFeatureStruc (0 arguments) This method functions as a sort of destructor. Since (sub)feature structures can be shared, the number of times that a feature structure is referenced to should be kept up to date. This is done by giving each node in a feature structure a link counter. A call to *deleteFeatureStruc* decreases the link counter of every node by one. If a link counter is equal to zero, the node is deleted.

addFeature (3 arguments) The *addFeature* method should normally only be used during the construction of new types. (Otherwise a feature structure would already have all the necessary features.) The first and second argument are feature name and feature value, respectively. The feature value is a feature structure. The third argument is a flag with which the link counter of the feature value can be controlled. If the flag is equal to zero, the link counter of all nodes in feature value are increased by one. If the flag is equal to 1, only the link counter of the root node of the feature value is increased. Any other value for the flag does not change the link counter.

assignFeatureValue (3 arguments) For assigning a new value to an existing feature, there are two methods. The first one has the same arguments as *addfeature*. The old value of the feature with the name given by the first argument is 'deleted' by a call to *deleteFeatureStruc*. Then the new value is assigned to the feature.

assignFeatureValue (3 arguments) The second form of *assignFeatureValue* uses an integer as first argument instead of an string. This integer indicates the position of a feature structure in the array of feature values. This feature structure is 'deleted' by *deleteFeatureStruc* and replaced by the new value. This method allows faster access to a feature value, since no string comparisons have to be executed. It can be used in cases where the exact position of a feature value is known.

equal (1 argument) The method *equal* takes another feature structure as argument and returns **true** if both feature structures are equal. Two feature structures are equal if they describe the same graph and have the same labels on the nodes and edges.

unify (1 argument) The *unify* method takes another feature structure as argument and tries to unify the current feature structure with this feature structure. If unification is possible, it returns a pointer to the resulting unifact. Otherwise it returns zero. The unification algorithm is described in section 2.3.

unify0 (1 argument) This method is an auxiliary method for *unify* and also takes one feature structure as argument. It returns **true**, if unification is possible. *Unify0* corresponds to the procedure *unifiable* in figure 2.4.

build (1 argument) The *build* method is a less restrictive variant of the *unify* method; it always succeeds to create a resulting feature structure. It is used to build of new types when the specification is parsed and should not be used during the parsing of sentences.

build0 (1 argument) This method is an auxiliary method for *build* and has the same function as *unify0*.

copyUnifact (1 argument) After a call to *unify0* or *build0* the feature structure contains the intermediate results to generate the unifact. The generation of the unifact is done by *copyUnifact*. The working of this method is described in section 2.3.

copy (0 arguments) This method returns an exact copy of the current feature structure. No nodes are shared.

incLinkCount (0 arguments) The *incLinkCount* method increases the link counter of all the nodes in the current feature structure by one.

forward (1 argument) The *forward* method sets the forward pointer to the feature structure pointer given by the argument. It also sets the *forwardMark* to the current generation.

dereference (0 arguments) *Dereference* returns the forward pointer, if the *forwardMark* is equal to the current generation and the forward pointer is not equal to zero. Otherwise a pointer to the current feature structure is returned.

find (1 argument) The *find* method returns the index of the feature with the name specified by the argument in the array of features. If the requested feature does not exist, the value -1 is returned.

setCoreferences (0 arguments) This method places markers in those nodes that are referenced more than once within the same feature structure. This method is called by the output operator before a feature structure is actually printed.

nextGeneration (0 arguments) *NextGeneration* increases the shared generation counter by one.

B.2 The *FeatureStrucList* Class

The *FeatureStrucList* class is used only used by the *Word* class. Ambiguous words are represented by a list of feature structures and a lexeme. The *FeatureStrucList* class uses the *unifact* member of the *FeatureStruc* class, which is a pointer to a feature structure, to link one feature structure to another.

FeatureStrucList (0 arguments) This constructor creates an empty list of feature structures.

FeatureStrucList (2 arguments) This constructor is called by the *mapToCode* method. The first argument is an integer indicating the length of the array of feature structures specified by the second argument.

insert (1 argument) *Insert* adds another feature structure to the list.

first (0 arguments) This method returns the head of the list.

next (0 arguments) The *next* method returns the next feature structure on the list. By making subsequent calls to *next* one can walk through the list. If the end of the list is reached, zero is returned.

B.3 The *Type* and the *TypeList* Class

The *Type* class is very simple. It is in fact only a record with some data. It contains a feature structure, an array of type identifiers and a pointer to another type (to make a linked list of types). The array of type identifiers is a row of the lower triangle least upper bound matrix as described in section 3.1 on page 15. New types should only be created by *TypeList* methods, since these methods make sure that the type lattice remains consistent. The *Type* class contains no methods, only a constructor to initialize a type.

The *TypeList* class is used to store all information about types and the type lattice. The *FeatureStruc* class uses a type list to create new feature structures and to determine the least upper bound of two types.

TypeList (2 arguments) The *TypeList* constructor has two arguments which can both be omitted. If no arguments are given an empty type list is created. Otherwise the first argument is the length of the array of types specified by the second argument. Giving zero or two arguments has an effect on the internal representation of types. In the case of zero arguments the type list assumes the dynamic linked list mode. This mode is used during the parsing of the specification. In the case of two arguments the type list assumes the static array mode. This mode is used during the parsing of sentences. In the static mode no more types can be added, but information in the list can be accessed faster.

get (1 argument) The *get* method returns a copy of the type whose type identifier is given by the argument.

put (3 arguments) The *put* method appends a new type to the end of the type list. The first argument is a feature structure, the second argument contains the length of an array of supertypes specified by the third argument. With this new type and the list of super types a new row can be added to the least upper bound matrix. The *put* method makes a call to *computeClosure* to update (if necessary) entries in this matrix.

leastUpperBound (2 arguments) *LeastUpperBound* takes two type identifiers as arguments and returns the type identifier of the least upper bound of the two corresponding types.

computeClosure (0 arguments) This method updates the least upper bound matrix after a new type has been added. The algorithm used by this method is explained in section 3.1.

B.4 The *Word* and the *Lexicon* Class

The lexicon is implemented as a binary tree. Nodes in the tree are objects of the *Word* class. Every word consists of a lexeme and a list of feature structures. Every feature structure stands for a different meaning of the word. Furthermore, every word has two pointers to two other words—the standard way to implement a tree. The *Word* class contains no methods, only a constructor to initialize a word.

The *Lexicon* class contains almost no data; only a pointer to the root of the word tree and a counter for the number of words in the tree. But the *Lexicon* class does have some methods to operate on the word tree:

Lexicon (2 arguments) Both arguments can be omitted. In that case an empty lexicon is created. The first argument is an integer that stands for the number of words in the tree given by the second argument. This number is not checked with the actual number of words in the lexicon.

insert (1 argument) The only argument for *insert* is a feature structure. This method inserts a new word or a new meaning for a word that already occurs in the lexicon. In the latter case the feature structure is added to the list of feature structures for this word. The lexeme of the word can be extracted from the name of the feature structure.

lookup (1 argument) The *lookup* method takes a lexeme as argument and returns a list of feature structures if the lexeme occurs in the lexicon. If the lexeme does not occur in the lexicon, it returns 0.

B.5 The *Rule* Class

The *Rule* class is derived from the *FeatureStruc* class. That is, it inherits the methods and data from the *FeatureStruc* class. The symbols of a grammar rule correspond to the features of a feature structure. So it is important that the features are ordered sequentially and that this order remains intact after every operation on a grammar rule. Fortunately, the implementation of the *FeatureStruc* class is such that this constraint is satisfied. In addition to the *FeatureStruc* methods, the *Rule* class has its own methods:

Rule (2 arguments) The *Rule* constructor takes two arguments. The first argument is a feature structure, which is a representation of a grammar rule. The second argument is the head of the grammar rule.

length (0 arguments) *Length* returns the number of symbols on the right-hand side of the grammar rule.

symbol (1 argument) The *symbol* method returns the n^{th} symbol of the grammar rule, where n is given by the argument. A call to *symbol* with $n = 0$ returns the symbol on the left-hand side.

featureStruc (1 argument) This method is similar to the *symbol* method. Instead of the symbol it returns the feature structure that corresponds to the n^{th} symbol.

head (0 arguments) *Head* returns the head of the grammar rule.

unify (2 arguments) The *unify* method takes two arguments. The first argument is a feature structure and the second argument is an integer. The feature structure is unified with the n^{th} feature structure of the grammar rule. The resulting rule is returned.

B.6 The *Grammar* Class

With the *Grammar* class a set or rules and the reflexive and transitive closure of the head-corner relation can be kept together. The set of rules is implemented as a simple array of rules and the head-corner relation is implemented as a two-dimensional array of booleans.

Grammar (5 arguments) The *Grammar* constructor takes five arguments. The last two arguments can be omitted. The first argument is an integer that specifies the (maximum) number of rules in the grammar. The second argument gives the number of symbols that occur in the grammar. This is determines the size of the head-corner matrix. The third argument is the start symbol of the grammar. The fourth and fifth argument are the array of rules and the head-corner relation, respectively. So, the fourth and fifth argument can only be used if every grammar rule known in advance. This is used when a $\mathcal{TFS}$ specification of the grammar has been read and the C++ code is to be generated.

size (0 arguments) *Size* returns the number of rules in the grammar.

insert (1 argument) With *insert* a rule can be inserted in the grammar (as long as the number of rules does not exceed the maximum number of rules, as specified with the constructor call).

get (1 argument) The *get* method returns the n^{th} grammar rule in the array, where n is an integer given by the argument.

startSymbol (0 arguments) *StartSymbol* returns the start symbol of the grammar.

computeClosure (0 arguments) The *computeClosure* method computes the reflexive and transitive closure of the head-corner relation. After all rules have been inserted in the grammar a call to *computeClosure* has to be made.

HCclosure (2 arguments) *HCclosure* takes two grammar symbols A and B as arguments and returns **true** iff $A >_h^* B$.

B.7 The *RuleIterator* Class

An iterator is a very general structure, that provides a way to access the elements in a collection one by one. The typical usage of any iterator is like this:

```
// initialize a new iterator:
Iterator* iterator = new Iterator(collection);
Element* element;

while (iterator->next(element)) {
    do something with element
}
```

In the example above `collection` is some collection containing elements of type `Element`. `Iterator*` is the C++ notation for ‘a pointer to an `Iterator`’. Likewise, `Element*` is ‘a pointer to an `Element`’.

The methods of the *RuleIterator* class are straightforward: they provide a simple way to ‘walk’ through all the rules in an object of the *Grammar* class.

RuleIterator (1 argument) The *RuleIterator* requires one argument: the grammar on which the iterator is going to operate.

next (1 argument) The *next* works different than any other method we have seen so far. Usually arguments function as input, but in this case the argument is used to return the next rule. The method itself returns **true** iff all rules have been accessed. See also the example above.

reset (0 arguments) This method reinitializes the iterator to the first element of the grammar.

done (0 arguments) *Done* returns **true** iff all rules have been accessed.

B.8 The *Item* Class and Its Subclasses

The head-corner parsing algorithm for typed feature structures has been explained in section 5.3. This algorithm has been implemented in the following way in C++. The common properties of the different kinds of items are described in the *Item* class. Data and methods specific for complete items, predict items and head-corner items are implemented in classes derived from the *Item* class. For every kind of item there is a different subclass. Every class has an constructor (of course) to create new items, a method called *combineWithChart* that tries to create new items by combining the current item with the items on the chart and method called *equal* that returns **true** iff the argument is equal to the current item. So there are different classes with methods that have the same name and do the same thing at the conceptual level, but work differently at the implementation level. The method *combineWithChart* calls several other methods to create new items. The number and implementation of these methods differ for the subclasses.

B.9 The *ItemList* Class and Related Classes

The *ItemList* class is used as the base class for the *Agenda* class and is also used in the implementation of the *Chart* class. The *ItemList* class is implemented as an ordinary first-in-first-out list; there is a method to put items on the list, a method to get items from the list and a method to check if the list is empty.

There is also an *ItemIterator* class that is implemented in the same way as the rule iterator. The only difference is that the *ItemIterator* constructor has *two* arguments: the first argument is an *ItemList* object and the second argument is the kind of item that the iterator must look for. So, if we set the second argument to complete item, then the *next* method will return only the complete items on the list and will skip any other items.

B.9.1 The Agenda Subclass

The *Agenda* class is almost the same as its superclass *ItemList*. There is only one difference. Putting items on the agenda works slightly different now. The method for putting items on a list is called *putItem* for both the *ItemList* class and the *Agenda* class, but the *putItem* method for the *Agenda* class takes *two* arguments. The first argument is a pointer to the chart and the second argument is an item. If the item already occurs on the chart, then *putItem* doesn't do anything. Otherwise the item is put on both the chart and agenda.

B.9.2 The Chart Class

The *Chart* is implemented as a matrix of item lists. Every entry $e_{i,j}$ in the matrix is a list of items that have position markers i and j. Since $0 \leq i \leq j \leq n$ the matrix is in fact an upper-triangular matrix. This kind of structure allows fast searching in the chart, if the position markers of the item that is searched for are already known. The constructor for the Chart class requires the sentence length as an argument to determine the size of the matrix. The interface 'on the outside' is the same as for the *ItemList*: the method *putItem* simply requires an *Item* as argument and adds it to the right item list

B.10 The *Scanner* Class

The scanner reads characters from the input file and returns initial items to the parser. It does so by using a lexical analyzer created by `flex++` (see also appendix C.2). The lexical returns one word at a time and skips unwanted characters (like spaces and line break characters). The scanner looks up every word in the lexicon and returns one complete item at a time. The *Scanner* class has the following methods:

Scanner (1 argument) The constructor takes one argument: a pointer to a lexicon. This lexicon is used to look up the feature structures that are associated with a certain string.

sentenceLength (0 arguments) *SentenceLength* returns the length of the sentence scanned so far.

lookup (1 argument) The string that is returned by the lexical analyzer is passed to *lookup*, which looks up the string in the dictionary. It puts the resulting list of feature structures (if any) in an internal variable of the scanner.

nextItem (0 arguments) This is the method that should be used by the parser to create the initial items. The *nextItem* method simply returns complete items until the end of the sentence is reached. At the end of a sentence the value 0 is returned. Note that ambiguous words are handled by the scanner alone; the parser does not see the difference between ambiguous or successive words.

reset (0 arguments) The *reset* method resets the internal variables of the scanner to enable it to scan another sentence.

B.11 The *Parser* Class

The parser uses all the previous classes to parse sentences. It reads the sentences from the standard input and after a sentence has been parsed the chart can be printed to a file or to the standard output. The *Parser* class does not need many methods, because most of the parsing is handled by other classes (like the *Item* class).

Parser (2 arguments) The constructor for the *Parser* class requires two arguments: a grammar and a scanner.

readInitialItems (0 arguments) The *readInitialItems* method reads the initial items by making subsequent calls to the *nextItem* method of the scanner until the end of an sentence is reached. It creates an initial chart and puts the initial items on the agenda.

parse (0 arguments) The *parse* parses one sentence and returns **true** iff a parse of the sentence has been found.

final (1 argument) This method returns **true** iff the argument is a final item, that is, a parse has been found.

B.12 The *SymbolTable* Class

A symbol table is the usual way that compilers use to store information about identifiers. The $\mathcal{TFS}$ parser stores information about types, words, rules and variables in this table. After a specification has been read, the entries are mapped to an internal representation. This means that types are added to a type list, the words are added to a lexicon and the rules are added to a grammar. After this has been done, these internal representations are mapped to C++ code. It is not necessary to map entries for variables to an internal representation or C++ code. These entries are only used to see if a variable is bound or unbound within a certain scope. At the end of the scope these entries are removed.

SymbolTable (0 arguments) The *SymbolTable* constructor does nothing else than initializing the internal data members of the class.

insertBasicTypes (0 arguments) The $\mathcal{TFS}$ parser provides many basic types that must be inserted with a call to *insertBasicTypes* before any other types are inserted in the symbol table. The basic types can be used in a specification. There are basic types for strings, numbers, but also for QLF entities such as the logical ‘and’.

makeBasicType (0 arguments) The *makeBasicType* is an auxiliary method for *insertBasicTypes*. It takes three arguments: a type identifier, an integer indicating how many features the new type will have and a name. This method will create a feature structure and give it a unique type identifier[2] and a name. At this moment only the *number* of features can be specified, not their names or values. Based on the second argument features are created with names op1, op2,... op$n$. The feature values are all set to $\perp$.

insert (3 arguments) There are three different methods to insert an entry in the the symbol table. It depends on the kind of entry which method is used. This method is intended to insert a type entry. The first argument is a feature structure, the second argument is an integer and the third argument is an array of type identifiers. The integer gives the length of the array. The array is meant to be a list of supertypes for the new type. The last two arguments are normally not used; they are used only by *insertBasicTypes*. Normally, the list of supertypes is constructed during parsing and is kept in an internal variable. The *insert* method gives the new type a unique identifier, puts the type in the type list that is used by the feature structures. The *insert* method also adds the unbound variables that occur in the QLF part of the type to the ‘unbound’ feature. Finally it inserts the type entry in the symbol table.

insert (3 arguments) This second version of *insert* can be used to insert a word or a variable. The first argument is the name of the word or variable, the second argument is a feature structure and the last argument is a flag indicating whether a word or a variable is to be inserted. In the case of a variable the feature structure is at this moment always equal to $\perp$, but in future versions of the $\mathcal{TFS}$ parser (when initial values for variables can be specified) this might change. A variable that is inserted, is added to the list of variables in the current scope. This list gets deleted at the end of the scope.

insert (2 arguments) The last version of *insert* can be used to insert grammar rules in the symbol table. The first argument is an integer indicating the position of the head in the rule and the second argument is a feature structure representation of the rule. The name of a rule is made equal to a character representation of the context-free grammar rule (like “`S --> NP *VP*`”). The start symbol of the grammar is made equal to the symbol on the left-hand side of the first rule that is inserted.

lookup (2 arguments) The *lookup* method looks up an entry with a name given by the first argument in the symbol table. With the second argument the kind of the desired entry can be specified. This implies that there can be multiple entries with the same name; as long as they have different kinds they can still be found. The *lookup* method returns the feature structure that belongs to the entry if an entry has been found. Otherwise the value 0 is returned.

appendEdge (2 arguments) This method appends an edge (the second argument) to feature path (the first argument). The new path is returned. A feature path is a feature structure where every node has only one feature (except of course for the node at the end of the path, which has no features). An edge is a feature structure is a path of length 1.

[2]The method that calls *makeBasicType* has to ensure that the identifier is indeed unique.

appendNode (2 arguments) The *appendNode* method replaces the node at the end of an path (the first argument) with another feature structure (the second argument). The new path is returned.

join (2 arguments) *Join* takes two paths as arguments and joins the ends (and the beginnings) of the paths. The resulting feature structure is returned.

unboundVariable (1 argument) The *unboundVariable* method adds a feature to the ‘unbound’ feature structure. This ‘unbound’ feature structure is reinitialized when a scope is entered. The feature names of ‘unbound’ are the unbound variables and the feature values are the feature structures that are associated with these names.

addUnboundVars (1 argument) At the end of the scope the ‘unbound’ feature structure can be added to a feature structure. This is done by the *addUnboundVars* method.

newType (0 arguments) This method should be called at the beginning of a new type definition. It initializes the variables that contain information about the super types.

addSuperType (1 argument) *AddSuperType* takes an type identifier as argument and adds it to the list of super types that is kept for the currently defined type.

beginScope (0 arguments) Currently scopes cannot be nested. So, the *beginScope* clears all old scope information and reinitializes the list of variables in the scope.

endScope (0 arguments) The *endScope* method deletes the list of variables in the scope.

hashpjw (1 argument) This is the hash function by P.J. Weinberger. According to the ‘dragon book’ Aho et al. (1986), pages 435–438, this is one of the best hash functions. It maps a string (the argument) to an index in the table.

B.13 The *Entry* Class and Its Subclasses

Similarly to the *Item* class, the common properties of entries in the symbol table have been defined in the *Entry* class and the different kind of entries are refined in different subclasses. Actually, objects of the subclasses *TypeEntry*, *WordEntry* and *RuleEntry* are nothing more than ‘wrappers’ around objects of the *FeatureStruc* or *Rule* class. These subclasses have a method called *mapToInternalRepresentation*, which adds an entry to a type list, a lexicon or a grammar. These methods are called when the entire specification has been read.

In addition to the subclasses mentioned above, there is a subclass *VarEntry*. This subclass can be used to store and retrieve information about variables.

Appendix C

Software Development Tools

Software development can be much easier if you have the right tools. Unfortunately, it can take quite some time to find out what kind of tools are available, which ones are the most useful and how they actually can be used. There is very little documentation about what software can be used and how it works[1]. In this appendix I shall give a short overview of the programs that I have used for the development of the $\mathcal{TFS}$ parser. In the first section I shall point out how the right editor can help in formatting source code and detecting simple errors before compilation. Also in this section I will give a short description of the Source Code Control System (SCCS), that can help to keep track of the changes in the source code.

In the next section the usage of a makefile, compilers and compiler-compilers is discussed. Finally, in section C.3 I will say a few things about debugging.

C.1 Source Code Editing and Maintenance

The design process of a program consists of several stages. After the conceptualization, where the task that the program is going to perform is defined, an initial implementation can be made. In this implementation several subtasks are identified. After this has been done these subtasks can be coded in C++. In general, a subtask corresponds to a class in C++. The right editor can make the creation of source code a bit easier. I have found the GNU Emacs editor particularly useful for my needs. Emacs has the following features that might be useful for editing a program:

highlighting of matching brackets This feature is not unique for Emacs; most editors nowadays show the matching opening bracket when a closing bracket is typed. But Emacs also ‘shows’ the opening bracket when it is not in the current window. In such a case Emacs says something like “`Matches int main() {`” (where `int main()` is the beginning of some procedure) on the command line.

C++ style indentation Emacs ‘knows’ C++ and it indents each line the right way. So, if a closing bracket is forgotten, it is noticed immediately, because the code will be indented one level to deep.

[1]This is not completely true; there is a lot information in the form of so-called man-pages on the network file system, but these pages can only be found if the user specifies where they are.

multiple buffers and windows Working on a large program often means that more than one file is edited at the same time. Emacs can hold multiple files at the same time. If necessary, multiple windows can be opened, so that more than one file can be seen at the same time. This is convenient when an implementation is made in one window of a header file in another window.

highlighting of keywords This feature cannot be used on a black and white display, but it seems that when a color display is used the keywords, variables and method names are given different colors. This would make it easier to detect typos.

Once source files are created, they can be added to the Source Code Control System (SCCS). With this system the user can keep track of changes that have been made to the source. If certain changes turn out to be wrong, an older version of the source file can be retrieved. Old versions can be retrieved based on the date they were created or on their revision number. To each version an optional comment describing the changes can be added. SCCS is also very useful when multiple users are working on the same program: SCCS registers which source files are 'checked out'. It registers what files are checked out, by who and if they are checked out for editing or just viewing. Files can be checked out only once for editing, but multiple times for viewing.

C.2 Compilers and Compiler-Compilers

The next step in the program development process is the compilation of the source files. Usually, there are many dependencies between the different files: implementation files depend on the header files and classes in one file often use objects of classes in another file. This means that when a file is changed, all other files that depend on this file need to be recompiled, too. With a so-called makefile these dependencies can be laid down. When the makefile is executed it checks which files have changed and which files need to be recompiled. The following example shows how dependencies can be specified in a makefile:

```
Main.o:             Main.c Parser.h Scanner.h
                    $(CC) $(CFLAGS) -c Main.c
```

This fragment says that the file `Main.o` depends on `Main.c`, `Parser.h` and `Scanner.h`. If one of those files is changed, the file `Main.c` is recompiled to create a new `Main.o`.

Most source files are C++ files and can be compiled with any C++ compiler. As far as I know, all the library functions that I have used (for e.g. handling I/O and strings) are part of the standard C/C++ libraries. So the $\mathcal{TFS}$ parser could be recompiled on any other platform that has a C++ compiler. The C++ compiler I used was the GNU C++ compiler `g++`, because it works nicely together with the GNU debugger (see section C.3) and because it comes with a lot of online documentation.

Besides the C++ files there also source files that contain specifications of the lexical analyzer and the parser of $\mathcal{TFS}$. With a lexical analysis program generator and a compiler-compiler these specifications can be turned into C++ code. The specification of the parser consists of a set of grammar rules, extended with appropriate semantic actions. The following example shows how this can be done (cf. appendix A).

```
path:               edge
                    { $$ = $1; }
                    | path edge
                    { $$ = symbolTable.appendEdge($1,$2); }
;
```

This piece of code corresponds to the context-free grammar rule $path \rightarrow edge \mid path\ edge$. The semantic actions are the C++ expressions between the curly brackets. For instance, `$$ = $1;` means that the semantic value of the left-hand side is equal to the semantic value of the right-hand side.

The advantages of using a compiler-compiler instead of coding the scanner and parser directly in C++ are obvious. A specification is easier to change, easier to read and syntax and semantics can be changed independently from each other.

The standard lexical analysis program generator is `lex` and the standard compiler-compiler is `yacc`, but these programs generate C code and not C++ code. This problem was solved by using the GNU versions of `lex` and `yacc`: `flex++` en `bison`. These programs can generate C++ code, so that the scanner and parser can be integrated easily into the rest of the program.

C.3 Debugging

The final steps in the development of a program are debugging and testing. If all compilation errors have been corrected, the program can be run. By turning on certain compiler options the program can give some extra information when a run-time error occurs. Usually a debugger is used to run the program. With a debugger the program can be halted at certain points or one can 'walk' statement by statement through the program. Also, the values of certain variables can be checked.

For debugging I used `xxgdb`, an X-Window front-end for the GNU debugger. This debugger uses four windows: (1) a window for giving commands, (2) a window to view the source that the program is currently executing, (3) a window to display the values of variables and (4) a sort of 'remote control' window which contains buttons for the most common debugging commands. With this debugger one can set breakpoints on certain methods, so that the exact behaviour of that particular method can be inspected.

In addition to the debugging information that the compiler adds to the program, I have added many lines of code for debugging purposes, that can be compiled conditionally. That is, by specifying certain options in the makefile some extra code is compiled. So, the performance of the program will not be affected if no option is specified. Typically such debugging code occurs at the beginning and at the end of a method. This code will then print the values of parameters and the result that is going to be returned. In total there are about 20 debug options. Usually an option corresponds to one method. Multiple options can be turned on at the same time.

Many run-time errors are caused by the usage of uninitialized variables, by indices that exceed array bounds and by the usage of pointers to freed memory ('dangling pointers'). A program that helps detecting these types of errors is `purify`. This program is invoked from within the makefile. It should be added to the line where the actual executable program is made. The `purify` program will add some extra debugging code to the program. Before a run-time error is actually made it halts the program. In the case of dangling pointers it will give information about where the memory was claimed, where it was freed and it will show the line in the source code were the error is about to occur.

Besides detecting these run-time errors, `purify` will also give information about so-called memory leaks. If program exits without run-time errors it is not necessarily so that no error occured. It might very well be that many pointer structures have claimed a lot of memory and that the program 'forgets' to free the memory when it is no longer needed. This type of error can become fatal if the prototype is going to be scaled up to a commercial application: the memory usage might grow exponentially with the size of the input. After the program has finished `purify` will show what memory is still in use. It will give information what kind of objects there are in memory and where they were created.

Appendix D

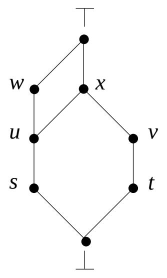
Test Results

Several test files were given as input to the $\mathcal{TFS}$ parser. In the beginning only the definition of types and the type lattice was tested. In the next step more complex operations like inheritance were tested. Finally, specifications of words and grammar rules were fed to the $\mathcal{TFS}$ parser. After the input was parsed, the contents of the symbol table was printed on the standard output (initially) or mapped to C++ code (in the end).

D.1 The Type Lattice Example

The computation of the least upper bound matrix was tested with the same example as in section 3.1. The (specification of the) lattice for which the least upper bound matrix has to be computed looks like this:

```
TYPE(s; ; ; )
TYPE(t; ; ; )
TYPE(u; s; ; )
TYPE(v; t; ; )
TYPE(w; u; ; )
TYPE(x; u,v; ; )
```



If the $\mathcal{TFS}$ parser has read this specification the least upper bound matrix looks like this:

```
           LUB |  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
               ---------------------------------------------------------------------------------------------------
     bottom  1 |  1
        top  2 |  2  2
    boolean  3 |  3  2  3
        not  4 |  4  2  4  4
        and  5 |  5  2  5  2  5
         or  6 |  6  2  6  2  2  6
      imply  7 |  7  2  7  2  2  2  7
  boolquant  8 |  8  2  8  2  2  2  2  8
     forall  9 |  9  2  9  2  2  2  2  9  9
     exists 10 | 10  2 10  2  2  2  2 10  2 10
    exists1 11 | 11  2 11  2  2  2  2 11  2  2 11
        set 12 | 12  2  2  2  2  2  2  2  2  2  2 12
       mood 13 | 13  2  2  2  2  2  2  2  2  2  2  2 13
     concat 14 | 14  2  2  2  2  2  2  2  2  2  2  2  2 14
<predicate> 15 | 15  2 15  2  2  2  2  2  2  2  2  2  2  2 15
    termrel 16 | 16  2 16  2  2  2  2  2  2  2  2  2  2  2  2 16
   <string> 17 | 17  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 17
   <number> 18 | 18  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 18
       card 19 | 19  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 19 19
          > 20 | 20  2 20  2  2  2  2  2  2  2  2  2  2  2  2 20  2  2  2 20
          < 21 | 21  2 21  2  2  2  2  2  2  2  2  2  2  2  2 21  2  2  2  2 21
          = 22 | 22  2 22  2  2  2  2  2  2  2  2  2  2  2  2 22  2  2  2  2  2 22
         >= 23 | 23  2 23  2  2  2  2  2  2  2  2  2  2  2  2 23  2  2  2  2  2  2 23
         <= 24 | 24  2 24  2  2  2  2  2  2  2  2  2  2  2  2 24  2  2  2  2  2  2  2 24
         != 25 | 25  2 25  2  2  2  2  2  2  2  2  2  2  2  2 25  2  2  2  2  2  2  2  2 25
          s 26 | 26  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 26
          t 27 | 27  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 31 27
          u 28 | 28  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 28 31 28
          v 29 | 29  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 31 29 31 29
          w 30 | 30  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 30  2 30  2 30
          x 31 | 31  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 31 31 31 31  2 31
               ---------------------------------------------------------------------------------------------------
```

The numbers in the matrix refer to the unique number that is given to each type. For example, the number for `top` (‘$\top$’) is 2. Not only the specified types have been inserted, but also all the basic types. If we compare the bottom right corner of this matrix with figure 3.2, we see that the least upper bound has been computed correctly.

D.2 The ‘Jan lives in Amsterdam’ Example

In this section the full specification is given of the example as it was used in section 3.4. After the specification has been read, the contents of the symbol table is sorted out into a type list, a lexicon and a grammar. These three objects are then printed. For the type list only the least upper bound matrix is shown. From this least upperbound matrix the entries for the basic types have been deleted, since they are the same as for the previous example and the matrix would otherwise not have fitted on the page.

```
// type section:
TYPE(numbertype;;;)
TYPE(singular;numbertype;;)
TYPE(plural;numbertype;;)
TYPE(persontype;;;)
TYPE(first;persontype;;)
TYPE(second;persontype;;)
TYPE(third;persontype;;)
TYPE(agreementtype;;
        <num> := numbertype,
        <pers> := persontype; )
TYPE(constituent;;
        <agr> := agreementtype; )
TYPE(thirdsing;constituent;
        <agr num> := singular,
        <agr pers> := third; )
TYPE(s;constituent;;)
TYPE(np;constituent;;)
TYPE(vp;constituent;;)
TYPE(noun;np;;)
TYPE(verb;constituent;
        <agrobj> := bottom; )
TYPE(pp;constituent;;)
TYPE(prep;constituent;;)
TYPE(propernoun;noun;;)
TYPE(location;agreementtype;;)
TYPE(inlocation;location;;)
TYPE(transitive;verb;;)
TYPE(person;noun;;
        EXISTS Person (personname(Person,Name)))

// lexical entries:
LEX("Jan"; propernoun, person, thirdsing;
        <unbound name> := "Jan"; )
LEX("lives"; transitive, thirdsing;
        <agr> := location;
        livesin(Subject,Object))
LEX("in"; prep; <agr> := inlocation;)
LEX("Amsterdam"; propernoun, thirdsing;
        <agr> := location;
        EXISTS Location (locationname(Location,"Amsterdam")))

// grammar rules:
RULE(s --> np *vp*;                    // Jan lives in Amsterdam
        <np agr> = <vp agr>,
        <s agr> = <vp agr>,
        <vp unbound subject> = <np qlf>,
        <s qlf> = <vp qlf>)
RULE(vp --> *verb* pp;                 // lives in Amsterdam
        <verb agr> = <pp agr>,
        <vp agr> = <verb agr>,
        <vp unbound subject> = <verb unbound subject>,
        <vp unbound object> = <verb unbound object>,
        <vp unbound object> = <pp qlf>,
        <vp qlf> = <verb qlf>)
RULE(pp --> *prep* np;                  // in Amsterdam
        <pp agr> = <prep agr>,
        <pp agr> = <np agr>,
        <pp qlf> = <np qlf>)
```

Given the above specification the least upper bound matrix for the types, the lexicon and the grammar look like this:

```
             LUB |  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47
             -----------------------------------------------------------------------------------------------
  numbertype  26 |  26
    singular  27 |  27  27
      plural  28 |  28   2  28
  persontype  29 |   2   2   2  29
       first  30 |   2   2   2  30  30
      second  31 |   2   2   2  31   2  31
       third  32 |   2   2   2  32   2   2  32
agreementtype33 |   2   2   2   2   2   2   2  33
constituent  34 |   2   2   2   2   2   2   2   2  34
   thirdsing  35 |   2   2   2   2   2   2   2   2  35  35
           s  36 |   2   2   2   2   2   2   2   2  36   2  36
          np  37 |   2   2   2   2   2   2   2   2  37   2   2  37
          vp  38 |   2   2   2   2   2   2   2   2  38   2   2   2  38
        noun  39 |   2   2   2   2   2   2   2   2  39   2   2  39   2  39
        verb  40 |   2   2   2   2   2   2   2   2  40   2   2   2   2   2  40
          pp  41 |   2   2   2   2   2   2   2   2  41   2   2   2   2   2   2  41
        prep  42 |   2   2   2   2   2   2   2   2  42   2   2   2   2   2   2   2  42
  propernoun  43 |   2   2   2   2   2   2   2   2  43   2   2  43   2  43   2   2   2  43
    location  44 |   2   2   2   2   2   2   2  44   2   2   2   2   2   2   2   2   2   2  44
  inlocation  45 |   2   2   2   2   2   2   2  45   2   2   2   2   2   2   2   2   2   2  45  45
  transitive  46 |   2   2   2   2   2   2   2   2  46   2   2   2   2   2  46   2   2   2   2   2  46
      person  47 |   2   2   2   2   2   2   2   2  47   2   2  47   2  47   2   2   2   2   2   2   2  47
             -----------------------------------------------------------------------------------------------


Lexicon:
---------------------
Amsterdam [agr: location [num: singular
                          pers: third
                         ]
           qlf: exists [op1: @1  Location
                        op2: locationname [op: concat [op1: @1
                                                       op2: Amsterdam
                                                      ] ]
                       ]
          ]

Jan [agr: agreementtype [num: singular
                         pers: third
                        ]
     qlf: exists [op1: @1  Person
                  op2: personname [op: concat [op1: @1
                                               op2: @2  Jan
                                              ] ]
                 ]
     unbound:  [Name: @2   ]
    ]

in [agr: inlocation [num: numbertype
                     pers: persontype
                    ] ]

lives [agr: location [num: singular
```

```
                      pers: third
                     ]
       agrobj: bottom
       qlf: livesin [op: concat [op1: @1  Subject
                                 op2: @2  Object
                                ] ]
       unbound:  [Subject: @1
                  Object: @2
                 ]
      ]


Grammar -- 3 rules
--------
vp --> *verb* pp [vp: vp [agr: @1  agreementtype [num: numbertype
                                                  pers: persontype
                                                 ]
                          unbound:  [subject: @2
                                     object: @3
                                    ]
                          qlf: @4
                         ]
                  verb: verb [agr: @1
                              agrobj: bottom
                              unbound:  [subject: @2
                                         object: @3
                                        ]
                              qlf: @4
                             ]
                  pp: pp [agr: @1
                          qlf: @3
                         ]
                 ]

pp --> *prep* np [pp: pp [agr: @1  agreementtype [num: numbertype
                                                  pers: persontype
                                                 ]
                          qlf: @2
                         ]
                  prep: prep [agr: @1   ]
                  np: np [agr: @1
                          qlf: @2
                         ]
                 ]

s --> np *vp* [s: s [agr: @1  agreementtype [num: numbertype
                                             pers: persontype
                                            ]
                     qlf: @3
                    ]
               np: np [agr: @1
                       qlf: @2
                      ]
               vp: vp [agr: @1
                       unbound:  [subject: @2   ]
                       qlf: @3
                      ]
              ]

--------
```