Annotation Grammars and their compilation into Annotation Transducers

Magisterarbeit im Fach
Allgemeine Sprachwissenschaft und Nebenfächer
Universität Tübingen

Master's thesis in the subject of
"Allgemeine Sprachwissenschaft und Nebenfächer"
University of Tuebingen

Holger Wunsch

April 27, 2003

Hiermit versichere ich, dass ich diese Magisterarbeit selbständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benützt habe.

Tübingen, den 27. April 2003 Holger Wunsch

Contents

Chapter 1

Introduction

The major part of information available today will be, at some point, represented in electronic form. It is therefore a natural step to use computers not only to store and manually edit the information, but also to automatically process the data to acquire information relevant to a specific task. The advantages are obvious: Using computers, large amounts of information can be processed in relatively short time.

In this context, two terms are important: *Information Retrieval (IR)* is an intelligent type of search: A set of documents is searched for a certain search term specified by the user; the documents which match the search term are returned to the user. It is up to the user to browse through the documents returned, to extract whatever information is needed and to represent it in a suitable way (a summary for a presentation, for example).

Information Extraction (IE) goes one step further. Information is restructured according to the user's needs by the system itself. An IE system will generate its own internal representation from the information it retrieved, and then transform this information according to the user's needs. An IE system might be capable of automatically generating the presentation itself. The results of a very sophisticated IE system could be fed into a natural language generator, that creates a new text based on the IE data in a different language, a way to build automatic translation tools.

Over the last decades quite an amount of research was spent on IR/IE technologies, improving the performance of existing IR/IE systems step by step. These systems are not compatible to each other, making it hard, sometimes impossible, to easily incorporate successful parts of one system into another, minimizing the well known effect of “reinventing the wheel over and over again”. There were several research projects that worked on developing standard architectures for IR/IE systems. One of these projects was the TIPSTER project, which will be described in a later chapter.

TIPSTER proposes a modular architecture for IE systems. All modules make use of well-defined interfaces to communicate with each other. This way, exchanging or adding new modules becomes quite easy. It turns out that such a modular architecture has been chosen for most existing IE systems irrespective of their TIPSTER conformance.

This thesis evolved in a cooperation of the University of Tübingen and IBM Germany Development’s TAF project. The TAF software (TAF is an acronym for “Text Annotation Framework”), provides the infrastructure needed in an IE system to gather and manipulate data about texts, which is represented in annotations.

While on higher levels, the processing of the information represented in annotations is probably language-independent, this is certainly not the case on the lower-level where basic linguistic processing (part of speech tagging, for example) occurs. But also more complex tasks, like the detection of company names in noun phrases are highly language-dependent. Company names frequently consist of syntactically complex compounds - and of course they occur in noun phrases with different syntax in different languages. The fact that a large amount of an IE system is highly language-dependent makes it difficult to adapt the system to new languages. Therefore tools are desirable that make the development of language independent parts in IE systems easier.

The approach taken in TAF to solve this problem was to integrate a mechanism that doesn’t require language dependent parts that manipulate annotations on text to be hardwired in the software, but instead implemented in sets of transducers that can be executed by the system at runtime. Such transducers are hard to implement and maintain, too, of course. The goal of the project described in this thesis was to pursue an idea well known from formal language theory, namely that languages accepted by transducers can be described by grammars (for certain classes of languages). There are two advantages of such an approach: First, it is much easier for humans to read and understand grammars than transducer descriptions or program code. Second, people developing grammars don't need to be familiar with programming languages - and linguists, who will frequently participate in language-dependent parts of IE systems are familiar with grammar formalisms.

This formalism was called *Annotation Grammars* due to the fact that the grammars actually describe manipulations on annotations. As part of the thesis work, the author implemented a compiler that translates annotation grammars into annotation transducers. The system has been deployed at IBM Germany in a number of projects of named entity recognition.

Chapter 2

Formal Foundations

In the chapters that follow, we will introduce the central components of the annotation grammar formalism: annotations, annotation transducers and annotation grammars. Essentially, all of these are based upon well-known formal concepts that we are going to discuss in this chapter.

The basic entities in this formalism are *annotations*, which will be the subject of chapter 3. Annotations contain data about linguistic objects, and are special kinds of typed feature structures. The logic of typed feature structures is discussed in-depth in Carpenter [Car92].

Similar to other types of transducers, like finite state transducers or pushdown transducers, *annotation transducers* (chapter 4) read in sequences of annotations and based on whether the sequence read can be matched or not, new annotations are created. Annotation grammars (chapter 5) will be used to describe annotation transducers, in a way similar to how context-free grammars describe pushdown automata.

So two basic concepts are the formal foundation in this formalism: Typed feature structures and context-free grammars. This chapter will deal with these two concepts.

2.1 Type Inheritance Hierarchies

Chapter 3 will give an overview of different ways of representing annotations. In the annotation grammar formalism, typed feature structures will be used to represent annotations.

In order to introduce typed feature structures, first a notion is needed what a *type* is and how types are related to each other.

According to Carpenter [Car92], a type inheritance hierarchy is a finite set Type of types that are ordered according to their specificity. A type τ is more specific than a type σ from the same type inheritance hierarchy if τ inherits information from σ. In such a case, we say that σ *subsumes* τ and write $\sigma \sqsubseteq \tau$. If $\sigma \sqsubseteq \tau$, σ is a *supertype* of τ, and τ is a *subtype* of σ.

The subsumption relation can be represented by ISA arcs between supertypes and subtypes. The full subsumption relation is the reflexive transitive closure of all ISA arcs.

We assume a most general type (which is the least specific type), called *bot*, written $\bot$, where $\bot \sqsubseteq \tau$ for all $\tau \in \mathsf{Type}$.

2.2 Typed Feature Structures

Definition 2.2.1 *A* feature structure *over a type inheritance hierarchy* $\langle \mathsf{Type}, \sqsubseteq \rangle$ *and a finite set* Feat *of features is a tuple* $F = \langle Q, \bar{q}, \theta, \delta \rangle$*, where*

- *Q: a finite set of* nodes *rooted at* $\bar{q}$
- *$\bar{q}$: the* root *node*
- $\theta : Q \to \mathsf{Type}$*: a total node* typing *function*
- $\delta : Q \times \mathsf{Feat} \to Q$*: a partial* feature value *function*

A feature structure can be conceptualized by a labeled rooted directed graph. Q is the set of nodes, θ determines the labels on the nodes, and there is an arc between two nodes q and q' labeled f if $\delta(f, q)$ is defined and $\delta(f, q) = q'$.

2.2.1 Paths

Definition 2.2.2 *A* path *is a sequence of features,* $\mathsf{Path} = \mathsf{Feat}*$. *The empty path containing no features is* ϵ. *The feature value function is extended for paths so that* $\delta(\pi, q)$ *is the node that can be reached by following the features in the path* π *from* q*:*

- $\delta(\epsilon, q) = q$
- $\delta(f\pi, q) = \delta(\pi, \delta(f, q))$

In the annotation grammar formalism, cyclic feature structures are not allowed. That is, for a node q, there may be no path π such that $\delta(\pi, q) = q$.

2.2.2 Subsumption

In type inheritance hierarchies, if a type τ is subsumed by a type σ, then τ inherits information from σ, being more specific at the same time. This concept is extended to feature structures.

We assume a fixed type inheritance hierarchy $\langle \mathsf{Type}, \sqsubseteq \rangle$ and a finite set of features Feat.

Definition 2.2.3 $F = \langle Q, \bar{q}, \theta, \delta \rangle$ subsumes $F' = \langle Q', \bar{q}', \theta', \delta' \rangle$ *if and only if there is a total function* $h : Q \to Q'$, *called a* morphism *such that:*

- $h(\bar{q}) = \bar{q}'$
- $\theta(q) \sqsubseteq \theta'(h(q))$ *for every* $q \in Q$
- $h(\delta(f, q)) = \delta'(f, h(q))$ *for every* $q \in Q$ *and feature* f *such that* $\delta(f, q)$ *is defined*

In the annotation grammar formalism, subsumption of feature structures (or annotations, respectively) will play an important role in the way how matching of sequences of annotations will be defined in annotation transducers (see section 4.2.1). Derivations of sequences of annotations in annotation grammars will also be based on subsumption.

2.2.3 Appropriateness

So far, there have been no further restrictions on feature structures except that every node has to be typed. Especially, any feature from Feat may occur on any node regardless of its type. An *appropriateness specification* over the inheritance hierarchy defines which features may and may not occur on a node with a certain type (see Carpenter [Car92], p. 86).

Definition 2.2.4 *An* appropriateness specification *over the inheritance hierarchy* $\langle \mathsf{Type}, \sqsubseteq \rangle$ *and features* Feat *is a partial function* $Approp : \mathsf{Feat} \times \mathsf{Type} \rightarrow \mathsf{Type}$ *that meets the following conditions:*

- (Feature Introduction)
  for every feature $f \in \mathsf{Feat}$*, there is a most general type* $Intro(f) \in \mathsf{Type}$ *such that* $Approp(f, Intro(f))$ *is defined.*
- (Upward Closure / Right Monotonicity)
  if $Approp(f, \sigma)$ *is defined and* $\sigma \sqsubseteq \tau$*, then* $Approp(f, \tau)$ *is also defined and* $Approp(f, \sigma) \sqsubseteq Approp(f, \tau)$.

The first condition, feature introduction, ensures that for any feature, there exists one most general type which the feature is appropriate for. This is necessary for correct type inference: given only a single feature it must be possible to infer one unique most general type which this feature is appropriate for. The second condition requires that all features that are appropriate for a type σ are also appropriate for all subtypes of σ, which is, as Carpenter points out, an intuitive property of subtypes.

2.2.4 Well-Typedness

Definition 2.2.5 *A feature structure* $F = \langle Q, \bar{q}, \theta, \delta \rangle$ *is said to be* well-typed *if whenever* $\delta(f, q)$ *is defined,* $Approp(f, \theta(q))$ *is defined, and such that* $Approp(f, \theta(q)) \sqsubseteq \theta(\delta(f, q))$.

Let $\mathcal{LF}$ *denote the collection of well-typed feature structures.*

Informally, in a well-typed feature structure, any feature that occurs on a node q must be appropriate for the type $\theta(q)$ and furthermore, the value of the feature must be of a type that is subsumed by the value-type declared in the appropriateness specification for the feature.

2.2.5 Total Well-Typedness

Definition 2.2.6 *$F = \langle Q, \bar{q}, \theta, \delta \rangle \in \mathcal{F}$ is* totally well-typed *if and only if it is well-typed and if $q \in Q$ and $f \in \mathsf{Feat}$ are such that $Approp(f, \theta(q))$ is defined, then $\delta(f, q)$ is defined.*

Total well-typedness requires that in addition to being well-typed, a feature structure must contain all features that are appropriate.

2.3 Context-free grammars

In this section, we will give a brief summary of standard context-free grammars over characters. A comprehensive discussion of CFGs of this kind can be found in Hopcroft, Motwani and Ullman [HMU01]. Chapter 5 will describe the properties of annotation grammars, which are context-free grammars with a few extensions.

A context-free grammar is a 4-tuple $G = \langle V, T, P, S \rangle$, where V is the set of *variables* (or *nonterminals*), T is the set of *terminals*, $P \subset V \times (T \cup V)^*$ is the set of *production rules*, and finally $S \in V$ is the *start symbol*. Furthermore, $V \cap T = \emptyset$. Any rule in P contains a left-hand side that consists of one symbol from V, and a right-hand side which consists of a string of both nonterminals and terminals. A rule with a symbol A on its left-hand side is called an *A-rule*. A symbol $A \in V$ on the right-hand side of a rule can be replaced with the string on the right-hand side of any A-rule in the grammar.

The *language $L(G)$ denoted* by the grammar G is the set of words that can be generated by the grammar. The definition of the language is based on a relation between two strings, called the *derives relation*. Two strings $\alpha A \beta$ and $\alpha \gamma \beta$

both from $(T \cup V)^*$ are in the derives relation, written $\alpha A\beta \underset{G}{\Longrightarrow} \alpha\gamma\beta$, if in G, there is an A-rule of the form $A \rightarrow \gamma$. Informally, the elements in the derives relation are those pairs of strings where a symbol in the first string was replaced by a substring in the second string, driven by an appropriate rule in the grammar.

Commonly, the derives relation is extended to its reflexive transitive closure, written $\overset{*}{\underset{G}{\Longrightarrow}}$, which also contains the pairs of strings that evolved from each other by multiple rewrite-steps.

The language $L(G)$ denoted by the grammar can now be defined to be the set of words that the start symbol derives: $L(G) = \{w \mid w \in T^*, S \overset{*}{\underset{G}{\Longrightarrow}} w\}$.

From a more procedural point of view, a derivation of a word in a grammar can be interpreted in two directions, either *top-down* or *bottom-up*. The top-down view is more a generative view: beginning with the start symbol, substrings are replaced by applying rules, ending up with a word. The bottom-up view is much more like a test: substrings in an existing word are replaced by nonterminals by applying appropriate rules "in reverse". This process ultimately yields the start symbol if the word was in the language of the grammar. The second way is how many parsers work: strings of symbols are grouped into higher-order categories, until the top-most category is reached.

An important property of CFGs is that the alphabet of nonterminals V and the alphabet of terminals T are two disjoint sets. That is, a nonterminal can be distinguished from a terminal in all cases. Furthermore, nonterminals never occur in words in $L(G)$ (they do occur in intermediate steps of a derivation, called *sentential forms*). As we will see, annotation grammars do not distinguish between "sentences" and "sentential forms".

Chapter 3

Annotations

The basic entities in this thesis are *annotations*. Annotations contain *linguistic data* about portions of a given document. In our context, the document will mostly be a text like a newspaper article, but in general it might also be a speech recording or a video film, for example.

Linguistic data can basically be anything of interest during linguistic processing. To perform a lexical analysis of a text, it is useful to have information about the parts of speech of all the words in the text, or their morphologic properties. If one is interested in the syntax of a text, it might be necessary to mark all the heads of noun-phrases, or the origin of a movement operation. For speech signals, a textual transcription makes analysis much easier or even just possible.

In all these cases, the document is *annotated* with additional important information, just like the notes a reader sometimes scribbles on the page margins when studying a textbook.

It is obvious that such an annotation cannot exist on its own; it must always be interpreted relative to the portion of the document about which it contains data. So regardless of how an annotation is represented, it has two properties: Its start and end position with respect to the document. For speech signals this might be points of time, for strings, this could be the positions of characters in the string. Later on in this thesis, we will restrict ourselves to annotations on written text in electronic form only. An annotation can be imagined to be a labeled arc between a start and an end point, where the label stands for the information contained in the annotation.

Annotations are widely used in linguistics, and there exist a large number of systems that are based on annotations, and almost just as many formats and strategies of how to store and manipulate annotations. In the following sections we will discuss basic ways of how annotations are represented. Then we will introduce two of these systems, ATLAS and JAPE, which are quite similar to the Annotation Grammar formalism.

3.1 Basic representation models of annotations

As described in the introduction, annotations contain data about linguistic entities. In addition to the data, they have two additional properties: a start and an end position. In actual applications, there are various ways of how to represent annotations, most of which fall in two basic groups: *additive* and *referential* annotations. We will discuss both types in what follows.

3.1.1 Additive annotations

An *additive* annotation is directly inserted into the text using an appropriate notation that allows readers (and computers) to distinguish between the text and the information about the text.

Bracketing

Annotations in their simplest form consist of brackets that are inserted into the text. In the following example, brackets are used to mark noun phrases:

[Brackets in this example] mark [noun phrases].

The opening brackets indicate the start position, and the closing brackets the end position of the respective annotation. The data contained in the annotation is not represented explicitly at all, but only by our definition that bracketed constituents are noun phrases. Obviously, this is not very powerful, especially because this way, only one type of information can be encoded without confusion. Extending the bracket notation with proper symbols that explicitly encode the information can solve this problem:

$[_{NP}$Brackets in this example$]_{NP}$ $[_{V}$mark$]_{V}$ $[_{NP}$ noun phrases and verbs$]_{NP}$.

Bracket notations are convenient in applications that analyze text serially, for example by using finite-state recognizers and transducers. An example for such an application that makes use of very complex bracket symbols can be found in Kaplan & Kay [KK94]. This article shows that apparently context-sensitive phonological rules can in fact be implemented by finite-state transducers (as long as rules are not applied on their own result). Brackets are inserted into the input at positions where rules applications are allowed. Then finite-state transducers are run over the input annotated this way, replacing phonemes between brackets. The brackets here annotate the context that licenses the application of one specific rule on the phonemes between the context.

Markup

Another way to represent additive annotations is by means of markup as used in languages derived from SGML, like HTML or XML. Of course there are many ways of encoding an annotation in a markup language (after all, this flexibility is one reason why markup languages became so popular), but one possibility would be to have the annotation expressed by a pair of tags, where the start tag is placed at the start position, and the end tag is placed at the end position. The actual information could be encoded as an attribute on the start tag:

```
<?xml version="1.0"?>
<text>
  <word pos="noun">Words</word>
  <word pos="prep">in</word>
  <word pos="det">this</word>
  <word pos="noun">text</word>
  <word pos="aux">are</word>
  <word pos="verb">annotated</word>
  <word pos="prep">with</word>
  <word pos<"noun">markup</word>
  <punct type="period">.</punct>
</text>
```

The above example shows how text can be annotated using XML. Every annotation is represented by a pair of tags, where the start tag indicates the start position of the annotation, and the end tag indicates the end position. The actual data contained in the annotation is represented by attributes on the start tag.

Evaluation

Using an additive approach for annotations as discussed in the previous sections has several advantages:

- **Readable**
  Both brackets and markup are, at least up to a certain complexity, human readable. Additional software that is capable to display annotations in an understandable way is not necessarily required.
- **Editable**
  It is not too difficult to manually insert additional annotations. To add a new annotation, brackets or markup have to be inserted properly at the positions where the annotation starts and ends. This can be done using an ordinary editor. Because start and end positions of annotations are encoded implicitly by the position of the markup, all subsequent annotations just move on with the text.
- **Platform independent**
  Almost any system can process text files. It is easy to interchange additively annotated text provided the annotation scheme is well documented.

However, there are also a number of disadvantages:

- **Restricted to text**
  Brackets or markup can only be used in text files. This is because they must be physically inserted into the text to be annotated. Text files may be encoded using various character encoding schemes (Unicode, ASCII etc.), annotations must use the same encoding. It is not possible to insert additive annotations to other type of input data (like speech recordings), since this would corrupt the original data.

- **Requires serial processing**
  Since additive annotations are integrated into the text flow, they must be processed in a serial manner. This is inefficient[1].

- **Alphabet conflicts**
  The fact that annotations are stored as part of the text requires that annotations be encoded with the same character set as the text itself. In order to clearly distinguish annotations from text, it it necessary to define a unique notation for annotations, which might be difficult when using a small character set.

3.1.2 Referential annotations

A *referential* annotation is stored separately from the data about which it contains information. The annotated data remains unchanged, the start and end positions are stored within the annotations by references into the annotated data.

[1]Actually, this issue can also be found in many systems that use XML - in order to get random access to all parts of a document, the complete document must be present in memory, which is often not possible because of the document's size. The solution is to analyze the document serially, which requires more processing time. In fact, this is also reflected in what types of XML parsers are available: DOM parsers (Document Object Model, [W3C02]) that expose a DOM tree of the document after completely parsing it, or SAX parsers (Simple API for XML, [Bro02]) which return every element in the sequence found. A widely used suite of DOM and SAX parsers can be found on the homepage of the Apache Xerces project [Apa02].

The TIPSTER model of annotations

An example for a referential annotation model is the one used in the TIPSTER architecture [Gri98], which we will introduce in this section. TIPSTER [NIS00] was a research program funded by the US Defense Advanced Research Projects Agency (DARPA) and several other US government agencies.

> "In its efforts to improve document processing efficiency and cost effectiveness TIPSTER focused on three underlying technologies.
>
> - **Document Detection**: the capability to locate documents containing the type of information the user wants from either a text stream or a store of documents.
> - **Information Extraction**: the capability to locate specified information within a text.
> - **Summarization**: the capability to condense the size of a document or collection while retaining the key ideas in the material" [NIS00].

The TIPSTER architecture uses an object-oriented approach. This means that all objects and manipulations on these objects needed to solve the aforementioned tasks are implemented as classes. Thus annotations in TIPSTER are instances of a class `Annotation`, which we will discuss in greater detail in a moment.

An annotation provides information about a certain portion of a document, which is specified by a sequence of *spans*. A span is an instance of a class `Span`, which is defined as follows:

Class Span
    Properties
        Start: Integer
        End: Integer
    Operations
        CreateSpan(start:integer, end:integer): Span

A span has two properties: 'Start' and 'End', which specify the beginning and the end of the span. The values are byte positions in the "raw data", which is the document in its original format, without any annotations or modifications. The "CreateSpan" operation is, using OOP terminology, a constructor that creates new instances of spans.

We will now move on to the actual structure of annotations in the TIPSTER framework. Annotations are instances of a class `Annotation`. Its definition is:

Class Annotation
    Type of AttributedObject
    Properties
        Id: string
        Type: string
        Spans: sequence of Span
    Operations
        CreateAnnotation(Type: string, Spans: sequence of Span,
            attributes: sequence of Attribute): Annotation

An annotation is a descendant of an `AttributedObject`, which is an object capable of maintaining a list of attributes, or features, and their values. The information contained in annotations is represented by means of attributes. The `Annotation` class has a property 'Spans', which is a sequence of spans. Using sequences of spans, one individual annotation can refer to multiple positions in the document, allowing for discontinuous linguistic elements, such as verb plus participle pairs, like in "I *gave* my gun *up*"[2].

Furthermore, the `AnnotationSet` class is defined in the TIPSTER architecture. This class provides a wide range of operations on sets of annotations, among those adding and removing annotations to or from a set, or iterating through a set of annotations.

We briefly mentioned that attributes and their values represent the actual information in annotations. It is necessary to define clearly what attributes and values will occur in an annotation representing one kind of information. *Annotation type declarations* serve this purpose. An annotation type declaration is a description of what attributes will occur on a special type of annotation, and what values these attributes will take on. Annotation type declarations are

[2]The example is taken from Grishman [Gri98], p. 15.

arranged in *type packages*, in which annotations of related types are described together.

TIPSTER annotations use `Span` instances to refer to their positions in a document. This way, the document itself remains completely unchanged. This is the most important difference to additive models, where manipulation of annotations always implies manipulation of the document.

Annotation Graphs

Another formalism that uses referential annotations is the *annotation graph model* by Bird et al. ([BL99], [BB00]). The goal of the model is to introduce a general mathematical foundation for the many different existing annotation formalisms.

An annotation graph can be formally defined as follows:

Definition 3.1.1 ([BL00]) *An **annotation graph** G over a label set L and timelines $\langle T_i, \leq_i \rangle$ is a 3-tuple $\langle N, A, \tau \rangle$ consisting of a node set N, a collection of arcs A labeled with elements of L, and a time function $\tau : N \rightharpoonup \bigcup T_i$, which satisfies the following conditions:*

1. *$\langle N, A \rangle$ is a labeled acyclic digraph containing no nodes of degree zero*
2. *for any path from node n_1 to n_2 in A, if $\tau(n_1)$ and $\tau(n_2)$ are defined, then there is a timeline i such that $\tau(n_1) \leq_i \tau(n_2)$.*

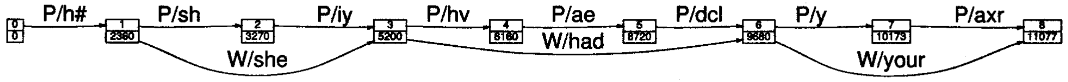
Most of the time, a large number of annotations is added to a text, some of which are to be interpreted as sequences of annotations of the same type (for example, annotations containing part of speech information about each word), and other groups of annotations that contain other aspects of information about the same text. Annotation graphs focus on this view of sets of annotations. An individual annotation is represented in an annotation graph by an arc between two nodes. The linguistic data contained in the annotation can be found in the arc label[3].

[3] A suitable structure to represent the data in the label must be found from application to application.

The time function τ assigns to each node a point in a timeline. A timeline is a formal concept with the property that the points on a timeline are totally ordered. In many cases the points on a timeline will be, as the name suggests, real points of time; however they need not be. They could also be character positions in a text, or sample numbers in an audio recording. Start and end positions of annotations are encoded by points on such a timeline in this formalism: An arc connecting two nodes n_i and n_j represents an annotation with the start position $\tau(n_i)$ and the end position $\tau(n_j)$.

The definition of annotation graphs is kept very general: The time function is a partial function, that is, not all nodes need to be assigned to a timeline. Most of the time, however, all nodes will be assigned to a timeline, in which case an annotation graph is called *totally anchored*. Furthermore, more than one timeline may exist. Timelines are the links between the abstract annotation structure and the actual structure of the annotated data. By allowing multiple timelines in a single annotation graph, it is possible to relate different types of annotated data to each other. This way, a speech recording (with timeline units in milliseconds) and its transcription (with timeline units in string positions) can be annotated in the same graph, clearly showing the relations between the annotations.

In most cases however (including the ones discussed in this thesis), only one timeline is needed. The following example (taken from Bird [BL00]) shows an annotation graph that contains annotations to the TIMIT corpus of read speech [GLF$^+$86].



Two types of annotations can be found in this graph: Annotations of type P contain transcriptions of spoken language, annotations of type W contain the spoken words in written form. The boxes indicate nodes. Each node has a unique ID (numbers in the upper half of the boxes). The numbers in the lower half are the values the time function τ assigns to each node. There is only one timeline in this example, and the values on the timeline are the numbers of the samples where the respective sound starts in the audio file (which was sampled at 16kHz).

Evaluation

The advantages in using a referential approach are:

- **Any type of document can be annotated**
  Annotations of the referential type don't have to be inserted into a document. This way, the representation of the annotation is independent of the format of the document, which makes it possible to annotate documents like sound recordings, videos etc.

- **Efficient processing**
  For referential annotations to process, it is not necessary to serially parse a document. Instead, annotations can be directly accessed using appropriate accessor functions provided a suitable way of storage.

On the downside, using a referential model may have the following drawbacks:

- **Difficult to read**
  The storage structure of referential annotations is likely to be in a format easily processed by computers, but less readable to humans. Because annotations are not stored in the same position they refer to, it can be hard to find out about where an annotation belongs.

- **Modifcations to document requires recalculation**
  Modifications to the annotated document requires that all start and end positions in annotations be properly readjusted. In large texts, this is nearly impossible without proper tools.

Comparing the points mentioned above with the evaluation in the previous section about additive annotation models, it seems that where one approach has its strengths, the other approach has its weaknesses. Which approach is ultimately chosen depends on the application.

3.2 Existing Annotation Systems

3.2.1 ATLAS

ATLAS (A Flexible and Extensible Architecture for Linguistic Annotations, [BDG$^+$00]) is an API to manipulate annotations. Its goal is to provide a unified API for handling annotations in order to facilitate application development.

ATLAS consists of three levels: the logical level, the physical level, and the application level.

- The API is placed on the **logical level** which provides access to the underlying formalism which is a generalization of the annotation graph model described in section 2. The API adapts the TIPSTER API for annotations, however internally, annotation graphs are used.
- The **physical level** implements persistence for objects on the logical level. Persistence is the ability of an object to be saved to or loaded from a storage system (e.g. disk or database).
- The **application level** is not actually part of ATLAS but rather consists of the applications that make use of the ATLAS API.

On the logical level, annotations and annotation graphs are represented by respective Java classes. The `Annotation` class, for example, provides methods to set the start and end positions of an annotation, or to set a feature in an annotation to a value. The `AnnotationSet` class contains methods to add or remove annotations to or from an annotation graph.

3.2.2 GATE and JAPE

The Java Annotation Patterns Engine [CMT00] is a system that can be used to describe manipulations of annotations. JAPE is part of GATE (General Architecture for Text Engineering, [CBPW00]), which was developed at the University of Sheffield. GATE is general-purpose development system for linguistic tasks (see also Cunningham [Cun00]), designed on the basis of the TIPSTER architecture. JAPE is a compiler that translates rules contained in a JAPE grammar into a set of Finite State Automata.

A rule in a JAPE grammar consists of two parts. The left-hand side is an expression that describes a sequence of annotations to be matched. The right-hand side consists of an action to be executed if the left-hand side matched. The following example of a JAPE rule is taken from Cunningham [CMT00]:

```
1  Rule:  NumbersAndUnit
2  (( {Token.kind == "number"} )+:numbers {Token.kind == "unit"})
3  -->
4  :numbers.Name = {rule = "NumbersAndUnit"}
```

This rule matches a sequence consisting of at least one number-token followed by a number token, then creates an annotation across the span of the numbers, setting the RULE feature of this annotation to `NumbersAndUnit`. Let us inspect this rule in greater detail.

- Rules are named in JAPE. In line 1, the name of the rule, `NumbersAnd-Unit`, is given.
- The left hand side of the rule is shown on line 2. The first part of the rule, `({Token.kind == "number"})+:numbers`, describes the sequence of at least one number-token. A token is represented by annotations of type `Token`[4], where KIND is a feature of this annotation. Its value, `number`, indicates that the token is a number token[5]. The == operator is the equality operator (like in C or JAVA). The + operator is the Kleene plus operator that allows for the repetition of the token. The `numbers` symbol following the colon is a *label*. Labels can be used to name spans that were matched. The label `number` names the span of all number-tokens matched.

  The second part of the rule, `{Token.kind == "unit"}` works analogously, requiring tokens of kind `unit`.
- Line 4 shows the right-hand side of the rule. Here, this is a simple ac-

[4]Indentifiers are case-sensitive

[5]Obviously, annotations resemble typed feature structures, but also `record` types in PASCAL or `structs` in C.

tion to create an annotation of type `Name`, with its feature `rule` set to `NumbersAndUnit`, that spans over the region previously labeled `numbers`.

Note the use of the assignment operator = in multiple contexts in line 4. While the left = assigns whole annotations, the right assignment operator assigns values to features. The expression `rule = "NumbersAndUnit"` is automatically expanded to a full annotation.

The "." operator is also used in two contexts: It serves as a feature selection operator (similar to C++'s member selection operator, and it is used to indicate the span over which an annotation is created (in line 4).

The reason for this multiple use of operators is that annotations as well as spans are JAVA classes. Features of annotations are data members of an annotation class, and spans are also classes which have a member which is the annotation at the position the spans represents.

In the example above, the action on the right-hand side was very simple. Actually, JAPE allows the developer to include blocks of arbitrarily complex Java code that targets the GATE API, thus making it possible to express a wide range of operations from within the grammar[6].

The JAPE application itself is a compiler that translates a grammar into a large nondeterministic Finite State Automaton. The general structure is shown in figure 3.1. The arcs that leave the initial state are all epsilon arcs, each of which leads to a sub automaton that matches the sequence on the left-hand side of the corresponding rule. To the final states of all FSAs, the action specified on the right-hand side of the corresponding rule is attached. An action is executed if the FSA reaches the final state to which it is attached.

Note that the nondeterministic FSA is just an intermediary step; the final result of the compilation is a deterministic FSA with a sophisticated matcher to eliminate multiple comparisons of shared substructures on arcs. See Cunningham [CMT00] for the complete presentation of JAPE.

[6]This is a similar concept to the one used in compiler-compilers like JFlex [Kle02] or CUP [Hud02], where actions may (and are supposed to) include Java code.

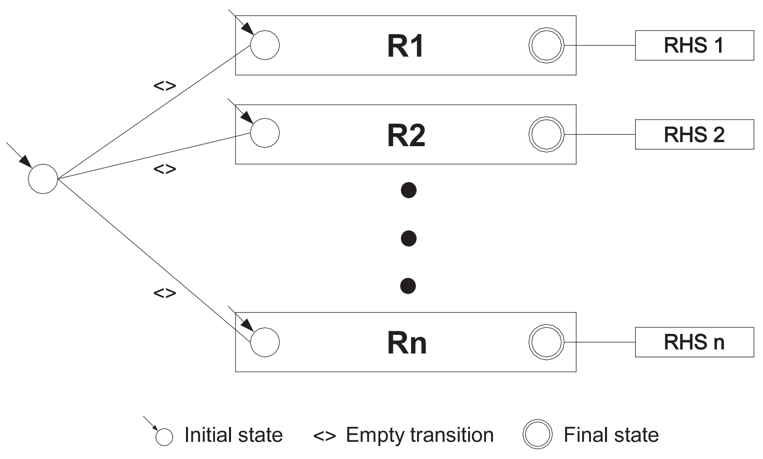


Figure 3.1: FSA generated from a JAPE grammar

3.3 Representation of annotations

In the previous section we introduced the two basic types of annotations, additive and referential, and several instances of these. We will now move on to how annotations will be represented in the remainder of this thesis.

In the annotation grammar framework, annotations are represented by triples consisting of a start position, an end position, and a well-typed feature structure that contains the data associated with the range in the string the annotation spans over. The start position and the end position are to be interpreted as character offsets with respect to the annotated input text. Essentially, we adapt a referential, TIPSTER-like model of annotations.

Formally, annotations can be defined as in definition 3.3.1. We assume a fixed type inheritance hierarchy $\langle \mathsf{Type}, \sqsubseteq \rangle$ and a finite set of features Feat (as described in chapter 2).

Definition 3.3.1 *An* ***annotation*** *is a triple $\langle b, e, \phi \rangle$, where $b \in \mathbb{N}$ is the* ***start position*** *of the annotation, $e \in \mathbb{N}$ is the* ***end position****, and $\phi \in \mathcal{LF}$ is a well-typed feature structure that represents the information contained in the annotation.*

We don't require the feature structures in annotations to be totally well-typed. This way, features in an annotation are not required to be present. Assume an annotation that represents company information, with features NAME and STOCKSYMBOL. It might still be feasible to annotate a company name in a text with an annotation of type *company_info*, even when no information about the company's stock symbol is available. If we required annotations to be totally well-typed, such annotations would be illegal. Another reason is the usage of annotations in descriptions (see section 3.5).

Definition 3.3.2 *A tuple $\langle \alpha_1, \ldots, \alpha_n \rangle$ of n annotations ($\alpha_i = \langle b_i, e_i, \phi_i \rangle$) assigned to the same input string is called a* ***sequence of annotations*** *if $b_i < e_i$ and $b_{i+1} \geq e_i$ for all $1 \leq i \leq n$.*

The concept of sequences of annotations is the equivalent to the concept of strings of characters. A string is a number of characters in a certain order, a sequence of annotations is a number of annotations in a certain order. The second part of the definition requires that annotations in a sequence may not overlap (by demanding that the start position of the following annotation is always greater or equal than the end position of the preceding annotation).

Our annotation model is very close to the TIPSTER model of annotations. In both models, the data is represented by a set of attribute/value pairs, where it is well defined what values an attribute may take on, and what attributes may occur in what types of annotations. In TIPSTER, this is accomplished by annotation type declarations. Our formalism uses typed feature structures. As described in chapter 2 on the formal foundations, typed feature structures are built over a type inheritance hierarchy, so typing is inherent in this formalism. Both models are referential models that explicitly represent the start and end positions as a property of annotations.

3.4 Example

For an example let us assume that the underlying input string is

an example

which is to be annotated with parts-of-speech. We first need a type inheritance hierarchy that defines the types and the features appropriate for these types.

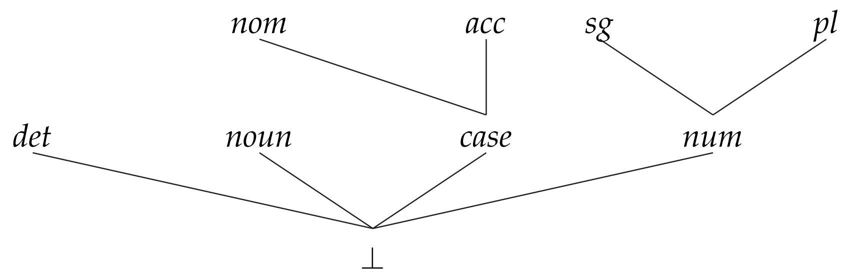


Figure 3.2: Example type inheritance hierarchy

In addition to the plain type inheritance hierarchy, appropriateness specifications are needed.

Type	**Appropriate Features**
det	CASE: *case* NUM: *num*
noun	CASE: *case* NUM: *num*

Figure 3.3: Appropriatness specifications for the example type inheritance hierarchy

As usual, there is a *bot* type (⊥) in the hierarchy. All other types are subtypes of *bot*. For each of the types *det* and *noun*, two features were declared to be appropriate: CASE, which is of type *case*, and NUM, which is of type *num*. The types *case* and *num* in turn have subtypes: *nom* and *acc* and *sg* and *pl*, respectively.

This type hierarchy declares the types necessary for two simple annotations to represent determiners and nouns.

$$_0\,\mathrm{a}\,_1\,\mathrm{n}\,_2\,\sqcup\,_3\,\mathrm{e}\,_4\,\mathrm{x}\,_5\,\mathrm{a}\,_6\,\mathrm{m}\,_7\,\mathrm{p}\,_8\,\mathrm{l}\,_9\,\mathrm{e}\,_{10}$$

The subscript numbers to the left and to the right of each character indicate string positions: The first 'a' starts at position 0 and ends at position 1, followed by an 'n' character from position 1 to position 2, and so on. The start and end positions in annotations refer to these string positions.

With the given type hierarchy, annotations can be created to annotate the parts of speech of the two words in "an example":

$$\left\langle 0, 2, {}_{det}\begin{bmatrix} \text{CASE} & \textit{nom} \\ \text{NUM} & \textit{sg} \end{bmatrix} \right\rangle$$

The annotation above is of type *det* and starts at string position 0 and ends at string position 2. So it annotates the determiner "an" with data represented by the feature structure that is the third element in the triple. Note that this feature structure is well-formed with respect to the type hierarchy.

$$\left\langle 3, 10, {}_{noun}\begin{bmatrix} \text{CASE} & \textit{nom} \\ \text{NUM} & \textit{sg} \end{bmatrix} \right\rangle$$

This annotation is of type *noun* and annotates the word "example" at position 3-10 with information about a noun.

3.5 Annotation Descriptions

An *annotation description* is a textual way to describe an annotation. Annotation descriptions serve the same purpose, and are syntactically very close to, *attribute-value descriptions* discussed in chapter 4 of Carpenter [Car92].

3.5.1 Syntax of annotation descriptions

The syntax of an annotation description is given by the following grammar.

annotation-description	→	*type* ('&' *path-conjunction*)
path-conjunction	→	*path-value* |
		path-value '&' *path-conjunction*
path-value	→	*path* '=' *variable-or-value*
path	→	*feature-name* |
		path ':' *feature-name*
variable-or-value	→	*variable-name* |
		value
type	→	':=' *type-name*

The following example shows the translation of an annotation containing some simple information about noun phrases into an annotation description.

$$\left\langle 0, 10, {}_{np}\left[\begin{array}{ll} \text{NOUN} & {}_{noun}\left[\begin{array}{ll}\text{CASE} & nom\\ \text{NUM} & sg\end{array}\right] \\ \text{DET} & {}_{det}\left[\begin{array}{ll}\text{CASE} & nom\\ \text{NUM} & sg\end{array}\right]\end{array}\right] \right\rangle$$

The corresponding annotation description is:

```
:= np &
noun:case = nom & noun:num = sg &
det:case = nom & det:num = sg
```

Note that, like the annotations themselves, annotation descriptions must be interpreted with respect to a type hierarchy. Paths and values may be ommitted in annotation descriptions if they can be determined using type inference.

Furthermore, as can be seen in the grammar of the annotation description syntax above, variables are allowed in annotation descriptions. The following example

```
:= np &
noun:case = X & noun:num = sg &
det:case = X & det:num = sg
```

is the description of an annotation where the NOUN | CASE and DET | CASE features have identical values.

Start and end positions of annotations that refer to underlying input are not part of annotation descriptions.

3.5.2 Subsumption of annotation descriptions

To simplify matters, we will say that an annotation description subsumes another annotation description if the annotation described by the first description subsumes the annotation described by the second description.

3.6 Summary

- Annotations contain linguistic data about a document.
- An annotation has three properties: Its start position and its end position with repsect to the document it annotates, and the actual linguistic data.
- There are two basic types of annotations: Additive annotations are directly inserted into the document, thus modifying it. Start and end positions are implicitly repesented by the position the additive annotation occurs at in the text.

  Referential annotations are stored separately from the document. They are an indirect way of annotating, start and end positions are stored as references into the document.
- Two examples for additive annotations are bracketing and markup. The TIPSTER annotation model and annotation graphs are instances of referential annotations.
- Annotations in this thesis are represented by triples consisting of a start position, an end position and the actual data represented by a well-typed feature structure. So they are of the referential type.
- Analogously to strings of characters, we introduce the term “sequence of annotations”.
- Annotation descriptions are textual descriptions of annotations.

Chapter 4

Annotation Transducers

In the previous chapter we introduced the concept of *annotations*. Annotations are objects that contain linguistic data. Information extraction systems will use annotations to represent the data gathered. This process of gathering and inferring data will include the manipulation of the annotations that represent this data. In this chapter, we will introduce in detail the devices to manipulate annotations, called *annotation transducers*, which can be described using annotation grammars.

Annotation transducers (abbreviated AT) don't directly operate on the input text; they only operate on annotations added to the text[1]. Their purpose is to provide a fast and efficient means of a first step of extracting information from a text. ATs match sequences of annotations and create new annotations. The information contained in these new annotations is a summarization of the information contained in the annotations just matched. Other ATs are applied on these newly created annotations in a subsequent step, resulting in an even higher layer of abstraction.

The general principle is similar to a technique by Abney called *partial parsing via finite state cascades*, or *chunk parsing* [Abn96]. Unlike traditional parsers, which generate one global parse-tree for a sentence, a chunk parser only produces a parse for a part of a sentence for which it is certain about (an *island*

[1]To be fully accurate, this requires that some other device than an annotation transducer must add a first level of annotations. This could be a part-of-speech tagger, or a lexical lookup device that annotates each word in the text with canonical lexical information.

of certainty) and leaves everything unchanged. In multiple iterations, these islands of certainty grow more and more, until a complete parse was found. One way of implementing such a chunk parser is by using cascades of finite-state transducers. The first set of transducers typically operates on part of speech information generated by a tagger. The result are higher-order syntactic categories (Det N yields NP, for example). Anything that can't be matched on this level is passed on ("punted") to the next level. In a second iteration, a new set of transducers is applied on these category symbols, generating even higher-order classifications. Multiple iterations of this kind occur until a final parse could be found.

Two important properties are to be noted: First, the parser only operates on pieces of the input, and doesn't require a global match. Second, parsing is done in multiple iterations, where on each iteration, only the input on the highest level is taken into account, and input on lower levels is ignored.

Although annotations transducers don't parse (in the sense that they don't reject input if they don't match) the same properties can be found in the way annotation transducers work: They create new annotations at positions they match, and leave everything else unchanged (most important, annotating doesn't fail). Furthermore, ATs are applied in a cascade – annotations containing lower-level information are created first, in subsequent iterations annotations containing higher-level information are added.

Annotation transducers are devices very closely related to pushdown transducers [2]. A pushdown transducer recognizes context-free languages just like a pushdown automaton, but in addition to this, it can generate output symbols at each transition. ATs recognize context-free languages over annotations; however they do not contain an explicit stack like PDTs do, instead, they are capable of *calling* other annotation transducers by means of a *call-statement* on an arc. This concept is familiar from programming languages like C or PASCAL that support procedures. Compilers implement support for procedure calls and especially returns using call stacks. This is the same approach that is used for annotation transducers. Thus, the stack found explicitly in PDTs can be found in the internal handling of call statements in ATs.

[2] A discussion of pushdown transducers can be found in Hopcroft & Ullman [HU79].

4.1 Definition of annotation transducers

Definition 4.1.1 *An* annotation transducer *is a 6-tuple $AT = \langle Q, q, I, O, \delta, F, n\rangle$, where*

- *Q is the set of states.*
- *$q \in Q$ is the initial state.*
- *$I \subseteq (\mathcal{AD} \cup \mathcal{CL})$ is the input alphabet ($\mathcal{AD}$ and $\mathcal{CL}$ will be explained in section 4.2).*
- *$O \subseteq \mathcal{INS}$ is the output alphabet, the set of valid output instructions (to be discussed in section 4.3).*
- *$\delta : Q \times I \to Q \times O$ is the transition function.*
- *n is the transducer name.*

In the next section we will introduce *call-statements*, which enable an AT to transfer execution to another AT. Calling other ATs requires of course the existance of multiple ATs. It is important that these ATs are uniquely identifiable to make sure that the right AT can be called. We introduce the concept of *sets of ATs* which is designed to meet these requirements.

Definition 4.1.2 *A* set of annotation transducers *is a set of ATs $\mathcal{AS} = \{t_1, \ldots, t_n\}$ where each $t_i = \langle Q_i, q_i, I_i, O_i, \delta_u, F_i, n_i\rangle$ is an AT as defined above and furthermore for any two t_i, t_j $(i \neq j)$: $n_i \neq n_j$.*

This concept provides us with a way to group ATs that can possibly call each other, and makes sure that all ATs in the set have a unique name.

4.2 The input alphabet

The input alphabet I is a subset of the union of the two sets $\mathcal{AD}$ and $\mathcal{CL}$.

$\mathcal{AD}$ is the set of valid annotation descriptions, as defined in section 3.5.1. An annotation description on the input side of an AT denotes an annotation that must match the next annotation in the input.

$\mathcal{CL}$ is the set of valid `call`-statements. We will return to the discussion of `call`-statements shortly.

4.2.1 Match criterion for annotations

For the most familiar types of transducers, which use characters as their input alphabet, matching of symbols in the input is quite simple: A transition from one state to the next state occurs if the symbol on the input side of the arc label is equal to the symbol at the next position to be read in the input. So the match criterion is equality in these cases. Equality could also be used as the match criterion for annotation transducers: A transition from one state to the next state in the AT would occur if the annotation denoted by the description on the arc connecting these two states was equal to the annotation to be read next from the input. However, equality of annotations turns out to be infeasible, as the following example will show.

Assume an AT to match a sequence of a determiner- and a noun-annotation, as in the example in section 3.4. In order to cover both singular and plural noun phrases, one would want to leave the NUM feature in the descriptions of both annotations underspecified, so the annotations denoted by the descriptions on the arcs would have this shape:

${}_{det}\begin{bmatrix}\text{CASE} & \textit{nom}\end{bmatrix}$ and ${}_{noun}\begin{bmatrix}\text{CASE} & \textit{nom}\end{bmatrix}$

If equality was chosen for the match criterion, such an AT would never match any sequence of annotations like the one shown in the example in section 3.4, because due to the missing NUM feature, the annotations are not equal:

$$\left\langle 0, 2, {}_{det}\begin{bmatrix}\text{CASE} & \textit{nom}\\ \text{NUM} & \textit{sg}\end{bmatrix}\right\rangle \quad \left\langle 3, 10, {}_{noun}\begin{bmatrix}\text{CASE} & \textit{nom}\\ \text{NUM} & \textit{sg}\end{bmatrix}\right\rangle$$

The power of feature structures is in their ability to express partial information, and restrictions on this information at points where the information itself is not even known, yet. Therefore subsumption of annotations seems to be the most appropriate match criterion. An annotation A on an arc matches an annotation B in the input, if the feature structure in A subsumes the feature structure in B.

By using subsumption as the match criterion, a description matches not only one unique annotation, but a set of multiple annotations (namely all those annotations subsumed by the annotation on the arc). This way, only one arc is needed with a description of an annotation A in an AT where otherwise many arcs containing descriptions of all annotations subsumed by A were needed to express the same phenomenon. This makes ATs more compact, and allows to express generalizations in a more natural way.

A similar concept can be found in a paper by van Noord and Gerdemann [vNG01]. This paper presents *predicate augmented finite state transducers*, which allow predicates over the transducer's alphabet Σ instead of only symbols from Σ. These predicates can also be interpreted as a description of a set of symbols to be matched. According to van Noord and Gerdemann, using predicates makes transducers more compact. An interesting property of the PFST formalism is that it is possible to determinize certain non-functional[3] transducers. Determinization algorithms designed for "classic" transducers require a transducer to be functional (see Roche & Schabes [RS97]).

4.2.2 `Call`-statements in the input

In the introduction to this chapter we mentioned that annotation transducers recognize context-free languages. This is because instead of matching an individual annotation, an annotation transducer can `call` another annotation transducer (or even itself). This way, ATs can recognize center-embedding languages in addition to left-recursive and right-recursive languages. The set of valid `call`-statements is called $\mathcal{CL}$.

Languages that are only right-recursive are regular languages (left-recursion can always be eliminated in favor of right-recursion). Intuitively speaking, when deriving a word in a regular language, the word only "grows" at its end

[3]A functional transducer is a transducer that for a given input creates at most one output.

– but never in the middle. In a context-free language, when deriving a word, it may also "grow" in the middle. This is called center-embedding. A typical example for a center-embedding language would be $\{a, aca, acca, accca, ...\}$, which is a context-free language.

With a `call`-statement an AT can call another AT that is contained in the same AT set. Each transducer t_i in the set has a unique name (within this set), n_i. With a `call`-statement, an AT t_i can delegate processing to another AT t_j. If AT t_j reaches a final state, then the `call`-statement behaves as if AT t_i matched an input symbol and a transition occurs. AT t_i resumes matching at the position after the last annotation matched and consumed by AT t_j. If AT t_i doesn't reach a final state, AT t_j will fail, too.

4.3 The output alphabet

As mentioned earlier, the output an AT generates (provided it successfully reaches a final state at some point) are new annotations to be added to a text. However, these new annotations are not created by the AT itself. The output of an AT is rather a description of what type of new annotation is to be created, at what place it is to be created and what the values in the annotation are going to be. These descriptions are called *output instructions*. Conceptually, output instructions resemble function calls in a computer program, and indeed they are the input to an additional external processor which interprets the output instructions, and then executes these instructions. As the result of running this "program" of output instructions, new annotations are created.

The set of valid output instructions is $\mathcal{INS}$. The supported instructions are:

- `begin(name)`
  Register a start position for a new annotation, that is named '`name`', at the start position of the annotation just matched.

- `end(name)`
  Register an end position for a new annotation, that is named '`name`', at the end position of the annotation just matched. A start position for `name`

must have been registered prior to a call of `end`. Matching goes from left to right, so the start position will always be on the left of the end position.

- `create(name, type)`
  Create a new annotation with name `name`, with the start and end positions previously registered by calls to `start` and `end`. The new annotation will have the type `type`.

- `set(name, path, value)`
  Set the feature identified by the path `path` to the value `value`. The feature and the value set must satisfy the restrictions imposed by the type hierarchy.

Output instructions can occur on any arc in theory, as long as they occur in the proper sequence (`begin` instructions before `end` and `create` instructions, `set` instructions after `create`, and so on). An AT could even create multiple annotation during one run. However the compiler described in chapter 6, which creates ATs from rules in annotation grammars, restricts the places where output instructions can occur: `begin` instructions occur only on arcs that leave the initial state, all other output instructions occur only on arcs that go into final states.

Output instructions aren't executed immediately, but only after the AT successfully matched the input. When matching fails, all output instructions that were generated up to the point where matching failed (in most cases this is only a `begin` instruction) are ignored. Furthermore, when the transducer needs to backtrack, in most cases no instruction at all needs to be removed from the buffer.

4.4 Handling of variables

As mentioned in section 3.5.1, variables may occur in annotation descriptions indicating that the values at all positions where the variable occurs must be identical. In annotation transducers, the scope of a variable reaches from the point where it first occurs over all subsequent arcs. When a variable is encountered the first time in a description on an arc, and this description matches the next annotation in the input, the variable is bound to the appropriate value from the annotation that was matched. After the variable has been bound to a value, subsequent descriptions that contain this variable only match the input if the value at the corresponding position is identical to that of the variable.

4.5 An example of an Annotation Transducer

This section gives a simple example of an AT that matches stock symbols. Stock symbols are a sequence of an opening bracket, a number of characters, a hyphen, some more characters and a closing bracket (like "`(IBM-N)`").

4.5.1 The type inheritance hierarchy

Since the feature structures in annotations are typed, first the type inheritance hierarchy to be used in the feature structures must be defined. This type hierarchy must be unique and the same for all annotations throughout the system.

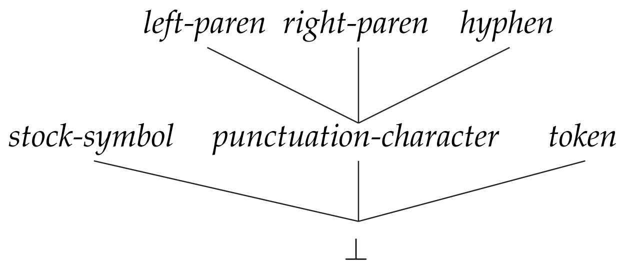
We assume the type hierarchy shown in figure 4.1.

4.5.2 The Annotation Transducer

Figure 4.2 shows an annotation transducer. The numbered circles represent the states, the arrows represent an arc. The text above each arc is the arc label. Each label consists of two parts, the input side and the output side, which are separated by a slash. Some labels don't generate output, in this case the output side is ϵ.

The initial state is marked with `>`, which is state 1. Next to the initial state, you can see the transducer's name (`stock_symbol1`).

On the input side of the label between state 1 and state 2 is the description of a *left_paren* annotation. On the output side is a `begin` output instruction, which marks the start position of the new annotation. This new annotation is named `a_1`. The labels between the following states up to state 5 are similar, however



Type	**Appropriate Features**
left_paren	- none -
right_paren	- none -
hyphen	- none -
stock_symbol	SYMBOL: *string* ID: *string*
punctuation_character	- none -
token	TEXT: *string*

Figure 4.1: Type inheritance hierarchy and appropriate features

they don't generate any output. On the output side of the label between the states 5 and 6 are finally quite a number of output instructions. First, the end position is marked. Then follows the command to create the annotation and to set the values of its features.

The final state (state 6) is marked with a double-circle.

A sequence accepted by the transducer would be:

$$\left\langle 0, 1, \textit{left\_paren} \right\rangle, \left\langle 1, 4, {}_{\textit{token}}\left[\text{TEXT} \quad \text{"IBM"}\right] \right\rangle, \left\langle 4, 5, \textit{hyphen} \right\rangle, \left\langle 5, 6, {}_{\textit{token}}\left[\text{TEXT} \quad \text{"N"}\right] \right\rangle, \left\langle 6, 7, \textit{right\_paren} \right\rangle$$

assuming that the underlying input string is:

$$_0\,(\,_1\,\mathrm{I}\,_2\,\mathrm{B}\,_3\,\mathrm{M}\,_4\,-\,_5\,\mathrm{N}\,_6\,)\,_7$$

Running the transducer over the above sequence would result in the following operations:

- The initial state is state 1 in the AT labeled `stock_symbol1`.
- Two arcs connect state 1 with state 2. The left arc is taken if the next

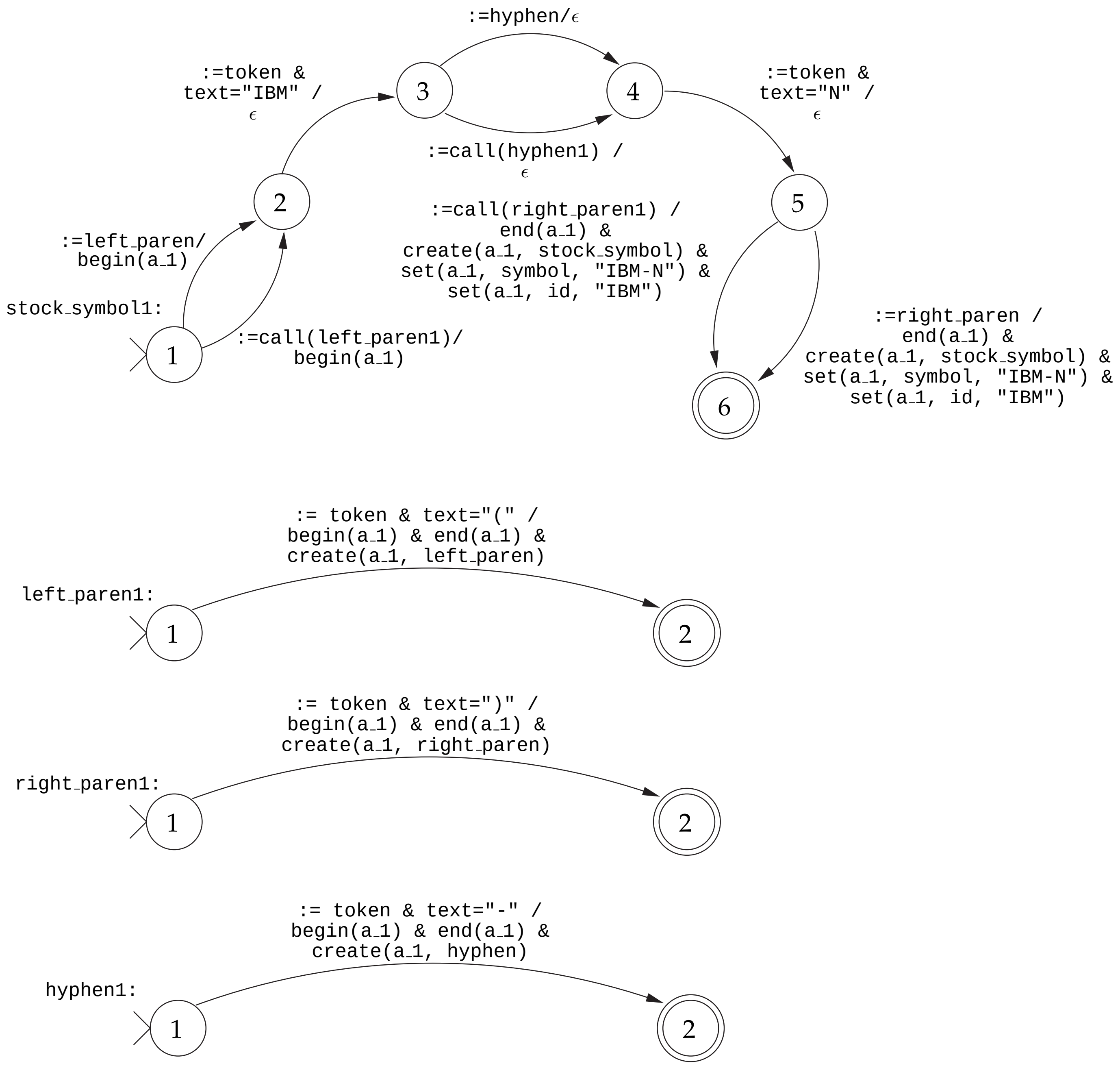

:=hyphen/ϵ
:=token & text="IBM" / ϵ
:=token & text="N" / ϵ
:=call(hyphen1) / ϵ
:=call(right_paren1) / end(a_1) & create(a_1, stock_symbol) & set(a_1, symbol, "IBM-N") & set(a_1, id, "IBM")
:=left_paren/ begin(a_1)
stock_symbol1:
:=call(left_paren1)/ begin(a_1)
:=right_paren / end(a_1) & create(a_1, stock_symbol) & set(a_1, symbol, "IBM-N") & set(a_1, id, "IBM")
1
2
3
4
5
6
:= token & text="(" / begin(a_1) & end(a_1) & create(a_1, left_paren)
left_paren1:
1
2
:= token & text=")" / begin(a_1) & end(a_1) & create(a_1, right_paren)
right_paren1:
1
2
:= token & text="-" / begin(a_1) & end(a_1) & create(a_1, hyphen)
hyphen1:
1
2

Figure 4.2: An AT to annotate the stock symbol (IBM-N)

annotation in the input is subsumed by the description on the input-side of the arc, that is, if the annotation in the input is of type *left_paren*. The `begin` instruction will drop a mark at the start position of the annotation matched (position 0).

The right arc is labeled with a *call* statement. This arc is taken if the next annotation is not of type *left_paren*. In this case, the AT labeled `left_paren1` is called immediately, and `stock_symbol1` waits until `left_paren1` finishes. If `left_paren1` reached a final state, a mark will be dropped at the position `left_paren1` was called, otherwise `stock_symbol1` will also fail.

- The AT advances to state 2.
- The next annotation to be matched must be of type *token*, with its feature TEXT set to `IBM`.
- The AT advances to state 3.
- Again, two arcs connect state 3 and 4. This time, either a *hyphen* must be matched, or the `hypen1` AT is called which matches a *token* and creates a *hyphen* annotation.
- The AT advances to state 4.
- Again, a *token* annotation must be matched between states 4 and 5.
- The AT advances to state 5.
- States 5 and 6 are connected by two arcs - either a *right_paren* annotation is matched or created. The sequence of output instructions on the output sides of both arcs will first drop a mark at the end position of the annotation last matched (position 7). Then the new annotation of type *stock_symbol* is created, and the two features SYMBOL and ID set to the values `IBM-N` and `IBM`, respectively.
- The AT advances to state 6. State 6 is a final state. All output instructions that were buffered in the external processor are now executed in the order they were generated.

As the result, this annotation is generated:

$$\left\langle 0, 7, {}_{\mathit{stock\_symbol}}\begin{bmatrix} \text{SYMBOL} & \texttt{"IBM-N"} \\ \text{ID} & \texttt{"IBM"} \end{bmatrix} \right\rangle$$

A larger example including ATs with variables is in the appendix.

4.6 Comparison to Augmented Transition Networks

A formalism which is similar to ATs are Augmented Transition Networks (ATNs), which are presented comprehensively in an article by Madeleine Bates [Bat78]. An ATN consists of a set of states that are connected by labeled arcs, with a distinguished start state and a set of distinguished final states. In addition to this, ATNs have access to a set of registers, which may contain any type of data. The labels on the arcs are operations of different types, such as

- testing, if the current input word is of a specific syntactic category
- testing, if the input is a specific word
- pushing the current state on a stack, then jumping to a different state
- jumping to a state previously pushed on the stack

Each operation consists at least of a *test* and a number of *actions*. A test is a predicate which can be used to check the value in one of the registers, or examine properties of the input (the syntactic category of the current input word, end of sentence, etc.). Actions can be used to manipulate the values of registers.

ATNs share quite a number of properties with ATs. First of all, due to their support of `push` and `pop`, which makes it possible to recognize center-embedding structures, they are of context-free complexity, just like ATs with `call`-statements. But furthermore, they offer a set of registers to store data gathered in previous operations, and a large number of manipulation functions on these registers. Due to the fact that taking an arc issues a function call, the only way to interpret an ATN is procedurally, from left to right. In ATs, attributes and their values in annotations take the role of registers, and the way output instructions are handled gives ATs a strong procedural bias.

4.7 Summary

- Annotation transducers match sequences of annotations and generate new annotations.
- The match criterion for annotations is subsumption.
- The output of ATs is a number of output instructions that are executed in the order of their appearance by an external processor.
- Due to `call`-statements, ATs can handle languages of context-free complexity.

Chapter 5

Annotation Grammars

In the previous sections, we have introduced annotations and annotation transducers. Annotations are objects that can be used to add information to a specific range in an input text. Therefore, they can be used in information extraction systems to represent data gathered during processing. Annotation transducers are devices that are capable of examining an input sequence of annotations, and creating new annotations. These new annotations can then be filled in with information which may be computed based on the information contained in the sequence of annotations matched.

A set of annotation transducers is basically a description of how information of interest is structured in a certain language. Consider an application that is able to detect company names in an English text. Company names are special kinds of noun phrases, frequently consisting of multiple words. The set of ATs in this application would contain ATs that match noun phrases containing company names, and then create new annotations that mark the company names found and contain information about the company, its full name, an abbreviation, and its stock symbol, for example. Depending on the complexity of the task, such AT sets might become very large.

Typically, applications are required to support multiple languages. This will also affect our noun phrase detector, of course. It will not be sufficient to just replace some strings in a resource file, it will rather be necessary to replace most of the AT set. Considering the possible complexity of such a transducer set, this task might take a long time to complete.

In such a scenario, it is desirable to have a more human-intelligible, high-level description language that is capable of expressing most of the power of annotation transducers, but hides their complicated details, like specifying output instructions in correct order, from the developer. It should be possible to compile descriptions in this language into annotation transducers automatically.

In this chapter, we will describe *annotation grammars* whose goal it is to provide such a description language. We will first give an overview over the requirements to be met by annotation grammars and the solutions found to satisfy these. Then, we will describe the concepts and the formalism of annotation grammars in detail, finishing the chapter with a detailed description of the syntax developed as part of this thesis.

5.1 Requirements

The following properties were found to be necessary for a description language to have in order to be useful in a scenario described above.

- The description language will be used mainly by linguists, so it should be close to well-known grammar formalisms.
- The expressive power of the description language should be as close as possible to that of annotation transducers.
- The description language should support the development of grammars for different languages, especially for languages with non-latin alphabets.
- Reuse of existing parts of grammars should be simplified, making it possible to build libraries of grammars for different phenomena in different languages.

5.2 Solutions

As a consequence of the requirements stated in the previous section, a description language and a compiler were developed that provide solutions to as much of the requirements mentioned as possible. The formalism is called *annotation grammar* (abbreviated AG). Annotation grammars have the following properties:

- Formally speaking, annotation grammars are context-free grammars over the domain of annotations. Recall that annotation transducers can call other ATs using the `call`-statement, which makes their expressive power context-free.
- Rules in an AG correspond to ATs.
- An AG may consist of multiple modules, where each module is stored in a different file. This keeps AGs more structured, making development and maintenance easier. Libraries of modules can be built and plugged together to solve different tasks.
- Every module can be saved using a different character set. This way, a module for Japanese can be developed using a Japanese character set, or Unicode, avoiding complicated escape codes.
- The AG syntax includes various extensions to simplify development, for example conditional compilation to selectively include or exclude parts of a grammar.

5.3 The Annotation Grammar Formalism

An annotation grammar is a tuple $G = \langle A, P, S, \mathsf{Type} \rangle$. $A \subset \mathsf{Descr}$ is the set of *symbols*, which are descriptions of annotations with respect to a type inheritance hierarchy Type. $P \subset A \times A^+$ is the set of *production rules*, and finally $S \in A$ is the *start symbol*[1].

[1] We will not admit an empty sequence of annotation descriptions, neither in production rules nor as words in a language.

Just like for context-free grammars, we will now introduce two relations $\underset{G}{\Longrightarrow}$ and $\overset{*}{\underset{G}{\Longrightarrow}}$ between two sequences of annotations from A^+ which will enable us to define the language denoted by the annotation grammar.

Two sequences of descriptions $\alpha a \beta$ and $\alpha \gamma \beta$ are in the $\underset{G}{\Longrightarrow}$ relation ($\alpha a \beta \underset{G}{\Longrightarrow} \alpha \gamma \beta$) if in G, there exists a production rule $b \rightarrow \gamma$ and $a \sqsubseteq b$. The reflexive transitive closure of $\underset{G}{\Longrightarrow}$ is written $\overset{*}{\underset{G}{\Longrightarrow}}$. $\alpha_1 \overset{*}{\underset{G}{\Longrightarrow}} \alpha_n$ if $\alpha_1 \underset{G}{\Longrightarrow} \ldots \underset{G}{\Longrightarrow} \alpha_n$.

The language denoted by the annotation grammar G can now be defined to be the set of sequences that are in relation to the start symbol: $L(G) = \{\alpha \mid \alpha \in A^+ \text{ and } S \overset{*}{\underset{G}{\Longrightarrow}} \alpha\}$.

The definition of annotation grammars is very similar to the classic definition of context-free grammars with a number of exceptions which we will now inspect in greater detail.

The alphabet used in annotation grammars consists of descriptions of annotations. The use of annotations as well as their descriptions requires that all types used in these annotations are defined in a type inheritance hierarchy. Therefore, a type inheritance hierarchy must be part of every annotation grammar.

There are two more major differences. In annotation grammars, no distinction is made between nonterminals and terminals. The sets V and T in context-free grammars are collapsed in one set A, the alphabet. The condition that a word in the language generated by a context-free grammar must be from T^* is omitted in the definition of languages generated by annotation grammars. As a consequence of this, a derivation may "stop" at arbitrary points. One could say that the language denoted by an annotation grammar consists of both sentences and sentential forms[2]. The reason for this is motivated technically rather than formally; therefore we will postpone the discussion of this point to the technical part in section 5.5.2.

Furthermore, in the definition of the derivation relation, equality of symbols is replaced by subsumption of annotations. The reasons for this have already been discussed in the chapter on annotation transducers: Using subsumption, features can be left underspecified in desriptions allowing the grammar developer to refer to whole classes of annotations in rules which otherwise would

[2]These two terms become obsolete in annotation grammars.

only be possible with huge disjunctions [3]

5.4 An Example

This section will give an example of an annotation grammar that describes how to mark up a company name ("International Business Machines") and a stock symbol ("IBM-N") with annotations[4]

```
:=company_info & fullname = X & stocksymbol = Z ->
  :=company_name & name = X & id = Y,
  :=stock_symbol & id = Y, symbol = Z.

:=company_name & name = "International Business Machines" & id = "IBM" ->
 :=token & text = "International",
 :=token & text = "Business",
 :=token & text = "Machines".

:=stock_symbol & symbol = "IBM-N" & id = "IBM" ->
  :=left_paren?,
  :=token & text = "IBM-N",
  :=right_paren?.

:=left_paren ->
  :=token & text = "(".

:=right_paren ->
  :=token & text = ")".
```

The grammar consists of five rules. By convention, the first rule in the grammar is the one where the derivation starts (one could also say that the left-hand side of the rule is the "start symbol"). This rule says that an annotation,

[3]Due to the fact that feature structures inside of annotations must use a fixed and finite type hierarchy and cyclic feature structures are not allowed, there is a finite number of possible well-formed feature structures that could all be listed.

[4]For clarity, we will omit meta statements which are required in an annotation grammar module. See the following sections.

which is of type *company_info* can be created at positions where an annotation of type *company_name* is followed by an annotation of type *stock_symbol*. In other words, the *company_info* annotation provides information about companies that consists of their name and their stock symbol.

The *company_info* annotation has two features: FULLNAME and STOCKSYMBOL. The "values" of these features are set to the values of the variables X and Z.

The right-hand side of the rule (line 2 and 3) contains the sequence of annotations which is annotated with the *company_info* annotation. The variable X takes on the value of the NAME feature in the *company_name* annotation, Z becomes equal to the SYMBOL feature in the *stock_symbol* annotation. These two values percolate up to the *company_name* annotation.

Note the third variable in this rule, the Y variable. It requires the ID features in their respective annotations to be equal. By requiring that the IDs of both company name and stock symbol are the same the rule makes sure that both stock symbol and company name actually belong to the same company.

The other rules in this example follow similar concepts. The option operators on lines 9 and 11 indicate that parentheses around the stock symbol don't need to be there (see section 5.7.7).

5.4.1 A derivation

The strings that are in the language denoted by a context-free grammar can be generated by deriving the strings starting at the start symbol. This concept is also present in annotation grammars, however with a number of conceptual differences which will become obvious in this example. Let us first look at a derivation based on the above grammar example. We will use a feature structure notation here instead of descriptions for better readability.

The "start symbol" is the left-hand side of the grammar, which is equivalent to the feature structure:

$${}_{\textit{company\_info}}\left[\begin{array}{ll}\text{FULLNAME} & \mathsf{X} \\ \text{STOCKSYMBOL} & \mathsf{Z}\end{array}\right]$$

This is the annotation to be created when this sequence is matched:

$$ {}_{\textit{company\_name}}\begin{bmatrix}\text{NAME} & \mathtt{X}\\ \text{ID} & \mathtt{Y}\end{bmatrix}, \quad {}_{\textit{stock\_symbol}}\begin{bmatrix}\text{ID} & \mathtt{Y}\\ \text{SYMBOL} & \mathtt{Z}\end{bmatrix} $$

By the definition of the derives relation, an annotation can be replaced by a sequence of annotation if this annotation subsumes another annotation on the left-hand side in the grammar. The *company_name* annotation indeed subsumes the annotation on the left-hand side of the rule starting on line 4. The same way, *stock_symbol* subsumes the annotation on the left-hand side of the rule starting on line 8.

$$ {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"International"}\end{bmatrix}, {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"Business"}\end{bmatrix}, {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"Machines"}\end{bmatrix} \\ \textit{left\_paren}, {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"IBM-N"}\end{bmatrix}, \textit{right\_paren} $$

Again, the *left_paren* and *right_paren* subsume the left-hand sides of the rules on line 12 and 14, respectively. So in another step we get:

$$ {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"International"}\end{bmatrix}, {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"Business"}\end{bmatrix}, {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"Machines"}\end{bmatrix} \\ {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"("}\end{bmatrix} {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{"IBM-N"}\end{bmatrix}, {}_{\textit{token}}\begin{bmatrix}\text{TEXT} & \mathtt{")"}\end{bmatrix} $$

Now, no more annotation can be replaced.

In this derivation, one thing happened "by magic": The proper assignment of variables (and more important, the restrictions placed on the possible sequences by the use of variables). How are the values of these variables determined? As mentioned before, there are a number of conceptual differences. The purpose of annotation grammars is to serve as a *description language for annotation transducers*. Annotation transducers work their way bottom-up, which is also the preferred view for annotation grammars. Variable binding is much more obvious, of course, when applying the rules bottom up from an existing input sequence.

A final note on the *token* annotations: Annotation grammars (as well as annotation transducers) never operate on strings directly, only on annotations. This means that the first level of annotations must be added by something else than an annotation transducer, a tokenizer for example, that accepts text as its input and generates annotations marking up tokens as its output.

5.5 Technical considerations

In the previous chapter, we introduced the formal properties of annotation grammars. Annotation grammars are essentially context-free grammars – with a number of exceptions resulting either from the usage of annotation descriptions as the alphabet, or from technical requirements. They are to be used in a production environment as a flexible means of describing manipulations of annotations. In this chapter we will discuss the additional properties of annotation grammars designed to satisfy these requirements from a technical point of view.

5.5.1 The meaning of rules in AGs

The main purpose of annotation grammars is to give developers a means to express manipulation of annotations in an intelligible description language rather than by coding low-level annotation transducers. Annotation transducers basically perform two tasks: matching of existing input sequences of annotations and creation of new annotations. Every rule in an AG will correspond to one individual AT. The right-hand sides of rules represent the input to be matched, while the left-hand sides describe the annotations to be created. This is actually quite intuitive - in many grammar formalisms the symbol on the left-hand side represents some higher category.

However, it is not desirable in any case that all left-hand side annotations become part of the output. Suppose, for example, a rule that is developed to be used from several other rules, something similar like a function in programming languages like C. The annotation on the left-hand side is just used as a means to propagate the important information up to the calling rule, and should not occur in the output. It is therefore possible to specify which rules should be translated in ATs that match a sequence and create a new annotation, and which rules should be translated in ATs that only match (that is, behave more like a recognizer than a transducer). The former rules will be called *"match-and-create rules"*, the latter *"match-only rules"*.

A **match-and-create** rule describes an annotation transducer that checks if the sequence of annotations on the right-hand side of the rule **matches** the sequence of annotations at the current read position of the AT in the input. If it does, the annotation on the left-hand side of the rule is **created**, that is, added to the set of annotations attached to the underlying input string. A rule becomes a match-and-create rule, if the type of the left-hand side annotation is an *external* type[5].

A **match-only** rule describes an Annotation Transducer that checks if the sequence of annotations on the right-hand-side of the rule **matches** the sequence of annotations at the current read position of the AT in the input. Even if the input is matched, the AT does **not create** the annotation on the left-hand side. A rule becomes a match-only rule, if the type of the annotation is an *internal* type[6].

Whether a type is external or internal must be defined in the type hierarchy file along with the actual type definition (this is shown in the example in the appendix).

It remains to specify when an annotation on the right-hand side of a rule just has to be matched, or to be replaced with another sequence of annotations (the equivalent to terminal and nonterminal symbols in CFGs). Assume there is a description of some annotation A on the right-hand side of the rule. If there exists a rule in the AG with a description of an annotation B and A subsumes B, then the sequence of annotations on the right-hand side of this rule must be matched instead of only A. Furthermore, if this rule is a match-and-create rule, after matching its right-hand side the annotation B will be created. If, on the other hand, no rule exists with an annotation on its left-hand side subsumed by A, then only A must be matched.

[5]"external" because the annotation becomes externally visible after its creation, that is as an annotation attached to the input sequence

[6]"internal" because the annotation data is only used to determine which rule applies, much like a nonterminal in CFGs.

5.5.2 Abolishing the distinction between terminal and non-terminal symbols

In the formal introduction of annotation grammars we mentioned that there is no notion of non-terminal and terminal symbols in annotation grammars. As a consequence, in the language denoted by an annotation grammar, there will be sequences of descriptions which are both what would be called sentence and sentential forms in the case of classic CFGs. Essentially, this means that sequences of annotations are legal sequences with respect to the AG even if they contain annotations on which other rules could apply. This way, sequences of annotations can be used as input which may have been pre-annotated by other tools than annotation transducers. An application might want to use a specialized recognizer for certain types of noun phrases for example, that creates NP annotations before the ATs described by the AG come into use. In such a case, the AT encounters sequences that already contain higher-level annotations, which it should be able to process anyway.

5.6 The module concept of annotation grammars

5.6.1 Why modules?

In the early days of computers, programs essentially consisted of a stack of punch-cards, later they were stored in long text files. Programs grew longer and longer, and they rapidly gained complexity. A point was reached when programs written this way became unmanagable. Today, this is called the *software crisis*. It became obvious that it was no longer possible to write programs the way this was done up to that time, but necessary to develop software just like engineers develop cars. *Software Engineering* was born. The goal of software engineering is to provide ways of how to develop reliable software. Sophisticated guidelines were created of how to plan, design and implement software. New programming concepts were invented, and new programming languages that supported these.

One of the most important concepts, and certainly one of the most successful, is the concept of *modules*.

According to Klaeren [Kla00], the three important properties of modules are:

- It is possible to replace a module without affecting the rest of the system.
- A module performs non-trivial tasks.
- A module is a self-contained functional unit.

With modules, large programs can be broken down into small units. Such a program can be understood much more easily by somebody who has to add a feature some time after the initial development. It is possible to replace one module with another when it turns out that there is a better approach to solve a specific task or the module didn't work properly. Furthermore, it is possible to reuse a module in a different program provided it was designed carefully. Careful design is addressed by the *information hiding principle*. The information hiding principle essentially requires that the least possible of the internals of a module is shown to the outside, but the parts that are shown are documented accurately.

One requirement for annotation grammars is that it should be possible to reuse as many parts as possible in other projects. The solution is to introduce a module concept for annotation grammars.

An annotation grammar is declarative, unlike a computer program. The meaning of the grammar is determined by the rules *all at once*, not by a number of functions in a certain order. The information hiding principle requires the strict distinction between an *interface* and and *implementation*, where the interface specifies what is to be done and the hidden implementation describes how it is done. For a module of an annotation grammar, there is no sensible distinction of an interface and an implementation, as grammars contain no implementations, just descriptions – the rules. Therefore, there is no information hiding principle in the classic sense.

However, annotation grammars are related to languages. Any annotation grammar is developed for one language. When a module is reused, it should be possible to make sure that the module can only be used if it was developed for the right language! It may well be possible that a module can be used in several different languages: Think of our company-name example. English company names and stock symbols may occur in English texts just the same as they occur in German texts, for example. After all, names won't be translated in most cases, neither will be stock symbols. So such a module could be used in both annotation grammars for English and German. In some sense, the "interface" in an annotation grammar module defines the languages the module was developed for, and the "implementation" are the rules in the module.

So the two important points that determine the structure of the annotation grammar module system are the improvement of reusability and intelligibility, and language-awareness.

5.6.2 A module system for annotation grammars

An annotation grammar built up out of modules has a general structure as shown in figure 5.1. There is one main module, which is the only module visible to the outside. The main module can include multiple submodules, which in turn can include other submodules, resulting in a tree-like inclusion structure.

5.6.3 Language-awareness in modules

An important feature is the language-awareness of the module system. For each module, the developer must specify what languages the module is valid for. Main modules can be valid for only one language, which is the language for which the AG is developed. Submodules, however, may be valid for more than one language. Consider a company name detection system for English. The main module would be declared to be valid for English. Now assume that German company names, found in an English text, should also be discovered. The developer could use a submodule for German company names that has been declared to be valid to be used AGs for both English and German, and include this module in the main module. This feature puts strict restrictions on what can be a submodule in another module: the set of languages the including module is valid for must be a subset of the set of languages of the submodule. Figure 5.2 illustrates this.

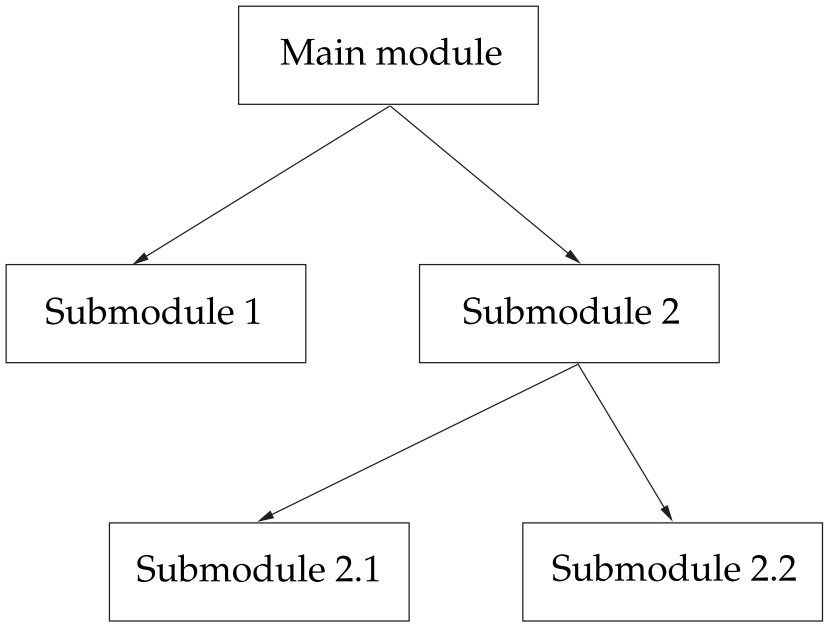


Figure 5.1: The general structure of an annotation grammar with respect to its modules.

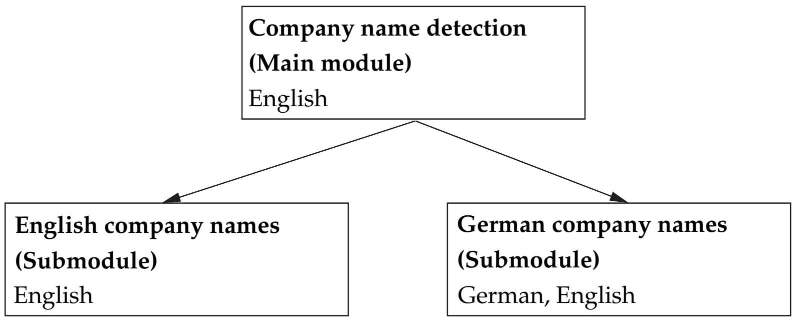

Company name detection (Main module) English
English company names (Submodule) English
German company names (Submodule) German, English

Figure 5.2: Language awareness in AG modules

A configuration as shown in figure 5.3, would be illegal, because the module "German company names" was only developed for German and French, but not for English. So the attempt to include this module in an English AG is illegal.

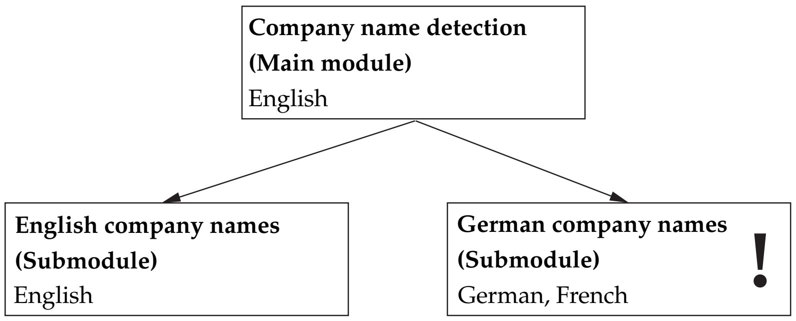


Figure 5.3: Language awareness in AG modules – invalid inclusion

5.7 Syntax of annotation grammars

This section will describe the syntax of annotation grammars in detail. In addition to rules, the annotation grammar syntax defines statements for the management of modules and declaration of language properties of a module. Because typed feature structures are used in annotations, the syntax provides statements to declare both internal and external types. Conditional compilation, familiar from programming languages like C++ or PASCAL, allows for flexible inclusion or exclusion of parts of rules within a module.

5.7.1 Module headers

The first line in any module must be the *module header*. The module header contains the type of the module (main module or submodule) and a number of *meta-variables* that contain information about the module. The `languages` meta-variable, for example, contains the list of languages the module is valid for, and the `encoding` meta-variable contains the name of the character encoding used. Meta-variables are also used for conditional compilation.

Syntax of main module headers

The syntax of headers for main modules is:

'`<main_module`' *meta-variable-list* '`>`'

meta-variable-list is a list of meta-variable equations, where each equation consists of a name and a value, for example `author="Holger Wunsch"`. Some meta-variables are mandatory in a main module header:

- `languages =` *language-id*
  The `languages` meta-variable specifies the language the main module is valid for. The value is a string containing exactly one *language-id*, which is a standardized language identifier like `en_US` or `de_DE`.
  Example: `languages = "en_US"`

- `encoding =` *encoding-name*
  The `encoding` meta-variable specifies the character set of the module. *encoding-name* must be an ecoding name that is recognized by the Java Virtual Machine [Sun02], like `iso-8859-1` or `UTF16`. If an international version of the JVM is used, all extended character sets implemented there are also supported.
  Example: `encoding="iso-8859-1"`

There is one more predefined meta-variable, the `author` variable, that contains the name of the developer. This variable is optional.

An example for a well-formed main module header is:

```
<main_module author="Holger Wunsch" languages="en_US"
encoding="utf-8">
```

Syntax of submodule headers

The syntax of headers for submodules is:

'`<sub_module`' *meta-variable-list* '`>`'

As with main modules, *meta-variable-list* is a list of meta-variable equations. The definition of two meta-variables is mandatory in submodule headers:

- `languages` *language-id-list*
  The `languages` meta-variable specifies the languages for which the submodule is valid. The value *language-id-list* is a string of comma-separated language-identifiers. Note the contrast to main modules: Only one language-identifier as the value of the `languages` meta-variable is allowed there, for submodules the meta-variable `languages` must be used with a *list* of language-identifiers[7].
  Example: `languages="en_US,en_GB,de_DE"`
- `encoding =` *encoding-name*
  The encoding meta variable is to be used the same way as in main modules.

An example for a well-formed submodule header is:

```
<sub_module author="Holger Wunsch" languages="en_US, en_GB"
encoding="utf-8">
```

In addition to the predefined meta-variables with special meanings an arbitrary number of user-defined meta-variables may be specified, for example a time-stamp of the last modification of the module or a flag whether the module is to be compiled for a debug environment. These meta-variables will be ignored by the compiler with the exception of meta-statements for conditional compilation where a set of rules may be included or excluded based on the value of a meta-variable.

5.7.2 Meta-statements

A meta-statement contains information about the module, or processing instruction to the compiler. Meta-statements are similar to preprocessor statements in programming languages like C. However, advanced features like the

[7]This list may well contain only one element

definition and expansion of macros are not supported. The syntax of meta-statements is similar to elements found in markup-languages like HTML or XML. Meta-statements are enclosed in angle brackets. The first word in the statement defines the type of meta-statement, similar to a tag name in XML, followed by a number of data items depending on the type of statements in an attribute-value notation.

<*meta-statement-type data-items*>

The following types of meta-statements may occur in an AG:

- Module headers
- Module inclusion statements
- Type definition inclusion statements
- Statements for conditional compilation

The usage of module headers has already been explained, in the following sections the other types of meta-statements will be introduced.

5.7.3 Module inclusion statements

As described in section 5.6, AGs can be split in several modules, one main module and several submodules. Module inclusion statements are used to include a submodule in a module, which can either be the main module, or another submodule. The syntax of a module inclusion statement is:

'`<include`' *filename* '`>`'

filename is the name of the module to be included. Only modules can be included whose language list is a superset of the language list of the including module. The submodule inherits all meta-variables that were defined in parent modules, so in the submodule, all meta-variables from the submodule and the parent modules are known and accessible. If a meta-variable that was defined in a parent module is defined again in a submodule, the definition in the submodule is ignored.

Example: `<include "d:\modules\comp_name_en_de">`

5.7.4 Type definition inclusion statements

In order to keep annotation descriptions in grammar rules as short as possible, the AG compiler uses type inference to create a complete annotation. To accomplish this and to validate the descriptions given, the compiler needs access to the type information about the annotations used in the current module. This type information is stored in *type hierarchy files* (see the example in the appendix). Here, all possible types of annotations are declared along with their appropriate features. Type definition inclusion statements can occur in any module. During parsing, the compiler reads in all type definition files for all modules and merges them into one big type hierarchy. Therefore the individual type inheritance hierarchies **must declare types that are consistent throughout the whole AG**. The syntax of the type definition inclusion statement is:

'`<include_type_hierarchy`' *filename* '`>`'

Example: `<include_type_hierarchy "hierarchy.thy">`

5.7.5 Statements for conditional compilation

The statements for conditional compilation allow a developer to include or exclude certain sets of rules in a module based on the value of a meta-variable. The `if`-statement introduces a conditional block, the `endif`-statement closes a conditional block. The syntax of the `if` - `endif` statement is as follows:

'`<if`' *meta-variable-name* '`=`' *meta-variable-value* '`>`'
rules
'`<endif>`'

If a meta-variable with the name *meta-variable-name* was defined either in the header of the current module or in a header of one of its parent modules, and the value of the meta-variable is equal to *meta-variable-value*, then the block of rules enclosed in the `<if>` statement is included in the compilation, otherwise it is not.

5.7.6 Basic rule syntax

In what follows, the syntax of AG rules will be described in further detail. The left-hand side of an AG rule consists of a description of an annotation to be created. Variables are allowed as values on paths, if a variable with the same name occurs at least once on the right-hand side. The value of the variable on the left-hand side will be bound to the same value as the variables on the right-hand side. On the right-hand side, there is a comma-separated list of sequences of descriptions of annotations to be matched. An example of such a rule is shown in figure 5.4.

```
:=np & noun:case=nom & noun:num=sg & det:case=nom &
    det:num=sg ->
    :=det & case=nom & num=sg,
    :=noun & case=nom & num=sg.
```

Figure 5.4: Example of an AG rule

5.7.7 Regular operators in rules

To simplify notation, a number of additional "regular" operators can be added to each element of a sequence. "Regular" because the operators are the Kleene star '*' and Kleene plus '+', as well as the option operator '?', that are used in regular expressions. Using these operators, multiple occurences of the same annotation in a sequence can be expressed in a more convenient way than by using multiple rules. If these operators are used, the compiler can also generate more efficient transducers.

The regular operators can be used to form rules like in figure 5.5.

The meaning of these rules is as expected:

Rule (1) describes a transducer to annotate a sequence of one determiner, an arbitrary number of adjectives, and a noun with an NP annotation.

1. `:=np -> :=det & case:nom, :=adj & case:nom*,`
   `:=noun & case:nom.`
2. `:=np -> :=det & case:nom, :=adj & case:nom+,`
   `:=noun & case:nom.`
3. `:=np -> :=det & case:nom, :=adj & case:nom?,`
   `:=noun & case:nom.`

Figure 5.5: Rules with regular operators.

Rule (2) describes a transducer to annotate a sequence of one determiner, at least one adjective, and a noun with an NP annotation.

Rule (3) describes a transducer to annotate a sequence of one determiner, one or no adjective, and a noun with an NP annotation.

Note that regular operators may only be attached to the end of an annotation description and their scope ranges over the whole description.

Regular operators don't add anything to the expressive power of the annotation grammar formalism. Rules 1 – 3 could have been expressed as in figure 5.6.

The type `adjs_internal` is an example of an *internal type*, that is the type of an annotation that will not be created if the right-hand side matches. Annotation descriptions of feature structures of internal types thus behave like standard non-terminal symbols in a CFG over characters.

5.8 Complete Annotation Grammar Sytnax

annotation-grammar	→	*module-header grammar-body*
module-header	→	*main-module-header* | *sub-module-header*
main-module-header	→	'`<main_module`' *meta-variable-list* '`>`'
sub-module-header	→	'`<sub_module`' *meta-variable-list* '`>`'
meta-variable-list	→	*required-meta-variable-list* (*optional-meta-variable-list*)

required-meta-variable-list	→	*languages-meta-variable-eqn encoding-meta-variable-eqn*
languages-meta-variable-eqn	→	’`languages`’ ’=’ ’"’ *language-id-list* ’"’
encoding-meta-variable-eqn	→	’`encoding`’ ’=’ ’"’ *encoding-id* ’"’
optional-meta-variable-list	→	*meta-variable-equation* (*optional-meta-variable-list*)
meta-variable-equation	→	*meta-variable-name* ’=’ *meta-variable-value*
meta-variable-name	→	string
meta-variable-value	→	’"’ string ’"’
grammar-body	→	(*typedef-inclusion-statements*)[8] (*module-inclusion-statements*) *rules*
typedef-inclusion-statements	→	*typedef-inclusion-stmt* (*typedef-inclusion-statements*)
typedef-inclusion-stmt	→	’`<include_type_declaration`’ *filename* ’>’
filename	→	’"’ string ’"’
module-inclusion-statements	→	*module-inclusion-stmt* (*module-inclusion-statements*)
module-inclusion-stmt	→	’`<include`’ *filename* ’>’
rules	→	*rule* (*rules*) |
		if-statement rules endif-statement
if-statement	→	’`<if`’ *meta-variable-equation* ’>’
endif-statement	→	’`<endif>`’
rule	→	*left-hand-side* ’`->`’ *right-hand-side* ’`.`’
left-hand-side	→	*annotation-description*
right-hand-side	→	*rhs-sequence*
rhs-sequence	→	*rhs-expr* (’`,`’ *rhs-sequence*)
rhs-expr	→	*annotation-description* (*regular-operator*)
regular-operator	→	’`*`’ | ’+’ | ’?’
annotation-escription	→	*type* (’&’ *path-conjunction*)
path-conjunction	→	*path-value* |
		path-conjunction ’&’ *path-value*
path-value	→	*path* ’=’ *variable-or-value*
path	→	*feature-name* |
		path ’`:`’ *feature-name*
variable-or-value	→	*variable-name* |
		value
type	→	’`:=`’ *type-name*
type-name	→	string
variable-name	→	string
value	→	string

[8]At least one type hierarchy inclusion statement must occur in the grammar.

1.
```
:=np -> :=det & case:nom, := adjs_internal & case:nom,
        :=noun & case:nom.
:=np -> := det & case:nom, := noun & case:nom.
:=adjs_internal & case:nom -> :=adj & case:nom.
:=adjs_internal & case:nom -> := adj & case:nom,
                              adjs_internal.
```

2.
```
:=np -> :=det & case:nom, := adjs_internal & case:nom,
        :=noun & case:nom.
:=adjs_internal & case:nom -> :=adj & case:nom.
:=adjs_internal & case:nom -> := adj & case:nom,
                              adjs_internal.
```

3.
```
:=np -> :=det & case:nom, := adj & case:nom,
        :=noun & case:nom.
:=np -> :=det & case:nom, :=noun & case:nom.
```

Figure 5.6: The rules from figure 5.5 without the use of regular operators.

5.9 Summary

- Annotation grammars provide an intelligible way to describe manipulations of sequences of annotations using ATs.
- AGs are context-free grammars over the alphabet of descriptions of annotations.
- Since annotations contain typed feature structures, the presence of a type inheritance hierarchy as part of the grammar is required.
- In the “derives” relation, equality of symbols has been replaced by subsumption of annotations.
- No distinction is made between terminal and nonterminal symbols. As a consequence, derivation may stop at any point. This formal property reflects the technical requirement of ATs to optionally accept annotations generated by other tools than ATs.

- Annotation grammars support a modular structure. This allows developers to split large AGs in smaller parts of functionally related rules that can be handled more easily.

- The language-awareness of modules supports the development of multi-language grammar libraries. The strict definition of what languages a module is valid for eliminates possible errors resulting from accidentally using a module not developed for the target language.

- Several syntactic enhancements (Kleene operators) simplify grammar development.

Chapter 6

A compiler for annotation grammars

In the previous chapters we described annotations, which are objects to represent information about a text, annotation transducers which are capable of manipulating annotations, and annotation grammars, which are descriptions of annotation transducers in a familar, grammar-like form. Now, we will fill the gap between annotation grammars and annotation transducers with a compiler that translates annotation grammars into annotation transducers.

6.1 Overview

This section gives a brief overview of the main features of the compiler.

- The compiler reads in an annotation grammar consisting of multiple modules, and generates a set of annotation transducers.
- There is a one-to-one correspondence between an AG rule and an AT: For each rule, one AT is generated.
- The compiler is implemented in Java. This makes it platform independent. The international version Java Virtual Machine furthermore offers built-in support of a rich set of different character encoding schemes.

Figure 6.1 shows the top-level structure of the compiler program.

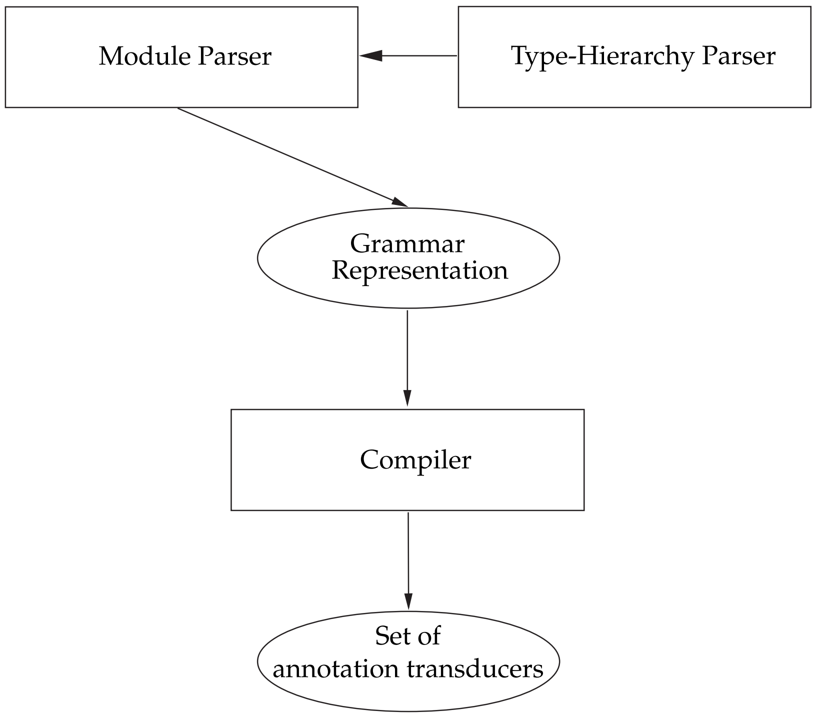


Figure 6.1: Structure of the compiler program.

There are two parsers: The *module parser* is the part of the program that reads in a module and all submodules, that evaluates all meta-variables and meta-statements. On this level, language validity checking is done as well as all operations needed for conditional compilation.

The second parser is the *type-hierarchy parser*. In order to generate an internal representation of the grammar, the module parser needs access to the type information about all the annotations used. As mentioned before, it is legal to specify multiple type hierarchy inclusion meta-statements in a module (for both external and internal types). All these “local” type inheritance hierarchies must be merged into one “global” type hierarchy that reflects the type system of the whole grammar. This task is solved by using a two-pass system: First, the type hierarchy parser scans all modules for type hierarchy inclusion statements and this way gradually builds up the complete type hierarchy. Then, after the type hierarchy is fully available, the module parser takes over and builds the in-memory representation of the grammar.

The result of the parse is a *representation of the complete grammar*. All objects in the grammar (rules, feature structures etc.) are represented by instances of respective Java classes. Annotation descriptions are not used any more at this stage. The grammar representation then becomes the input for the *compiler*. The compiler generates a set of Annotation Transducers, one transducer for each rule.

In the sections that follow, these functional blocks will be discussed in detail.

6.2 Grammar representation

The result of the parsing process is the internal grammar representation, which will be used by the compiler to generate ATs. A grammar is essentially a list of statements or *expressions*. One individual feature structure on the right-hand side, a Kleene star, or a whole rule are examples of expressions. It is therefore natural to use a hierarchy of Java classes to represent different types of expressions, and to implement the actual grammar object as a container for expressions. Figure 6.2 shows this class hierarchy.

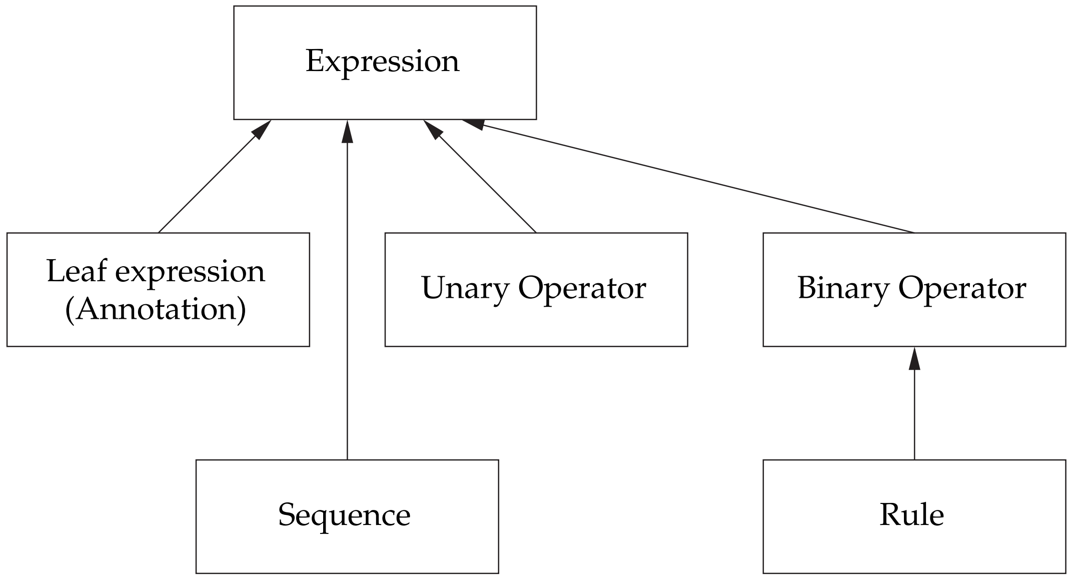


Figure 6.2: The Java class hierarchy of grammar objects

Expression: The *Expression* class is the root of the class hierarchy. It does not correspond to a concrete grammar object. It provides the capability to maintain information about a grammar object in a derived class, for example the line and column position of a grammar object in its respective module file, which are stored in *MetaInformation* objects.

Class Expression
  Properties
    MetaInfo: MetaInformation

Leaf expression: A *Leaf Expression* is a wrapper around one individual annotation. The annotation on the left-hand side of a rule, for example, is represented by a *Leaf Expression*.

Class LeafExpression
  Type of Expression
    Annotation: Annotation

Unary operator: The *Unary Operator* class represents the three regular operators Kleene star, Kleene plus and option. Instances of this class contain an instance of a leaf expression, which is the operand of the operator.

Class UnaryOperator
  Type of Expression
  Properties
    Type: one of `star, plus, option`
    Operand: Expression

Binary operator: The *Binary Operator* class represents the “arrow operator”. Instances of this class contain an instance of a leaf expression, which is the left-hand side operand, and an instance of a class derived from the general expression class, which is the right-hand side operand.

Class BinaryOperator
  Type of Expression
  Properties
    FirstOperand: Expression
    SecondOperand: Expression

Rule: A *Rule* is a specialized binary operator. It adds rule-specific functionality such as accessor functions and a means to determine the shortest possible sequence on which a rule applies (this is useful to detect rules that accept the empty string[1]).

Class Rule
  Type of BinaryOperator
  Operations
    GetLHS( ): Expression
    GetRHS( ): Expression
    LengthOfShortestSequence( ): integer

Sequence: A sequence is, as the name implies, a sequence of expressions on the right-hand side of a rule. It corresponds to the comma-operator in AG syntax.

Class Sequence
  Type of Expression
  Properties
    Expressions: sequence of Expression
  Operations
    Add(Expression: Expression)
    Length( ): integer
    Get(n: integer): Expression)

A *Grammar* class then consists of a list of instances of *Rule*, along with useful operations to access and set rules and find out the language of the grammar, for example (these operations are not listed here).

Class Grammar
  Properties
    Rules: sequence of Rule
    TypeHierarchy: TypeHierarchy

[1]Note that rules that accept the empty string are not allowed in this framework. A rule with a shortest possible accepted sequence of length 0 is therefore illegal.

6.3 The type hierarchy parser

As shown in figure 6.1, the parser does its work in two passes, where the first pass is to acquire the full type hierarchy. In each module, type inclusion meta-statements may occur that define types local to that module. In order to infer types correctly, all these local type definitions must be merged into one global type hierarchy that must be available before the first feature structure is parsed. Therefore it is necessary to implement the type hierarchy parser as a separate pass.

In the type hierarchy parser, several features of the module parser are duplicated, of course. Module inclusion statements must be read and parsed using the same stack-based approach as in the module parser. This means that the type hierarchy needs its own instance of the encoding detector (see below). However, it doesn't perform language validity checks.

6.4 The module parser

The module parser is responsible for reading all module files for an annotation grammar, parsing the modules and building the grammar representation. The module parser consists of several functional blocks that are shown in figure 6.3.

6.4.1 Encoding detection

As part of the internationalization support of the AG formalism, modules may be encoded in a number of different character encoding schemes. The exact encoding of the module must be specified by the developer in the module header. In the majority of cases, this information is sufficient to set up the proper conversion routines to transform the input into Unicode, which is the only encoding scheme used internally.

The detection engine first makes a guess about the encoding by looking at the first 3 bytes in the module file to be read. By definition, the first character in

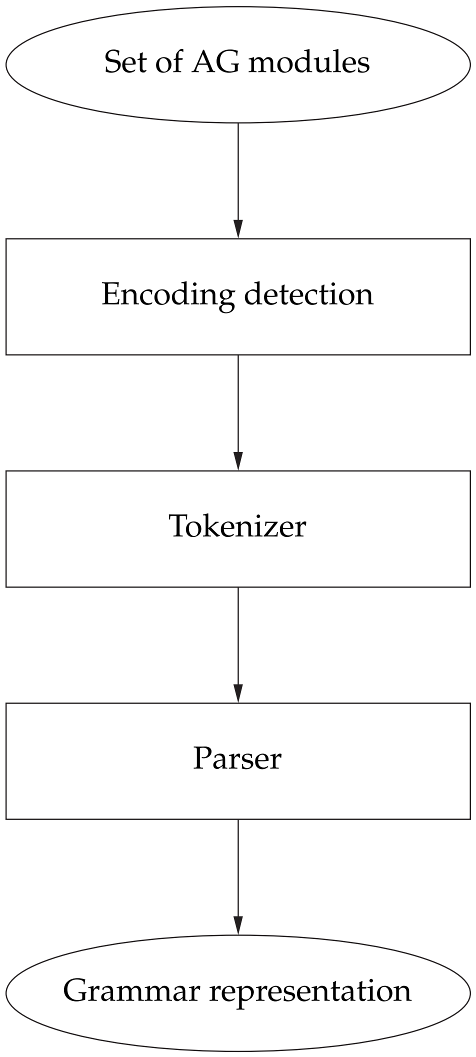


Figure 6.3: The functional blocks in the module parser

the module file must be an opening angle bracket, the opening bracket of the module header. In Unicode UTF-16, the first two bytes in a file specify the endianness of the character data. If the detection engine finds a UTF-16 byte order mark, it will read the module header assuming a UTF-16 encoding. If the byte order mark is missing, the detection engine assumes that the first byte is the character '<'. Most encoding schemes use the ASCII order for the lower 128 characters, so it is sufficient to ensure that the first byte equals to the ASCII-code of '<'. However, there are encoding systems that don't use the ASCII order (EBCDIC, for example). To handle these cases, the detection algorithm maintains a table of the character code of '<' in each of these encodings. The algorithm tries to find an encoding in the table where the character code of '<' is equal to the value of the first byte in the file, and then tries to find the string `encoding` (the name of the mandatory `encoding` meta-variable) in the header. If it finds this string with the currently guessed encoding, it verifies its guess using the given value. For a flowchart of the detection process, see figure 6.4.

6.4.2 Tokenizer

The tokenizer splits the incoming Unicode character stream in tokens. Tokens basically fall in three classes:

- Start and end markers of meta-statements
- Operators
- Identifiers. Identifiers are, for example, names of meta-variables or a complete annotation description.

6.4.3 Parser

The parser takes the tokens returned by the tokenizer as its input, and performs the following tasks:

- **Build the grammar representation**
  As the initial step in the parsing process, the parser first creates an empty

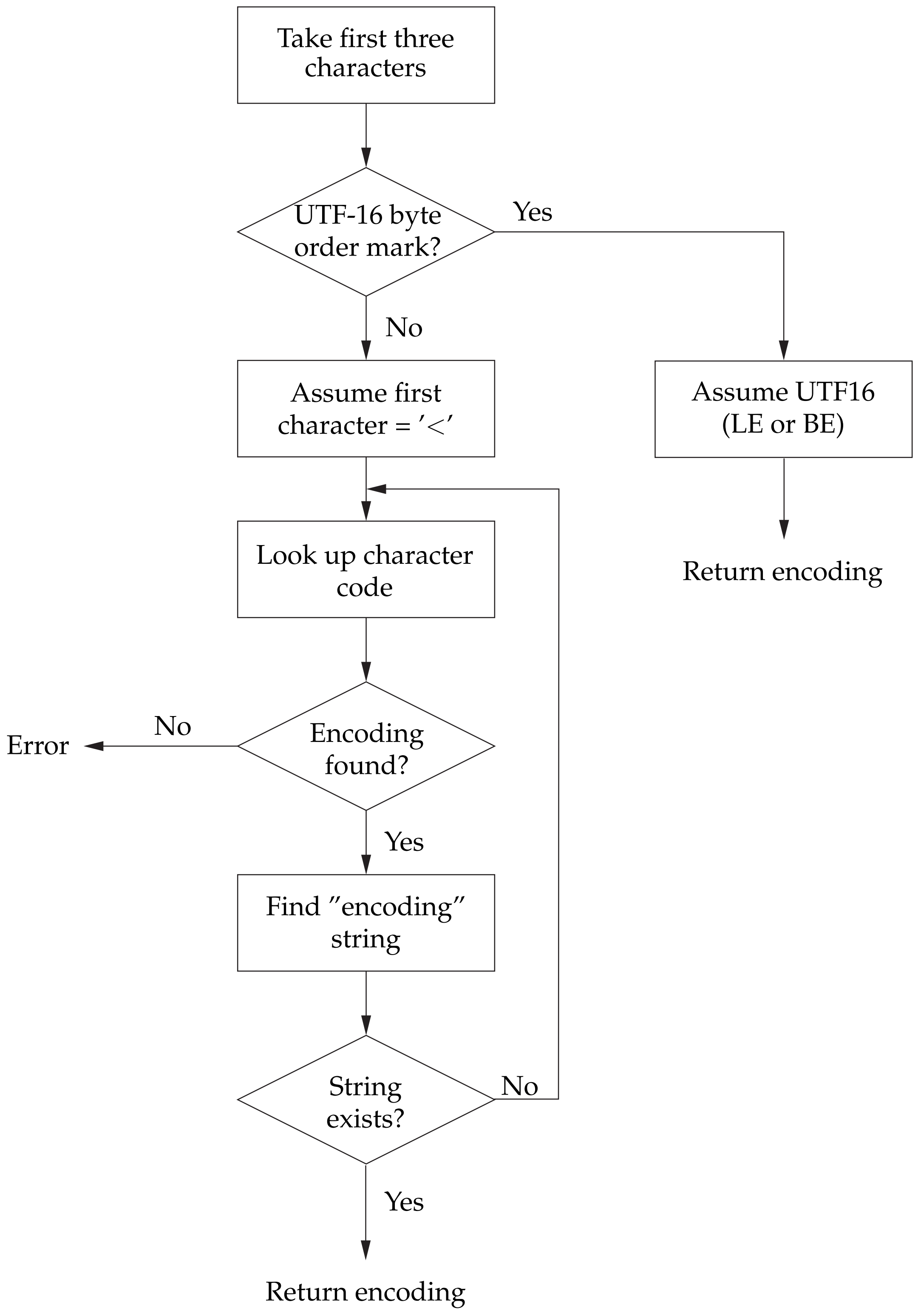


Figure 6.4: Flowchart of the encoding detection system

Rule object. The first token to be returned to the parser is a description of the annotation on the left-hand side that is delegated to a specialized parser that was developed as part of the IBM project and is not a subject of this thesis. This parser returns an annotation object that contains a well-typed feature structure. To accomplish this, the underlying type hierarchy is needed which is supplied by the type hierarchy parser. After having created the left-hand side (which becomes a *leaf expression* object, that is used to fill in the left-hand side data member of the *rule* object), the parser advances to the right-hand side, creating and filling in the appropriate structures for operators, sequences, and so on. The result is essentially a parse-tree with objects from the class hierarchy shown in figure 6.2.

- **Handle module inclusion**
  To support module inclusion, the parser maintains a stack of `FileReaders`[2]. When the parser encounters a module inclusion meta-statement, it creates a new file reader for the module to be included, pushes this file reader on the stack and passes it on to the tokenizer, which then returns tokens from the newly opened file. When the end-of-file token is returned to the parser, it pops the file reader from the stack, closes the file, and processing goes on with the next file reader on top of the stack. This way, module inclusion only affects a very small number of points in the parser and is transparent to all other parts.

- **Maintain a table of all meta-variables that were defined**
  The parser keeps track of the names and values of all meta-variables defined in the headers of the currently processed module and all its parent modules. For each module, the parser creates a hashtable that maps the names of the meta-variables declared in the header of the current module to its values. These hashtables in turn are pushed onto a stack. This way, all meta-variables down to the current module are available. On completion of the module, its hashtable is removed from the stack. One could say that a module "inherits" all meta-variables from its parent module, but siblings only have in common the meta-variables of their parent modules.

[2]A `FileReader` is a Java class that handles file I/O with real-time mapping of the character encoding used in the file to be read into Unicode.

- **Perform language validity checks on all modules included**
  Before the rules of a module are included in the grammar representation, the parser checks whether the languages, that the module to be included was developed for, are compatible with the languages of the parent modules (and ultimately of the unique language of the complete grammar). For a module to be compatible with its parent module, it must be valid for at least all of the languages its parent is valid for. Since language compatibility relies only on the standard subset relation, which is transitive, language compatibility is also a transitive relation and it is sufficient to examine the list of languages of the immediate parent only.

- **Handle conditional compilation**
  If the parser encounters an `if` meta-statement, it first checks if the meta-variable referred to has been defined. If so, it checks whether the value of the meta-variable is equal to the value given in the `if` statement or not. Only if both conditions are met, the parser goes on parsing until it reaches the closing `endif` statement or a new nested `if` statement. Otherwise, if one of the conditions is not met, everything up to the corresponding `endif` statement is ignored.

  There is one syntactic exception in the way how the parser checks values of `languages` meta-variables. The value of a languages meta-variable is a list rather than an atomic value. In this case the enclosed block of rules is compiled only if the language of the main module is included in this list[3].

6.5 The compiler

The compiler finally is the part of the system that translates a grammar representation into a set of annotation transducers. Essentially, the compiler creates one new AT for each rule found in the list of rules in the grammar representation. ATs, like all other objects so far, are represented by instances of an *Annotation Transducer* Java class.

[3]Recall that submodules may be valid for multiple languages. In such modules, special rules could be added or excluded based on the target language of the grammar.

6.5.1 Compilation of rules

The grammar representation that is the result of the parsing process is essentially a list of instances of *Rule* objects that are numbered from 0 to n (a `Vector` of rules, in Java terms). For each rule, the compiler generates one AT. Every AT is given a name that is used by the `call`-statement. The name is calculated by appending to the type name of the annotation on the left-hand side of the rule the number of the rule in the grammar. This way, it is guaranteed that the given name is unique.

6.5.2 Compilation of the right-hand side

The right-hand side is a sequence of *Expressions*, which can either be a singleton feature structure (a *Leaf Expression*), a Kleene star or plus operator, or an option operator. The latter three are represented by an instance of a *Unary Operator* class[4], which in turn contains a feature structure object.

The algorithm builds the AT in several steps. In the initial step an AT is created that contains one more state than there are elements on the right-hand side of the rule. States are numbered from 0 to $n{+}1$, where n is the number of elements on the RHS. The $(n+1)^{st}$ state is the only final state in the AT.

If no regular operators are used on the right-hand side, the states can be connected by arcs in a fairly straightforward way: State n and state $n+1$ are connected by an arc with a label whose input side is the description of the n^{th} annotation on the right-hand side. This arc will be taken when the annotation described subsumes the annotation encountered in the input at the current read position. An additional arc is added between states n and $n+1$ if there exists a rule in the grammar with a feature structure on its left-hand side that is subsumed by the n^{th} feature structure on the RHS. The input side of the label is a `call`-statement, that calls the transducer with the name corresponding to the other rule. The output side of all arc labels is set to ϵ.

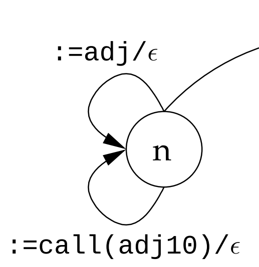
The situation becomes more difficult if regular operators are used. While an operator could be expressed by multiple rules, thus creating multiple ATs, this

[4]See figure 6.2 to recall the hierarchy of grammar representation classes.

is not desirable because many `call`-statements would have to be executed in order to loop. Instead, the AT is constructed using an algorithm based on the Thompson Construction which is normally used to construct a Finite State Automaton from a regular language:

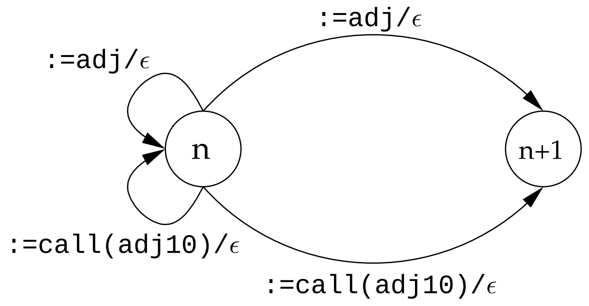
- If a Kleene star is attached to the n^{th} feature structure on the RHS, one or two arcs are added to state n that loop back to state n, one labeled with an annotation description, the other, if present, labeled with a `call` statement. An ϵ-arc (an arc with both input and output side ϵ) leads to state $n + 1$.

  Example: Assuming that the rule with number 10 has `:= adj` on its left-hand side, `:=adj*` becomes[5]:



- If a Kleene plus is attached to the n^{th} feature structure, the looping arcs are added just the same way, but the ϵ arc is replaced by one or two arcs with the same labels as on the looping arcs.

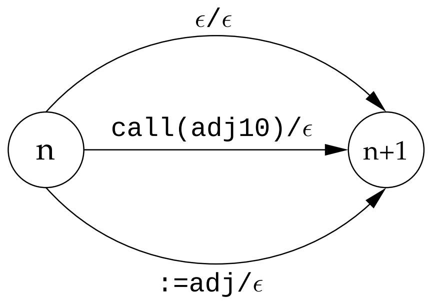
  Example: Assuming again that the rule with number 10 has `:= adj` on its left-hand side, `:=adj+` becomes:



[5]The number of the rule is important because its type `adj` and its number 10 will be used to calculate the name of the corresponding AT: `adj10`. This name will be used in the generated `call`-statement.

- If an option operator is attached to the feature structure, an ϵ-arc is added to the one or two other arcs.

  Example: `:=adj?` becomes:

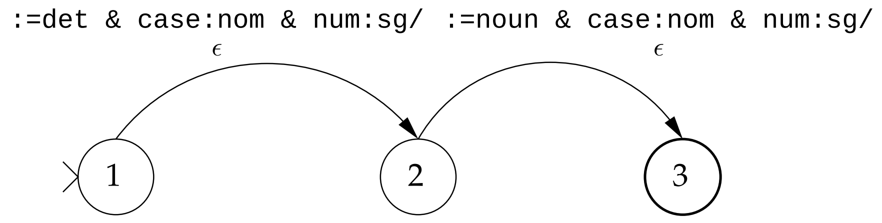


6.5.3 Compilation of the left-hand side

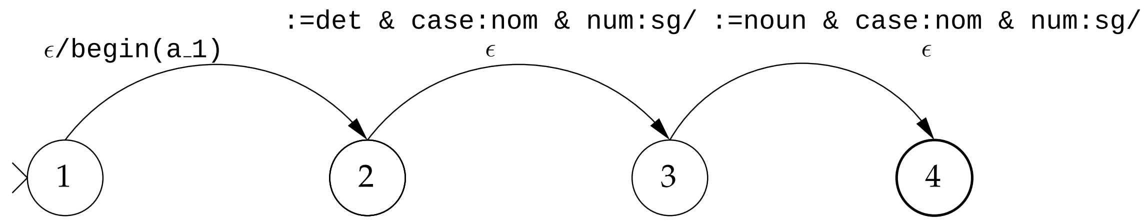
On the left-hand side, there is never more than one description of an annotation. If this annotation is of internal type, it will not be reflected at all in the resulting AT. If such an AT is called from another AT, it won't do anything else but match or fail depending on the input. However, if the annotation is of external type, arcs with output instructions to create an annotation are added to the AT in the following way:

- A new state is added to the AT that becomes the new initial state. The new and the old initial state are then connected with an arc whose label is ϵ on the input side and a `begin` instruction on the output side.
- Another new state is added to the AT that becomes the new final state. The old final state and the new one are connected with an arc, whose label is ϵ on the input side, just like above. On the output side, multiple output instructions are placed: An `end` instruction that drops an end marker for the annotation, a `create` instruction for the annotation and a number of `set` instructions that set all features to the values specified in the description of the annotation.

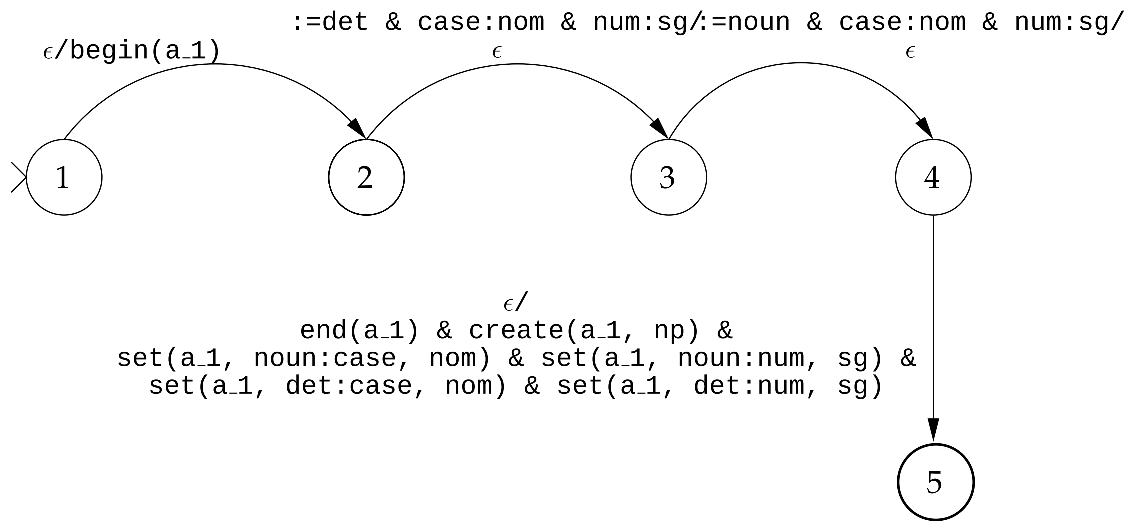
Example: For an example of the addition of output instructions, let us look at the following AT. After creation of the input sides, the AT looks like this:



Then a `begin` output-statement is added to the left of the transducer:



In the next step, the output instructions to register the end position (`end`), to create the annotation (`create`), and to set values of features (`set`) are added at the end of the transducer:


ϵ/begin(a_1)
:=det & case:nom & num:sg/
ϵ
:=noun & case:nom & num:sg/
ϵ
ϵ/
end(a_1) & create(a_1, np) &
set(a_1, noun:case, nom) & set(a_1, noun:num, sg) &
set(a_1, det:case, nom) & set(a_1, det:num, sg)
1
2
3
4
5

The result is a transducer with arcs that either contain statements on the input side, or output instructions on the output side.

6.5.4 ϵ-removal and storage

As a result of the Thompson Construction used to build the AT, multiple ϵ-transitions were added to the AT when creating output instructions, when translating Kleene star and the option operator. An ϵ-transition is a transition that has ϵ on its input side (that is a transition that matches the empty sequence). The occurrence of an ϵ on the output side is not important for ϵ-transitions.

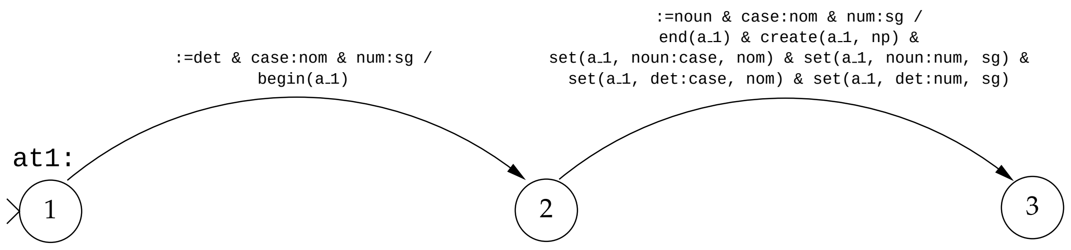
However, the execution device for ATs does not support ϵ-transitions in ATs, so they must all be removed before the AT can actually be used. In Roche & Schabes [RS97] a standard algorithm is given to remove ϵ-transitions. This algorithm requires determinized and minimized transducers. It is not possible to determinize or minimize transducers in general (consider labels with the same input symbol but different output symbols), however, an AT can be treated as an FSA for this purpose by interpreting the input and output side as one atomic symbol, which is an approach commonly used.

The objects on arcs of ATs are actually annotations. In order to remove ϵ-transitions from ATs, annotation descriptions as well as output instructions are taken to be atomic strings, and for determinization and minimization, these strings are compared. This works fine because for two annotations that are equal, the compiler generates two fully indentical annotation descriptions or output instructions, respectively.

So ϵ-transition are removed in three steps:

1. The underlying FSA is determinized.
2. The underlying FSA is minimized.
3. The ϵ-arcs (that is, arcs with ϵ on the input side) of the AT are removed by calculating the ϵ-closure over all input sides and delaying output as long as the input side remains ϵ. This is a standard ϵ removal technique for transducers.

The result of this is the following AT:


:=noun & case:nom & num:sg /
end(a_1) & create(a_1, np) &
set(a_1, noun:case, nom) & set(a_1, noun:num, sg) &
set(a_1, det:case, nom) & set(a_1, det:num, sg)
:=det & case:nom & num:sg /
begin(a_1)
at1:
1
2
3

As the last step, the completed AT is written to a file.

The result of the compilation is one file for each AT. There are as many files as there are rules in the AG. The filenames of the file are the names of the ATs, with the extension `.fst`.

6.6 Summary

- The compiler translates an AG into a set of ATs.
- There are several functional blocks: Type hierarchy parser, module parser, grammar representation, and compiler.
- The type hierarchy parser builds one global type hierarchy from all local type hierarchies included in different modules.
- The module parser reads in all modules and creates the grammar representation using the global type hierarchy obtained from the type hierarchy parser. It also handles conditional compilation and performs language-validity checking as well as character encoding detection.
- The grammar representation is a container that contains instances from a Java class hierarchy of classes that represent the different components of an AG grammar.
- The compiler functional block takes the grammar representation as its input and translates it into a set of ATs.

Chapter 7

Comparison to the JAPE system

After having described the annotation grammar formalism in its entirety, we will compare it with the JAPE system sketched in section 3.2.2.

The basic purpose is the same for both systems: They provide a general framework to manipulate annotations assigned to text based on matched input patterns. As such, they can be used as a first tier of larger text processing systems (for example, information extraction systems).

In both systems, the interface to the user is a grammar-like input syntax that describes sequences of existing annotations to be matched and actions to be taken based on the match. These grammars are then compiled into objects similar to finite-state automata for JAPE and annotation transducers for AGs. These devices target APIs provided by the annotation systems they are built upon. For JAPE, this is the GATE system, ATs are executed within IBM's Text Annotation Framework.

7.1 The grammar

In order to compare JAPE grammars with AGs , let us again consider the example given earlier in section 3.2.2.

```
Rule:  NumbersAndUnit
(({Token.kind == "number"})+:numbers {Token.kind == "unit"})
```

```
3   -->
4   :numbers.Name = {rule = "NumbersAndUnit"}
```

The example above is a rule in JAPE syntax (the example is taken from Cunningham [CMT00]). It says 'match sequences of numbers followed by a unit; create a `Name` annotation across the span of the numbers, and attribute rule with value `NumbersAndUnit`'.

We will now translate this rule in AG syntax:

```
1   :=aux_type ->
2   :=name & rule = "NumbersAndUnit", :=token & kind = "unit".
3
4   :=name & rule = "NumbersAndUnit" ->
5   :=token & kind = "number"+.
```

The result of an application of this AG rule will be the same as for the JAPE rule above (ignoring, of course, the differences in the representation of annotations). The most obvious difference is that there are two rules in the AG example instead of only one. The reason for this is that the AG formalism does not support labeling parts of matched sequences, like in line 2 of the JAPE example. By referring to the `numbers` label, the `Name`-annotation can be created such that it only spans over the `number` tokens, but not over the `unit` token. Start and end positions are implicitly set to the full extent of the sequence matched in AG rules, so we need two rules: The first rule makes sure that the right context conditions are met: Number token(s) must be followed by a unit token. We assume that in the type hierarchy, the type *aux_type* is declared to be an internal type, that is an annotation of this type is not created (so here it is just interpreted as the start symbol). The first symbol on the right-hand side is an annotation that subsumes the annotation on the left-hand side of the rule starting on line 4. So this rule can be applied. It matches one or more annotations of type *token* with the KIND feature set to `"number"`[1]. If such a sequence can be matched, an annotation of type *name* is created, its RULE feature set to `"NumbersAndUnit"`. The result of the application of both rules is the same as in the JAPE example.

[1]Note that a "token" is just an annotation of a special type. This reflects the general idea of the AG system to encode as many different linguistic objects as possible with a simple basic type, the annotation.

The above example shows quite well the similarities and differences of JAPE and AG rules.

- In JAPE, labels can be assigned to mark parts of matched sequences. These labels can be referred to on the action side (right-hand) side, which makes it possible to create an annotation over only part of a sequence matched. This is not possible in AG rules.
- Rules in an AG are arranged in a tree-like structure like rules in other CFGs: There is one "start symbol", and subsequent rules apply on annotations on the right-hand sides of other rules. The rules in a JAPE grammar are individual regular expressions that are applied independently[2].
- JAPE grammars have regular power, while annotation grammars have context-free power.
- The AG syntax has built-in support for conditional compilation, modular development and language validation. These features are absent in the JAPE syntax.
- The JAPE syntax supports the definition of macros which can be expanded at any point in the grammar. The AG syntax does not support macros.

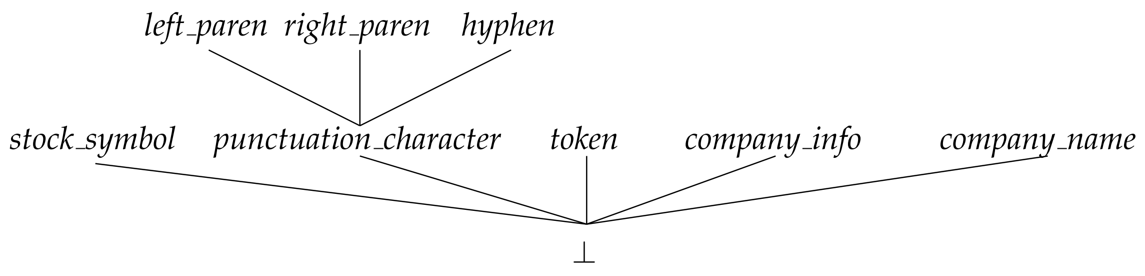
[2]Actually, the way JAPE rules apply can be chosen in the grammar: Using the "Brill-style", all rules apply at all places in the input they match, independently of each other, using the "Appelt-style", only the rule with the longest match applies. See Cunningham [CMT00] for more information.

Appendix A

A Complete Example

A.1 The type hierarchy

A.1.1 Graphical representation



Type	**Appropriate Features**
company_info	FULLNAME: *string* STOCKSYMBOL: *string*
company_name	NAME: *string* ID: *string*
stock_symbol	SYMBOL: *string* ID: *string*
left_paren	
right_paren	
hyphen	
punctuation_character	
token	TEXT: *string*

A.1.2 Textual representation `hierarchy.thy`

```
company_info (fullname:string, stocksymbol:string) external;
company_name (name:string, id:string) external;
stock_symbol (symbol:string, id:string) external;
punctuation_character external;
  left_paren external;
  right_paren external;
  hyphen external;
token (text:string) external;
```

A.2 The annotation grammar

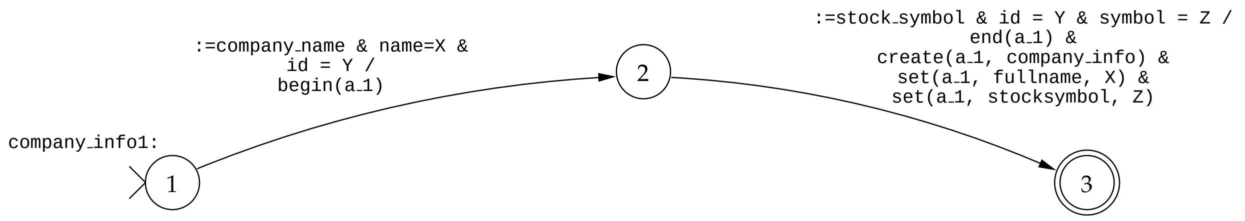
A.2.1 The main module

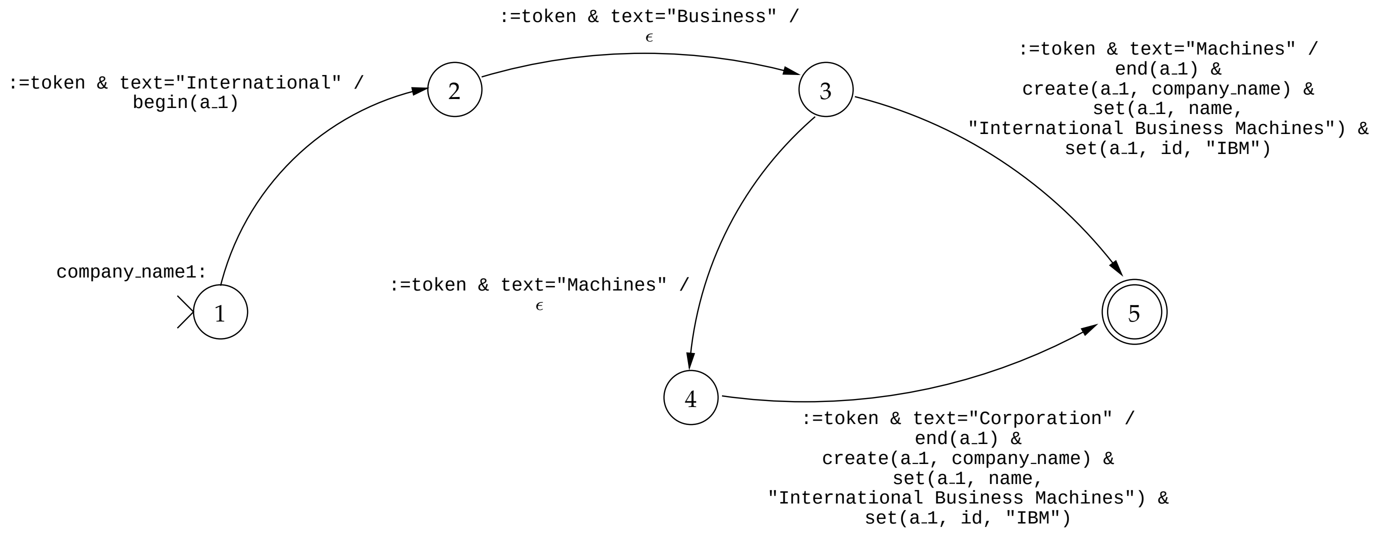
```
<main_module author="Holger Wunsch" languages="en_US"
 encoding="iso-8859-1">
<include_type_hierarchy "hierarchy.thy">

:=company_info & fullname = X & stocksymbol = Z ->
  :=company_name & name = X & id = Y,
  :=stock_symbol & id = Y, symbol = Z.

:=company_name & name = "International Business Machines" & id = "IBM" ->
 :=token & text = "International",
 :=token & text = "Business",
 :=token & text = "Machines",
 :=token & text = "Corporation"?.

<include "stocksyms.atg">
```

A.2.2 The submodule, `stocksyms.atg`

```
<sub_module author="Holger Wunsch" languages="en_US,de_DE"
 encoding="iso-8859-1">
<include_type_hierarchy "hierarchy.thy">

:=stock_symbol & symbol = "IBM-N" & id = "IBM" ->
 :=left_paren,
 :=token & text = "IBM-N",
 :=right_paren.

:=left_paren ->
 :=token & text = "(".

:=right_paren ->
 :=token & text = ")".
```

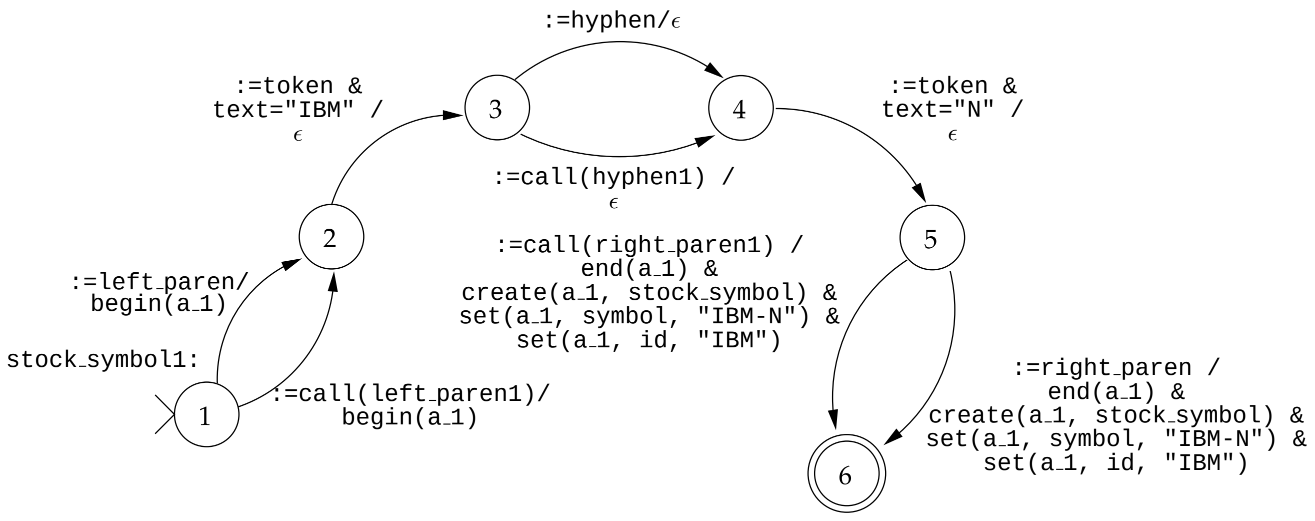
A.3 The annotation transducers

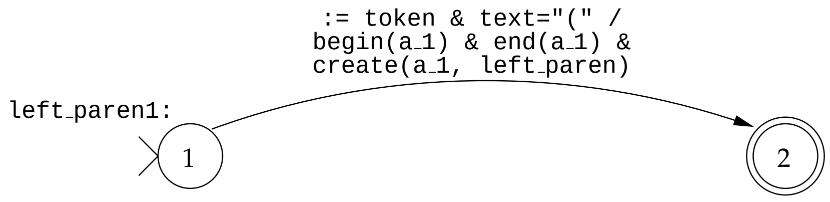

company_info1:
1
:=company_name & name=X & id = Y / begin(a_1)
2
:=stock_symbol & id = Y & symbol = Z / end(a_1) & create(a_1, company_info) & set(a_1, fullname, X) & set(a_1, stocksymbol, Z)
3

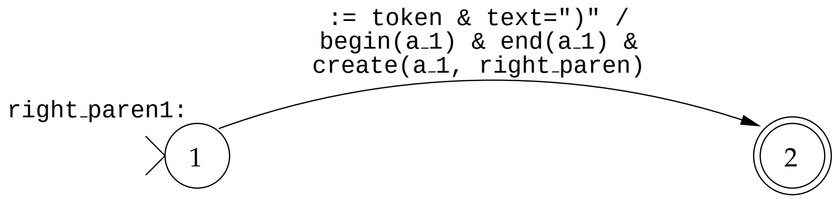

company_name1:
1
:=token & text="International" / begin(a_1)
2
:=token & text="Business" / ϵ
3
:=token & text="Machines" / end(a_1) & create(a_1, company_name) & set(a_1, name, "International Business Machines") & set(a_1, id, "IBM")
:=token & text="Machines" / ϵ
4
:=token & text="Corporation" / end(a_1) & create(a_1, company_name) & set(a_1, name, "International Business Machines") & set(a_1, id, "IBM")
5

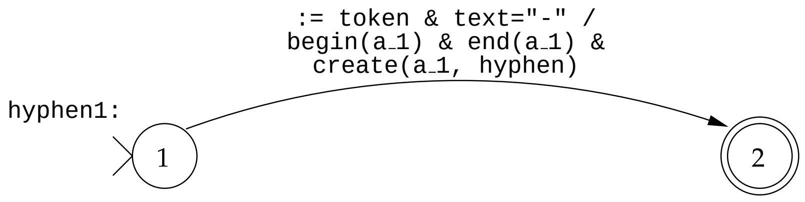
Note how variables are used in the AT `company_info1`. The variables `X` and `Y` are bound to the values found when matching a *company_name* annotation between states 1 and 2. `X` and `Y` keep their values from then on. This means that between states 2 and 3, an annotation matches only if its ID feature has the same value as the `Y` variable and the ID feature in the previously matched *company_name* annotation.

Furthermore, the variables are used to set the values of the FULLNAME and STOCKSYMBOL features in the *company_info* annotation to be created.


stock_symbol1:
:=left_paren/ begin(a_1)
:=call(left_paren1)/ begin(a_1)
:=token & text="IBM" / ϵ
:=hyphen/ϵ
:=call(hyphen1) / ϵ
:=token & text="N" / ϵ
:=call(right_paren1) / end(a_1) & create(a_1, stock_symbol) & set(a_1, symbol, "IBM-N") & set(a_1, id, "IBM")
:=right_paren / end(a_1) & create(a_1, stock_symbol) & set(a_1, symbol, "IBM-N") & set(a_1, id, "IBM")
1
2
3
4
5
6


:= token & text="(" / begin(a_1) & end(a_1) & create(a_1, left_paren)
left_paren1:
1
2


:= token & text=")" / begin(a_1) & end(a_1) & create(a_1, right_paren)
right_paren1:
1
2


:= token & text="-" / begin(a_1) & end(a_1) & create(a_1, hyphen)
hyphen1:
1
2

Appendix B

Deutsche Zusammenfassung / German Summary

Diese Magisterarbeit beschreibt Annotationsgrammatiken und ihre Übersetzung in Annotations Transducer (ATs).

Innerhalb eines Information Retrieval Systems, das automatisch relevante Information aus Texten herausfiltern und aufbereiten kann, ist der erste Schritt, den Text syntaktisch und semantisch zu analysieren. Für viele Aufgaben ist es dabei nicht nötig, den Text vollständig abzudecken, sondern nur einzelne Teile daraus zu betrachten. Um z.B. Firmennamen innerhalb eines Textes herauszufinden, ist nur die Analyse von Nominalphrasen notwendig.

Die Entwicklung solcher Software setzt eine eingehende linguistische Analyse der Gegebenheiten voraus, zudem erfordert eine Portierung eines existierenden Softwareprodukts in eine neue Sprache die Neuentwicklung zentraler Teile. Es ist daher wünschenswert, diese Teile der Software aus dem eigentlichen Programm herauszunehmen und eine Entwicklungsumgebung zu schaffen, die auch für Personen ohne weitreichende Programmierkenntnisse verständlich ist.

Diese Magisterarbeit entstand in Kooperation mit dem TAF-Projekt der IBM Deutschland GmbH, in welchem dieses Ziel verfolgt wird. Im TAF-System (TAF: Text Annotation Framework) wird die linguistische Information in *Annotationen* dargestellt. Anstelle Annotationen innerhalb eines festverdrahteten

Programms zu manipulieren, werden dafür *Annotation Transducers* verwendet, eine Verallgemeinerung von Pushdown Transducers, die über Annotationen arbeiten. Diese werden vom eigentlichen Programm ausgeführt. Solche Transducer von Hand für größere ambitionierte Projekte zu entwickeln ist sehr aufwändig. Um dieses Problem zu lösen, wird in dieser Magisterarbeit folgender Ansatz verfolgt: Anstelle Annotation Transducer von Hand implementieren zu müssen, können Entwickler eine Beschreibungssprache verwenden, die sich sehr stark an die Syntax kontextfreier Grammatiken anlehnt, und deshalb *Annotationsgrammatik* genannt wird. Diese Annotationsgrammatiken werden mit einem Compiler in Annotation Transducer übersetzt.

Diese Strategie bringt einige Vorteile mit sich. Linguisten, die typischerweise an der Entwicklung der genannten Analysemodule mitarbeiten, sind mit Grammatikformalismen bestens vertraut. Insgesamt sind Annotationsgrammatiken leichter verständlich und somit besser implementierbar und einfacher zu warten.

Zusätzlich werden in den Annotationsgrammatik-Formalismus Elemente integriert, die es erlauben, Grammatiken modular zu entwickeln. Inhaltlich zusammenhängende Regeln können so in einem Modul gruppiert und von anderen Regeln strukturell getrennt werden. Dadurch werden große Grammatiken übersichtlicher. Zudem wird durch die Modularität die Wiederverwendung von Teilen einer Grammatik in neuen Projekten vereinfacht.

Der Annotationsgrammatik-Formalismus und der Compiler werden bei IBM Deutschland in einigen Projekten zur Named Entity Recognition eingesetzt.

Bibliography

[Abn91] Steven P. Abney. Parsing by chunks. In Robert Berwick, Steven Abney, and Carol Tenny, editors, *Principle-Based Parsing*. Kluwer Academic Publishers, Dordrecht, 1991.

[Abn96] Steven P. Abney. Partial parsing via finite-state cascades, 1996.

[Apa02] Apache XML Project. Xerces2 java parser. `http://xml.apache.org/xerces2-j/index.html`, 2002.

[Bat78] Madeleine Bates. The theory and practice of augmented transition network grammars. In Leonard Bolc, editor, *Natural Language Communication with Computers*, volume 63 of *Lecture Notes in Computer Science*, pages 191–259. Springer Verlag, Berlin, Heidelberg, New York, 1978.

[BB00] S. Bird and P. Buneman. Towards a query language for annotation graphs, 2000.

[BDG$^+$00] S. Bird, D. Day, J. Garofolo, J. Henderson, C. Laprun, and M. Liberman. ATLAS: A Flexible and Extensible Architecture for Linguistic Annotation, 2000. In Proceedings of the Second International Language Resources and Evaluation Conference. Paris: European Language Resources Association.
`http://citeseer.nj.nec.com/bird00atlas.html`.

[BL99] S. Bird and M. Liberman. Annotation graphs as a framework for multidimensional linguistic data analysis, 1999.

[BL00] Steven Bird and Mark Liberman. A formal framework for linguistic annotation. Technical report MS-CIS-99-01, Department of

Computer and Information Science, University of Pennsylvania, 2000. `http://xxx.lanl.gov/abs/cs.CL/9903003`.

[Bra98] Sascha Brawer. Patti – Compiling Unification-Based Finite State Automata into Machine Instructions for a Superscalar Pipelined RISC Processor. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, 1998.

[Bro02] David Brownell. The sax project page. `http://www.saxproject.org`, 2002.

[Car92] Bob Carpenter. *The logic of typed feature structures*. Cambridge University Press, New York, NY, 1992.

[CBPW00] H. Cunningham, K. Bontcheva, W. Peters, and Y. Wilks. Uniform language resource access and distribution in the context of a General Architecture for Text Engineering (GATE). *Proceedings of the Workshop on Ontologies and Language Resources (OntoLex 2000)*, 2000. `http://gate.ac.uk/sale/ontolex/ontolex.ps`.

[CHGW97] H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. Software infrastructure for natural language processing, 1997.

[CMT00] Hamish Cunningham, Diana Maynard, and Valentin Tablan. JAPE: A Java Annotation Patterns Engine, 2000. Research memo CS-00-10, Institute for Language, Speech and Hearing (ILASH), and Department of Computer Science, University of Sheffield, UK, `http://www.dcs.shef.ac.uk/˜hamish`.

[CP94] Bob Carpenter and Gerald Penn. ALE User's Guide Version 2.0. Technical report, Laboratory for Computational Linguistics, Carnegie Mellon University, Pittsburgh, 1994.

[Cun99] Hamish Cunningham. Information extraction - a user guide, 1999. Research memo CS-00-10, Institute for Language, Speech and Hearing (ILASH), and Department of Computer Science, University of Sheffield, UK, `http://www.dcs.shef.ac.uk/˜hamish`.

[Cun00] Hamish Cunningham. *Software Architecture for Language Engineering*. PhD thesis, University of Sheffield, June 2000.

[GLF$^+$86] John S. Garofolo, Lori F. Lamel, William M. Fisher, Jonathon G. Fiscus, David S. Pallett, and Nanc L. Dahlgren. The DARPA TIMIT Acoustic-Phonetic Continuous Speech Corpus CDROM. NIST, 1986. `www.ldc.upenn.edu/lol/docs/TIMIT.html`.

[Gri98] Ralph Grishman. Tipster text architecture design. Technical report, New York University, 1998.

[GW01] Thilo Götz and Holger Wunsch. *An Abstract Machine Approach to Finite State Transduction Over Large Character Sets*, 2001.

[HAB$^+$97] Jerry R. Hobbs, Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel, and Mabry Tyson. FASTUS: A Cascaded Finite-State Transducer for Extracting Information from Natural-Language Text. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Natural Language Processing*, pages 383 – 406. MIT Press, Cambridge, MA, 1997.

[HM01] Elliotte Rusty Harold and W. Scott Means. *XML in a nutshell*. O'Reilly, 2001.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Boston, München, second edition, 2001.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Reading, MA, 1979.

[Hud02] Scott E. Hudson. Cup parser generator for java. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`, 2002.

[KK94] Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 1994.

[Kla00] Herbert Klaeren. *Skriptum zur Softwaretechnik-Vorlesung im Sommersemester 2000*. Universität Tübingen, 2000.

[Kle02] Gerwin Klein. JFlex - The Fast Scanner Generator for Java. `http://www.jflex.de`, 2002.

[LC98] Laura Lemay and Rogers Cadenhead. *Java 1.2 programmieren in 21 Tagen*. Markt & Technik (SAMS), München, 1998. Original English title: "Teach yourself Java 1.2 in 21 Days", SAMS Publishing.

[NIS00] National Institute for Standards and Technology, `http://www.itl.nist.gov/iaui/894.02/related_projects/tipster/`. *TIPSTER Text Program*, 2000.

[Pen95] Gerald Penn. Compiling Typed Attribute-Value Logic Grammars. In H. Bunt and M. Tomita, editors, *Current Issues in Parsing Technologies*, volume 2. Kluwer, 1995.

[RS97] Emmanuel Roche and Yves Schabes. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Natural Language Processing*. MIT Press, Cambridge, MA, 1997.

[Str92] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison Wesley, second edition, 1992. Original English title: "The C++ programming language, second edition, reprinted with corrections".

[Sun02] Sun Microsystems, Inc. *Java 2 SDK, Standard Edition, Documentation*. `http://java.sun.com`, 2002.

[vN00] Gertjan van Noord. Treatment of epsilon moves in subset construction. *Computational Linguistics*, 2000.

[vNG01] Gertjan van Noord and Dale Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:263 – 286, 2001.

[W3C02] W3C. *Document Object Model (DOM)*, 2002. `http://www.w3.org/DOM/`.

[Woo70a] W. A. Woods. An experimental parsing system for transition network grammars. In R. Rustin, editor, *Natural Language Processing*, pages 111 – 154. Algorithmics Press, 1970.

[Woo70b] W. A. Woods. Transition network grammars for natural language analysis. *C ACM 3(10)*, pages 591 – 666, 1970.