UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Feature grammar systems. Incremental maintenance of indexes to digital media warehouses

Windhouwer, M.A.

Publication date
2003

Link to publication

Citation for published version (APA):
Windhouwer, M. A. (2003). *Feature grammar systems. Incremental maintenance of indexes to digital media warehouses*. [Thesis, externally prepared, Universiteit van Amsterdam].

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:18 Nov 2026

Chapter 3

Feature Grammar Language

> *There is nothing that can be said by mathematical symbols and relations which cannot also be said by words. The converse, however, is false. Much that can be and is said by words cannot successfully be put into equations, because it is nonsense.*
>
> C. Truesdell

The mathematical notation introduced and used in the previous chapter is, although precise, not very convenient in everyday life. In this chapter a more practical notation is introduced: the feature grammar language. This language describes how a feature grammar is specified, *i. e.* it is a *meta-language*.

In Appendix A the complete specification of the language is given using the *Extended Backus-Naur Form* (EBNF). The ancestor of EBNF, *Backus-Naur Form* (BNF), is a CF grammar notation mostly used for specifying programming languages, *i. e.* it was first used to define ALGOL 60. In the upcoming sections parts of the language are introduced by language snippets related to the example feature grammar. Appendix B contains the collection of feature grammars, which has been build for the various case studies (see Chapter 7).

The next section introduces the core of the language which directly maps on the formalization of feature grammar systems. Subsequently additions to this language will be described whose main purpose is to provide shortcuts for developers or to steer the FDE or FDS in their analysis of the grammar.

3.1 The Basic Feature Grammar Language

The core of the feature grammar system is formed by the shared alphabet $V = (D \cup N \cup T)$, the production rules as distributed over the various grammar components, the start and stop conditions of these components, and the start symbol. In the subsequent subsections the notation for these core components is shown.

3.1.1 Production Rules

Production rules form the heart of every grammar. As Definition 2.1 and the section on ambiguous grammars showed there are alternative interpretations possible for one non-terminal. The extended notation (in this specific form also called *regular right part grammars* (RRPG) [LaL77]) makes it possible to combine these alternatives syntactically into one rule.

```
1  Image      : Location ( Color Class )?;
2  Color      : RGB* Number Prevalent Saturation;
3  Class      : Graphic | Photo Skin Faces;
4  RGB        : Red Green Blue;

5  Location   : url;
6  Number     : int;
7  Prevalent  : flt;
8  Saturation : flt;
9  Graphic    : bit;
10 Photo      : bit;
11 Skin       : bitmap;
12 Faces      : int;
13 Red        : int;
14 Green      : int;
15 Blue       : int;
```

The RGB non-terminal is introduced into the example to make it rich enough to illustrate some extended features.

In this notation a production rule's LHS and RHS are separated by a colon and the rule is terminated with a semicolon. Alternative representations for the same LHS are grouped together and then separated by vertical bars, | (see the $Class$ rule). Symbol sequences, *i. e.* optional, star and positive closure, are indicated by respectively ?, $*$ and $+$ occurrence indicators (see the $Image$ and $Color$ rules). Furthermore, symbols can be combined into groups using brackets. These groups can have their own occurrence indicators and embedded alternatives (see once more the $Image$ rule).

All these extended constructs for production rules can be rewritten into the basic formal version of the production rules. For symbol sequences there are two methods. The first method of rewriting uses the *recursive* interpretation. In this interpretation the RGB^* sequence is rewritten into the following formal rules:

$$
\begin{aligned}
Color &\rightarrow Number\ Prevalent\ Saturation \\
Color &\rightarrow \alpha\ Number\ Prevalent\ Saturation \\
\alpha &\rightarrow RGB \\
\alpha &\rightarrow RGB\ \alpha \\
RGB &\rightarrow Red\ Green\ Blue
\end{aligned}
$$

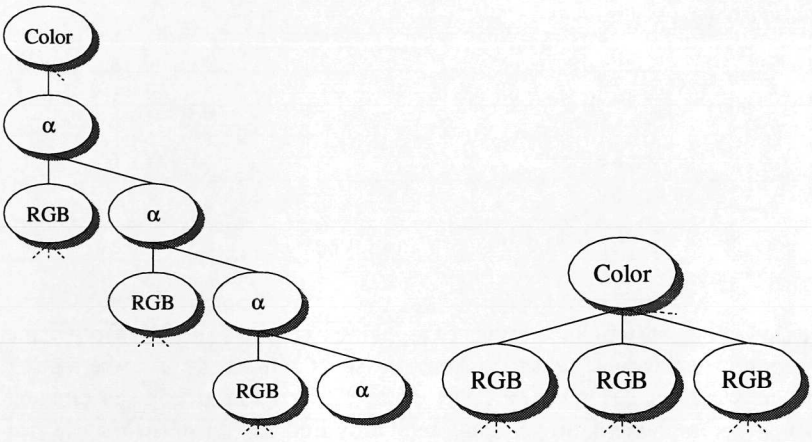

Figure 3.1: (a) A right-recursive and (b) an iterative interpretation of a list construct

This interpretation has as advantage that it is easy to explain because the transformation to a basic CF grammar is simple. Disadvantages are the introduction of anonymous variables (in this case α) and the lopsided parse tree (see Figure 3.1.a), which in generally does not correspond to ones intuition.

The *iterative* interpretation of symbol sequences is more intuitive. It sees the RGB^* sequence as an abbreviation of:

$$
\begin{aligned}
&Color \rightarrow Number\ Prevalent\ Saturation \\
&Color \rightarrow RGB\ Number\ Prevalent\ Saturation \\
&Color \rightarrow RGB\ RGB\ Number\ Prevalent\ Saturation \\
&Color \rightarrow RGB\ RGB\ RGB\ Number\ Prevalent\ Saturation \\
&Color \rightarrow \ldots \\
&\quad \ldots \\
&RGB \rightarrow Red\ Green\ Blue
\end{aligned}
$$

The advantage of this interpretation is a beautiful parse tree (see Figure 3.1.b), but has as disadvantages that it involves an infinite number of production rules and that the nodes in the parse tree have a varying fan-out.

Next to rewrites the occurrence indicators may also directly be interpreted by the parser implementation (see [GJ98]), *i. e.* using IF-statements for optional symbols, WHILE-statements for star closure and REPEAT-statements for positive closure. The resulting parse tree will adhere to the iterative interpretation. However, this implements these indicators in a greedy fashion and thus favors greedy alternatives. Notice

Atom name	*Substitution language (as a regular expression)*
bit	`^(0|1|(true)|(false))$`
chr	`^.$`
int	`^-?[0-9]+$`
flt	`^-?[0-9]+\.[0-9]+([Ee][-+]?[0-9]+)?$`
str	`^.*$`

Table 3.1: Default atom types

that is the case with most implementations of regular expressions [Fri02]. Only some languages support special constructs to make expressions non-greedy, *e. g.* the regular expression syntax used by *Tcl 8* and *Perl 5* supports `*?` for a non-greedy star-closure. A parser using this implementation strategy may only find one parse $(a(b(c\ d)))$ for the input sentence $(c\ d)$ using this grammar:

```
1 a: b e?;
2 b: c d?;
3 e: d;
```

A second parse tree, which postpones the consumption of d, *i. e.* $a(b(c)\ e(d))$, will *not* be found by this parser implementation.

Which of these alternative interpretations of symbol sequences is chosen is a FDE implementation decision. In Chapter 4 the actual implementation of the FDE will be discussed and in Section 4.3.1 this specific decision will be made and explained.

Symbol groups are rewritten by introducing anonymous symbols.

$$\begin{aligned} Image &\rightarrow Location \\ Image &\rightarrow Location\ \beta \\ \beta &\rightarrow Color\ Class \end{aligned}$$

The anonymous symbol, in this case β, inherits the occurrence indicators and embedded alternatives of the symbol group. Notice that the $Image$ rule is duplicated to eliminate the sequence once.

3.1.2 Atoms

The feature grammar language supports a default set of atom types, *e. g.* several numeric types and strings (see Table 3.1). However, additional types may be supported by the DBMS, for example by using an extension mechanism. To be able for a feature grammar to use these new types they have to be defined and a REG validation language can be added.

```
%atom          image::bitmap {^[01]*$};
```

The *bitmap* type, available in the extension module *image*, is defined. The *bitmap* specific rule defines the REG substitution language from which a valid *bitmap* atom value is a member. Using this regular expression the FDE can validate a specific *bitmap* instantiation. When there is no regular expression the default validation code provided by the system will always accept each terminal of this atomic type.

Next to the atom definitions there are also atom declarations. Using an atom declaration several production rules can be summarized in one command.

```
%atom flt      Prevalent, Saturation;
```

This declaration is a simple shortcut for these two rules:

```
Prevalent  : flt;
Saturation : flt;
```

Notice, once more (see also Section 2.2.2), that the application of this shortcut helps to make the production rules semantically more meaningful, *i. e.* more useful for human consumption.

3.1.3 Detectors

The production rules already provide insight in which symbol belongs to the set of terminals and non-terminals, *i. e.* non-terminals appear as the LHS of a rule. To get the set of detector symbols those are explicitly declared.

```
%detector    Color(preceding::Location/url);

%detector    Graphic(preceding::Number/int,
                 preceding::Prevalent/flt,
                 preceding::Saturation/flt);
%detector    Photo(preceding::Color);

%detector    Skin(preceding::Location/url);
%detector    Faces(preceding::bitmap);
```

The output/input dependency is formally represented by the REG language R_i for detector d_i. In the detector declaration for d_i R_i is represented as a set of regular path expressions. In Section 2.1.1 with the introduction of REG grammars it was already stated that there are many extended regular expression languages, *e. g.* like the one used for the specification of the atom substitution languages. For the specification of the regular path expressions the *XPath* language [W3C01d] is adopted by the feature grammar language.

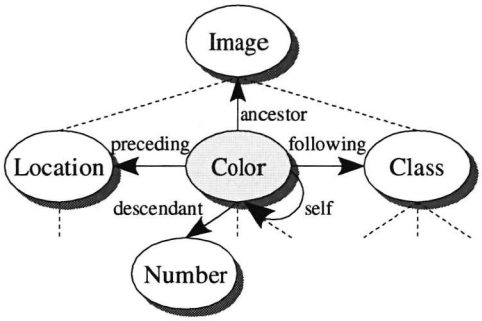

Figure 3.2: The XPath axes

The XPath Language

The XPath language is a W3C standard for describing paths in a document in the eXtensible Markup Language (XML) [W3C00]. XML, derived from the older Standard Generalized Markup Language (SGML) standard, is becoming a standard document format for data exchange between systems on the WWW. An XML document describes a tree structure of nodes. Each tree node is identified by an element name, encloses an area in the document identified by the opening and closing tags, may have zero or more textual values, may have a set of zero or more attributes, *i. e.* name/value pairs, and may be the parent of zero or more child nodes. The parse trees, as encountered until now, map naturally on this document format.

The basic XPath expression consist of a sequence of steps separated by / or //, where the first indicates "goto a matching child" and the latter "goto a matching descendant". An example XPath expression is: `preceding::Location/url [ xf:starts-with(., "http://") ]` .

The first step, `preceding::Location`, is evaluated in the context of the current node, for example the new *Color* detector node. The evaluation of this step may result in zero or more result nodes. In this case it leads to the *Location* node. The next step, `url[xf:starts-with(.,"http://")]`, is evaluated for each of these result nodes. And selects the node *url* which satisfies the predicate. As this is the last step the result of this expression contains only this node. The upcoming paragraphs will describe the basic building blocks of each step: the optional axis, the nodetest and the set of zero or more step qualifiers.

The XPath specification defines a total of 13 axes to traverse from one node to another. These axes describe the whole parse tree from the view point of the context node: the *ancestor*, *descendant*, *following*, *preceding* and *self* axes (see also Figure 3.2). Other axes form sub- and supersets from these basic axes: the *child*, *parent*, *descendant-or-self*, *following-sibling*, *preceding-sibling* and *ancestor-or-self* axes. Most of these axes are forward axes, *i. e.* nodes are traversed in the order in

which they were added to the tree (the document order). In contrast, the *parent*, *ancestor*, *ancestor-or-self*, *preceding*, and *preceding-sibling* axes traverse the nodes in reverse document order.

The axis is used to select the set of possible nodes to move to. This set may be further reduced using a node test: either an exact match on the symbol name or a wildcard match.

The last part of the step specification is a, possibly empty, set of step qualifiers. A step qualifier takes the form of a predicate, *i. e.* a boolean expression on the context of the step. In this expression logical comparisons and an extensive set of functions may be used. The example expression uses the `xf:starts-with` function to check if the value of the terminal *url* uses the HTTP protocol. The value of the terminal is indicated by the single dot, which is the abbreviated syntax for `self::node()`.

All kinds of (user defined) functions are part of the XPath standard. Due to this a full fledged XPath expression may have a selective power that goes beyond a standard regular expression language. To stay in line with the formal feature grammar system of Chapter 2 the XPath expressions used for detector parameter binding are limited to axis steps and symbol name node tests[1].

This subset of XPath expression language is adopted by the feature grammar language with (initially) two convenient changes:

1. as parameter path expressions should, enforced by the lPC rewriting mechanism, point backward into the tree the default axis in the first step of a path expression is `preceding::` instead of `child::`, notice that in all subsequent steps `child::` is still the default;

2. XPath expressions always return a node set, however, in a feature grammar the interest is mainly for, depending on the axis (forward or reverse), the first or last item in the nodeset (sorted on document order).

Using these path expressions the input sentence for a detector d_i can be selected. When one of the expressions results in an empty node set, *i. e.* $R_i \not\subseteq path(t_j)$, the start condition of the grammar component is not satisfied, and the detector function f_i will not be executed, which in the leftmost derivation process results in rejection of the detector symbol.

To retrieve the value of a node the XPath function `text()` is used. This function returns a concatenation of all values under the inner node. This allows the use of a non-terminal, which may carry more semantics, to indicate a terminal's value, *e. g.* the XPath expression `Location` is equal to `Location/url`. This helps once more to keep the feature grammar specification more semantic oriented (see Section 3.1.2).

[1] The extension to a more complete version of the XPath standard would need a reevaluation of the role of R_i.

Detector Confidence

All detectors should at least return one information token (see Definition 2.9). As indicated in Section 2.2.4 the disambiguation process would benefit from knowledge about the detector confidence. By enforcing the output of this confidence level, *i. e.* making it a default rewrite of a detector production rule, the developer of a feature grammar can conceptually write detectors which output an empty sentence. This can be used to allow the use of the detector symbol to model a binary decision, *i. e.* using the partiality of the detector function. The presence of the detector symbol in the parse forest then indicates the success of the function and thus the (maybe in-confident) validity of a concept. The absence of the confidence level, and thus of the symbol, is then used to model the failure.

These feature grammar declarations and rules:

```
1 %detector    Color(Location);

2 %detector    Graphic(Number, Prevalent, Saturation);
3 %detector    Photo(Color);

4 %detector    Skin(Location);
5 %detector    Faces(bitmap);

6 Color       : RGB* Number Prevalent Saturation;
7 Class       : Graphic | Photo Skin Faces;
```

are rewritten into the following formal rules:

$$
\begin{aligned}
Color &\rightarrow \rho\ Number\ Prevalent\ Saturation \\
Color &\rightarrow \rho\ \alpha\ Number\ Prevalent\ Saturation \\
Graphic &\rightarrow \rho \\
Photo &\rightarrow \rho \\
Skin &\rightarrow \rho\ bitmap \\
Faces &\rightarrow \rho\ int
\end{aligned}
$$

Notice that the feature grammar does not contain any explicit rules for the *Graphic* and *Photo* detectors, they are added to feature grammar system by the rewrite. The special terminal ρ describes the confidence value. This value may be seen as an annotation (or attribute) of the LHS (see also Figure 2.5).

3.1.4 The Start Symbol

The feature grammar language as defined until now only misses the declaration of the start symbol.

```
1 | %start        Image(child::Location);
```

This start declaration provides the information to construct the special grammar component G_S. First of all the dummy start symbol S_S directly passes control on to the *Image* symbol, in this case a non-terminal. Furthermore, the parameter defines which initial words the special detector function f_S should generate, *i. e.* in this case one word: a *Location* instantiation. How f_S determines this instantiation depends on the implementation, *e. g.* ask the user or look for the sentence on the command line of the FDE.

For specification of these parameters a special subset of XPath expressions is, once more, adopted. In this case only forward steps are allowed, *i. e.* the parse tree can only be traversed from the root to its descendants. This also means that the original default axis of XPath is used in these path expressions, *i. e.* `child::`.

3.2 The Extended Feature Grammar Language

The core of the language has been defined in the previous section. However, to make the life of a developer easier several shortcuts have been introduced. Other additions provide the developer with the possibility to influence the analysis and usage of the feature grammar by the tools in the Acoi system architecture.

3.2.1 Production Rules

Additional Sequence Types

Symbol sequences, like the positive or star closure, lead to collections of symbols. The most natural form for storing this collection in a DBMS is a list, as it guarantees that the symbols can be reconstructed in exactly the same order.

```
1 | Color       : RGB[*] Number Prevalent Saturation;
```

Notice that this rule is equivalent to the original *Color* rule, as list is the default type. However, this may not always be needed and, as keeping information about the order of the symbols costs storage space, the feature grammar developer can give a hint that the collection type may be changed to a set.

```
1 | Color       : RGB{*} Number Prevalent Saturation;
```

When a list is limited in size another optimization may be to turn the type into a tuple, which has better selection properties.

```
1 | Color       : RGB<16:16> Number Prevalent Saturation;
```

This example shows another addition to the feature grammar language: it allows to exactly specify, *i. e.* with a lower and upper bound, how many symbols are allowed in the collection (which may be of any type). This case is equivalent to a rule where the *Color* rule contains exactly 16 *RGB* non-terminals. When the lower bound is omitted it defaults to zero.

These constructs are all very tight related to the storage model for the parse trees, and thus will be revisited in Chapter 5.

Constants

The extended language also allows a constant of a builtin type to be placed as a symbol in the right-hand side of a rule.

```
1 Segment     : Scene*;
2 Scene       : Begin End Type;
3 Type        : "tennis" Tennis;
4 Type        : "closeup";
5 Type        : "audience";
6 Type        : "other";
```

This set of example rules describes the type of scenes found in a video of a tennis match. For each scene the *Segment* detector finds it also determines the type and puts this as a string token in the output sentence. This token now determines which alternative rule of *Type* is validated, *i. e.* for a tennis scene the *Tennis* detector will be called. This spares the need for an explicit whitebox detector (to be discussed in the next section).

```
1 %detector  TennisType [ str = "tennis" ];

2 Type       : str TennisType Tennis;
```

The other types can be handled in the same fashion. Furthermore, notice that another approach may be to add a normal detector for each type, which would implement the type detection algorithm now present in the *Segment* detector. The need for the constant would then disappear, and it would become possible to have several types for the same scene, *i. e.* ambiguity.

3.2.2 Detectors

Whitebox Detectors

One step in the annotation extraction process is the combination of the low-level features into high-level concepts. One way to do this combination is through a binary decision rule. Recall the description of the XPath language: the last part of the step specification consists of a, possibly empty, set of step qualifiers. Each step qualifier is

a predicate on the context of the step. *Whitebox detectors* provide the preferred way to embed such a predicate in the feature grammar.

In step qualifiers all XPath expressions are allowed, but not all of them will result in a boolean value. For these situations the XPath specification [W3C01d] provides these rules to derive the predicate truth value:

1. if the result of the expression is an empty node set, the predicate truth value is *false*;
2. if the result is one numeric value, this value is rounded and compared to the context position, *i. e.* the position of the context node in the processed sequence of nodes, this will always be *one* as the processed sequence contains only the detector node;
3. if the result is one boolean value, the predicate truth value is equal to this boolean value;
4. if the result node set contains at least one node, the predicate truth value is *true*;
5. otherwise there is a serious error and a runtime error will be raised.

When the predicate truth value equals *true* the detector symbol will be accepted, and otherwise rejected.

This partial feature grammar shows the simple decision rule for the *Photo* detector:

```
%detector    Photo    [
                        Number      > 200
                    and Prevalent   <   0.26
                    and Saturation  <   0.67 ];
```

On the basis of the number of colors, the existence of a prevalent color and the average saturation of the colors the image is classified as a photo (or not). But XPath allows also more advanced expressions, *e. g.* these quantified expressions to extract a color map concept:

```
%detector    ColorMap [
                  some $RGB in RGB satisfies
                       $RGB/Red != $RGB/Green
                    or $RGB/Red != $RGB/Blue ];

%detector    GrayMap  [
                 every $RGB in RGB satisfies
                       $RGB/Red = $RGB/Green
                   and $RGB/Red = $RGB/Blue ];

Color       : RGB* Map Number Prevalent Saturation;
Map         : ColorMap | GrayMap;
```

Plugin name	*Description*
exec	Execute an external program
perl	Execute a *Perl* script
matlab	Execute a *Matlab* script

Table 3.2: Default plugins

The *ColorMap* concept is validated by checking the existence of one or more non-gray pixels. While the *GrayMap* does the opposite check: the existence of only gray pixels.

To emphasize the difference between whitebox detectors the original detectors, where the implementation is not specified within the feature grammar itself, are called *blackbox detectors*.

Plugins

The blackbox detectors can do basically everything that is supported by the host language. However, this may also ask for extensive knowledge of the in and outs of the host language and its interfaces (APIs) by the developer. In this section *plugins* are introduced. A plugin frees the developer from this low-level knowledge, either by completely taking over the burden of low-level coding or by transferring the coding task to an external application. This external application may provide the developer with a higher-level language to code his annotation extraction algorithms in.

The task implemented by a plugin always follows the same steps. However, these steps need to be adapted to the specific detector. The detector specific knowledge embedded in a feature grammar is the in- and output specification. So the plugin needs access to this information. This access is provided by the symbol table. The symbol table contains information about all the symbols in the grammar and their associations described in the component production rules. Due to these privileges plugins should only be developed by experts with knowledge about the Acoi API.

```
1 %detector    matlab::Color(Location);
```

This detector communicates with the Matlab engine [Mat01], which is linked into the FDE or runs as a separate server. It assumes that the *Color* command exists in Matlab and calls it with the *Location* parameter. The answer of the Matlab engine is parsed and the data is returned as the output sentence.

The feature grammar language also allows a plugin to get its implementation directly from the feature grammar, *i. e.* as in the case of whitebox detectors. In fact whitebox detectors are by default available as the *xpath* plugin.

```
1 %detector    xpath::Photo [
2                          Number      > 200
```

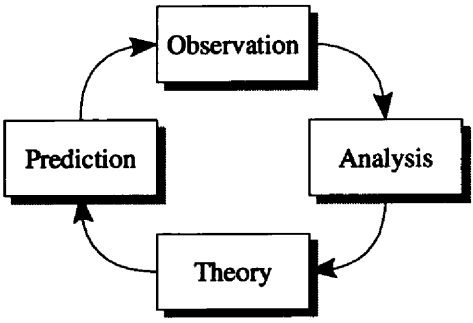

Figure 3.3: Emperical cycle

```
3                    and Prevalent  <   0.26
4                    and Saturation <   0.67 ];
```

This code snippet is equivalent to the previous *Photo* whitebox detector declaration.

In the same vain other whitebox plugins can be developed. The Acoi system provides a default library function to instantiate templates with embedded XPath expressions.

Classifiers

Classifiers are an alternative to whitebox detectors to validate a concept. Instead of decision rules, most likely constructed by a human expert, they provide bridges to machine learning algorithms.

Machine learning algorithms can be characterized by the cycle shown in Figure 3.3. The process starts with a number of observations. These observations are analyzed to find patterns. If patterns are found a theory, or hypothesis, is formulated to explain the pattern. This theory is used to predict new phenomena that can be verified by new observations.

Take for example this classifier:

```
1 %classifier  bpnn::Faces(Skin);
```

The classifier *Faces* makes use of the plugin *bpnn*. The *bpnn* plugin, written by an expert, manages *back propagation neural networks*. The plugin can handle the generic steps of the classification cycle for this specific machine learning algorithm.

A back propagation neural network learns incrementally, *i. e.* the theory is updated after each observation is seen by the algorithm. The observations consist of a set of input sentences the developer feeds to the FDE. An observation for this classifier would contain the fact that the image contains one face. When the *Faces* detector is encountered its input, the `Skin/bitmap` terminal, is selected from the parse tree. The

Classifier plugin name	*Description*
decrules	Learn/use C4.5 decision rules [Qui93]
bpnn	Train/use a neural network [Mit97]

Table 3.3: Default classifier plugins

developer provided the expected output of the $Faces$ detector: the sentence $int(1)$. The neural network is then trained by analyzing the actual and the expected output. On the basis of the error between these the internal parameters of neural network will be adapted.

When the developer did not provide an observation the theory is used to predict the output. The theory is not updated until new observations are fed to the classifier.

Other algorithms may learn per batch. In this case the algorithm will collect and store the observations. When there is no observation the theory will be build using this collected data. It will then use this, just-in-time, formulated theory to predict the actual output.

Formally the classifier is split into two detectors:

$$\begin{aligned} Faces &\rightarrow int\ Faces.analyze \\ Faces &\rightarrow Faces.predict \\ Faces.analyze &\rightarrow \rho \\ Faces.predict &\rightarrow \rho\ int \end{aligned}$$

$Faces$ is a non-terminal, while $Faces.analyze$ and $Faces.predict$ are both detectors. Due to the fact that an integer, *i. e.* the observation, is present in the parsed sentence a specific alternative is chosen. The first alternative will take this integer as input and train the neural network, while the second alternative will use the net to predict the value of the integer. The natural value for ρ in the case of an observation is 1.0, *i. e.* its confidence is high as its been provided by the developer.

The Acoi implementation provides a default set of classifier plugins. The *bpnn* plugin is an example of an incremental learning algorithm. *Decision rules*, a batch learning algorithm, are available through the *decrules* plugin.

3.2.3 The Start Symbol

References

Until now there was always only one parse forest. But the main goal is to build and maintain a database of such parse forests, or in fact its building blocks: parse trees. It is possible to keep all these parse trees independent of each other in the construction phase, and to postpone resolving dependencies to the moment of insertion

into the database. In fact this was done in early versions of the Acoi system (see Chapter 7). However, as this knowledge would not be explicit and the system could thus not exploit it, the coding burden would be completely passed on to the developer.

Take for example the WWW multimedia search engine. On the WWW HTML pages contain the links which connect all the web objects together. By adding a detector which parses the HTML and extracts the links the structure of the WWW can be described by a feature grammar.

Example 3.1.

```
1 %start        WebObject(Location);

2 %detector     HTML_type [ Location\[ends-with(.,".html")\] ];

3 %detector     HTML(Location);

4 %atom str     Title;

5 WebObject   : Location WebBody;
6 WebBody     : HTML_type HTML;
7 HTML        : Title? WebObject*;
```

□

However, the WWW is not a tree but a graph. And this graph may contain cycles, which would lead to infinitely deep parse forests. This would make the feature grammar impractical. How to prevent the FDE from reanalyzing an object over and over again, thus storing redundant information in the parse tree? The developer may add detectors, which check if a web object has already been analyzed, and alter production rules, *i. e.* add optionality such that a partial parse tree containing foreign key information is also valid.

Once more the FDE can manage these dependencies better if they are made explicit. The HTML feature grammar illustrates when a cycle in the graph may be introduced: the start symbol *WebObject* is reused in a production rule. The start declaration already contains information on the data items an initial sentence should contain to start a parsing process. When this information is viewed as key information the FDE gets the ability to check the database for the existence of this specific (partial) parse tree. However, if the parse tree does not exist yet this key information is also enough to start building the parse tree.

This solves the case when the start symbol is reused, however, in other cases information would still be redundant. Keyword search can be added to the WWW application by extending the HTML module with these production rules:

```
1 %atom str     Word;
```

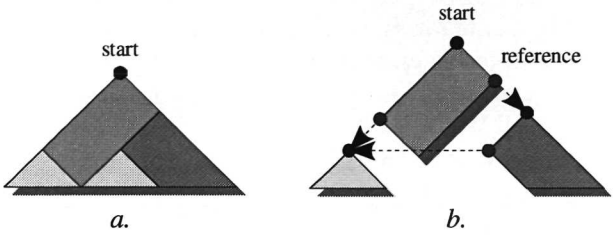

Figure 3.4: Parse tree structure (a) without and (b) with references

```
2 HTML        : Title? Body WebObject*;
3 Body        : KeyWord*;

4 KeyWord     : Word Synset*;
5 Synset      : Synonym* Hypernym* Hyponym*;
6 Synonym     : Keyword;
7 Hypernym    : Keyword;
8 Hyponym     : Keyword;
```

From the body of the HTML page keywords are extracted. These keywords are related with WordNet synsets [CSL01], which allow to expand the search with synonyms, hypernyms and hyponyms. If for each keyword encountered on the WWW a new partial parse tree would be added the database would explode in no time. Using the WWW case study a sample of 150,000 web objects lead to 850,000 keywords which were on average 40 times reused. This explosion is prevented by permitting multiple start symbols, *e. g.* a start declaration for the *Keyword* symbol is added.

```
1 %start        KeyWord(Word);
```

The FDE can now use the same strategy for binding a parse tree for this start symbol. A start symbol which reoccurs in the RHS of a production rule is also known as a *reference*. This emphasized in the language by adding a & prefix to the symbol occurrence. Figure 3.4 shows how this addition of references to start symbols affects the parse tree structure.

This conversion of a tree into a graph structure has also implications for the XPath expressions as used to specify either a detector input or a whitebox predicate. Resolving these paths could lead to endless loops, *e. g.* the XPath expression `preceding::Location` could reenter the same tree over and over again by encountering the same *WebObject* symbol again. To resolve this, references have to be crossed explicitly, which basically means that a XPath expression can not cross over to a start symbol. This splits the global parse tree, as shown in Figure 3.4.a, into several parse trees combined by references in a graph, as shown in

Figure 3.4.b. Take once more the XPath expression `preceding::Location`. Using this convention only *Location* symbols in the local parse tree can be found. To cross a reference, *i. e.* a start symbol, the XPath expression would look as follows: `preceding::&WebObject//Location`. The & prefix, which is not part of the XPath standard, allows the adapted XPath interpreter to resolve the `//Location` path in the parse tree to which a *WebObject* reference is bound. In the next chapter this feature grammar specific addition to the XPath language will be translated back into native XPath expressions, however, this depends on the implementation specific representation of feature grammar parse trees in XML documents.

A drawback of this is the reintroduction of deadlock situations when binding the detector parameters. The XPath expression may lead to a (indirect) self-reference, *i. e.* a detector needs access to x_r and thus violates the lPC regulated rewrite rules and thus the deadlock prevention strategy (see Section 2.2.3). As there is no general solution for this, an exception is passed on to the detectors implementation. The developer may, for example, know how to retrieve the requested token from x_r or which default value to pick.

The use of references is also related to the quasi-nodes introduced in Section 2.2.4. Quasi-nodes were introduced to handle the binding of ambiguous parameters to the right detector calls. Start symbols are always detectors, remember the introduction of S_S. The reference is the quasi-root containing the (ambiguous) binding information. The refereed S_S is the quasi-foot.

Notice also that with the addition of multiple start symbols the feature grammar now describes formally a set of feature grammar systems where one is instantiated on the moment a start symbol is chosen, *i. e.* $S_S \rightarrow \ldots$ is generated on-the-fly.

3.2.4 Feature Grammar Modules

One of the main goals of feature grammar systems is to keep detectors generic, so they are easily applicable in another context. This goal is further supported by the concept of feature grammar modules. In such a module related detectors, atoms and their production rules are grouped and available for reuse by other feature grammars.

```
1 %module      WWW;

2 %start       WebObject(Location);

3 %detector    WebHeader(Location);

4 %atom        www::url {^http://([^ :/]*)(:[0-9]*)?/?(.*)$};
5 %atom        temporal::date;

6 %atom        url  Location;
7 %atom        date Modification;
8 %atom        lng  Length;
```

```
9  WebObject   : Location WebHeader WebBody;
10 WebHeader   : Modification Length;
```

This `WWW` feature grammar module gives basic support for the WWW. The module offers a start symbol, *WebObject*, and a detector to retrieve basic HTTP header information, *WebHeader*. Furthermore, it contains the definition of two atomic types: *url* and *date*. These feature grammar statements show how the `Image` feature grammar uses the `WWW` module:

```
1 %module      Image;
2 %use         WWW;

3 WebBody      : Image;
4 Image        : Color Class;
```

In this example both the *Local* and *WebBody* symbols are unique for the union of the `Image` and `WWW` feature grammars. However, when this is not the case naming conflicts arise. Assuming that the module names are unique, they are used as namespaces to resolve such conflicts. An explicit namespace is added as a prefix to the symbol name, *e. g.* *WWW* :: *WebBody*.

Declarations and definitions always happen in the scope of the active namespace, *i. e.* the module in which the declaration or definition takes place. This means that atoms and detectors can not be redefined by another feature grammar module. However, additional production rules are allowed, *i. e.* to add an alternative view. The `Image` feature grammar uses this construction to add the `Image` view on a *WebBody* symbol. In this way feature grammars are easily extensible.

3.2.5 Change Detection

In Chapter 1 several sources of change were identified:

internal sources either the dependency description or the implementation of detector functions change;

external sources the source data changes, the system may check for these changes by itself (polling) or may be alerted by the librarian.

The upcoming sections will describe the features the feature grammar language offers to the Acoi system to detect some of the changes.

Versions One of the triggers for incremental maintenance of a set of parse trees are changes in detector implementations and the production rules. Changes in implementation may not always be reflected in the feature grammar, *e. g.* a bug fix does not have to lead to changes in the output, and thus the production rules, of a detector. To indicate these changes detectors have a version.

```
%version    Skin 1.23.4;
```

Increments of the version indicate a change in implementation and forces the FDS to determine if any persistent stored parse trees are affected and should be updated. The priority which is assigned to this change is depending on the level of change: major (the first number), minor (the second number) or service (the last number).

Polling Detector versions handle the notification of internal detector changes. To poll for external changes an (optional) poll detector related to a start symbol is added. The arguments of this detector are at least the required initial sentence as part of the start declaration.

```
%detector    WebObject.poll(Location,WebHeader/Modification);
```

An implementation of this poll detector will be called by the FDS (see Chapter 6) on a regular basis to check if a stored parse tree needs to be updated. When no poll detector is defined the default poll detector is used, which always returns *true*, *i. e.* always indicates an external change.

3.3 Discussion

This chapter described the feature grammar language: a convenient notation for feature grammar systems, including several syntactic shortcuts. The language aims at being readable, although a clear understanding of the use of CS grammars is still needed to construct one, *e. g.* understand the relationship between rules, trees and regular path expressions. This should not prevent the usability of the language as its target audience are DMW and annotation extraction algorithm developers, who will be schooled in basic computer science topics.

With the rise of XML more and more languages are expressed in XML and thus have the advantage of standardized tools like SAX parsing and DOM parse trees. But those languages tend to be verbose and non-transparent. However, an XML-ized version of the feature grammar language may be constructed as an intermediate format to gain those advantages. This language could be defined as an extension on one of the upcoming XML schema languages (see Section 5.2.1).