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 4

Feature Detector Engine

...Sharpen line forty-eight between twenty point twenty-seven.
...Profile trace.
...Stop. Back up.
...Stop.
...Enhance.
...Seesaw.
...Stop!
...Enhance.
...Enhance.
...
...Hey!

Rick Deckard – *Blade Runner*

Grammars are mostly used to validate a sentence's membership of a specific language. This process of validation, called parsing, may lead to the construction of a parse tree, *i. e.* an internal representation of the structure of the sentence. The parsing process forms the heart of the *Feature Detector Engine* (FDE). During this process the FDE encounters detector symbols, binds their input sentence, executes the associated algorithms, and validates their output sentence (the actual parsing). There are many different parsing algorithms. Yet only a few of these algorithms are suited for processing a feature grammar system. The next section will review the predominant parsing algorithms for CF grammars. The remainder of the chapter is focused on the implementation decisions for the FDE, based on a parsing algorithm well suited for feature grammar systems.

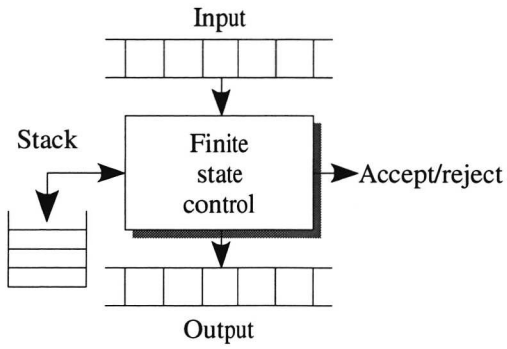

Figure 4.1: The basic PDT

4.1 A Parser Primer

The membership of a sentence w in a language $L(G)$ can be checked using a parser, which is especially constructed for that grammar. Parsers are based on specific types of (finite) automata. For each grammar type or language family a specific type of (finite) automata is used. A REG language is parsed using a *finite automaton* (FA), while a CF language needs the more advanced functionality of a *push-down automaton* (PDA).

The basic automaton is an acceptor, *i. e.* accepts or rejects an input sentence. When the automaton also has additional output, *e. g.* a structural description of the symbols encountered, it is called a transducer.

The transducer version of the PDA, *i. e.* the *push-down transducer* (PDT), is basically a FA with a stack-based memory and an output tape, see Figure 4.1. The read head of the input tape can read one symbol at a time from the tape. The same goes for the write head of the output tape: it can write one symbol at the time. Both the read and write head advance to the next symbol after reading or writing. The end of both the input sentence and the stack is indicated by the special symbol \$. The stack-based memory allows the PDT to push and pop symbols on or from the stack in a last-in first-out (LIFO) way.

The transducer description includes a set of states. The states of the PDT are restricted to these types, which are directly related to its capabilities:

start the PDT;

read a single symbol from the input tape;

write a symbol on the output tape;

push a single symbol on top of the stack;

pop the topmost symbol from the stack;

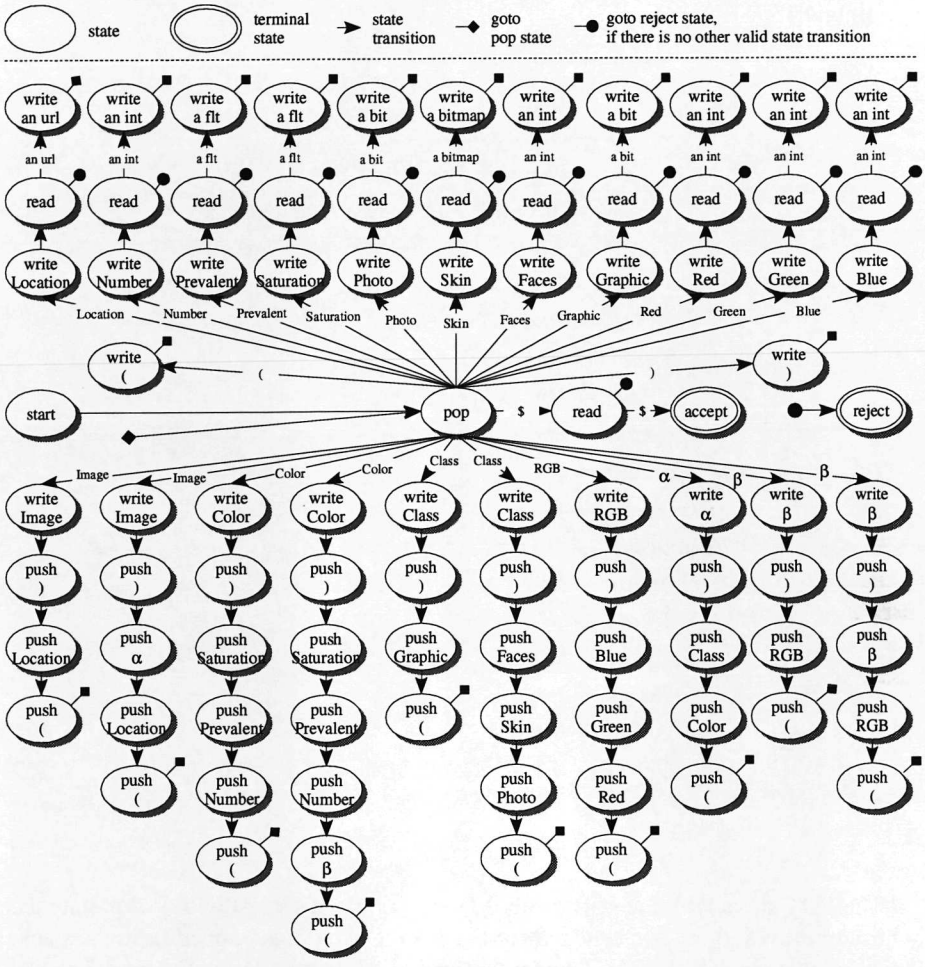

Figure 4.2: A non-deterministic PDT for the `Image` CF grammar

accept the input string and stop;

reject the input string and stop.

Using these states and a set of state transitions the PDT can be used to implement a parsing algorithm. To illustrate this Figure 4.2 shows a PDT for this example CF grammar:

Example 4.1.

$$
\begin{aligned}
\mathit{Image} &\rightarrow \mathit{Location} \\
\mathit{Image} &\rightarrow \mathit{Location}\ \alpha; \\
\alpha &\rightarrow \mathit{Color\ Class} \\
\mathit{Color} &\rightarrow \mathit{Number\ Prevalent\ Saturation} \\
\mathit{Color} &\rightarrow \beta\ \mathit{Number\ Prevalent\ Saturation} \\
\beta &\rightarrow \mathit{RGB} \\
\beta &\rightarrow \mathit{RGB}\ \beta \\
\mathit{RGB} &\rightarrow \mathit{Red\ Green\ Blue} \\
\mathit{Class} &\rightarrow \mathit{Graphic} \\
\mathit{Class} &\rightarrow \mathit{Photo\ Skin\ Faces} \\
\mathit{Location} &\rightarrow \mathit{url} \\
\mathit{Number} &\rightarrow \mathit{int} \\
\mathit{Prevalent} &\rightarrow \mathit{flt} \\
\mathit{Saturation} &\rightarrow \mathit{flt} \\
\mathit{Graphic} &\rightarrow \mathit{bit} \\
\mathit{Photo} &\rightarrow \mathit{bit} \\
\mathit{Skin} &\rightarrow \mathit{bitmap} \\
\mathit{Faces} &\rightarrow \mathit{int} \\
\mathit{Red} &\rightarrow \mathit{int} \\
\mathit{Green} &\rightarrow \mathit{int} \\
\mathit{Blue} &\rightarrow \mathit{int}
\end{aligned}
$$

□

The PDT starts with the start symbol *Image* on its stack. After the *start* state the controller moves to the *pop* state, where the *Image* symbol is popped from the stack. Based on this symbol the PDT chooses a transition to a next state. Figure 4.3 shows the (possible) condition of the PDT the second time it visits the *pop* state. The *accept* state is reached when both the stack and the read tape are empty (reached by both popping and reading a $ symbol). The output tape will then contain a textual description of the parse tree, *e. g.* `Image ( Location http://... α ( Color ( Number 29053 ...) ...) )`.

On its way to the *accept* or *reject* state the controller has to choose a move to a next valid state. In the case of either a pop or a read state, the valid options are determined by respectively the symbol on top of the stack or under the read head of the input tape. If there is always precisely one valid move, the PDA or PDT is called *deterministic* (DPDA or DPDT), when there are more valid options the PDA or PDT is called *non-deterministic* (NPDA or NPDT). The PDT in Figure 4.2 is non-deterministic, *e. g.* after a pop of the *Image* symbol there are two valid moves (due to the alternative

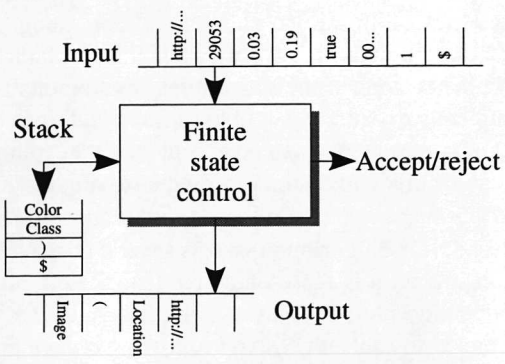

Figure 4.3: The condition of the PDT

production rules). In such a case the NPDT follows both choices, *i. e.* it is in several states at the same time. However, some of these choices will never lead to an accept state and will thus not yield a valid parse tree. On the other hand several choices may lead to valid parse trees. When more than one valid parse tree can describe the same input sentence the grammar is *ambiguous* (see also Section 3).

A PDA or PDT can be automatically generated from a CF grammar. The algorithm implemented in the PDT and sketched above performs a top-down parsing strategy – an intuitive method. In the next section a short overview of other methods and their most important properties are given.

4.1.1 More Parsing Algorithms for Context-free Grammars

A *top-down algorithm*, as has been sketched and implemented in a PDT in the previous section, starts with the start symbol and tries, by traversing the production rules in a smart way, to *produce* the input sentence. A *bottom-up algorithm* works just the other way around: it starts with the input sentence and tries to *reduce* it back to the start symbol.

Both algorithms can be used to parse the input sentence of Figure 2.3 using the example CF grammar. Due to space considerations a simplified version of the grammar is used (see Figure 4.4) to illustrate the behavior of the two basic algorithms.

Furthermore, the end of the input sentence is once more indicated with the special \$ terminal. This terminal reappears at the end of the rule for the start symbol, Im. This rule takes care for a correct detection of the end of the input sentence.

Figure 4.5 shows the steps taken by a specific top-down parsing algorithm, *i. e.* a depth-first algorithm. The information used by this algorithm consists of two parts: the active rules and the sentence. The current position within a rule (or the sentence) is indicated with a bullet (•). This bullet splits a rule in a *match* part and a *prediction*

part. The algorithm always follows one alternative. For example, in step j the Gr alternative of the Cl non-terminal is tried, only when this one fails the next alternative, Ph, is tried in step k. When none of the active rules has a prediction left and the input sentence is also completely consumed, and both these conditions are enforced by the start rules for Im, the input sentence can be accepted. By keeping track of the active rules the parse tree can be gradually build during the parsing process.

The application of a bottom-up algorithm, *i. e.* a breadth-first algorithm, is shown in Figure 4.6. The breadth-first algorithm inspects several possible parses in each step. Each parse under consideration is represented by a stack with attached partial parse trees. Each step in this algorithm consists of two phases. In the *shift* phase the next input symbol is appended to each stack. The following *reduce* phase then examines all stacks and if they allow one or more reductions copies of the stack are made and the reductions applied to them. These reductions produce the partial parse trees. The first reduction is applied in step d: the shift phase added the Sa token to the first stack, which enabled the reduction of the $Nu\ Pr\ Sa$ symbol sequence to the Co non-terminal. This process continues until there is no input left. In the total of six (partial) trees left in step h there is only one which contains the start symbol, Im, which is also the root of the parse tree.

Both parsing algorithms process the input sentence from left to right, *i. e.* they are *directional.* However, there are also some algorithms which are *non-directional.* These methods may access the input sentence in any order they like. This requires the input sentence to be completely available upfront, while conventional algorithms work on a stream of tokens. To illustrate this: the breadth-first used algorithm in Figure 4.6 is well suited for on-line parsing where a source outside of the parser produces the input sentence gradually.

Table 4.1 shows a taxonomy of parsing algorithms (based on [GJ98], where these algorithms are described in more depth). The taxonomy shows that directional parsing algorithms can be further grouped. The description of top-down and bottom-up

The simplified CF grammar:		The simplified input sentence:
1	$Im \rightarrow Lo\ \$$	$Lo\ Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
2	$Im \rightarrow Lo\ \alpha\ \$$	
3	$\alpha \rightarrow Co\ Cl$	
4	$Co \rightarrow Nu\ Pr\ Sa$	
5	$Co \rightarrow \beta\ Nu\ Pr\ Sa$	
6	$\beta \rightarrow RGB$	
7	$\beta \rightarrow RGB\ \beta$	
8	$RGB \rightarrow R\ G\ B$	
9	$Cl \rightarrow Gr$	
10	$Cl \rightarrow Ph\ Sk\ Fa$	

Figure 4.4: The simplified CF grammar and input sentence

	Active rules	*Sentence*
a.	1. $Im \rightarrow \bullet Lo\ \$$	$\bullet Lo\ Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
b.	1. $Im \rightarrow Lo \bullet \$$	$Lo \bullet Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
c.	1. $Im \rightarrow \bullet Lo\ \alpha\ \$$	$\bullet Lo\ Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
d.	1. $Im \rightarrow Lo \bullet \alpha\ \$$	$Lo \bullet Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
e.	1. $Im \rightarrow Lo\ \alpha \bullet\ \$$	$Lo \bullet Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
	2. $\alpha \rightarrow \bullet Co\ Cl$	$Lo \bullet Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
f.	2. $\alpha \rightarrow Co \bullet Cl$	$Lo \bullet Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
	3. $Co \rightarrow \bullet Nu\ Pr\ Sa$	$Lo \bullet Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$$
g.	3. $Co \rightarrow Nu \bullet Pr\ Sa$	$Lo\ Nu \bullet Pr\ Sa\ Ph\ Sk\ Fa\ \$$
h.	3. $Co \rightarrow Nu\ Pr \bullet Sa$	$Lo\ Nu\ Pr \bullet Sa\ Ph\ Sk\ Fa\ \$$
i.	3. $Co \rightarrow Nu\ Pr\ Sa \bullet$	$Lo\ Nu\ Pr\ Sa \bullet Ph\ Sk\ Fa\ \$$
j.	2. $\alpha \rightarrow Co\ Cl \bullet$	$Lo\ Nu\ Pr\ Sa \bullet Ph\ Sk\ Fa\ \$$
	3. $Cl \rightarrow \bullet Gr$	$Lo\ Nu\ Pr\ Sa \bullet Ph\ Sk\ Fa\ \$$
k.	3. $Cl \rightarrow \bullet Ph\ Sk\ Fa$	$Lo\ Nu\ Pr\ Sa \bullet Ph\ Sk\ Fa\ \$$
l.	3. $Cl \rightarrow Ph \bullet Sk\ Fa$	$Lo\ Nu\ Pr\ Sa\ Ph \bullet Sk\ Fa\ \$$
m.	3. $Cl \rightarrow Ph\ Sk \bullet Fa$	$Lo\ Nu\ Pr\ Sa\ Ph\ Sk \bullet Fa\ \$$
n.	3. $Cl \rightarrow Ph\ Sk\ Fa \bullet$	$Lo\ Nu\ Pr\ Sa\ Ph\ Sk\ Fa \bullet \$$
o.	1. $Im \rightarrow Lo\ \alpha\ \$ \bullet$	$Lo\ Nu\ Pr\ Sa\ Ph\ Sk\ Fa\ \$ \bullet$

Figure 4.5: A top-down parse for the example CF grammar

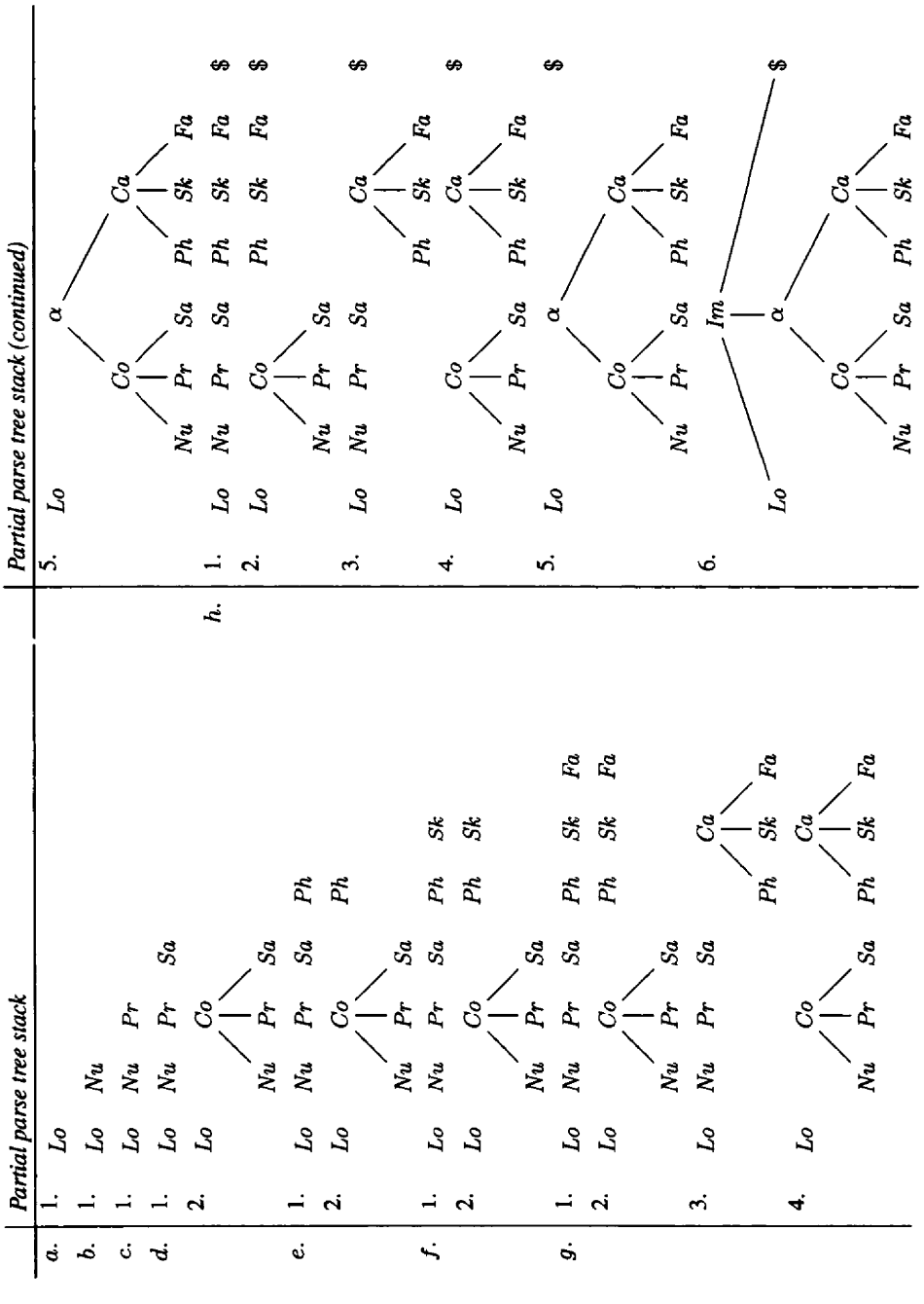

Figure 4.6: A bottom-up parse for the example CF grammar

	top-down	*bottom-up*
non-directional	Unger parser	CYK parser
directional	predict/match automaton	shift/reduce automaton
	1. depth-first	1. depth-first
	1.*a*. backtracking	
	1.*b*. exhaustive backtracking	
	2. breadth-first	2. breadth-first
	2.*a*. deterministic breadth-first	2.*a*. restricted breadth-first
	2.*a*.1. LL(k)	2.*a*.1. Earley
		2.*a*.2. Tomita
		2.*b*. deterministic breadth-first
		2.*b*.1. LR(k)
		2.*b*.2. SLR(1)
		2.*b*.3. LALR(1)

Table 4.1: A taxonomy of parsing algorithms

parsing already showed that either a depth-first or a breadth-first search strategy can be applied. Research on efficient algorithms, *i. e.* algorithms with linear complexity, has mainly focused on bottom-up, directional and deterministic methods. They use some form of look-ahead, *i. e.* one or more tokens of the input sentence, to decide which production rule to follow. Bottom-up parsers are more powerful for deterministic parsing as they will use more context, *i. e.* have seen more of the input sentence, before making a decision [Par93]. Although these variants are not shown in this table, deterministic algorithms can be generalized, *i. e.* made non-deterministic, by adding (pseudo-)parallel features [Lan74, Rek91].

4.2 Parsing Feature Grammar Systems

As summarized in the previous section, there exists a plethora of techniques to parse sentences and validate their membership of a CF language. However, are these parsing techniques also applicable to grammar systems and feature grammar systems in particular?

Grammar systems have been mostly studied in theory, however, some first steps have been taken to use them for practical purposes [PS98]. One step in this process is to investigate the use of (existing) parsing algorithms. In [MM96] the authors take a first step by investigating the deterministic subclass of grammar systems as a basis for parsing. However, as identified in the previous chapters the application domain of feature grammar systems benefits from non-determinism. In this section a suitable non-deterministic parsing algorithm for feature grammar systems is selected.

In a grammar system parsing operates on two different levels: the global grammar system, *i. e.* transfer of control between grammar components, and the local grammar

component, *i. e.* the actual parsing of a (partial) sentence. This is also reflected in the basic ingredients of the derivation process for a feature grammar system, as formally described in Chapter 2:

bind a grammar component G_i gathers its input sentence by binding its REG expression R_i with the *path* metamorphosis of the partial parse tree t_j;

detect a detector function f_{d_i} maps the input into a, just-in-time produced, partial sentence;

parse the partial sentence z_{d_i} is parsed, and thus validated, by the corresponding grammar component G_i, resulting in an extended partial parse tree;

(un)nest the yield z of the partial parse tree derived using G_i contains the words in z_{d_i} enveloped by arbitrary sequences of detectors, as described by the REG language derived by f_D from z_{d_i}.

The just-in-time behavior determines where the control of the system lies initially: with the "dummy" detector S_S. This implies a top-down algorithm, which is confirmed by the needs of the binding step. As this last step depends on the availability of a (partial) parse tree which can be transformed into a set of neat paths in which the regular expression, R_i, can be resolved. The nesting of detector components asks for a component to hand over the control to another component. As stated in Section 2.2.3, the lPC rewriting mechanism has been added to prevent deadlock situations and prefers leftmost derivation on the control or grammar system level. So the grammar system level calls for a top-down leftmost, *i. e.* directional, parsing algorithm.

Within a component a complete sentence z_{d_i} is available, which in principle may be parsed with any of the non-deterministic parsing algorithms described in the previous section. What complicates this parsing process is the nesting of detectors. Upon encountering a detector there are two alternatives: (1) delay validation of the detector until the stop condition of the grammar component is satisfied, or (2) first validate the detector and then go on with validation of the output sentence. The first alternative closely follows the formal derivation method as described in Chapter 2, but does not fit within any CF parsing algorithm. The second alternative allows the use of a standard top-down algorithm, *i. e.* control is handed over to the detector and handed back after validation.

Both levels allow, and even favor, the use of an adapted top-down algorithm. There are even more, general, reasons for the use of a top-down instead of a bottom-up algorithm:

1. people parse sentences top-down [AS88, RJ99], *i. e.* debugging a top-down parse is thus more intuitive for a feature grammar developer;

2. these algorithms provide better support for the addition of semantic actions [Par93], *e. g.* detector functions, as they provide more context information, *i. e.* the same reason why detector parameters can be bound;

3. the same context gives also easy support for informative error reporting [GJ98], which, once more, helps during debugging.

A top-down algorithm has been implemented in the current version of the FDE and will be described in more detail in the next subsection.

4.2.1 Exhaustive Backtracking for Feature Grammar Systems

The top-down algorithm used within the FDE is based on an *exhaustive backtracking* algorithm. Backtracking indicates depth-first behavior: one alternative is chosen and followed until it either fails or succeeds. Upon failure the algorithm backtracks until an untried alternative is found and tries that one. The adjective exhaustive means that the algorithm used by the FDE also backtracks when the alternative is successful. By doing this the algorithm handles ambiguous feature grammars and constructs the parse forest.

To show the algorithm in action a basic feature grammar is constructed in relatively the same manner as the CF grammar in Figure 4.4. Figure 4.7 shows this simplified feature grammar. The same figure shows the formal feature grammar system derived from the grammar. The rewrite of the rules involves the introduction of anonymous symbols, *i. e.* α and β, for the handling of a symbol group and sequences.

Figure 4.8 shows the various parsing actions, grouped per controlling grammar component. The actions are directly associated with the basic ingredients described before. A component which gets control starts with an empty output sentence. The REG expression associated with the detector is *binded* in the parse forest (see Figure 4.9 for the basic AND/OR graph) resulting in the input sentence. The output sentence is then filled by the *detect* action, *i. e.* the mapping function is applied. The parsing process of this sentence is then interleaved with control transfers to nested detectors. To be able to resume the parsing process the output sentence is pushed on the stack of sentences under inspection when control is transferred to a nested detector. This allows the delayed evaluation of the remainder of the stop condition by popping this stack when control is transferred back.

The exhaustive backtracking behavior of the algorithm is illustrated in step a, when the second alternative rule of Im is considered (and found valid in step k), even after the first rule has already been found valid.

Most algorithms pose limitations on the grammars they can parse. This is also true for a top-down parsing algorithm like exhaustive backtracking. The next subsection will investigate these limitations. The last two subsections will look at optimizations to make the algorithm more efficient by avoiding unnecessary backtracking and doing double work.

The simplified feature grammar:

```
1  | %start    Im(Lo);
2
3  %detector Co(Lo);
4  %detector Gr(Nu,Pr,Sa);
5  %detector Ph(Nu,Pr,Sa);
6  %detector Sk(Lo);
7  %detector Fa(Sk);
8
9  %atom     Lo,R,G,B,Nu,
10           Pr,Sa,Sk,Fa;
11
12 Im       : Lo ( Co Cl )?;
13 Co       : RGB* Nu Pr Sa;
14 RGB      : R G B;
15 Cl       : Gr | Ph Sk Fa;
```

The simplified feature grammar system:

$$
\begin{aligned}
\Gamma = (&\\
&D = \{S_S, Co, Gr, Ph, Sk, Fa\},\\
&N = \{Im, \alpha, Cl, \beta, RGB\},\\
&T = \{\rho, Lo, R, G, B, Nu, Pr, Sa, \$\},\\
&P_N = \{(Im \rightarrow Lo), (Im \rightarrow Lo\ \alpha), (\alpha \rightarrow Co\ Cl),\\
&\quad (\beta \rightarrow RGB), (\beta \rightarrow RGB\ \beta),\\
&\quad (RGB \rightarrow R\ G\ B), (Cl \rightarrow Gr), (Cl \rightarrow Ph\ Sk\ Fa)\},\\
&G_{Co} = (V, P_{Co} = \{(Co \rightarrow \rho\ \beta\ Nu\ Pr\ Sa\ \$), (Co \rightarrow \rho\ Nu\ Pr\ Sa\ \$)\}\\
&\quad \cup P_N, R_{Co} = (.^* \cdot Lo), f_{Co}),\\
&G_{Gr} = (V, P_{Gr} = \{(Gr \rightarrow \rho\ \$)\} \cup P_N, R_{Gr} = (.^* \cdot Nu) + (.^* \cdot Pr) + (.^* \cdot Sa),\\
&\quad f_{Gr}),\\
&G_{Ph} = (V, P_{Ph} = \{(Ph \rightarrow \rho\ \$)\} \cup P_N, R_{Ph} = (.^* \cdot Nu) + (.^* \cdot Pr) + (.^* \cdot Sa),\\
&\quad f_{Ph}),\\
&G_{Sk} = (V, P_{Sk} = \{(Sk \rightarrow \rho\ \$)\} \cup P_N, R_{Sk} = (.^* \cdot Lo), f_{Sk}),\\
&G_{Fa} = (V, P_{Fa} = \{(Fa \rightarrow \rho\ \$)\} \cup P_N, R_{Fa} = (.^* \cdot Sk), f_{Fa}),\\
&G_S = (V, P_S = \{(S_S \rightarrow \rho\ Im\ \$)\} \cup P_N, R_S = \emptyset, f_S),\\
&S = S_S\\
&)
\end{aligned}
$$

Figure 4.7: The simplified feature grammar (system)

	Component	Action	Sentence		Component	Action	Sentence
a.	G_S	*bind:* $\emptyset$	$\$$		G_S	*backtrack:* Cl has an alternative	$\rho\ Lo \bullet \$$
		detect: $f_S(\lambda)$ detects the initial sentence	$\bullet\rho\ Lo\ \$$			*parse:*	$\rho\ Lo \bullet \$$
		parse:	$\bullet\rho\ Lo\ \$$			4. $Cl \rightarrow \bullet Ph\ Sk\ Fa$	$\rho\ Lo \bullet \$$
		1. $S_S \rightarrow \bullet\rho\ Im\ \$$	$\bullet\rho\ Lo\ \$$			4. $Cl \rightarrow Ph \bullet Sk\ Fa$	$\rho\ Lo \bullet \$$
		1. $S_S \rightarrow \rho \bullet Im\ \$$	$\rho \bullet Lo\ \$$			*next:* transfer control to G_{Ph}	$\rho\ Lo \bullet \$$
		1. $S_S \rightarrow \rho\ Im \bullet \$$	$\rho \bullet Lo\ \$$				
		2. $Im \rightarrow \bullet Lo$	$\rho \bullet Lo\ \$$	f.	G_{Ph}	*bind:* $(.^* \cdot Nu) + (.^* \cdot Pr) + (.^* \cdot Sa)$	$\bullet\$$
		2. $Im \rightarrow Lo \bullet$	$\rho\ Lo \bullet \$$			*detect:* $f_{Ph}(Nu\ Pr\ Sa)$ detects the photo class	$\bullet\rho\ \$$
		backtrack: Im has an alternative	$\rho \bullet Lo\ \$$			*parse:*	$\bullet\rho\ \$$
		parse:	$\rho \bullet Lo\ \$$			5. $Ph \rightarrow \bullet\rho\ \$$	$\bullet\rho\ \$$
		2. $Im \rightarrow \bullet Lo\ \alpha$	$\rho \bullet Lo\ \$$			5. $Ph \rightarrow \rho \bullet \$$	$\rho \bullet \$$
		2. $Im \rightarrow Lo \bullet \alpha$	$\rho\ Lo \bullet \$$			5. $Ph \rightarrow \rho\ \$\bullet$	$\rho\ \$\bullet$
		2. $Im \rightarrow Lo\ \alpha \bullet$	$\rho\ Lo \bullet \$$			*unnest:* transfer control back to G_S	$\rho\ \$\bullet$
		3. $\alpha \rightarrow \bullet Co\ Cl$	$\rho\ Lo \bullet \$$				
		3. $\alpha \rightarrow Co \bullet Cl$	$\rho\ Lo \bullet \$$	g.	G_S	*parse:*	$\rho\ Lo \bullet \$$
		next: transfer control to G_{Co}	$\rho\ Lo \bullet \$$			5. $Cl \rightarrow Ph\ Sk \bullet Fa$	$\rho\ Lo \bullet \$$
						next: transfer control to G_{Sk}	$\rho\ Lo \bullet \$$
b.	G_{Co}	*bind:* $(.^* \cdot Lo)$	$\bullet\$$				
		detect: $f_{Co}(Lo)$ detects the color features	$\bullet\rho\ Nu\ Pr\ Sa\ \$$	h.	G_{Sk}	*bind:* $(.^* \cdot Lo)$	$\bullet\$$
		parse:	$\bullet\rho\ Nu\ Pr\ Sa\ \$$			*detect:* $f_{Sk}(Lo)$ detects the skin pixels	$\bullet\rho\ \$$
		4. $Co \rightarrow \bullet\rho\ \beta\ Nu\ Pr\ Sa\ \$$	$\bullet\rho\ Nu\ Pr\ Sa\ \$$			*parse:*	$\bullet\rho\ \$$
		4. $Co \rightarrow \rho \bullet \beta\ Nu\ Pr\ Sa\ \$$	$\rho \bullet Nu\ Pr\ Sa\ \$$			5. $Sk \rightarrow \bullet\rho\ \$$	$\bullet\rho\ \$$
		4. $Co \rightarrow \rho\ \beta \bullet Nu\ Pr\ Sa\ \$$	$\rho \bullet Nu\ Pr\ Sa\ \$$			5. $Sk \rightarrow \rho \bullet \$$	$\rho \bullet \$$
		5. $\beta \rightarrow \bullet RGB$	$\rho \bullet Nu\ Pr\ Sa\ \$$			5. $Sk \rightarrow \rho\ \$\bullet$	$\rho\ \$\bullet$
		5. $\beta \rightarrow \bullet RGB\bullet$	$\rho \bullet Nu\ Pr\ Sa\ \$$			*unnest:* transfer control back to G_S	$\rho \bullet \$$
		5. $\beta \rightarrow RGB\bullet$	$\rho \bullet Nu\ Pr\ Sa\ \$$				
		6. $RGB \rightarrow \bullet R\ G\ B$	$\rho \bullet Nu\ Pr\ Sa\ \$$				
		invalid: R can't be popped	$\rho \bullet Nu\ Pr\ Sa\ \$$	i.	G_S	*parse:*	$\rho\ Lo \bullet \$$
		backtrack: Co has an alternative without RGB	$\bullet\rho\ Nu\ Pr\ Sa\ \$$			4. $Cl \rightarrow Ph\ Sk\ Fa\bullet$	$\rho\ Lo \bullet \$$
		parse:	$\bullet\rho\ Nu\ Pr\ Sa\ \$$			*next:* transfer control to G_{Fa}	$\rho\ Lo \bullet \$$
		4. $Co \rightarrow \bullet\rho\ Nu\ Pr\ Sa\ \$$	$\rho \bullet Nu\ Pr\ Sa\ \$$				
		4. $Co \rightarrow \rho \bullet Nu\ Pr\ Sa\ \$$	$\rho \bullet Nu\ Pr\ Sa\ \$$	j.	G_{Fa}	*bind:* $(.^* \cdot Sk)$	$\bullet\$$
		4. $Co \rightarrow \rho\ Nu \bullet Pr\ Sa\ \$$	$\rho\ Nu \bullet Pr\ Sa\ \$$			*detect:* $f_{Fa}(Sk)$ detects the faces	$\bullet\rho\ \$$
		4. $Co \rightarrow \rho\ Nu\ Pr \bullet Sa\ \$$	$\rho\ Nu\ Pr \bullet Sa\ \$$			*parse:*	$\bullet\rho\ \$$
		4. $Co \rightarrow \rho\ Nu\ Pr\ Sa \bullet \$$	$\rho\ Nu\ Pr\ Sa \bullet \$$			5. $Fa \rightarrow \bullet\rho\ \$$	$\bullet\rho\ \$$
		4. $Co \rightarrow \rho\ Nu\ Pr\ Sa\ \$\bullet$	$\rho\ Nu\ Pr\ Sa\ \$\bullet$			5. $Fa \rightarrow \rho \bullet \$$	$\rho \bullet \$$
		unnest: transfer control back to G_S	$\rho\ Nu\ Pr\ Sa\ \$\bullet$			5. $Fa \rightarrow \rho\ \$\bullet$	$\rho\ \$\bullet$
						unnest: transfer control back to G_S	$\rho\ \$\bullet$
c.	G_S	*parse:*	$\rho\ Lo \bullet \$$				
		3. $\alpha \rightarrow Co\ Cl\bullet$	$\rho\ Lo \bullet \$$	k.	G_S	*parse:*	$\rho\ Lo \bullet \$$
		4. $Cl \rightarrow \bullet Gr$	$\rho\ Lo \bullet \$$			1. $S_S \rightarrow \rho\ Im\ \$\bullet$	$\rho\ Lo\ \$\bullet$
		4. $Cl \rightarrow Gr\bullet$	$\rho\ Lo \bullet \$$				
		next: transfer control to G_{Gr}	$\rho\ Lo \bullet \$$				
d.	G_{Gr}	*bind:* $(.^* \cdot Nu) + (.^* \cdot Pr) + (.^* \cdot Sa)$	$\bullet\$$				
		detect: $f_{Gr}(Nu\ Pr\ Sa)$ doesn't detect the graphic class	$\bullet\$$				
		parse:	$\bullet\$$				
		5. $Gr \rightarrow \bullet\rho\ \$$	$\bullet\$$				
		invalid: ρ can't be popped	$\bullet\$$				
		unnest: transfer control back to G_S	$\bullet\$$				

Figure 4.8: A top-down parse for the simplified feature grammar system

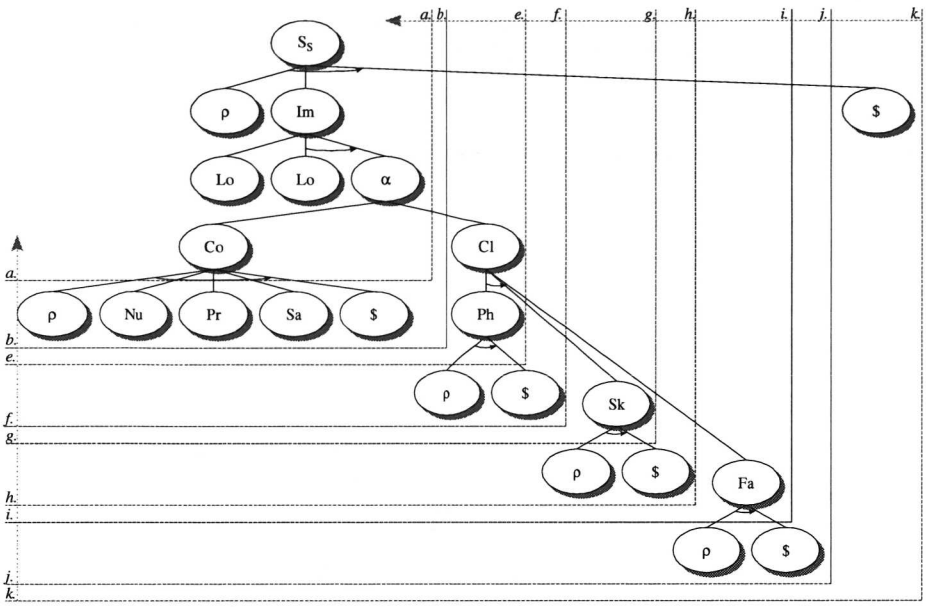

Figure 4.9: Partial parse forests for the steps in Figure 4.8

Left-recursion

The major limitation of top-down methods are their inability to handle left-recursive grammars. To illustrate this problem consider this, direct left-recursive, grammar:

$$S \rightarrow S\ a$$
$$S \rightarrow b$$

To validate the production rule of S the parser will try to validate S over and over again, thus entering an endless loop. Fortunately standard rewrite rules are available for left-recursion elimination. For example this grammar generates the equivalent language:

$$S \rightarrow b\ \alpha$$
$$S \rightarrow b$$
$$\alpha \rightarrow a\ \alpha$$
$$\alpha \rightarrow a$$

This is the same result as when the *right-recursive* interpretation for symbol sequences is used, *i. e.* both grammars are equivalent with this rule in the extended notation of the feature grammar language(see Section 3.1.1):

```
1 | S          : b a*;
```

These rewritten grammars show that any finite valid input sentence will have to start with a *b* terminal, followed by an optional tail of *a* terminals.

Indirect left-recursion is the case where left-recursion takes place after encountering several other non-terminals, *e. g.* as is the case in this grammar:

$$S \rightarrow A\ B\ c$$
$$B \rightarrow C\ d$$
$$B \rightarrow A\ B\ f$$
$$C \rightarrow S\ e$$
$$A \rightarrow \lambda$$

Recursion elimination in this grammar takes extensive rewrites (see for the algorithm pages 176 – 178 in [ASU86]): elimination of empty rules, elimination of unit rules and finally flatting of the rules interleaved with elimination of direct recursion. This whole process (using the rewrite rules based on the basic CF grammar notation) results in this grammar:

$$S \rightarrow B\ c$$
$$B \rightarrow \alpha\ \beta$$
$$B \rightarrow \alpha$$
$$\alpha \rightarrow \gamma$$
$$\gamma \rightarrow c\ e\ d$$
$$\gamma \rightarrow f\ c\ e\ d$$
$$\gamma \rightarrow f\ \beta\ c\ e\ d$$
$$\beta \rightarrow f$$
$$\beta \rightarrow f\beta$$

Notice that during the application of the rewrite rules symbols disappear and new anonymous symbols are added. Unfortunately this hinders the automatic application of the rewrite rules, especially when detector symbols are involved. The FDE can on one hand not decide to call the detector just once, as is the case with the rewrite rule for direct recursion which uses the extended notation. And on the other hand it can also not split the detector in two: one detector which produces the head (S) and one which produces the tail (α). The same goes for indirect left-recursion elimination.

This moves the burden of removing left-recursion of a detector symbol to the developer, *i. e.* the developer should manually decide when the detector fails and end the infinite production. The need for explicit rewrites by the developer is not uncommon in the world of grammar driven tools, *e. g.* a parser generator like Yacc [LMB92] does not rewrite the grammar rules, but only warns the developer. The main reason for this is that, not unlike the detector functions in a feature grammar system, actions are associated to the grammar rules. And the developer has to modify these actions along with the grammar rules. However, the FDE offers support by warning the developer when left-recursion appears.

Lookahead

Deterministic top-down parsing algorithms, and also some bottom-up variants, depend on lookahead. The algorithm looks ahead in the stream of tokens to be parsed to determine which alternative of a rule to choose. Depending on the lookahead depth the alternatives can share longer prefixes. In theory a lookahead of more then one token ($k > 1$) has been studied [RS70, PQ95], however, due to the exponential explosion in time and space ($|T|^k$) practical parsers have almost always implemented a lookahead of only one token.

The most common form of lookahead is implemented by two sets: $FIRST_k$ and $FOLLOW_k$. Both are based on the k-prefix of a string, $w = a_1 \dots a_n$:

$$k : w = \begin{cases} w & |w| \leq k \\ a_1 \dots a_k & |w| > k \end{cases}$$

Using this prefix operation the $FIRST_k$ and $FOLLOW_k$ sets are defined as follows:

$$FIRST_k(\alpha) = \{k : w | \alpha \stackrel{*}{\Rightarrow} w\}$$
$$FOLLOW_k(A) = \{FIRST_k(\beta) | S \stackrel{*}{\Rightarrow} \beta A \gamma\}$$

where

$$w \in T^*, A \in N, \alpha \in V^*, \beta \in T^*, \gamma \in V^*$$

The parse table is now constructed as follows: for every $(A \rightarrow \alpha)$ α is added to the (A, w) entry of the table for every w in $FIRST_k(\alpha FOLLOW_k(A))$ (see [GJ98]).

In [PQ96] the authors argue for the use of more lookahead to make grammars more natural. The rewrite from a $LL(k)$ or $LR(k)$ grammar to a $LL(1)$ or $LR(1)$ grammar may involve the introduction of many new (anonymous) symbols, *i. e.* to left-factor the rules, thus leading to obfuscation of the semantic meaning. The penalty for the use of more lookahead is the extra space needed for the lookahead table and the extra

time spent to check this table and make the decision. In [Par93] the author describes a linear, approximate, lookahead operation, $LOOK_k^1$, which should minimize this penalty ($|T| * k$). This operation is defined as follows:

$$FIRST_k^1(\alpha) = \{a | \alpha \stackrel{*}{\Rightarrow} w \wedge w = xay \wedge x \in T^{k-1}\}$$
$$FOLLOW_k^1(A) = \{FIRST_k^1(\beta) | S \stackrel{*}{\Rightarrow} \alpha A \beta\}$$
$$LOOK_k^1(A \rightarrow \alpha \bullet \beta) = \{FIRST_k^1(\beta FOLLOW_k^1(A))\}$$

where

$$a \in T, y \in V^*, \alpha, \beta \in V^*$$

A set of $LOOK_k^1$ tables now allows to look at just the discriminating token τ_i, instead of having to inspect up to all k tokens.

In the FDE non-determinism is allowed. However, lookahead is still useful to prevent time consuming parsing and superfluous execution of detectors. By augmenting the exhaustive backtracking algorithm with some form of lookahead the FDE will be able to skip (many of) these dead alleyways.

In a feature grammar system the lookahead is restricted to the sentence belonging to one grammar component. So the sets and the table are constructed on a per component basis. To simulate a complete grammar a default erasing production is added for each detector symbol, including the component detector itself, appearing within the grammar component.

Using the $LOOK_k^1$ operation the parser can skip the validation of a Co alternative (see step b in Figure 4.8) by looking at the second token in the lookahead. When this token is R choose alternative ($Co \rightarrow \rho\ \beta\ \ldots$), when it is Nu validate the rule ($Co \rightarrow \rho\ Nu\ \ldots$).

Normally the lookahead depth is determined by steadily incrementing k until all decisions have become deterministic. In a feature grammar system two or more alternatives may completely overlap within the grammar component, *i. e.* the terminals are only interleaved with (at least one) different detectors. This may result in a $LOOK_k^1$ table which will still contain two or more rules for one set of lookahead values.

Memoization

Several parsing algorithms, *e. g.* chart parsers, depend for their efficiency on a well-know technique from dynamic programming: memoization [Mic68]. This technique basically means that each part of the input sentence is only parsed once. When, due to for example backtracking, the same partial sentence is reparsed the memoized parse tree is returned, thus saving processing time. In [Nor91] the author shows that by using this technique in a simple (deterministic) top-down parser the efficiency becomes equivalent to the much more advanced Earley parser, *i. e.* $O(n^3)$ (where n is the length of the sentence).

The same technique can be applied within the FDE, but it can also be taken one step further. Remember that the target of the Acoi system is to store the constructed parse trees persistently in a database. Also, references were added to the language in Section 3.2.3. These references make it possible to share (partial) parse trees. This can be generalized even more by sharing detector executions as stored in the database. This is possible as, stated in Chapter 2, detectors are (deterministic) functions, *i. e.* the same input always results in the same output. Once a detector has been called with a certain input the output may be memoized and reused, thus preventing superfluous execution. However, memoized detector functions should really be side effect free. Memoization will, for example, spoil the value of an internal counter which needs to be incremented to reflect the actual number of symbol instances.

When detector parse trees are memoized the storage will contain two kind of trees: *elementary trees* and *auxiliary trees*. Elementary trees are rooted by start symbols, they exist individually. Auxiliary trees are rooted by other detectors, they always need to be (indirectly) associated to a elementary tree. This distinction is also known in a NLP technique: *tree adjoining grammars* (TAG) [AR01]. In some variants of TAG trees are also described by D-Theory and quasi-nodes are used to perform substitution and adjoining. Substitution is, in the case of feature grammar systems, the binding of a specific auxiliary detector tree to an elementary tree.

Memoization may also partially resolve the problems with left-recursion (see Section 4.2.1), depending on the type of repetition. If the recursive structure also repeats the instantiations, this instantiation will be memoized, be referenced the next time it is encountered and thus break the recursion in the parser. The recursion in the constructed graph will be retained by the memoization reference.

4.3 The Feature Detector Engine

This section will describe the actual implementation of the exhaustive backtracking algorithm in the FDE. Before going into the details of the various components within the FDE, the actual form of the FDE needs to be determined.

A grammar can be used in two basic ways: (1) it can be interpreted by a generic parser, or (2) it can be input to a generator which produces a specific parser. These two ways lead to two basic architectures as shown in Figure 4.10. Of course both architectures have their advantages and disadvantages.

The main advantage of the generic parser is its adaptability. A change in the grammar leads to updates of its internal bookkeeping structures, and because those are not hardcoded the changes can be done during runtime [HKR90]. This adaptability comes at a loss of performance, which is the main advantage of a specialized, generated, parser. But in this case changes to the grammar can only be reflected by regeneration and recompilation. To prevent the FDS from having to manage these (possibly complicated) steps the FDE is implemented as a feature grammar driven parser.

Figure 4.10 shows that the parser is preceded by a lexer. In traditional parsers the

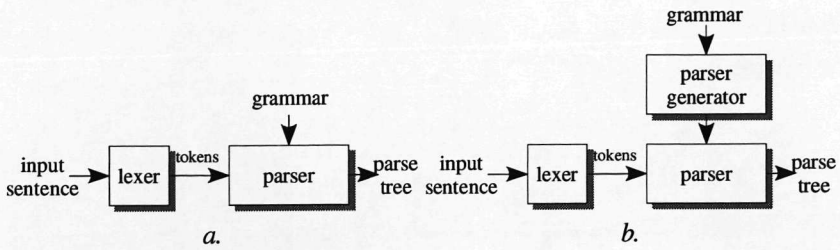

Figure 4.10: (a) A generic parser and (b) a specialized parser

lexer, which performs the lexical analysis, splits the input byte stream into meaningful tokens. In the FDE only a subset of the lexical analysis is needed, as the initial sentence and the output sentences produced by the detectors are already split into tokens. However, their validity is still checked using the specific atom validation rules (see Sections 2.2.2 and 3.1.2).

The internal architecture of the FDE is shown in Figure 4.11 and contains these components:

the symbol table is filled by a specific parser (based on the EBNF grammar in Appendix A) for the feature grammar language and contains all the information derived from the specific feature grammar, which is constructed by the developer;

the set of detectors are implemented by the developer and each of them can dynamically be loaded into the FDE;

the set of plugins are implemented by an expert and can take over the role of a detector, they can also be dynamically loaded into the FDE;

the set of tokens is gradually extended with the output of detectors, in fact multiple sets of tokens exist concurrently (one for each grammar component);

the controller uses the symbol table to call the detectors, to parse the tokens, and to gradually build the parse forest, *i. e.* implements the exhaustive backtracking parsing algorithm;

the parse forest is a DOM tree and can, when the parsing process has ended successfully, be dumped as an XML document containing all valid parse trees.

In the next subsections these components will be revisited and their specific implementation and optimization will be discussed.

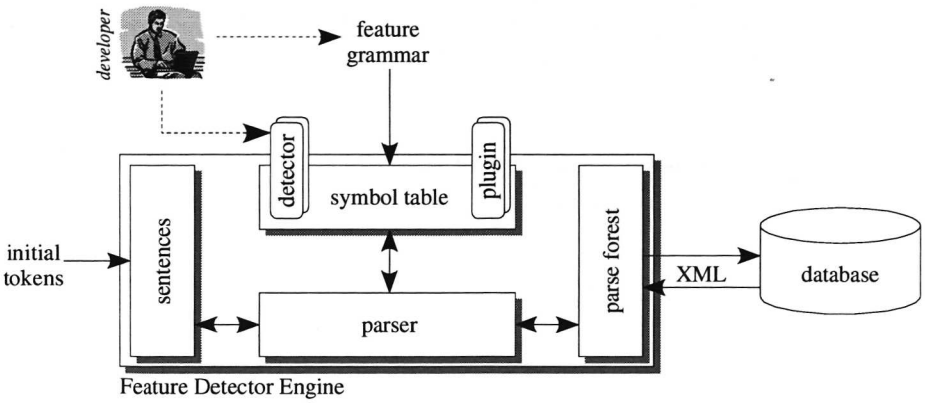

Figure 4.11: The FDE components

4.3.1 The Symbol Table

The symbol table is the basic bookkeeping structure of the FDE. It contains all information derived by parsing a specific feature grammar (which conforms to the language of Appendix A). This parsing step ensures the syntactic validity of the grammar. As shown in Chapter 3 some of the language constructs need additional semantic completion, *i. e.* rewrites. When the feature grammar system is complete a semantic check is needed to validate some additional constraints and warn the developer of some (unwanted) properties of the grammar. The rewrites and semantic checks are the topics of the upcoming subsections.

Rewriting

The use of the feature grammar language allows a developer to describe a feature grammar system in a intuitive fashion. However, to achieve this some symbols and rules have become implicit. At some points during the parsing of a feature grammar these symbols and rules are made explicit by applying specific rewrites or adding annotations to the symbol entry in the symbol table.

Symbol sequences In the FDE symbol sequences are not rewritten but the occurrence indicators are translated into a lower and upper bound. These bounds are checked by a WHILE-statement in the parser implementation (see Section 3.1.1), *i. e.* greedy alternatives are favored.

Symbol groups For each symbol group an anonymous is introduced, according to the rewrites shown in Section 3.1.1. Extra care is taken to prevent these symbols to clog up the parse forest by the use of *edge folding*.

Detector confidences The compulsory confidence value (see Section 6) is enforced by the implementation skeleton of detectors, this will be illustrated in Section 4.3.5.

Classifiers Once more the formal rewrite is embedded within the parser instead of applying the rewrite explicitly. Due to the specific entry in the symbol table the FDE knows when and how to call the *analyze* or the *predict* detectors (see Section 3.2.2).

Notice that the greediness of this implementation would not notice the ambiguity of the example parse in Section 4.2.1. Only one alternative of the *Image* rules will be found. The greedy implementation conforms more to the usual semantic meaning of optionality: the symbol exists or not, *i. e.* both alternatives are not considered at the same time. As indicated in Section 3.1.1 the greedy implementation results in a iterative interpretation of symbol sequences. This interpretation circumvents the introduction of anonymous symbols and keeps resolving the XPath expressions relatively easy.

Semantic Checks

The semantic analysis of the grammar ensures that the symbol table and the embedded grammar rules are semantically consistent. Furthermore, a series of checks is performed on the grammar to warn the developer of "unwanted" properties:

Check for unknown symbols When a symbol appears in a RHS, which has no rules but is also not a terminal or a detector, the symbol table does not know it yet. These unknown symbols become non-terminals with an, implicit, empty rule.

Check for naming conflicts A naming conflict happens when there are several (imported) namespaces to which a symbol can be bound.

Check for unique rules A warning is issued when a non-terminal contains exactly the same production rule more than once.

Check for factors The rules are checked for possible shared pre- and suffixes, a warning is issued when such a possibility is found.

Check for recursion Left-recursive non-terminals may lead to infinite parses. The FDE issues a warning when left-recursion is found, however, only the developer can resolve these or may have already solved them in the detector implementation.

Check for non-reachable symbols This check issues warnings about symbols which may never be reached from a specific start symbol. Notice that these symbols may be reachable from another valid start symbol.

Check for valid path expressions Using the detector dependency graph (as will be discussed in Chapter 6) the FDE checks if all the paths point to one or a set of other nodes.

Check for independent alternatives Path expressions may not point into other alternatives of the same context node, as each alternative will belong to a different parse tree and this will make the alternatives order dependent.

Check for possible deadlocks Check if a reference crossing in a parameter path expression may lead to violation of the linear ordering of detectors.

During the parsing process the controller uses the production rules and symbol information from the table to adapt its generic implementation of the exhaustive backtracking algorithm to the specific feature grammar system.

4.3.2 The Parser

Recursive descent is a popular method to implement exhaustive backtracking. In this method specialized functions are generated for each non-terminal, which are recursively called according to the exact semantics of the production rules. In this case, where the FDE is a generic parser, the specialized function is replaced by a generic one which adapts its behavior on the basis of knowledge from the symbol table and the production rules. The implementation of this generic function is shown in pseudo code in Figure 4.12. The other parsing functions (see lines 10 to 21) are all variations on this function. For example the *parse-detector* function will create a local new sentence s by executing the detector function (after successfully binding the input sentence), and will check if it is empty before declaring itself valid.

The next sections will focus on the various components the parser interacts with: the set of sentences and the set of parse trees, *i. e.* the parse forest.

4.3.3 The Parse Forest

The parse forest is the main result of the FDE. Due to the, possible, ambiguous nature of a feature grammar system and its mild context-sensitivity the parse forest is a rather complex data structure. To manage this structure several control mechanisms have been introduced in Section 2.2.4. Before discussing the actual implementation and use of these mechanisms the global (standardized) data structure is introduced.

XML and DOM

As has been shortly mentioned in Section 3.1.3 XML documents describe tree structures [W3C00]. Due to the fact that XML is very popular as an exchange format on the WWW it, and many related standards, has been quickly adopted and implemented

```
function parse-non-terminal
input
  T: the parse trees under construction
  s: the remainder of the input sentence under inspection for ctxt
  nt: the non-terminal being validated
output when nt is valid: (T, S)
  T: the extended parse trees
  S: the input sentence remainders to be inspected
output when nt is invalid: ∅
implementation
  T' = expand t in T with a node for nt in s.ctxt
  valid = ∅
  for each production rule r of symbol nt:
   if s matches the lookahead requirements of rule r:
    s' = copy s in (new) context s'.ctxt derived from s.ctxt
    for each symbol rhs in production rule r:
     for each s'' in s':
      bound = 0
      while bound < the higher bound of rhs:
       if the rhs symbol is a start symbol:
        res = parse-start(T', s'',the rhs symbol)
       else if the rhs symbol is a classifier:
        res = parse-classifier(T', s'',the rhs symbol)
       else if the rhs symbol is a detector:
        res = parse-detector(T', s'',the rhs symbol)
       else if the rhs symbol is anonymous:
        res = parse-anonymous(T', s'',the rhs symbol)
       else if the rhs symbol is a non-terminal:
        res = parse-non-terminal(T', s'',the rhs symbol)
       else if the rhs symbol is a terminal:
        res = consume-terminal(T', s'',the rhs symbol)
       if res == ∅:
        break
       bound = bound + 1
      end
      if bound < the lower bound of rhs:
       rule r is invalid
       break
      end
      if rule r is valid:
       s' = res.s
     end
    end
    if rule r is valid:
     add s' to valid
    else:
     delete all nodes related to r from T'
   end
  end
  if valid == ∅:
   delete T' from T
   return ∅
  end
  return (T', valid)
```

Figure 4.12: Implementation of the generic parsing function

for a wide range of operating systems and programming languages. The FDE implementation uses an implementation [Vei03] of the Document Object Model (DOM) standard [W3C01a] as an internal representation of the parse tree. This DOM tree can be easily accessed by XPath expressions, *i. e.* whitebox detectors and parameter expressions are easily resolved.

Labeling Parse Trees

In the parse forest as introduced in Section 2.2.4 each node is labeled with a specific *context*, *i. e.* the parse trees the node is a member of. This context is a list of binary flags, where each flag represents a parse tree. When the flag is *true* the node belongs to the parse tree. The disadvantage of this rather simple scheme becomes clear when a new tree is added to the forest. All known nodes have to be revisited to indicate if they belong to the new tree (or not). To prevent these superfluous runs through the forest the context of a node should only be set when the parsing algorithm visits this node, *i. e.* in a *pre-* or *post*-visitation.

A pre-visitation takes place when the parser starts the validation of a non-terminal. At that moment the parser only knowns the intermediate number of trees in the forest: this number is called the *scope* of the context. In principle the node is a possible member of all new parse trees which are added later on, however, those trees are outside its current scope. A new tree (except for the initial tree) always shares nodes with an older tree, *e. g.* its ancestors or the trees it took its detector parameters from. At least the root of the forest is shared by all trees.

After validation of the production rules of a non-terminal the node receives a post-visitation. At that moment the parser knows how many parse trees have been added by these rules and the scope of the context can be enlarged.

To illustrate the use of the context and scope in pre- and post-visitation this, rather artificial but highly ambiguous, feature grammar is used:

```
1  %module          ambigue;

2  %start           S();

3  %detector        b [ return i = 1 ];
4  %detector        c [ return i = 10 ];
5  %detector        d [ return j = 100 ];

6  %detector        e [ return i = a//i * 2 ];
7  %detector        g [ return i = a//i + 2 ];
8  %detector        h [ return i = a//i - 2 ];

9  %atom            i, j;

10 S                : a e?;
```

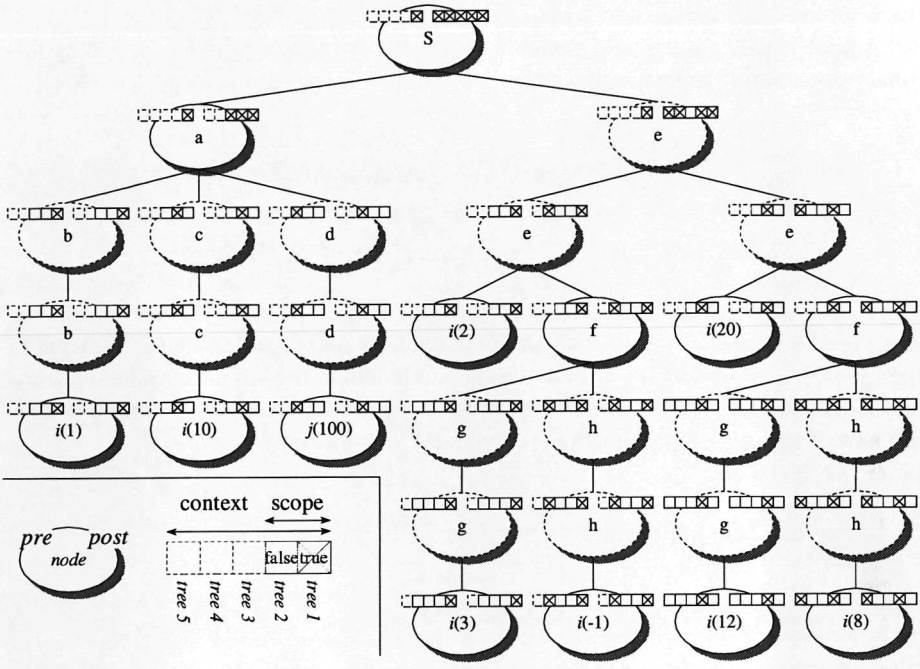

Figure 4.13: A parse forest

```
11 a          : b | c | d;
12 b          : i;
13 c          : i;
14 e          : i f;
15 d          : j;
16 f          : g | h;
17 g          : i;
18 h          : i;
```

A run of the FDE for this feature grammar (which has only one possible run) results in the parse forest shown in Figure 4.13. This forest contains 5 trees. The scope of the node contexts increases with the top-down left to right construction of the parse tree. The non-terminal a has three valid alternatives leading to the addition of two new trees, as the first alternative extends the existing tree. The parameter of the detector e has now an ambiguous binding: either $i(1)$ or $i(10)$. This leads to two quasi-foots representing two executions of the detector function f_e in the two contexts. The non-terminal f has once more two alternatives leading to the addition of two new trees, each within their specific context. The g subtrees extend the existing trees, while

the h subtrees are derived new trees.

A node can determine which other nodes in the forest belong to its context by this binary operation ($ indicates the current node and @ indicates the inspected node):

$$npad(@scope, @context) \;\&\; npad(\$scope, \$context)$$
$$=$$
$$npad(max(@scope, \$scope), max(@context, \$context))$$

The $npad$ function sets all flags outside of the context scope to the default value $true$. The max operations determine which of the nodes is deeper and further to the right of the forest, *i. e.* more specific as nodes higher and more to the left have a smaller scope and are shared more. Take for example the two possible h roots. The first one does not have $i(10)$ in its scope and context (where $t = true$ and $f = false$):

$$npad(3, ftf) \;\&\; npad(4, tfft) \neq npad(max(3,4), max(ftf, tfft))$$
$$ttftf \;\&\; ttfft \neq npad(4, tfft)$$
$$ttfff \neq ttfft$$

Doing the same inspection for the second h root results in a positive match:

$$npad(3, ftf) \;\&\; npad(5, tfftf) = npad(max(3,5), max(ftf, tfftf))$$
$$ttftf \;\&\; tfftf = npad(5, tfftf)$$
$$tfftf = tfftf$$

This also shows that the validity contexts of the h roots are in fact determined by their ancestor, the e quasi-foots.

In the post-visitation all contexts of the compulsory children of the node, *i. e.* those with a lower bound of one or more, are unified. See for example the quasi-root of e. The third tree does not contain an e node, however, this symbol is optional leading to a valid S node and thus to a valid third parse tree. The post context replaces the pre context.

This matching operation is used for resolving ambiguous parameter bindings by adding a feature grammar system specific nodetests to the XPath expression.

Memoized Parse Trees

Persistently memoized parse trees function for the FDE as a persistent lookup table of detector calls. Each detector call is identified by a quasi-foot which contains information about a specific input sentence. As a detector is a partial function this input

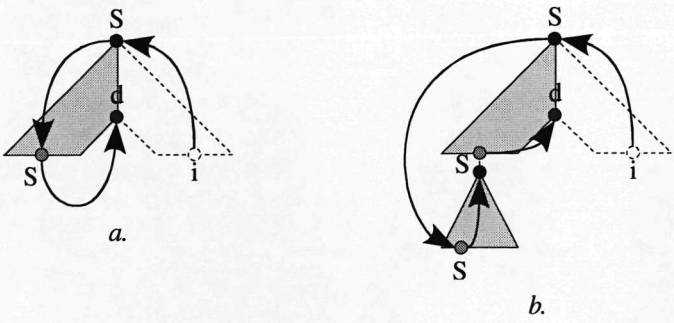

Figure 4.14: A deadlock situation due to (a) a direct and (b) an indirect self reference

sentence always maps to the same, stored, output sentence. The FDS, which will be discussed in detail in Chapter 6, manages the lookup table.

The moment the FDE has assembled a complete input sentence a request for the parse tree is send to the FDS. When there exists a mapping for this input sentence the FDS will return the unique identifier for the tree and its availability, the FDE will then take the appropriate action:

1. when the parse tree is *available*, the identifier is stored within the quasi-foot as a place holder;
2. when the parse tree is *under construction*, the FDE will have to wait till it is know if the mapping exists, *i. e.* the parse tree becomes available, is unknown or a deadlock situation occurs (which will be discussed in the next paragraph);
3. when the mapping *does not exist*, the detector symbol can be rejected by the parser;
4. when the mapping of a black- or whitebox detectors is *unknown* the FDE will inform the FDS that it will execute the detector to instantiate the parse tree, *i. e.* the parse tree becomes under construction.

In principle parse trees are not loaded from the lookup table, until a value is needed as part of an input sentence. The FDE then sends a request for the complete parse tree or the specific value, depending on the abilities of the underlying XML storage structure, to the FDS. When the parse tree is still under construction there may be a deadlock situation. Such a situation occurs when, by a reference, the linear ordering is violated. Figure 4.14 illustrates the two basic deadlock forms: due to a direct self-reference, *e. g.* `d(&S//i)`, or an indirect self-reference, *e. g.* `d(&S//&S//i)`. As a global deadlock resolution strategy is not possible the detector is informed and expected to handle the situation leading to a memoizable parse tree (see Section 4.3.5).

In the previous section the trees within the parse forest have been labeled using a scope and context mechanism. However, these elementary and auxiliary parse trees

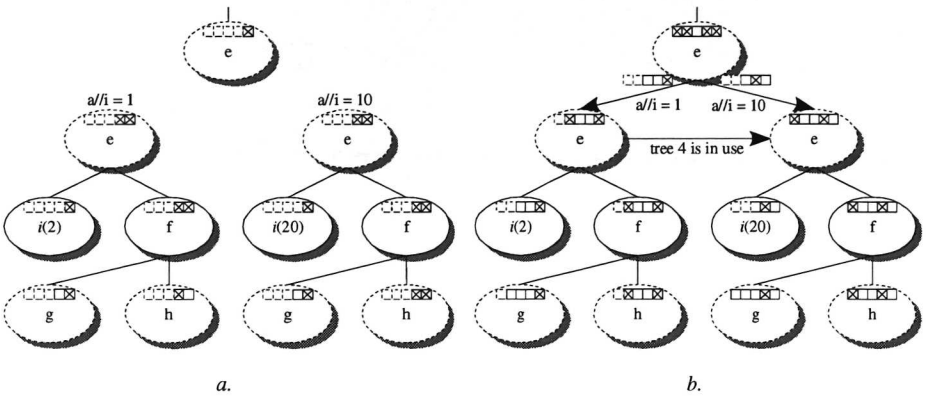

Figure 4.15: (a) Local parse forests are (b) combined in a global forest.

will be memoized. A memoized parse tree may be loaded in another forest with a different context. During saving, the context has to be localized, while during loading the context has to be globalized. Localization means that the parse tree loses the inherited global context, only the local context remains. Globalization then reinstates a, possibly different, global context.

Figure 4.15 illustrates this process. The local forests are derived from Figure 4.13. Notice that bits in use by siblings are stripped out, *e. g.* bit 4 for the second alternative of e. Figure 4.15.b globalizes the context once more by replacing bit 1 by the global context. As the first alternative claims another bit, *i. e.* creating a difference between the scopes of the quasi-root and the current scope, bit 4 is once more inserted for the second alternative. This dynamic behavior of the context bits makes it useless to persistently store the post-context as a change in one of the memoized trees may use up more bits.

4.3.4 The Sentences

Sentences are produced per grammar component by the detector function. Internally a sentence is a simple linked list of tokens. Figure 4.12 shows that for each alternative production rule a copy of the sentence is made (s'). In fact only a copy of the token pointer is made, so each alternative points to its own current position in the sentence. Each copy is associated with a context, *i. e.* corresponds with a specific parse tree within the parse forest.

The stack of sentences under inspection, needed for resuming the validation of the sentence after control has been temporarily transfered to another grammar component (see the upcoming Section 4.2.1), is implicit, as each sentence is a local variable of a specific call of the *parse-detector* function.

4.3.5 Detectors

Detector Input

Detector parameters are identified by XPath expressions. These XPath expressions are normalized by the feature grammar parser. In this process these rewrites are applied:

1. The default axis for the first step is `preceding::`;
2. By default only the last match is returned, *i. e.* add `[fn:position() = 1]` for a reverse axis and `[fn:position() = fn:last()]` for a forward axis.
3. The feature grammar specific reference operation *&node* is translated into a *node* `[fg:bind(@id)]` call. This FDE specific XSLT extension function returns a nodeset containing the root node of the refereed (memoized) parse tree, *i. e.* this may have to be loaded just-in-time from the database.
4. The parse forest may contain several types of anonymous nodes, *e. g.* quasi-foots. The developer does not know about those nodes and thus will not take care of them within his XPath expressions. Between each two steps a skip expression is inserted in the vain of `/descendant::*[contains(@type, '.q.')]/`. This a rather expensive solution. It is cheaper to prevent creating these nodes at all. This can be done with anonymous nodes which do not contain additional information, *e. g.* group nodes. These parse forests stay closer to the semantic grammar and are also called *Reduced Derivation Trees* (RDT) [JS98].
5. Detector parameters may only be bound within the context of the current node. This XPath nodetest will only allow nodes which are within the current context scope:

```
[ fg:and(
    fg:npad( @scope, @ctxt),
    fg:npad( current()/@scope, current()/@ctxt) )
  =
  fg:npad( fg:max( @scope, current()/@scope),
    fg:max( @ctxt, current()/@ctxt) ) ]
```

The resulting XPath expressions can be resolved against the internal DOM tree. The result may be several sets of input parameters for different contexts. For each context a detector call will be bound to a quasi-foot.

```
1  (Confidence,Sentence) Skin(Token myLocation) {
2       Sentence  mySentence = newSentence();
3       Image     myImage    = openImage(getValue(myLocation));
4       Bitmap    myBitmap   = deriveBitmap(myImage, false);
5       Iterator  myPixels   = newIterator(getPixels(myImage));
6       while(hasMore(myPixels))
7            if (isSkin(nextElement(myPixels)))
8                 nextBit(myBitmap, true);
9       putToken(mySentence, "Skin/bitmap", myBitmap);
10      return (0.95, mySentence);
11 }
```

Figure 4.16: Implementation of the *Skin* blackbox detector in pseudo code

Blackbox Detectors

Blackbox detectors are implemented in the host language of the FDE, *i. e.* a *general purpose language* (GPL) like *C*. Figure 4.16 shows an implementation of the *Skin* detector in pseudo code.

The detector receives its input sentence as a set of tokens from the parse tree. It uses this information, *i. e.* the *Location* of the *Image*, to load the image. A new bitmap is created and filled by iterating over the pixels of the image and determining if they are a skin pixel or not. The new *bitmap* token is then added to the newly created output sentence which is returned to the FDE. Next to the sentence also the compulsory confidence information is returned: the *Skin* detector knows for 95% sure that these pixels are really skin.

Plugins

Plugins take over a large part of the coding burden from the developer by implementing a generic detector. Plugins come in the two basic variants of detectors: blackbox and whitebox. In the first case only the input parameters are provided, while in the latter case those are embedded within a template in a *domain specific language* (DSL), like XPath.

Figure 4.17 shows the implementation of the *matlab* plugin. The plugin receives a list of requested parameters belonging to one context. Using the symbols name, *e. g.* *Color*, a command call is constructed. When the command was successfully executed

```
(Confidence, Sentence) matlab(Symbol mySymbol, List myParams) {
    Engine myEngine = startEngine(getProperty("matlab"));
    if (myEngine) {

        String myCommand = getName(mySymbol) + "(";
        Iterator myIterator = newIterator(myParams);
        if (hasMore(myIterator))
            myCommand += getValue(nextElement(myIterator));
        while(hasMore(myIterator))
            myCommand += "," + getValue(nextElement(myIterator));
        myCommand += ")";

        Sentence mySentence = runEngine(myEngine, myCommand);
        if (closeEngine(myEngine) && mySentence)
            return (1.0, mySentence);
    }
    return (0.0, newSentence());
}
```

Figure 4.17: Implementation of the *matlab* plugin in pseudo code

the output sentence and a confidence of 100% is returned to the FDE. When the execution was unsuccessful a zero confidence is returned, which will lead to rejection of the symbol.

The same process happens for whitebox plugins although the FDE handles, instead of the list of parameters, the instantiated template over to the plugin implementation. So binding detector parameters is always done by the FDE, just like with blackbox detectors. But a plugin has additional access to the symbol table and can thus adapt its course on the actual rule context of the symbol.

Classifiers

Classifiers are special in the sense that they imply two detectors, both are in fact implemented as a plugin. Figure 4.18 and Figure 4.19 show the implementation of these two detectors for the *bpnn* classifier.

Start Symbols and References

Start symbols and references are once more implemented as plugins, *i. e.* the feature grammar developer does not have to provide any code for these detectors.

Only one start symbol is instantiated in a specific FDE run. This detector looks in the environment of the FDE for the required initial tokens. This environment consists of notifications of the FDS, the command line of the FDE or interaction with the

```
(Confidence,Sentence)bpnn.analyze(Symbol mySymbol,List myParams){
    Confidence myResult = 1.0;

    bpnn myNN = openBPNN(getName(mySymbol)+".net");
    if (!myNN)
        myNN = newBPNN(getLength(getParameters(mySymbol)),4,2);

    Iterator myIterator = newIterator(myParams);

    targetBPNN(myNN, 1, myResult);
    targetBPNN(myNN, 2, atoi(getValue(nextElement(myIterator))));

    integer i = 1;
    while (hasMore(myIterator))
        inputBPNN(myNN, i++, getValue(nextElement(myIterator)));

    trainBPNN(myNN);
    saveBPNN(myNN, getName(mySymbol)+".net");
    closeBPNN(myNN);

    return (myResult, newSentence());
}
```

Figure 4.18: Implementation of the *bpnn.analyze* detector in pseudo code

librarian. When all tokens are available the parsing algorithm starts the validation process.

References take their required tokens from the sentence under inspection. Then they request the FDS for the identifier and status of the parse tree belonging to the sentence constructed from these tokens (see Section 4.3.3). If the parse tree is not yet known the FDE can build the parse tree, as the input sentence is available, and it needs to know if the tree is valid.

Deadlock Resolution

Sections 3.2.3 and 4.3.3 identified that deadlocks have to be resolved by the developer within the detector implementation. For this the developer will have to check if one of the tokens received from the FDE is empty[1]. The developer has then three options: (1) use a default value, (2) know how to retrieve the value, which will only work when the token is part of the output sentence of this detector, or (3) let the detector fail. In the case of failure the detector symbol will not be accepted by the FDE.

[1]This means a self reference because when the token is just not available in the parse forest the detector would not have been executed, *i. e.* its start condition is not valid.

```
(Confidence,Sentence)bpnn.predict(Symbol mySymbol,List myParams){
    Confidence myResult = 0.0;

    bpnn myNN = openBPNN(getName(mySymbol)+".net");
    if (!myNN) {
        myNN = newBPNN(getLength(getParameters(mySymbol)),4,2);
        saveBPNN(myNN, getName(mySymbol)+".net");
    }

    integer i = 1;
    Iterator myIterator = newIterator(myParams);
    while (hasMore(myIterator))
        inputBPNN(myNN, i++, getValue(nextElement(myIterator)));

    feedforwardBPNN(myNN);
    Confidence myResult   = outputBPNN(myNN, 1);
    Sentence   mySentence = newSentence(outputBPNN(myNN, 2));

    closeBPNN(myNN);

    return (myResult, mySentence);
}
```

Figure 4.19: Implementation of the *bpnn.predict* detector in pseudo code

4.4 Discussion

This chapter contained a detailed description of the design and implementation decisions made for the FDE. The FDE steers the actual annotation extraction process by interpreting a specific feature grammar system described by a feature grammar. The top-down parsing algorithm, implemented in the FDE, is interrupted by the execution of detector algorithms.

This execution model may seem not too different from the way actions are associated to attribute grammars [GJ98] and interrupt the parser, *e. g.* as in parsers generated by Yacc [LMB92]. However, those actions can only intervene in a limited way in the parsed sentence, *e. g.* push a token back on the stack. The parsed sentence is completely available, while in the FDE the parsed sentence is extended just-in-time. This limits the parser severely in taking decisions based on lookahead. As discussed, lookahead can only be used within a grammar component, where the complete sentence is available. Bottom-up algorithms, like used in Yacc, may be used within individual components. However, the control transfer between components complicates this. Postponing this transfer may enable the use of, in general, more efficient bottom-up algorithms, and is thus an interesting topic for future research.

Performance can also be boosted by replacing the depth-first algorithm with a breadth-first algorithm, *i. e.* each parse tree gets its own parsing thread. Detectors should already be side-effect free, but shared data structures, like the parse forest, will have to be guarded by critical sections or replaced by localized copies. Investigation of the theory of PC grammar systems may also be of interest here.

The current implementation is in *C*. However, other implementation strategies are well possible, *e. g.* in a functional language or in the form of generation of ToolBus scripts or translating context dependencies into output/input dependency for a dataflow or a daemon architecture (see Section 2.3). However, the *C* implementation gave more freedom in staying close to a well known parsing algorithm and thus study the impact of the extensions of feature grammar systems. A future ToolBus or daemon implementation may allow to incorporate more concurrency, and may also allow relaxation of the deadlock prevention strategy.