15

Non-Chomsky Grammars and Their Parsers

Just as the existence of non-stick pans points to user dissatisfaction with "sticky" pans, the existence of non-Chomsky grammars points to user dissatisfaction with the traditional Chomsky hierarchy. In both cases ease of use is the issue.

As we have seen in Section 2.3, the Chomsky hierarchy consists of five levels:

- phrase structure (PS),
- context-sensitive (CS),
- context-free (CF),
- regular (finite-state, FS) and
- finite-choice (FC).

Although each of the boundaries between the types is clear-cut, some boundaries are more important than others. Two boundaries specifically stand out: that between context-sensitive and context-free and that between regular (finite-state) and finite-choice. The significance of the latter is trivial, being the difference between productive and non-productive, but the former is profound.

The border between CS and CF is that between global correlation and local independence. Once a non-terminal has been produced in a sentential form in a CF grammar, its further development is independent of the rest of the sentential form, but a non-terminal in a sentential form of a CS grammar has to look at its neighbors on the left and on the right, to see what production rules are allowed for it. The local production independence in CF grammars means that certain long-range correlations cannot be expressed by them. Such correlations are, however, often very interesting, since they embody fundamental properties of the input text, like the consistent use of variables in a program or the recurrence of a theme in a musical composition.

15.1 The Unsuitability of Context-Sensitive Grammars

The obvious approach would be using a CS grammar to express the correlations (= the context-sensitivity) but here we find our way obstructed by three practical

rather than fundamental problem areas: understandability, parsability, and semantic suitability.

15.1.1 Understanding Context-Sensitive Grammars

CS grammars *can* express the proper correlations but not in a way a human can understand. It is in this respect instructive to compare the CF grammars in Section 2.3.2 to the one CS grammar we have seen that really expresses a context-dependency, the grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ from Figure 2.7, repeated here in Figure 15.1. The grammar for

```
 Ss  -->  abc | aSQ
bQc  -->  bbcc
 cQ  -->  Qc
```

Fig. 15.1. Context-sensitive grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

the contents of a book (Figure 2.10) immediately suggests the form of the book, but the grammar of Figure 15.1 hardly suggests anything, even if we can still remember how it was constructed and how it works. This is not caused by the use of short names like **Q**: a version with more informative names (Figure 15.2) is still puzzling.

```
           Ss  -->  a b c | a S bc_pack
b bc_pack c  -->  b b c c
  c bc_pack  -->  bc_pack c
```

Fig. 15.2. Context-sensitive grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ with more informative names

Also, one would expect that, having constructed a grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$, making one for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n\mathbf{d}^n$ would be straightforward. That is not the case; a grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n\mathbf{d}^n$ requires rethinking of the problem (see Problem 15.1).

The cause of this misery is that CS and PS grammars derive their power to enforce global relationships from "just slightly more than local dependency". Theoretically, just looking at the neighbors can be proved to be enough to express any global relation, but the enforcement of a long-range relation through this mechanism causes information to flow through the sentential form over long distances. In the production process of, for example $\mathbf{a}^4\mathbf{b}^4\mathbf{c}^4$, we see several **bc_pack**s wind their way through the sentential form, and in any serious CS grammar, many messengers run up and down the sentential form to convey information about developments in far-away places. However interesting this imagery may be, it requires almost all rules to know something about almost all other rules; this makes the grammar absurdly complex.

15.1.2 Parsing with Context-Sensitive Grammars

Parsing speed is also an issue. FS parsing can always be done in linear time; many CF grammars automatically lead to linear-time parsers; CF parsing needs never be more expensive than $O(n^3)$ and is usually much better; but no efficient parsing algorithms for CS or PS grammars are known. CS parsing is basically exponential in general — although (Web)Section 18.1.1 reports some remarkable efforts — and PS parsing is not even solvable in theory.

Still, many CS languages can be recognized in linear time, using standard CF parsing techniques almost exclusively. The language $a^n b^n c^n$ is a good example. We have already observed (at the beginning of Chapter 13) that it is the intersection of two CF languages, $a^n b^n c^m$ and $a^m b^n c^n$: the first language forces the numbers of `a`s and `b`s to be equal, the second does the same for the `b`s and the `c`s. We can easily create linear-time recognizers for both languages, for example by writing LL(1) grammars for them. We can then test the string with both recognizers, and if they both recognize the string, it belongs to the language $a^n b^n c^n$; otherwise it is rejected. This test can be done in linear time. So it is not totally unreasonable to demand good recognition speed for at least some non-CF languages. Constructing a parse tree depends on the exact form of the grammar and may be much more expensive; the result may not even be a tree but a dag, as it was in Figure 2.8.

15.1.3 Expressing Semantics in Context-Sensitive Grammars

A third problem concerns the semantic suitability. Although this book has not emphasized the semantic aspect of language processing (see Section 2.11), that aspect is of course important for anybody who is interested in the results of a parsing. In FS systems, semantic actions can be attached to regular expressions or transitions (see Section 5.9). CF grammars are very convenient for expressing semantics: to each production rule $A \rightarrow A_1 A_2 \cdots A_k$ code can be attached that composes the semantics of A from that of its children A_1, A_2, ..., A_k. But it is less than clear where we can find or attach semantics in a CS rule like `b bc_pack c --> b b c c`.

15.1.4 Error Handling in Context-Sensitive Grammars

Less important than the above but still an issue in practice is the behavior of a parsing technique on incorrect input: error detection ("Is there an error?"), error reporting ("Where exactly is the error and what is it?") and error repair ("Can we repair and continue?"). As we shall see in Chapter 16, error handling with CF grammars is a difficult area in which only moderately good answers are known. Error handling with *non-CF* grammars can be a nightmare. Already error *detection* can be a serious problem, since the parser is easily tempted to try an infinite number of increasingly complex hypotheses to explain the unexplainable: incorrect input then leads to non-termination. And given a non-Chomsky parser for the language *ww*, where *w* is an arbitrary string of `a`s and `b`s, and the input `aabbaabbab`, where exactly is the error, and what would be a sensible error message?

15.1.5 Alternatives

Many grammar forms have been put forward to mitigate the above problems and make long-range relationships more easily expressible. We shall look at several of them, with a special eye to understandability, parsability, semantic suitability, and error handling. More in particular we will look at VW grammars, attribute grammars, affix grammars, tree-adjoining grammars, coupled grammars, ordered grammars, recognition systems, Boolean grammars, and §-calculus. Of these, the recognition systems and to a certain extent §-calculus are particularly interesting since they question the wisdom of describing sets by generative means at all. Given the large variety of non-Chomsky systems, the relative immaturity of most of them, and the limited space, our descriptions of these systems will be shorter than those in the rest of this book. The bibliography in (Web)Section 18.2.6 contains explanations of several other non-Chomsky systems.

One interesting possibility not explored here is to modify the CF grammar under the influence of parser actions. This leads to *dynamic grammars*, also called *modifiable grammars*, or *adaptable grammars*. As the names show, the field is still in flux. See Rußmann [280] for theory and practice of LL(1) parsing of dynamic grammars. (Web)Section 18.2.6 contains many more references on the subject.

Each of the non-Chomsky systems should come with a paradigm, telling the user how to look at his problem so as to profit best from it. For a CF grammar this paradigm is fairly obvious, but even for a CS grammar it is not, as the grammar of Figure 15.1 amply shows. With the exception of VW grammars and attribute grammars, not enough experience has been gathered to date with any of the non-Chomsky systems for a paradigm to emerge. Also, VW grammars and attribute grammars are the only ones of the methods discussed here that can more or less conveniently describe large real-world context-sensitive systems, as the ALGOL 68 report [244] and several compilers based on attribute grammars attest.

There is one example of a non-Chomsky grammar type for CF languages: Floyd productions; they were already discussed in Section 9.3.2. Push-down automata (Section 6.2) could be considered another.

15.2 Two-Level Grammars

It is not quite true that CF grammars cannot express long-range relations; they can only express a finite number of them. If we have a language the strings of which consist of a **begin**, a **middle** and an **end** and suppose there are three types of **begin**s and **end**s, then the CF grammar of Figure 15.3 will enforce that the type of the **end** will properly match that of the **begin**, independent of the length of **middle**.

We can think of **(** and **)** for **begin1** and **end1**, **[** and **]** for **begin2** and **end2** and **{** and **}** for **begin3** and **end3**; the CF grammar will then ensure that each closing parenthesis will match the corresponding open parenthesis.

```
textS  -->  begin1 middle end1
         |  begin2 middle end2
         |  begin3 middle end3
```

Fig. 15.3. A long-range relation-enforcing CF grammar

By making the CF grammar larger and larger, we can express more and more long-range relations; if we make it infinitely large, we can express any number of long-range relations and have achieved full context-sensitivity. Now we come to the fundamental idea behind two-level grammars. The rules of the infinite-size CF grammar form an infinite set of strings; in other words, it is a language, which can in turn be described by a grammar. This explains the name *two-level grammar*.

The type of two-level grammar described in this section was invented by van Wijngaarden [244] and is often called a *VW grammar*. There are other kinds of two-level grammars, for example those by Krulee [270]; and some ordered grammars (Section 15.6) also use two grammars.

15.2.1 VW Grammars

To introduce the concepts and techniques we shall give here an informal construction of a VW grammar for the language $L = \mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ for $n \geq 1$ from the previous section. We shall use the VW notation as explained in Section 2.3.2.3: the names of terminal symbols end in **symbol** and their representations are given separately; rules are terminated by a dot (**.**); alternatives are separated by semicolons (**;**); members inside alternatives are separated by commas, allowing us to have spaces in the names of non-terminals; and a colon (**:**) is used instead of an arrow to separate left- and right-hand side.

Using this notation, we could describe the language L through a context-free grammar if grammars of infinite size were allowed:

```
textS:  a symbol, b symbol, c symbol;
        a symbol, a symbol,
              b symbol, b symbol,
              c symbol, c symbol;
        a symbol, a symbol, a symbol,
              b symbol, b symbol, b symbol,
              c symbol, c symbol, c symbol;
        ...         ...
```

We shall now try to master this infinity by constructing a grammar which allows us to produce the above grammar as far as needed. We first introduce an infinite number of names of non-terminals:

```
textS:  ai, bi, ci;
        aii, bii, cii;
        aiii, biii, ciii;
        ...        ...
```

together with three infinite groups of rules for these non-terminals:

```
ai:    a symbol.
aii:   a symbol, ai.
aiii:  a symbol, aii.
...    ...

bi:    b symbol.
bii:   b symbol, bi.
biii:  b symbol, bii.
...    ...

ci:    c symbol.
cii:   c symbol, ci.
ciii:  c symbol, cii.
...    ...
```

Here the **i** characters count the number of **a**s, **b**s and **c**s. Next we introduce a special kind of name called a *metanotion*. Rather than being capable of producing (part of) a sentence in the language, it is capable of producing (part of) a name in a grammar rule. In our example we want to catch the repetitions of **i**s in a metanotion **N**, for which we give a context-free production rule (a *metarule*):

```
N :: i ; i N .
```

Note that we use a slightly different notation for metarules: left-hand side and right-hand side are separated by a double colon (**::**) rather than by a single colon and members are separated by a blank ( ) rather than by a comma; also, the metanotion names consist of upper case letters only (but see the note on numbered metanotions in Section 15.2.2). The set of metarules in a VW grammar is called the *metagrammar*. The metanotion **N** produces the segments **i**, **ii**, **iii**, etc., which are exactly the parts of the non-terminal names we need.

We can use the production rules of **N** to collapse the four infinite groups of rules into four *finite* rule templates called *hyperrules*, as shown in Figure 15.4.

```
N ::     i ; i N .

textS:   a N, b N, c N.

a i:     a symbol.
a i N:   a symbol, a N.

b i:     b symbol.
b i N:   b symbol, b N.

c i:     c symbol.
c i N:   c symbol, c N.
```

Fig. 15.4. A VW grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

Each original rule can be obtained from one of the hyperrules by substituting a production of `N` from the metarules for each occurrence of `N` in that hyperrule, provided that *the same production* of `N` is used consistently throughout; this form of substitution is called *consistent substitution*. To distinguish them from normal names, these half-finished combinations of lower case letters and metanotions (like `a N` or `b i N`) are called *hypernotions*. Substituting, for example, `N=iii` in the hyperrule

```
b i N: b symbol, b N.
```

yields the CF rule for the CF non-terminal `biiii`:

```
biiii: b symbol, biii.
```

We can also use this technique to condense the *finite* parts of a grammar by having a metarule `A` for the symbols `a`, `b` and `c`. ("`A`" stands for "alphabetic".) Again the rules of the game require that the metanotion `A` be replaced consistently. The final result is shown in Figure 15.5. We see that even the names of the terminal symbols

```
N ::     i ; i N .
A ::     a ; b ; c .

texts:   a N, b N, c N.
A i:     A symbol.
A i N:   A symbol, A N.
```

Fig. 15.5. The final VW grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

are generated by the grammar; this feature is exploited further in Section 15.2.5.

This grammar gives a clear indication of the language it describes: once the "value" of the metanotion `N` is chosen, production is straightforward. It is now trivial to extend the grammar to $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n\mathbf{d}^n$. It is also clear how long-range relations are established without having confusing messengers in the sentential form: they are established *before* they become long-range, through consistent substitution of metanotions in simple right-hand sides. The "consistent substitution rule" for metanotions is essential to the two-level mechanism; without it, VW grammars would be equivalent to CF grammars (Meersman and Rozenberg [253]).

A very good and detailed explanation of VW grammars has been written by Cleaveland and Uzgalis [252], who also show many applications. Sintzoff [241] has proved that VW grammars are as powerful as PS grammars, which also shows that adding a third level to the building cannot increase its powers. van Wijngaarden [249] has shown that the metagrammar need only be regular (although simpler grammars may be possible if it is allowed to be CF).

When the metagrammar is restricted to a finite-choice grammar, that is, each metanotion just generates a finite list of words, the generation of the CF grammar rules from the hyperrules can be performed completely, and the result is a (much larger) set of CF rules. Conversely, the use of a finite metalevel can often reduce considerably the number of rules in a grammar; we used this in the grammar of

Figure 15.5 when we condensed the **a**, **b** and **c** into a metanotion **A**. In linguistics, the attributes of words are very often finite:

```
GENDER ::  masculine ; feminine ; neuter .
NUMBER ::  singular ; plural .
MODE ::    indicative ; subjunctive ; optative .
...
```

and using them as (finite-choice) metanotions can help reduce the complexity of the grammar.

15.2.2 Expressing Semantics in a VW Grammar

Now that we have seen that the understandability of VW grammars is excellent, we will turn to the semantic suitability. Here we are in for a pleasant surprise: van Wijngaarden [257] has shown that VW grammars can produce the semantics of a program *together with* that program. In short, we do not have to leave the formalism to express the semantics.

For an almost trivial example, let us assume the semantics of a string $a^i b^i c^i$ is the string "**OK**" if $i > 5$ and "**KO**" otherwise. The grammar in Figure 15.6 then produces strings of the form

```
aaa...bbb...ccc...=>[OK|KO]
```

with the proper "**OK**" or "**KO**".

```
1.  N ::                   i ; i N .
2.  A ::                   a ; b ; c .

3.  text_S:                a N, b N, c N,
                               result symbol, semantics N.
4.  A i:                   A symbol.
5.  A i N:                 A symbol, A N.

6.  semantics iiiii N:     ok symbol.
7.  semantics N:           where N N1 equals iiiiii, ko symbol.
8.  where N equals N:      .
```

Fig. 15.6. VW grammar for $a^n b^n c^n$ => [OK|KO]

Rule number 6 in the grammar of Figure 15.6 says that if the original **N** can be split up in five **i**s (the **iiiii**) and a sequence of at least one more **i** (the **N**), then the semantics is "**OK**". Rule number 7 uses a so-called *predicate*, a rule that controls the production process by either producing nothing (success) or getting stuck (failure). The hypernotion **where N N1 equals iiiiii** succeeds only when **N1** can be chosen so that **N N1** forms **iiiiii**, that is, when there is a number **N1** larger than zero that can be added to **N** to form 6, that is, when **N** is less than 6. Any sentential form that includes a notion like **where i equals ii** is a blind alley since

no hyperrule will produce a CF rule with `where i equals ii` as its left-hand side. The methods and techniques used here belong to the two-level programming paradigm; the ALGOL 68 report [244] is full of them.

The above paragraph uses another standard feature of VW grammars, the creation of independent copies of a metanotion by appending a number to its name. The **N1** in the above hypernotion is an independent copy of the metanotion **N** and is different from the **N** that occurs in the same rule. All **N**s must be substituted consistently, and so must all **N1**s, etc., as far as applicable.

If we consider the VW grammar to be a grammar for a programming language, the above technique produces sentences consisting of programs (sequences $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ in the above example) with their semantics. We can carry the semantic expression process one step further, leave out the program at all and just produce the result from a formulation of the problem in a VW grammar. The following small VW grammar produces the result of the multiplication **N1×N2**, given the above definition of **N** (note the similarity to definitions from mathematics and functional programming):

```
produce N1 times N2 i: produce N1 times N2, write N1.
produce N1 times i: write N1.
write N i: write N, i symbol.
write i: i symbol.
```

Given the start symbol `produce iii times iiii`, this grammar produces one string: `iiiiiiiiiiii`. So rather than having a grammar that we use to produce a program that we run to obtain a result, we have a grammar that we run to obtain a result. This explains the title *Languageless programming* of the paper in which van Wijngaarden [257] describes this technique. Małuszyński [261] develops the idea further. Grune [260] describes a sentence producing program for VW grammars; the above examples run correctly on this program. The first 8 lines of the output for the grammar from Figure 15.6 are given in Figure 15.7.

```
abc=>KO.
aabbcc=>KO.
aaabbbccc=>KO.
aaaabbbbcccc=>KO.
aaaaabbbbbccccc=>KO.
aaaaaabbbbbbcccccc=>OK.
aaaaaaabbbbbbbccccccc=>OK.
aaaaaaaabbbbbbbbcccccccc=>OK.
```

Fig. 15.7. The first 8 lines of output for the grammar from Figure 15.6

These examples are almost trivial, but they do show an outline of what can be done: it is a paradigm in its infancy. If VW programming ever becomes a full-fledged paradigm, we will no doubt find the style presented here as archaic as we find today machine code of the 1950s.

15.2.3 Parsing with VW Grammars

Parsing with VW grammars is an interesting subject. On the down side it cannot be done: it can be proved that there cannot be a general parser/processor for VW grammars. On the up side, with some reasonable restrictions, a lot can be done.

There are several known techniques; we mention here the CF-skeleton technique, the definite clause/Prolog technique, and LL(1) parsing, but the literature references in (Web)Section 18.2.6 contain additional descriptions. The Definite Clause/Prolog technique is the most convenient, and we will discuss it here in some depth. We will also briefly introduce the CF-skeleton technique. The LL(1) parsing technique is very interesting, very complicated and quite powerful; we refer the reader to the papers by Gerevich [258] and Fisher [263, 273].

In the CF-skeleton technique a skeleton grammar is extracted from the VW grammar by ignoring the metanotions, so the hypernotions reduce to simple CF non-terminals. An essential ingredient for this is an algorithm for finding out whether a given hypernotion, occurring in the right-hand side of a hyperrule, can ever match a given hyperrule. For example, for the above grammar the algorithm should be able to find out that the **write N1** from the hyperrule **produce N1 times i: write N1** can be expanded into something that can be matched by the hyperrule **write N i: write N, i symbol**. One says that the algorithm should solve the *cross-reference problem*. This seems easy enough for our examples, but it can be proved that no such algorithm can exist: the cross-reference problem for VW grammars is unsolvable. But, as usual, with some ingenuity one can construct an approximation algorithm, or one can impose restrictions on the grammar.

When we apply the CF-skeleton transformation to the grammar from Figure 15.5 it vanishes almost completely (which immediately shows that such techniques do not work for every VW grammar), so we turn to the less “vehemently two-level” grammar from Figure 15.4 for an example. A likely result would be the skeleton grammar (in CF notation)

```
textS:  A B C.
A: a.   A: a A.
B: b.   B: b B.
C: c.   C: c C.
```

The input is then parsed using this CF grammar and any suitable CF parsing method; a CF parse forest results in which segments of the input are identified as produced by the CF remainders of the hypernotions. Various techniques are used to extract information about the metanotions from this structure (see for example Dembiński and Małuszyński [254]). This information is then checked, used to reduce the number of trees in the parse forest, and correlated with the resulting tree(s) to yield the semantics.

The Prolog approach also uses a CF skeleton grammar and converts it to a Prolog program using the Definite Clause technique, as explained in Section 6.7. In the VW version of the Definite Clause technique, a Prolog rule is defined for each hyperrule. Its structure derives from the skeleton grammar and the names of the goals in it

correspond to the non-terminals in that grammar. The metanotions in the hyperrule are added as logic variables, in addition to the **S** and **R** resulting from the conversion to definite clauses.

This narrows down the "suitable CF parsing method" mentioned above to full backtracking top-down parsing, and the "checking and correlating" to unification of logic variables. Both features are built-in in Prolog, and are well studied and well understood.

For many VW grammars, the result is a reasonably understandable Prolog program which is a reasonably effective parser for the VW grammar. The translation of our grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ shown in Figure 15.8 is a good example. To avoid clutter

```
text(S,N,R):- a_n(S,N,R1), b_n(R1,N,R2), c_n(R2,N,R).

a_n(S,[i],R):- symbol(S,a,R).
a_n(S,[i|N],R):- symbol(S,a,R1), a_n(R1,N,R).

b_n(S,[i],R):- symbol(S,b,R).
b_n(S,[i|N],R):- symbol(S,b,R1), b_n(R1,N,R).

c_n(S,[i],R):- symbol(S,c,R).
c_n(S,[i|N],R):- symbol(S,c,R1), c_n(R1,N,R).

symbol([A|R],A,R).
```

Fig. 15.8. A recognizer for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ in Prolog

we have abbreviated **Sentence** to **S** and **Remainder** to **R**, and we have replaced clause names that start with a capital letter like **A** by a form like **a_n**, as we did in Section 6.7. Presented with the query

```
| ?- text([a,a,a,b,b,b,c,c,c], N, []).
```

the system answers

```
N = [i,i,i]
```

and to the query **text([a,a,a,b,b,c,c,c], N, [])** it answers **no**.

Surprisingly, even the VW grammar from Figure 15.5, which had a CF skeleton grammar with only one, nameless, non-terminal, leads to a Definite Clause program that works, as Figure 15.9 shows. We have named the corresponding Prolog rule **x**.

(There are the usual real-world problems with this approach; for example the rule name **c** in Figure 15.8 must be changed before running the program because **c** is a system predicate in Cprolog and cannot be redefined.)

Parsing time requirements are exponential in principle, or even infinite, but for many grammars the parser runs in linear time. As usual in all top-down parsers, left recursion causes problems; see the section on cancellation parsing (Section 6.8) for possible solutions.

```
text(S,N,R):- x(S,a,N,R1), x(R1,b,N,R2), x(R2,c,N,R).

x(S,A,[i],R):- symbol(S,A,R).
x(S,A,[i|N],R):- symbol(S,A,R1), x(R1,A,N,R).

symbol([A|R],A,R).
```

Fig. 15.9. An even shorter recognizer for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ in Prolog

Edupuganty and Bryant [262] implement a similar parser that is independent of Prolog or DCGs.

15.2.4 Error Handling in VW Grammars

Error handling is a problem. As with any backtracking system, when a Definite Clause parser cannot find what it is looking for, it backtracks, and when whatever it is looking for is not there, it backtracks all the way: it returns to the initial position and proudly announces "No!"; see Section 7.1 and especially Figure 7.5. This is not very helpful in debugging the grammar or in finding an error in the input. It is not even possible to add a logic variable to obtain the longest prefix of **Sentence** that the system has managed to match: the successive failures will uninstantiate this variable again and again, and it will not be set when the parser fails. In the end the trace facility of the Prolog system will have to come to the rescue, an often effective but always unelegant solution.

15.2.5 Infinite Symbol Sets

In a sense, VW grammars are even more powerful than PS grammars: since the name of a symbol can be generated by the grammar, they can easily handle infinite symbol sets. Of course this just shifts the problem: there must be a (finite) mapping from symbol names to symbols somewhere. The VW grammar of Figure 15.10 generates sentences consisting of arbitrary numbers of equal-length stretches of equal symbols, for example, $s_1s_1s_1s_2s_2s_2$ or $s_1s_1s_2s_2s_3s_3s_4s_4s_5s_5$, where s_n is the representation of the $\mathbf{i}^n$ **symbol**. See Grune [274] for more details.

```
N ::        n N; ε.
C ::        i; i C.

textS:      N i tail.
N C tail:   ε; N C, N C i tail.
N n C :     C symbol, N C.
C :         ε.
```

Fig. 15.10. A grammar handling an infinite alphabet

15.3 Attribute and Affix Grammars

The desire to bridge the gap between VW grammars — very powerful but next to unparsable — and CF grammars — annoyingly lacking in power but quite parsable — gave rise to two developments, one down from VW grammars and one up from CF grammars. The first yielded affix grammars (Koster and Meertens [237]), the second attribute grammars (Knuth [243]). They meet in the middle in the EAGs, (Watt and Madsen [259]) which may stand for Extended Attribute Grammars or Extended Affix Grammars.

Although a comparison of attribute and affix grammars finds far more similarities than differences (Koster [269]), their provenance, history and realm of application differ so much that it is useful to treat them in separate sections.

15.3.1 Attribute Grammars

An *attribute grammar* is a context-free grammar extended with two features. First, the non-terminals, and usually also the terminals, in the grammar have values from some programming language P attached to them; these values are called "attributes". And second, each grammar rule has attached to it a piece of code in the language P, its *attribute evaluation rule*, for establishing these values.

15.3.1.1 Attributes and Evaluation Rules

All rules for a non-terminal N specify the same set of attributes; each grammar rule R for N specifies its own evaluation rule. We will see later on why this has to be so. The attributes and their evaluation rules have to be supplied by the grammar writer. There are two forms of evaluation rules: functions and checks. An evaluation rule for a rule R for a non-terminal N can express some attribute A of N or of one of its children in R as functions of some other attributes, thus expressing a part of the semantics of the rule R:

$$A := func_{A,R}(A_p, \ldots, A_q);$$

where $func_{A,R}$ is the function in rule R responsible for the evaluation of attribute A. The evaluation code can also check some attributes of N in order to impose context-sensitive restrictions:

$$check_R(A_p, \ldots, A_q);$$

In some systems, the evaluation rules can also steer the parsing; their use is then similar to that of the conflict resolvers from Section 8.2.5.3.

We see that again two levels are involved, a CF level to express the syntax and an attribute evaluation rule level to express the context dependencies; as in VW grammars, the second level can at the same time express the semantics. We will now look in more detail at the use of the attributes and their evaluation rules.

Although the grammar rules for a non-terminal N are written only once in the grammar, many nodes for N, possibly stemming from different rules for N, can be

present in the parse tree. Each of these nodes has room for the same set of attributes; their values can, and usually will, differ from node to node. These values can be anything the programming language allows: numbers, strings, etc. For a given node for *N*, the attribute evaluation rules compute the values of some attributes of *N* and of some attributes of *N*'s children, using other attribute values of *N* and its children. So the rules provide local communication between the attributes of *N* and those of its children; and since *N* has a parent unless it is the root, and *N*'s children have again children unless they are terminal symbols, this scheme allows the computation of relationships all throughout the parse tree, between the terminal symbols and the root. As we have seen, these relationships can be used to implement context-sensitive conditions, and to compose semantics.

If the attribute evaluation rules of a grammar rule *R* for *N* compute an attribute of the left-hand side of *R*, that attribute is a *synthesized attribute* of *N*; all evaluation rules for *N* must have functions for the computation of all synthesized attributes of *N*. (Synthesized attributes are also known as *derived attributes*.) If the attribute evaluation rules compute an attribute of one of the non-terminals *A* in the right-hand side of *R*, then that attribute is an *inherited attribute* of *A*, and all grammar rules that have *A* in their right-hand side must have functions for the computation of all inherited attributes of *A*. Terminal symbols have their own built-in synthesized attributes; for example, a token **3** could have a string-valued synthesized attribute with value **"3"** or an integer-valued synthesized attribute with value 3 or 51 (its ASCII code). This way there is a function for the computation of each attribute in each node in the parse tree. The synthesized attributes of the root can be viewed as the semantics of the entire input string.

Figure 15.11 shows an attribute grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$, where the attribute evaluation rules are given in curly brackets, and **syn** and **inh** indicate the modes of the attributes. The notation used is ad hoc, but sufficient for our purposes; existing sys-

```
textS(syn int n):  A(na) B(nb) C(nc)  {nb:=na; nc:=na; n:=na}.

A(syn int n):      'a'                {n:=1;}.
A(syn int n):      'a' A(na)          {n:=na+1;}.

B(inh int n):      'b'                {check(n==1);}.
B(inh int n):      'b' B(nb)          {check(n>1); nb:=n-1;}.

C(inh int n):      'c'                {check(n==1);}.
C(inh int n):      'c' C(nc)          {check(n>1); nc:=n-1;}.
```

Fig. 15.11. Attribute grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

tems have much more elaborate notations. Rather than letting the rules for **A**, **B**, and **C** synthesize the number of letters they collect and then check the equality of these numbers at the top level, we let **A** do the synthesizing, pass the resulting value to **B**

and `C` as inherited attributes, and let these two check when they recognize their last token. This has the advantage of giving an error message on the first possible token.

15.3.1.2 Parsing and Attribute Evaluation

Parsing can be done using any suitable CF method; for the above attribute grammar LL(1) could be used. There are three ways to evaluate the attributes: bottom-up (data-driven), top-down (demand-driven), and very clever ("ordered"). We will briefly discuss the first two methods, using the input string **`aabbcc`**. The CF parse tree is given in Figure 15.12.

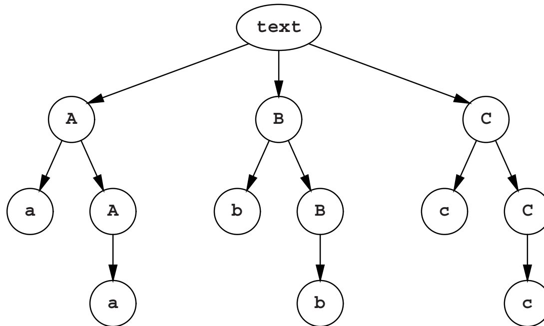


Fig. 15.12. The not yet attributed CF parse tree for **`aabbcc`**

In the bottom-up method, only the attributes for the leaves are known initially, but as soon as sufficient attributes in a right-hand side of a grammar rule are known, we can use its attribute evaluation rule to compute an attribute of its left-hand side or of a child. Initially only the attribute **`n`** of the node **`A(n):'a'`** is known. This allows us to compute the **`n`** in **`A(n):'a'A(na)`**, which in turn gives us the attributes of **`n`**, **`na`** and **`nb`** of **`text(n):A(na)B(nb)C(nc)`**. Since the **`nb`** is the inherited attribute **`n`** of **`B(n):'b'B(nb)`**, the next bottom-up scan will be able to compute the attribute **`nb`** of that rule. This way the attribute values (semantics) spread over the tree, finally reach the start symbol and provide us with the semantics of the whole sentence. If there are inherited attributes, repeated scans will be needed, so this method is primarily indicated for attribute grammars with synthesized attributes only.

In the top-down method, we demand to know the value of the attribute **`n`** of the root **`text`**. Since this is computed by **`n:=na`** and we do not know **`na`** yet, we have to descend into the tree for **`A`**, meeting assignments of the form **`n:=na+1`** and postponing them, until we finally reach the assignment **`n:=1`**. We can then do all the postponed assignments, and finally come up with the value of **`n`** in **`text`**. So, as far as semantics is concerned, we are finished now, but if we want to do checking, we have to evaluate (top-down) the arguments of the **`check`**s. (The large discrepancy between checking and semantics here is an artifact of the highly redundant input.)

These two methods are characterized by a fixed and grammar-independent evaluation order of the attribute code. For ordered attribute grammars (Kastens [255]) an optimal evaluation order can be derived from the grammar, which allows very efficient attribute evaluation. Constructing such evaluation orders is outside the scope of this book; see, for example, Grune et al. [414, Ch. 3].

The fully attributed tree is shown in Figure 15.13. The arrows show the directions in which the information has flowed; the checks are not shown.

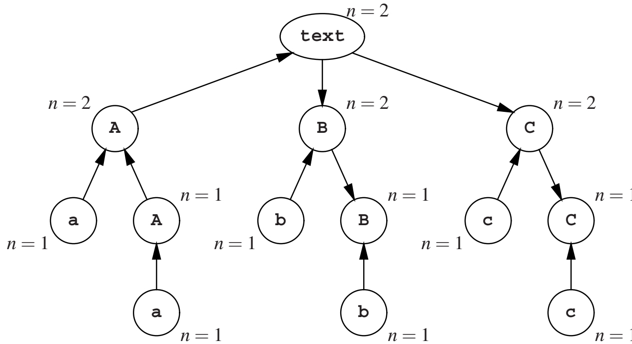


Fig. 15.13. The fully attributed parse tree for input **`aabbcc`**

The error handling in attribute grammars can be divided in two parts, the syntactic error handling, which comes with the CF parsing method, and the context-sensitive error handling. The latter is implemented in the checks in the attribute evaluation rules attached to the rules, and is therefore fully the responsibility of the programmer. This division of responsibilities, between automation and user intervention, is characteristic of attribute grammars.

We have seen that attribute grammars are a successful mix of CF parsing, a programming language, context-dependencies, establishing semantics, and conflict resolvers. What *can* be automated *is* automated (CF parsing and the order in which the attribute evaluation rules are performed), and for the rest the grammar writer has almost the full power of a programming language.

Attribute grammars constitute a very powerful method of handling the CF aspect, the context-sensitive aspect and the semantics of a language, and are at present probably the most convenient means to do so. It is doubtful, however, that they are generative grammars, since it is next to impossible to use them as a sentence *production* mechanism; they should rather be classified as recognition systems.

15.3.2 Affix Grammars

Like attribute grammars, *affix grammars* (Koster [245, 269]) endow the non-terminals with parameters, called *affixes* here; and like attributes these are divided into synthesized, called "derived" here, and inherited. But where attribute grammars

have evaluation rules which express checks and semantics and which have no direct grammatical significance, affix grammars have *predicates*, which are a special kind of non-terminals which can only produce {ε}, the set containing only the empty string, or the empty set { }, which contains nothing. This allows affix grammars to be used as generative grammars.

15.3.2.1 Producing with an Affix Grammar

During the production process, a non-terminal can be replaced by any of the strings that are in the set of its terminal productions, and the same applies to predicates. If a predicate P has {ε} as the set of its terminal productions, that set contains only one item, ε, so P gets replaced by it, disappears, and the production process continues. But if P has { } as the set of its terminal productions, P cannot be replaced by anything, and the production process halts. Note that this hinges on the difference between the empty set and a set containing one element, the empty string.

Figure 15.14 shows how this mechanism controls the production process. The

```
start(δ int n):  A(n) B(n) C(n).

A(δ int n):      n = 1, 'a'.
A(δ int n):      'a', A(na), n = na+1.

B(ι int n):      n = 1, 'b'.
B(ι int n):      n > 1, 'b', nb = n-1, B(nb).

C(ι int n):      n = 1, 'c'.
C(ι int n):      n > 1, 'c', nc = n-1, C(nc).
```

Fig. 15.14. Affix grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

forms **n=1**, **nb=n-1**, etc. are predicates; they produce ε when the affixes obey the test, and the empty set otherwise. In this figure δ indicates a derived affix and ι an inherited one, but the difference plays no role during production. To produce **aaabbbccc** we begin with **start(3)**. This leads to **A(3)**, but the first rule for **A** fails since the predicate **n=1** produces nothing. The second rule produces **a** followed by **na a**s, but production can only continue if we choose **na** to be 2, to pass the predicate **n=na+1**. Related considerations apply to the production of the segments **bbb** and **ccc**.

The predicates **n>1** are not really necessary: entering the rule for **B** with **n** equal to 1 would cause **nb** to be forced to 0, and attempts to produce from **B(0)** would lead to an infinite dead end. We have blocked these infinite dead ends for esthetic reasons.

15.3.2.2 Parsing with an Affix Grammar

Although the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ is symmetrical in **a**, **b**, and **c**, the grammar in Figure 15.14 is not. The reason is that it is a *well-formed affix grammar*, which means that the form of the predicates, their positions, and the δ and ι information together allow the affixes to be evaluated during parsing. The precise requirements for well-formedness are given by Koster [245], but effectively they are the same as for attribute evaluation in attribute grammars.

All the usual general parsing techniques are possible, and they handle a failing predicate just as bumping into an unexpected token. A simple top-down parser with **aaabbbccc** as input would first try the first rule for **A**, derive **n** equal to 1 from it, continue with **B(1)**, find there is no **b**, backtrack, try the second rule for **A**, try **na** equal to 1, fail, backtrack, etc., until all 3 **a** are consumed. The derived value of **n** of the top-level **A** is then 3, which is passed as inherited affix to **B** and **C**. The rest is straightforward.

Watt and Madsen [259] have extended the affix grammars with a transduction mechanism, resulting in the EAGs. These can be viewed as attribute grammars with generative power, or as affix grammars with semantics.

15.3.2.3 Affix Grammars over a Finite Lattice

Affixes can be of any type, but an especially useful kind of affix grammar, the *AGFL*, is created by restricting the affixes to finite lattices (Koster [268]). A lattice is a set of values that can be compared for rank; we shall use $\succ$ as the ranking operator. If x and y are compared for rank, the answer may be “smaller”, “larger”, or “not ordered”; it may also be “equal” but then x and y are the same value. An important condition is that there cannot be a value x for which we have $x \succ \cdots \succ x$. A lattice corresponds to a directed acyclic graph, a “dag”. A formal definition of AGFLs is given by Nederhof and Sarbo [349].

Such lattices are very useful for encoding attributes of linguistic forms, like gender, tense, active/passive, etc. A simple example is gender in, for example, German, Russian, and several other languages:

```
GENDER ::  masculine; feminine; neuter.
```

Here **GENDER** $\succ$ **masculine**, **GENDER** $\succ$ **feminine**, and **GENDER** $\succ$ **neuter**. But the usefulness of lattices is demonstrated better in a language that has a more complicated gender structure. One such language is Burushaski, which has four genders (Figure 15.15), **hm** (human masculine), **hf** (human feminine), **x** (countable non-human) and **y** (non-countable); examples of non-countable are “salt”, “love”, etc. This structure is shown in the first group of affix rules for **GENDER**; **hm**, **hf**, **x**, and **y** are affix terminals; the upper case words are affix non-terminals; and the **::** identifies the rules as affix rules.

But some parts of the Burushaski grammar treat feminine nouns quite differently from the rest, so a division in feminine and non-feminine is also appropriate; see the second group of affix rules. The dag for this affix type is shown in Figure 15.16

```
GENDER ::        COUNTABLE; UNCOUNTABLE.
COUNTABLE ::     HUMAN ; x.
HUMAN ::         hm ; hf.
UNCOUNTABLE ::   y.

GENDER ::        FEMININE; NONFEMININE.
FEMININE ::      hf.
NONFEMININE ::   hm; x; y.
```

Fig. 15.15. Finite Lattice affix rules for gender in the Burushaski language

in traditional lattice representation: if v_1 is higher than v_2 and connected by lines, then $v_1 \succ v_2$; the mathematical definition of lattices requires a top element ($\top$) and a bottom element ($\bot$). It will be clear why lattices are called lattices.

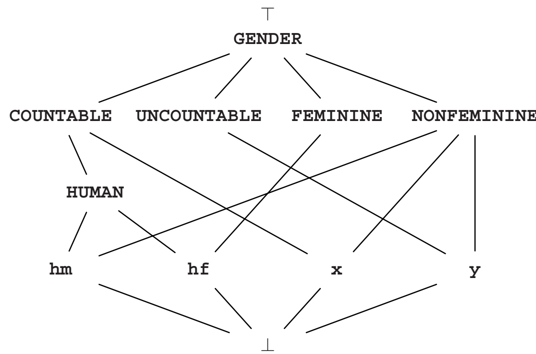


Fig. 15.16. Lattice of the FL affix notion **GENDER** of Figure 15.15

The following (very naive) grammar fragment shows the use of the affixes:

```
noun phrase (GENDER) :
      [article (GENDER)], noun (GENDER), infix (GENDER).
...
noun (HUMAN) : family member (HUMAN).
...
infix (hf) : "mu" .
infix (NONFEMININE) : .
...
```

The first rule says that gender in a noun phrase distributes over an optional article, the noun, and the infix. As in VW grammars, the affix variable must be substituted consistently. If the actual gender is in the subset **HUMAN**, the noun may be a family member of the same gender. If the gender is **hf**, the infix must be **mu**, but if it is in the subset **NONFEMININE** it must be left out.

Since a lattice does not contain cycles, a grammar which is a lattice can produce only a finite number of terminal productions: it is a Type 4 grammar (page 33). This

means that one could substitute out all affixes in an AGFL and obtain a context-free grammar, but such a grammar would be much larger and less convenient. So an AGFL is much weaker than an general affix grammar, but its strength lies in its ease of use for linguistic purposes and in its compactness.

AGFLs can be parsed by any CF method. The problem lies in managing the affixes: naive methods cause an explosive growth of the data structures. An efficient solution is given by Nederhof and Sarbo [349].

15.4 Tree-Adjoining Grammars

As languages go, English is a rather context-free language in that it exhibits few long-range relationships. Verbs do correlate with the subject ("I walk" versus "he walks"), but the subject is seldom far away. It has composite verbs (for example "to throw away"), but the two parts usually stay close together: "She finally threw the old towels, which she had inherited from her grandmother and which over the years had assumed paler and paler shades of gray, away" is awkward English. "The brush I painted the garage door green with .." (where "with" relates back to "brush") is better, but many speakers would prefer "The brush with which . . . ".

This is not true for many other languages. Verbs may agree with subjects, direct and indirect objects simultaneously in complicated ways (as in Georgian and many American Indian languages), and even languages closely related to English have composite verbs and other composite word classes, the parts of which can and often must move arbitrarily far away from each other. Examples are Dutch and Swiss German, and to a lesser degree Standard German. Especially the first two exhibit so-called *cross-dependencies*, situations in which the lines that connect parts of the same unit cross. Such relationships do not correspond to a tree, and CF grammars cannot produce them. Cross-dependencies have long worried linguists, and if linguistics want to explain such languages in a generative way, an alternative system is needed. *Tree Adjoining Grammars* (or *TAGs*) is one such system; it was developed by Joshi [250]. Yngve [375] describes a simpler and much earlier system.

15.4.1 Cross-Dependencies

As an example of a cross-dependency we will take the Dutch sentence

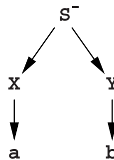
```
Daar doe ik niet aan mee.
```

This is a completely normal everyday Dutch sentence, but its structure is complex. It contains the words `ik` = "I", `niet` = "not", `meedoen` = "participate", and `daaraan` = "to that", and it means "I do not participate in that", or more idiomatically "I will have no part in that". (Since English has no cross-dependencies, Dutch sentences that use it have no literal translation in English.) We see that the words `daaraan` and `meedoen` have split up, and the dependencies connecting the parts cross:

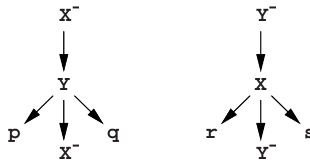
It is clear that there cannot be a CF grammar which produces this sentence so that **daar** and **aan** derive from one non-terminal and **doe** and **mee** derive from another. (We will not go into the fact that there is more than one kind of cross-dependency, nor into their linguistic relevancy here.)

TAGs were designed to solve this problem. We shall first explain the principles and then see how to create a TAG that will produce the above sentence in a satisfactory way. Where a CF grammar has rules, a TAG has trees. A CF rule names a non-terminal and expresses it in terminals and other non-terminals, and a TAG tree names a node and expresses it in terminals and other nodes. The top node of a tree is labeled with the name of the tree, and all the internal nodes are labeled with non-terminals; the leaves are usually terminals but can occasionally be labeled with a non-terminal. If a tree has terminal leaves only, it is a *terminal tree*, and it represents a string in the language generated by the TAG; that string is the sequence of terminals spelled by the leaves. An example of such a terminal tree will be shown in Figure 15.19.

Whereas a CF grammar has one start symbol, a TAG has several start trees called *elementary trees*. They usually represent the basic sentence types of a language: active sentences, passive sentences, questions, etc., which is why they are also called *sentential trees*. All the leaves of an elementary tree are terminals, so an elementary tree might look like this:

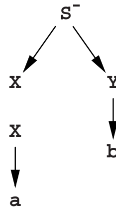


and it represents the string **ab**. (The meaning of the **-** marker next to the **S** will be explained below.) The other trees in the grammar (called *auxiliary trees*) have the same structure as elementary trees, except that one leaf is not a terminal but is labeled with the name of the tree:

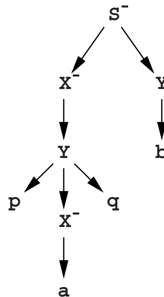


This leaf is called the “foot” of the tree. Note that there is exactly one path from the top of the tree to its foot.

Auxiliary trees do not represent strings in the language, but serve to expand nodes in elementary trees in a process called *adjoining*, as follows. Suppose we want to expand the node **X** in the elementary tree **S** by using the tree for **X**. We first cut the node **X** in **S** in two to detach its subtree from **S**:



Next we put the tree for **X** in between and connect the top to the open **X** node in **S** and the foot to the detached subtree:



Since tops and feet are labeled identically, this does not disrupt the structure of the tree. The result represents the string **paqb**. Since the **.a.b** come from one tree and the **p.q.** come from another, we have already created our first cross-dependency! And we can also see how it works: the tree **X** "spreads its wings" to the left and the right of the **a**, thus creating a (moderately) long-range correlation between its left wing and its right wing.

Some nodes in the above pictures are marked with a **-** marker; this marker indicates that the marked node cannot be expanded. We shall also meet nodes marked with a **+**; such nodes *must* be expanded for a tree to count as a terminal tree. Other varieties of TAGs may define other types of markers with their requirements.

We now turn to the sample sentence "**daar doe ik niet aan mee**", and start by creating a tree for **daar ... aan**. Sentences that do not start with the subject have the form of an "inverted sentence", one in which the verb precedes the subject; we will make a tree called **INV_S** for these. So the sentence consists of four pieces:

$$\texttt{S = daar INV_S}_l\texttt{ aan INV_S}_r$$

where **INV_S**$_l$ is the left wing of **INV_S** and **INV_S**$_r$ is its right wing. What is missing to turn this into an auxiliary tree is the position of the foot **S**. To determine that position, we need information about the language. In Dutch, no words can be inserted between **aan** and **INV_S**$_r$, but **INV_S**$_l$ and **aan** can be separated by words. So that gives us the proper place for the foot:

S = daar INV_S$_l$ **S**$_{foot}$ **aan INV_S**$_r$

This leads to the middle tree in Figure 15.17, where the **-** markers show that after adjoining the two nodes **S** cannot be adjoined again, and the **+** shows that **INV_S** must be adjoined to obtain a sentence. Figure 15.17 also shows the elementary tree **S** to start the process and the result of adjoining the tree for **S** to it.

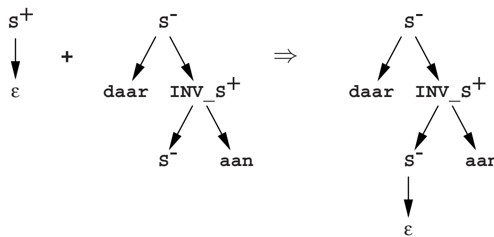


Fig. 15.17. Elementary tree, tree for **S** and result of adjoining

In a real grammar there will be many trees for **INV_S**; we construct one here for a negative inverted sentence. It contains an inverted verb part **INV_VP**. So the remainder of the sentence

INV_S = doe ik niet INV_S$_{foot}$ **mee**

corresponds to

INV_S = INV_VP$_l$ **niet INV_S**$_{foot}$ **INV_VP**$_r$

The tree is shown in Figure 15.18(*a*). This leaves

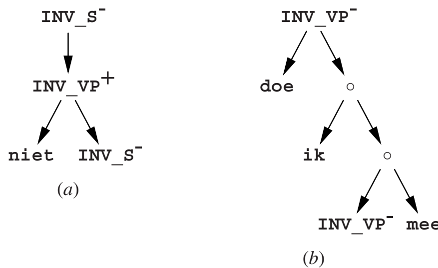


Fig. 15.18. Trees for **INV_S** and **INV_VP**

doe ik INV_VP$_{\text{foot}}$ **mee**

for **INV_VP**. The three words in it are strongly interdependent, so for simplicity we will accept this segment as a single tree. The tree is shown in Figure 15.18(*b*) and shows two anonymous nodes; if a particular formalism does not allow this, dummy names could be assigned to them. Figure 15.19 shows the complete tree; we see that it contains no nodes marked with a **+**, and that its outer rim spells

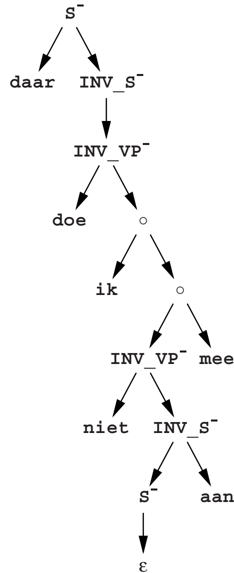


Fig. 15.19. The complete tree for the Dutch sentence with cross-dependency

"**daar doe ik niet aan mee**".

In a real-world system the tree for **doe ik ... mee** would be split up into one for a subject **ik** and one for a conjugated verb **doe ... mee**, but since the verb agrees with the subject (**doe ik ... mee** versus **doet hij ... mee**, etc.) another type of correlation must be established. In many systems, the trees and the nodes in them are provided with attributes, often called *features* in linguistics, that are required to agree for adjoining to be allowed. Since these attributes have only a finite (and usually very small) number of values, all the trees with all their values could in principle be written out, and the attributes are a means to complexity reduction only, just as a finite-choice metagrammar is to a VW grammar.

The TAG design process shows clearly how each detail of Dutch syntax is expressed in a particular auxiliary tree. There is a large TAG for the English language available on the Internet from the University of Pennsylvania.

TAGs are (somewhat) stronger than CF grammars, since they can produce the language $a^nb^nc^n$, which is not CF, but they are (much) weaker than CS grammars.

They cannot, for example, produce the language $a^n b^n c^n d^n e^n$. Remarkably, TAGs can create only 4 copies of the same number.

The semantics of a Tree Adjoining Grammar is attached to the trees, and composition of the semantics is straightforward from the adjoining process.

15.4.2 Parsing with TAGs

Input described by a TAG can be parsed in polynomial time, using a bottom-up algorithm described by Vijay-Shankar and Joshi [264]. The algorithm, which is very similar to CYK (Section 4.2), is not difficult, but since TAGs are more complicated than CF grammars there are many more details, and we will just sketch the algorithm here. We assume that no node in any elementary or auxiliary tree has more than two children; it is easy to get a TAG into this "2-form" by splitting nodes where necessary.

Rather than having a two-dimensional recognition table $R_{i,l}$ the entries of which contain non-terminals that produce $t_{i..i+l-1}$ where $t_{1..n}$ is the input string, we have a four-dimensional recognition table $A_{i,j,k,l}$ the entries of which contain tree nodes X in trees for Y that produce the segments $t_{i..j}$ and $t_{k..l}$, where the gap $t_{j+1,k-1}$ between them is to be produced by the tree hanging from the foot node of Y. Note that in the description of CYK we used starting position and length to describe a segment; here it is more convenient to use the start and end positions of both segments.

The meaning of the entry $A_{i,j,k,l}$ is shown in Figure 15.20; the drawn lines de-

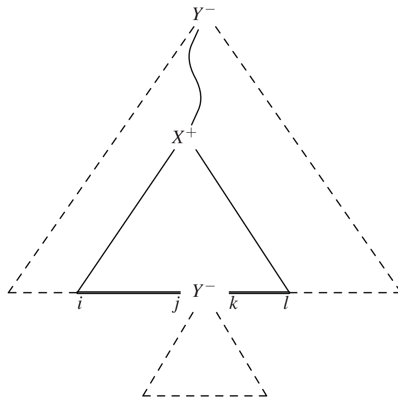


Fig. 15.20. Recognized footed tree for the node X in a tree for Y

marcate the recognized region, the broken lines show the region that must still be recognized to fully recognize a node of type Y. Note that it is the lower part of rule Y that has been recognized, in accordance with the bottom-up nature of the CYK algorithm. It is a fully recognized subtree, except that one of its leaves is the foot of Y; we shall call such subtrees "footed trees". The whole input string $t_{1..n}$ is recognized when at least one of the entries $A_{1,j-1,j,n}$ contains the root of an elementary tree, for

any value of j. Once the table has been filled in, we can find parse trees by working from the contents of the $A_{1,j-1,j,n}$ downwards, in a way similar to that of Section 4.2.5.

The actual contents of the entries in the table A are paths of the form $Y \cdots X$, where Y is the top of a tree from the grammar, and X is the node carrying the footed tree. Such a path can, for example, be implemented by giving the start node Y and a string of left-right-left directives that tell how to reach the right X from Y. Including Y is necessary to allow us to adjoin a completed tree to the foot of the part recognized by X, and including the explicit path is necessary because Y may contain more than one occurrence of X.

The essential step in the CYK algorithm is the combination of two recognized regions into a third, larger region. Doing these steps in the right order allows one to fill the table A in one sweep, visiting and filling the entries by combining the contents of entries that have already been filled. The table has n^4 entries, so filing it costs n^4 times the cost of the actions for one entry.

In TAGs there are two fundamental combination steps, one side-ways and one upwards. There are several other combination steps, but they are the mirror images of the fundamental ones, are special cases for elementary trees, or are trivial.

The most characteristic combination step for TAGs is the upwards combination, shown in Figure 15.21. It extends a footed tree for the path $Y \cdots X$ with a completely

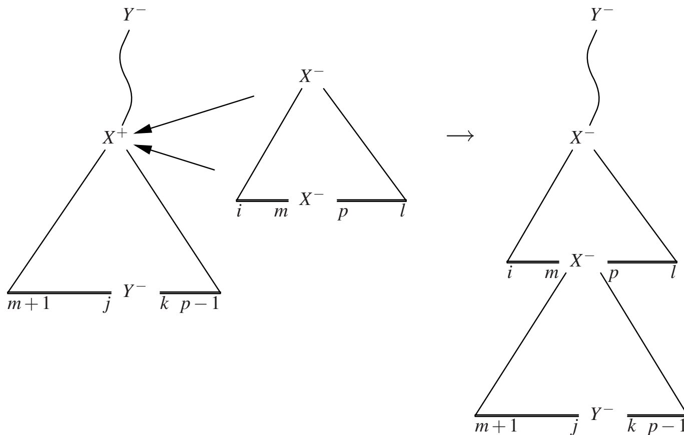


Fig. 15.21. The upwards-combination step for Tree Adjoining Grammars

recognized auxiliary tree for X, to form a new, larger, footed tree for the path $Y \cdots X$. The completely recognized tree for X covers the sections $t_{i..m}$ and $t_{p..l}$; we can find out that a tree is completely recognized from its entry in the table A, which has the form $X \cdots X$, that is, the length of the path is 0. The footed tree for $Y \cdots X$ must then start at position $m+1$ and end at $p-1$. Suppose its foot spans a gap from j to k; then the combined footed tree, still identified by $Y \cdots X$, can be inserted in $A_{i,j,k,l}$. So, to fill the entry $A_{i,j,k,l}$ with all results of the upwards combination step, we must search

for values for m and p such that the conditions of Figure 15.21 are fulfilled. There can be at most $O(n)$ such values for each of the two variables, so this step costs at most $O(n^2)$ actions per entry.

The sideways combination increases the size of a not completely recognized tree, as shown in Figure 15.22. It concentrates on those nodes Z in trees Y that have one already recognized child which a Y-footed tree $Y \cdots X$, and a second recognized child $Y \cdots W$, whose leftmost recognized token is next to the rightmost token of $Y \cdots X$; this happens at the $p, p+1$ junction in Figure 15.22. These two children can then combine

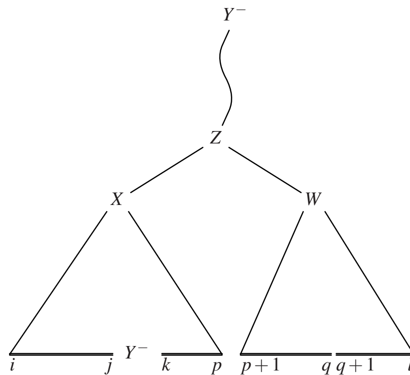


Fig. 15.22. The sideways-combination step for Tree Adjoining Grammars

sideways into a footed tree for $Y \cdots Z$. Of course $Y \cdots W$ cannot be a footed tree, since the auxiliary tree Y has only one foot. So, to fill the entry $A_{i,j,k,l}$ with all results of the sideways combination, we must search for values for p and q such that the conditions of Figure 15.22 are fulfilled. In principle there can again be at most $O(n)$ such values, so this step could cost $O(n^2)$ per entry. The value of q, however, is fairly arbitrary and can be taken out of the game by keeping a separate two-dimensional table $B_{i,l}$, containing copies of the nodes in $A_{i,p,p+1,l}$ for all values of p. All suitable nodes $Y \cdots W$ can now be found in $O(n)$ actions. The table also helps in finding top-level recognitions of the form E in $A_{1,p,p+1,n}$, where E is the root of an elementary tree. Although this will speed up our algorithm, in does not help for the overall complexity since the upwards combination step already takes $O(n^2)$.

We still have to answer the question where the initial values in the table A come from. They come from two sources, terminal symbols and empty footed trees. Each terminal symbol t directly attached to a node X in a tree for Y gives rise to an entry $Y \cdots X$ in $A_{i,i,i+1,i}$ for each position i in the input where t is found; that is, the token is absorbed in the left wing of the footed tree and the right wing is empty. And empty footed trees $Y \cdots Y$ for all auxiliary rules Y are entered in all $A_{i,i-1,j,j-1}$ for all $1 \leq i \leq j \leq n$; that is, footed trees for foot nodes with empty wings are recognized everywhere. At first they will grow sideways, using the finished subtrees originating from the terminal symbols.

We have now seen a general CYK-like parsing algorithm for TAGs in "2-form"; its time complexity is $O(n^6)$, because it has to fill in n^4 entries at a cost of at most $O(n^2)$ each. Satta [276] proves that if we can do general TAG parsing in $O(n^p)$, we can do Boolean matrix multiplication in $O(n^{2+p/6})$; note that for $p = 6$ this amounts to the standard complexities for both processes. Since Boolean matrix multiplication in time less than $O(n^3)$ is very difficult, it is probable that general tree parsing in time less than $O(n^6)$ is also very difficult.

Very little is known about error recovery for TAGs. Perhaps the techniques available for the CYK algorithm could be adapted.

CYK parsing is not the only possibility for TAGs. Schabes and Joshi [271] describe an Earley parser for TAGs, and Nederhof [281] and Prolo [282] explore LR parsers for TAGs, with their problems.

15.5 Coupled Grammars

Coupled grammars establish long-range relations by creating all parties to the relation simultaneously and keeping track of them as they go their separate ways. The non-terminals in a coupled grammar consist of fragments called *components*. During the production process, all the components of a non-terminal N must be replaced at the same time, using the same alternative for N. Suppose we have the sentential form

$$\ldots\ N_1\ \ldots\ N_2\ \ldots\ N_3\ \ldots$$

where N_1, N_2, and N_3 are components that were created simultaneously, and the following grammar rule for N:

$$N_1,\ N_2,\ N_3 \rightarrow a\ P_1\ b,\ c\ d,\ e\ P_2\ f\ \mid\ Q_1\ R_1\ R_2,\ R_3,\ Q_2\ R_4\ Q_3$$

where P consists of 2, Q of 3, and R of 4 components. Then two new sentential forms result from the simultaneous substitution process:

$$\ldots\ a\ P_1\ b\ \ldots\ c\ d\ \ldots\ e\ P_2\ f\ \ldots$$
$$\ldots\ Q_1\ R_1\ R_2\ \ldots\ R_3\ \ldots\ Q_2\ R_4\ Q_3\ \ldots$$

Th requirement that "N_1, N_2, and N_3 were created simultaneously" is essential, since other occurrences of N_1, N_2, and N_3 may be present in the sentential form, unrelated to the ones shown, and of course they are completely free in *their* substitution (as long as they obey their own consistent substitution) restriction. A better representation of the original sentential form would therefore be

$$\ldots \texttt{N[1]} \frown \ldots \texttt{N[2]} \frown \ldots \texttt{N[3]} \ldots$$

and this is indeed how such a form must be implemented in a program.

Coupled grammars can easily express non-CF languages like $a^n b^n c^n$:

$$S_s \rightarrow A_1\ A_2\ A_3$$
$$A_1,\ A_2,\ A_3 \rightarrow a\ A_1,\ b\ A_2,\ c\ A_3\ \mid\ \varepsilon,\ \varepsilon,\ \varepsilon$$

and a coupled grammar for the language *ww*, where *w* is an arbitrary string of **a**s and **b**s, is trivial. They can also provide finite-choice abbreviations in other grammars, for example:

$$
\begin{array}{lll}
\mathtt{COND_1,\ COND_2,\ COND_3,\ COND_4} & \rightarrow & \mathtt{if,\ then,\ else,\ fi} \\
 & | & \mathtt{(\ ,\ |\ ,\ |\ ,\ )} \\
 & | & \mathtt{si,\ alors,\ sinon,\ fsi}
\end{array}
$$

which allows the rule

$$
\begin{array}{lll}
\mathtt{conditional\ statement} & \rightarrow & \mathtt{COND_1\ Boolean\ expression} \\
 & & \mathtt{COND_2\ statement\ sequence} \\
 & & \mathtt{COND_3\ statement\ sequence\ COND_4}
\end{array}
$$

to produce various forms of the conditional statement in a programming language (but only the consistent ones!).

The power of coupled grammars depends on the number of components that one allows. It can be proved that for large numbers of components the power of coupled grammars approaches that of the CS grammars. There is little reason to put a hard limit on the number of components, so the attainable power is just under that of CS grammars.

15.5.1 Parsing with Coupled Grammars

Full backtracking top-down parsing works almost without modification for non-left-recursive coupled grammars. A sample parsing for the string **aabbcc** using the above grammar for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ is shown in Figure 15.23 in a format similar to that of Figure 6.11. Note that the subscripts in the recognized (left) part of each snap-

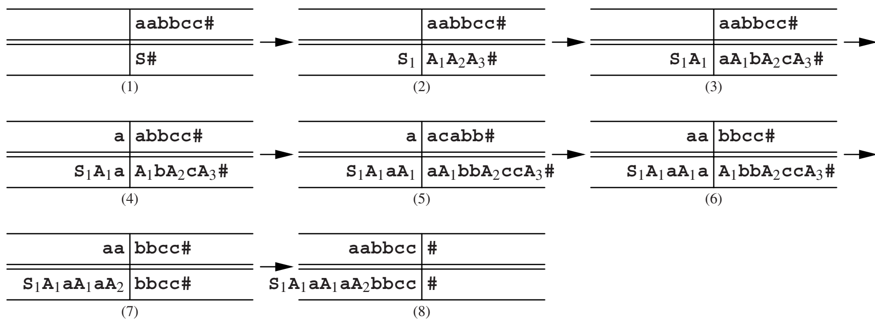


Fig. 15.23. Coupled-grammar parsing for the string **aabbcc**

shot represent the numbers of the chosen alternatives as they did in Figure 6.11, and those in the unrecognized (right) part represent numbers of components, as they do in a coupled grammar. So $\mathtt{A_2}$ on the left in the last snapshot identifies the rule $\mathtt{A_1,A_2,A_3} \rightarrow \varepsilon,\varepsilon,\varepsilon$ and $\mathtt{A_2}$ on the right in other snapshots identifies the second component of **A**. In the present case the backtracking is not activated (except for finding

out that there is no alternative parsing), but see Problem 15.10 for a parsing problem that requires backtracking.

The resulting parse tree, shown in Figure 15.24, is exactly the way one wants it, and semantics is attached easily.

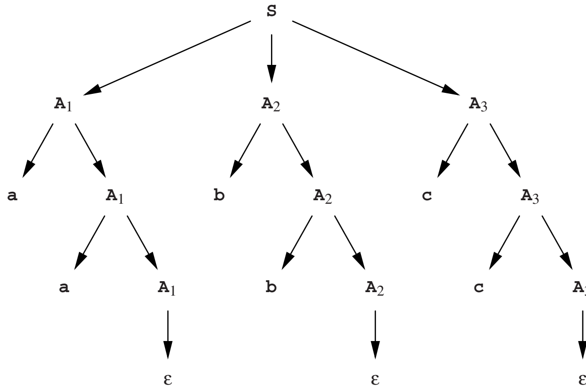


Fig. 15.24. The parse tree for the parsing of Figure 15.23

Pitsch [275, 277] gives algorithms for LL(1) and LR(1) parsing of coupled grammars, and Seki presents a general bottom-up parser based on CYK.

Coupled grammars were invented by Hotz in 1967 (Hotz [240, 278]) but received little publicity until the mid 1990s. They seem to be a convenient means of expressing mild context sensitivity, their main problem today being the lack of experience in their use.

15.6 Ordered Grammars

Ordered grammars produce non-CF languages by restricting the CF production process rather than by establishing long-range relationships. It is not clear whether this is a good idea in practice, but the ideas involved are certainly interesting, which is why we will discuss briefly two types of ordered grammars here.

15.6.1 Rule Ordering by Control Grammar

When using a CF grammar to produce a sentence, one is free to apply any production rule at any time, provided the required non-terminal is present in the sentential form. One type of ordered grammars restrict this freedom by requiring that the sequence of applied rules must obey a second grammar, the *control grammar*. Here, the rules in the original CF grammar are considered tokens in the control grammar, and a sequence of rules produced by the control grammar is called a *control sequence*.

It suffices to give just the control grammar, since the rules of the CF grammar are included in it.

Figure 15.25 gives an ordered grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ in an EBNF notation. It produces, among many others, the control sequence

```
controlS  =>  [textS -> A B C]
              ([A -> aA] [B -> bB] [C -> cC])*
              [A -> ε] [B -> ε] [C -> ε]
```

Fig. 15.25. An ordered grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

```
[textS -> A B C]
[A -> aA] [B -> bB] [C -> cC]
[A -> aA] [B -> bB] [C -> cC]
[A -> ε] [B -> ε] [C -> ε]
```

which in turn produces the final string **aabbcc**. Extension of the grammar to more than 3 tokens is trivial, and many other non-CF languages are also easily expressed. Usually a regular grammar is sufficient for the control grammar, as it was above.

The production process with ordered grammars can get stuck, even in more than one way. A control sequence could specify a CF rule for a non-terminal *A* that is not present in the CF sentential form. And when the control sequence is exhausted, the sentential form may still contain non-terminals. In both cases the attempted production was a blind alley.

15.6.2 Parsing with Rule-Ordered Grammars

Full backtracking top-down parsing, similar to the one explained for coupled grammars in Section 15.5.1, is sometimes possible for ordered grammars too. The basic action consists of making a prediction choice in the control grammar, which results in a rule to apply to the prediction for the rest of the input. If the rule cannot be applied or when the result contradicts the input, we backtrack over this choice and take the next one. If we can continue, we may get stuck further on, in which case we backtrack; or we may find a parsing based on this choice, and then if we need only one parsing we are done, but if we want all parsings we again backtrack.

The problem with this technique is that, just as in Definite Clause parsing, the process may not terminate. One reason is left recursion, but it can also happen that none of the prediction choices in the control grammar ever expands the leftmost non-terminal in the prediction, so the choices are never contradicted by the input, but the process does not get stuck either.

Since the input is recognized by the CF grammar, a normal parse tree results, to which semantics can be attached as to any CF parse tree.

For more information about this type of ordered grammars, see Friš [242] and Lepistö [247].

15.6.3 Marked Ordered Grammars

Although the “token” in the control sequence dictates what kind of non-terminal, for example A, is going to be substituted next during the production process, the system does not specify exactly which A is meant, in case there is more than one. This gives a certain non-determinism to the production process that seems to be alien to it: the control sequence does not really control it all. This is remedied in a variant described by Kulkarni [279], as follows. In a *marked ordered grammar* one member in the right-hand side of each CF rule is marked as the next one to be substituted. The control sequence must then provide the proper rule, and must end exactly when we find a marked terminal or a marked ε; otherwise the production is a blind alley.

Now the control grammar is in full control, but it is fairly clear that this cannot be the whole story: what about the other, non-marked non-terminals that may appear in right-hand sides and therefore in sentential forms? The answer is that a new control sequence is started for each of them.

Figure 15.26 gives a marked ordered grammar for the language $a^n b^{3n} c^n$, where the marked members are between square brackets; rather than including the rules

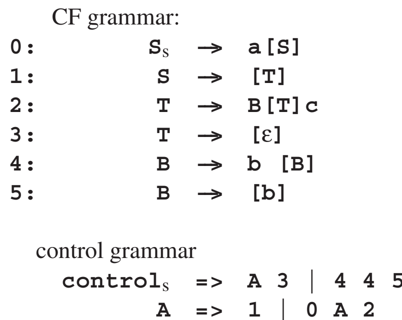


Fig. 15.26. A marked ordered grammar for the language $a^n b^{3n} c^n$

in the control grammar, we have named them 0 to 5 and refer to them by those names in the control grammar. For demonstration purposes the stretches of 3 bs are produced by a second control sequence. The start symbol $control_S$ generates two sets of control sequences, $0^n 1 2^n 3$ and 445. The sentential form starts as S, which is unmarked and so starts a new control sequence; 445 works on B only and would lead to a blind alley, so let us choose 001223. This produces

aaBBεcc

We see that the nesting in the control grammar rule A=>0A2 forces equal numbers of as, Bs and cs to be produced. Next, the Bs are developed, using new control sequences. The control sequence 445 applies twice, and the end result is aabbbbbbcc, as expected.

There is a different way in which the CS restriction on the language can be viewed: the CF grammar produces strings with parse trees, but a parse tree is only acceptable when the path following marked nodes downwards from each non-marked node spells a word in the control language. This view is depicted in Figure 15.27, where the thicker lines represent the paths that have to obey the control grammar. We see that the long middle path correlates the left part of the tree with the right

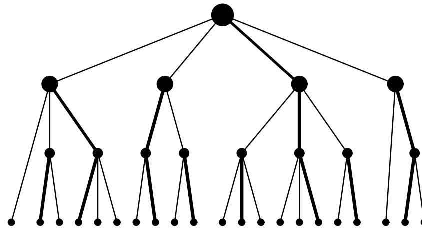


Fig. 15.27. Marked ordered grammar restrictions on a parse tree

part. The shorter paths arrange correlations in the subtrees only, and the subtrees of these are controlled by still shorter paths, and so on, so that control is distributed in an almost fractal way. The main path can, for example, correlate the subject, verb and object in a sentences; shorter paths can then correlate the adjectives in the subject with the noun in the subject, and the adjectives in the object with the noun in the object, as is needed, for example in French: *"Les éléphants asiatiques ont des petites oreilles."* where "*asiatiques*" (masculine plural of "*asiatique*", "asian") correlates with "*éléphants* (masculine plural), and "*petites*" (feminine plural of "*petit*", "small") correlates with "*oreilles*" (feminine plural, "ears").

15.6.4 Parsing with Marked Ordered Grammars

Parsing with marked ordered grammars is easier than with "normal" ordered grammars, since here it is possible to always expand the leftmost non-terminal in the prediction for the rest of the input, and thus produce a leftmost derivation. We start with the CF start symbol as the initial prediction; we start a control string predictor based on the control grammar, and try to develop the prediction to match the input. If the leftmost non-terminal in the prediction is marked, we continue its production guided by the control string we predict from the control grammar. If the leftmost non-terminal is not marked, we stack the control string predictor for the leftmost marked non-terminal and start a new one, as we did for the start symbol. So the data structure in our parser will be a stack of control string predictors (in addition to the prediction), where each control string predictor consists of a prediction for the rest of the control tokens plus some backtrack administration.

We will now demonstrate the parsing of the input string **`aabbbbbbcc`** with the grammar from Figure 15.26. Depicting the full two-level parsing process would require a figure in the style of Figure 4.3, but it would be very big and less than

useful, since all branches but one would be dead ends. We will therefore show only the successful choices for the control string; these are indicated by a double arrow (⇒). All other moves — the application of the CF grammar rule, the matching of the correctly predicted input symbol, the creation of a new control string predictor — are forced, and will be indicated by single arrows (→), without comment. The predicted control stack is shown in curly braces `{}` right after the non-terminal to which it pertains.

We start with the prediction `S→S{control}`, since it is not marked; we then get:

```
S{control} ⇒ S{A3} ⇒ S{0A23} → a[S]{A23} →
[S]{A23} ⇒ [S]{0A223} → a[S]{A223} →
[S]{A223} ⇒ [S]{1223} → [T]{223} → B[T]{23}c →
B{control}[T]{23}c ⇒ B{445}[T]{23}c →
b[B]{45}[T]{23}c → [B]{45}[T]{23}c → b[B]{5}[T]{23}c
→ [B]{5}[T]{23}c → [b]{}[T]{23}c → b[T]{23}c →
[T]{23}c → B[T]{3}cc →
B{control}[T]{3}cc ⇒ B{445}[T]{3}cc → b[B]{45}[T]{3}cc
→ [B]{45}[T]{3}cc → b[B]{5}[T]{3}cc → [B]{5}[T]{3}cc
→ [b]{}[T]{3}cc → b[T]{3}cc → [T]{3}cc → [ε]{}cc →
cc → c → ε
```

We see that there are two more points where a new control string predictor is started.

As with the "normal" ordered grammars, the input is recognized by the CF grammar and a normal parse tree results, so semantics can be attached in the usual way.

Kulkarni and Shankar [279] give very efficient LL(1) and LR(1) parsers for marked ordered grammars.

15.7 Recognition Systems

As said before, non-Chomsky grammars find their origin in user objections to Type 0 and Type 1 grammars. Proponents of recognition systems take the criticism one step further and ask: "Why do we cling to a generative mechanism for the description of our languages, from which we then laboriously derive recognizers,[1] when almost all we ever do is recognizing text? Why don't we specify our languages directly by a recognizer?" Some people answer these two questions by "We shouldn't" and "We should", respectively.

Several recognition systems have been described over the years; examples are the analytic grammars by Gilbert [239], the TMGs by Birman and Ullman [246], and S/SL (Syntax/Semantics Language) by Barnard and Cordy [265, 50] (used in compiler construction). More modern, much more extensive recognition systems are PEGs (Parsing Expression Grammars) by Ford [286] and §-calculus by Jackson [291].

[1] People even write — and read — books about it.

15.7.1 Properties of a Recognition System

Basically a *recognition system* is a program for recognizing and possibly structuring a string. It can in principle be written in any programming language, but is in practice always written in a very specialized programming language, designed expressly for the purpose. Programs in such a programming language look like grammars, but their interpretation is profoundly different.

A very simple recognition expression is `a`, which recognizes the token `a` and passes over it; this means that it moves the read pointer in the input, which was in front of the `a`, to behind the `a`. If the expression is the entire program and `a` is the entire input, the program terminates successfully; what that means depends on the system, but it would be reasonable to assume that a data structure is made available, representing the recognized input, to which semantic operations can be applied. Likewise, the expression `a b` (or simply `ab`) recognizes `ab`, but if the input is `ac` it will fail and leave the read pointer before the `a`.

A more interesting expression is

$$\texttt{a \& } \varepsilon$$

It recognizes an `a` but does not pass over it, so the read pointer remains where it was; the expression then continues to recognize the empty expression after the `&`, which of course succeeds. So the expression `a&`ε succeeds, but the end of the input is not reached, so if this is the entire program, it fails. More generally, an expression of the form

$$e_1 \texttt{ \& } \cdots \texttt{ \& } e_n$$

succeeds if all expressions e_1, $\cdots$, e_n are recognized starting from the present read pointer, and it leaves the read pointer where e_n left it. If one of the expressions e_1, $\cdots$, e_n fails, the whole expression fails and the read pointer is not moved.

In addition to these AND expressions there are OR expressions, which look like normal grammar alternatives, but behave differently. The expression

```
a / b / c
```

recognizes an `a`, a `b` or a `c`, but unlike its counterpart in a grammar it is "prioritized": the choices are tried in order and the first one to succeed wins. We shall see that this difference in interpretation with context-free grammars has several far-reaching consequences.

The expression `ab/a` will preferably recognize `ab` but will settle for `a` if there is no `ab`. On the other hand the expression `a/ab` will only recognize an `a`; if the input is `ab`, the first alternative still takes priority. This means that the second alternative in `a/ab` is useless, but unlike context-free grammars there is no algorithm to clean up a recognition program: it can be proved (Ford [286]) that it is in general undecidable whether an expression is useful (that is, whether it will ever match anything).

Another consequence of the "first matching alternative wins" principle is that a given expression E starting at a given position in the input recognizes and passes over exactly one segment of the input (or fails): there is no ambiguity and the matching

is unique. The recognition algorithm is fully deterministic. We shall see that this is a great help in designing an efficient (in fact linear-time) recognition algorithm.

Recognition expressions can be named, and the names can be used in other expressions or even in the same expression:

```
P  <-  ( P ) / [ P ] / ε
```

defines an expression **P** which recognizes correctly nested sequences of round and square parentheses. **P** can then be used in another recognition expression, for example

```
( & P
```

which recognizes the same strings as **P** except that the strings have to start with a **(**. This is a restriction that would be hard to express in a context-free grammar; see Problem 15.14.

A further consequence of the unique matching is that left-recursive rules are useless. For a rule like $A \leftarrow A\alpha$ to recognize a segment of the input both A and $A\alpha$ would have to match that segment. Since A can match only one segment due to the unique matching, this means that α must be ε, which turns the rule into $A \leftarrow A$, which just says that we have recognized an A when we have recognized an A. The recognition system PEG, for example, forbids left-recursive rules; unlike usefulness, left recursion can be tested.

As in EBNF, entities can be repeated by following them by a superscript asterisk: the expression $\mathbf{a}^*$ recognizes zero or more **a**s. Actually, it does more than that: since it is equivalent to a call of **A** where **A** is defined by

```
A  <-  a A / ε
```

it recognizes the longest possible sequence of **a**s. The reason is that the first alternative of **A** continues to succeed as long as there are **a**s left. As with the alternatives above, repetitions like $\mathbf{a}^*$ will match only one segment of the input: the longest one. A consequence of this is that the expression $\mathbf{a}^*\mathbf{a}$ does not match any string: the $\mathbf{a}^*$ moves over all **a**s present and at the end there will be no **a** left to match the trailing **a** in the expression.

Many recognition systems, including PEG, feature negation: the expression **!P** fails and recognizes nothing if **P** succeeds at this input position, and it succeeds and recognizes nothing if **P** fails at this point. Negation is generally useful but especially so in writing lexical analyzers:

```
\n / \t / !\\ .
```

recognizes **\n**, **\t** and any character except the backslash. (In a convention taken from lexical analyzers, the dot (**.**) at the end of the expression matches any character.)

Some programming languages have complicated conventions for comments. An example is the nesting comment in Pascal. In its basic form it consists of an opener, **(***, some text, and a closer, ***)**. This construct is already non-trivial to recognize

since the text may contain `*`s and `)`s, just not the sequence `*)`. The matter is complicated, however, by the fact that the text may again contain comments; this is useful for commenting out code segments that already contain comments. So

```
(* i := -1; (* should actually start at 0 *) *)
```

is a correct comment in Pascal. Such comments are recognized by the expression

```
       Comment  ←  (* CommentElement* *)
CommentElement  ←  Comment / !*) .
```

which matches fairly well the description in the Pascal Manual. The idea is that a **CommentElement** is either a complete **Comment** or any character (`.`) provided we are not looking at the string `*)`. Ford [286] shows that PEG is quite suitable for integrating lexical analysis and parsing.

At the beginning of this chapter (page 475) we pointed out that the language $a^n b^n c^n$ is the intersection of two CF languages, $a^n b^n c^m$ and $a^m b^n c^n$, and we exploit that fact in the recognition program in Figure 15.28. A call of **S** first recognizes

```
S  ←  A c* !. & a* C
A  ←  a A b / ε
C  ←  b C c / ε
```

Fig. 15.28. A recognition program for $a^n b^n c^n$

the string $a^n b^n c^m$, makes sure there are no left-over characters, backtracks over the string, and then recognizes $a^m b^n c^n$, which only succeeds if $m = n$. This shows that recognition systems can handle languages that are not CF.

15.7.2 Implementing a Recognition System

One of the best and most surprising features of recognition systems is that they can be converted relatively easily into linear-time recognizers. The first step is to identify all subexpressions in the recognition program. What is exactly a subexpression depends on the details of the algorithm, but for the recognition program in Figure 15.28 we can identify the following 12 subexpressions:

S, **Ac*!.**, **a*C**, **A**, **C**, **a***, **c***, **a**, **b**, **c**, **.**, and ε

Slightly different algorithms might also require subexpressions like **Ac*** or **c*!.**, but for our explanation the above set suffices.

We now construct a recognition table T, very similar to the one in tabular parsing explained in Section 4.3. The horizontal axis is labeled with the terminal symbols in the input string and the vertical axis is labeled with the subexpressions identified above. The entry $T_{e,i}$ contains the length of the input segment the subexpression e recognizes at position i; since an expression can recognize only one segment in a given position, we can be sure that an entry of T can never contain more than one

length. This makes the process of filling the table much easier. If the subexpression e does not recognize a segment in position i, the entry $T_{e,i}$ is empty.

As in Section 4.3 there are two ways to fill in the recognition table: top-down and bottom-up. In the top-down approach we start by trying to find out what length of input is recognized by **S** from position 1. This test is implemented as a call of a routine for **S** with 1 as the position parameter. By the time the length has been determined and the call returns, the answer is stored in $T_{\mathtt{S},1}$; it is either one integer or "no". In addition, all intermediate answers obtained in computing $T_{\mathtt{S},1}$ are stored as well, so no entry is computed more than once.

Since recognition systems cannot contain left-recursive rules, we are safe from loops caused by the computation of $T_{A,k}$ resulting in another call of $T_{A,k}$, and so the sequence of calls will terminate properly.

The result of recognizing the string **aaabbbccc** with the program of Figure 15.28 is shown in Figure 15.29. For example, the entry $T_{\mathtt{a}^*\mathtt{C},1}$ has a 9 because $T_{\mathtt{a}^*,1}$

S	9									
Ac*!.	9									
a*C	9									
A	6	4	2	0						
C				6	4	2	0			
a*	3	2	1	0						
c*							3	2	1	0
a	1	1	1	-						
b				1	1	1	-			
c							1	1	1	-
.										-
ε				0			0			0
Input:	**a**	**a**	**a**	**b**	**b**	**b**	**c**	**c**	**c**	**#**
Position:	1	2	3	4	5	6	7	8	9	10

Fig. 15.29. Packrat parsing of the string **aaabbbccc**

has a 3 and $T_{\mathtt{C},4}$ has a 6. The entries marked - are entries where the recursive descent has found the absence of a match; the empty entries are never touched by the algorithm. This algorithm is called *packrat parsing* by Ford [284], because, like a packrat, it stores and remembers everything it has ever seen.

In the bottom-up method, the entries are filled starting at the bottom right corner, working upwards through the columns and working leftwards from column to column (Birman and Ullman [246]). The top left element is the last to be filled in, and

concludes the recognition of the input string. See Figure 15.30, which shows that the bottom-up method computes many more values.

Subexpressions that have parts which recognize the empty string are handled correctly, since the empty string recognition is available as an entry with value 0. For example, an **a*C** of length 6 is recognized correctly in position 4 because an **a*** of length 0 was recognized in position 4.

The order in which the subexpressions appear in the first index of T requires some care in the bottom-up method, since it would be wrong, for example, to compute $T_{\mathtt{a}^*,k}$ before $T_{\mathtt{a},1}$. In fact the subexpressions must be ordered so that the first member of a subexpression comes lower in the table than the subexpression itself. For example, **Ac* ! .** must come higher up in the table than **A**, as indeed it does in Figure 15.30. Since the values in the columns are computed from the bottom upwards, this causes

S	9	-	-	-	-	-	-	-	-	0
Ac* ! .	9	-	-	-	-	-	-	-	-	0
a*C	9	8	7	6	4	2	0	0	0	0
A	6	4	2	0	0	0	0	0	0	0
C	0	0	0	6	4	2	0	0	0	0
a*	3	2	1	0	0	0	0	0	0	0
c*	0	0	0	0	0	0	3	2	1	0
a	1	1	1	-	-	-	-	-	-	-
b	-	-	-	1	1	1	-	-	-	-
c	-	-	-	-	-	-	1	1	1	-
.	1	1	1	1	1	1	1	1	1	-
ε	0	0	0	0	0	0	0	0	0	0
Input:	**a**	**a**	**a**	**b**	**b**	**b**	**c**	**c**	**c**	**#**
Position:	1	2	3	4	5	6	7	8	9	10

Fig. 15.30. Bottom-up tabular parsing of the string **aaabbbccc**

$T_{\mathtt{A},k}$ to be computed before $T_{\mathtt{Ac}^*\mathtt{!.},k}$. Such an ordering is always possible since the expressions cannot be left-recursive. Determining a correct order is pretty tedious and the top-down method is more efficient anyway; but see Problem 15.15.

It is interesting to note that the recognition system works without modification for input sequences that contain non-terminals, for example resulting from previous parsings.

15.7.3 Parsing with Recognition Systems

Although recognition systems are definitely non-Chomskian, top-down parsing with them results in recognition tables very similar to those in CYK parsing, and the techniques presented in Chapter 4 to extract parsings from them can be applied almost unchanged. We will discuss here the conversion to a parse-forest grammar. A few details have to be considered. Recognition systems usually have at least two constructs not occurring in CF grammars: the AND separator **&** and the negation **!**; also, the interpretation of the OR separator **/** differs slightly from that of the alternatives separator | in CF grammars.

The AND separator, for example in **P←A&B**, means that **P** is established by two parsings, both starting at the same input position, where the length of **P** is determined by the length of the last subexpression. Parse forest grammars have little trouble expressing this:

$$\begin{array}{rcl} \mathtt{P}_k_l & \leftarrow & \mathtt{A}_k_m \;\; \& \;\; \mathtt{B}_k_l \\ \mathtt{A}_k_m & \leftarrow & \cdots \\ \mathtt{B}_k_l & \leftarrow & \cdots \end{array}$$

where m may be smaller than, equal to, or larger than l.

The negation **!A** succeeds at an input position k when **A** cannot be recognized there; it produces the rule

$$\mathtt{!A}_k_\mathtt{0} \;\; \leftarrow \;\; \varepsilon$$

It fails when **A** is recognized, and then no rule for it is produced. As a side effect, however, rules have been produced for the recognition of **A**. These rules are now unreachable and can be cleaned out in the usual way (Section 2.9.5).

The same side effect occurs in OR separators. Suppose we have a rule **R←AB/CD/EF**, and suppose **A** is found, **B** is not found and **C** and **D** are found. Then **R** is parsed as **CD**; this results in the rule $\mathtt{R}_k_l \leftarrow \mathtt{C}_k_m \;\; \mathtt{D}_(k+m)_(l-m)$. But in the process a rule $\mathtt{A}_k_p \leftarrow \cdots$ has been created, which is now unreachable.

Since the result of a recognition system is unambiguous, the process yields a parse-tree grammar rather than a parse forest grammar. The parse tree grammar resulting from the table in Figure 15.29 does not require clean-up. It is given in Figure 15.31.

15.7.4 Expressing Semantics in Recognition Systems

Since an almost normal parse tree results, semantics can be attached to it in the usual way. The two constructs that make it different from a CF parse tree are the repetition expression and the AND expression. Handling the semantics of the AND expression is straightforward:

```
P  ←  Q & R  {P.attr := func(Q.attr, R.attr);}
```

The semantics of the repetition allows two interpretations, as in Section 2.3.2.4, but the iterative one is the most natural here. $\mathtt{A}^*$ could yield an attribute which is an array of the attributes of the separate **A**s.

```
       S_1_9  ←  Ac*!._1_9 & a*C_1_9
   Ac*!._1_9  ←  A_1_6 c*_7_3 !._10_0
     a*C_1_9  ←  a*_1_3 C_4_6
       A_1_6  ←  a_1_1 A_2_4 b_6_1
       A_2_4  ←  a_2_1 A_3_2 b_5_1
       A_3_2  ←  a_3_1 A_4_0 b_4_1
       A_4_0  ←  ε
       C_4_6  ←  b_4_1 C_5_4 c_9_1
       C_5_4  ←  b_5_1 C_6_2 c_8_1
       C_6_2  ←  b_6_1 C_7_0 c_7_1
       C_7_0  ←  ε
      a*_1_3  ←  a_1_1 a*_2_2
      a*_2_2  ←  a_2_1 a*_3_1
      a*_3_1  ←  a_3_1 a*_4_0
      a*_4_0  ←  ε
      c*_7_3  ←  c_7_1 c*_8_2
      c*_8_2  ←  c_8_1 c*_9_1
      c*_9_1  ←  c_9_1 c*_10_0
     c*_10_0  ←  ε
     !._10_0  ←  ε
```

Fig. 15.31. Parse tree grammar from the packrat parsing in Figure 15.29

15.7.5 Error Handling in Recognition Systems

Without special measures, a recognition system behaves as badly on incorrect input as any backtracking top-down system: it rejects all hypotheses it generated, backtracks to the beginning of the input and says: "No". Special measures are, however, possible in this case. The idea is to have a failing attempt to recognize an expression return the reason why it failed, just as a successful attempt returns the length of the recognized segment (Grimm [287]). This is demonstrated in Figure 15.32, where the input has been changed to **aaabbccc**. The attempt to recognize **S** in position 1 results in a call to **Ac***, which leads to a call of **A** in position 1. The **a** is recognized, and so is the **A** of length 4 in position 2, but the attempt to combine these into an **A** at position 1 fails, because there is no **b** in position 6. This information is made into the result of the call to **A** and is passed upwards, to **S**, where an error message is derived from it.

There is one situation in which error information needs to be merged: when an OR expression fails. Suppose we have an expression

```
trailing_Z_option  ←  Z / !.
```

which recognizes a **Z** or end-of-file; suppose **trailing_Z_option** is called at input position k; and a **Z** is present at position k but contains a syntax error. Then **Z** will return with error information about a position $l > k$, and next **!.** will fail with error information about position k. Since the first is more informative, we should then let **trailing_Z_option** fail with **Z**'s error information about position l.

S	no b at pos. 6								
Ac*!.	no b at pos. 6								
a*C									
A	no b at pos. 6	4	2	0					
C									
a*									
c*									
a	1	1	1	no a at pos. 4					
b				1	1	no b at pos. 6			
c									
.									
ε									
Input:	a	a	a	b	b	c	c	c	#
Position:	1	2	3	4	5	6	7	8	9

Fig. 15.32. Bottom-up tabular parsing of the string **aaabbbccc**

15.8 Boolean Grammars

One way to obtain more than context-free power and not stray too much from context-free grammars is to extend them with the Boolean combination operators *negation* ($\neg A$, all strings not produced by A) and *intersection* ($A \cap B$, all strings produced both by A and B). This yields the *Boolean grammars* or *Boolean closure grammars*.

This approach was pioneered by Schuler [248] who in 1974 gave a Turing machine recognizing languages described by Boolean formulas over CF languages and showed how to use it to define a context-sensitive fragment of ALGOL 60. In 1991 Heilbrunner and Schmitz [267] gave an $O(n^3)$ recognizer for Boolean grammars, based on an adapted Earley parser. Recently (2004-2005), Okhotin [288, 290, 289] has described properties of Boolean grammars and given several parsers for them. Even better, the author [290] shows how to use them to enforce the context conditions in a simple C-like programming language, including checks for use of undefined identifiers, multiple definitions, and calling a function with the wrong number of parameters, all of that in about 4 pages.

15.8.1 Expressing Context Checks in Boolean Grammars

The principle of the paradigm is “A correct program is the intersection of a syntactically correct program with one or more context-enforcing languages.” This requires

building context-enforcing languages; their nature depends on the kind of context conditions that need to be checked.

To demonstrate the paradigm we will use an abstract form of a very, very simple programming language. Programs consist of an open brace, a set of definitions containing one or more different identifiers, a semicolon, zero or more applications of the defined identifiers, and a close brace. Identifiers are one letter long. An example of a “program” is `{ij;ijiiij}`, which could be an abstract of the C-like block `{int i,j;i=4;j=i*i;print i,j;}`.

The context conditions are: no multiple identifiers in the definitions section, and no undefined identifiers after the semicolon. The main tool for checking context conditions on identifiers is a language that matches pairs of identifiers: the set $\{wcw\}$, where the two ws are the same identifier, and c is any sequence of tokens. An example is `j;ijiiij`. A Boolean grammar for this demo language is given in Figure 15.33.

```
Program_S --> '(' Body ')'
Body --> Definitions ';' Idf_Seq_Opt & No_Undefined_Idfs
Definitions --> Idf_Seq & No_Multiple_Idfs
No_Undefined_Idfs -->
        Idf_Seq ';' |
        No_Undefined_Idfs Idf & Last_Idf_Is_Defined
Last_Idf_Is_Defined -->
        Head_and_Tail_Idfs_Match |
        Idf Last_Idf_Is_Defined
No_Multiple_Idfs --> ¬ Multiple_Idfs
Multiple_Idfs -->
        Idf_Seq_Opt Head_and_Tail_Idfs_Match Idf_Seq_Opt

Any_Letter --> 'a' | ... | 'z'
Any_Char --> Any_Letter | ';'
Any_Seq --> Any_Seq Any_Char | ε
Idf --> Any_Letter
Idf_Seq --> Idf | Idf_Seq Idf
Idf_Seq_Opt --> Idf_Seq | ε
Head_and_Tail_Idfs_Match -->
        'a' Any_Seq 'a' | ... | 'z' Any_Seq 'z'
```

Fig. 15.33. A Boolean grammar for the context-checked specification of a very, very simple programming language

The first rule specifies the syntactic shape of a program. A `Body` has the form `Definitions ';' Idf_Seq_Opt` and has no undefined applied identifiers. A body with no undefined applied identifiers `No_Undefined_Idfs` is either a body with no applied identifiers at all, or an already checked body followed by an identifier and that last identifier is also defined. The last identifier of a string `Idf` $\cdots$ `;` $\cdots$ `Idf` (which is what we are looking at by now) is defined if the head and tail identifiers

match (because the first **Idf** is in the definition section), or we take away the first identifier and then a string in which the last identifier is defined remains. This makes sure all identifiers are defined.

Next, **Definitions** is a sequence of identifiers with no multiple copies in it. **No_Multiple_Idfs** is of course the negation of **Multiple_Idfs**. And a sequence of identifiers with multiple copies is any sequence which contains a subsequence the head and tail items of which match. Elementary, my dear Watson... The last few rules complete the full context-checked specification of the very simple programming language. Okhotin [290] gives a much more extensive example, based on similar principles.

When we compare our exposition to the above paper, we see that we have not only restricted ourselves to a simplification of a simplification of a programming language, but that we have also cut an enormous corner. Specifying the language **Head_and_Tail_Idfs_Match** for identifiers of unrestricted length is much more complicated than for identifiers of length 1, and requires substantial trickery, which is explained in Okhotin [283].

It should also be noted that negation in a production system has weird properties, and soon leads to paradoxes. The simple rule **S**$\rightarrow\neg$**S** describes the set of strings that it does not contain; in other words, a string is in **S** if it is not in **S**. Only slightly better is the grammar **S**$\rightarrow\neg$**T; T**$\rightarrow$**T**. Since **T** produces nothing, **S** produces all strings, i.e. Σ^*. And **S**$\rightarrow$**0|1|**$\neg$**S[0|1]** contains any string Z over **[0|1]*** provided it does not contain the string obtained by removing the last token from Z; it is unclear what that means. For ways to catch and/or tame these paradoxes see Heilbrunner and Schmitz [267] and Okhotin [288].

15.8.2 Parsing with Boolean Grammars

Boolean grammars can be parsed using tabular parsing, in a way similar to recognition systems in Section 15.7.3; see Heilbrunner and Schmitz [267] and Okhotin [288]. Generalized LR and LL parsing is also possible (Okhotin [289]).

15.8.3 §-Calculus

§-calculus (Jackson [285, 291]) adds another feature to the non-Chomsky arsenal of Boolean grammars: the possibility to assign a recognized segment of the input plus its parse tree to a variable of type non-terminal and to use that variable further on in the rule.

For one thing, the definition of **head_and_tail_idfs_match** becomes trivial with this facility:

```
grammar head_and_tail_idfs_match {
    S ::= $x(Idf) Any_Sequence x;
};
```

Here the first identifier is read and assigned to the local variable **x**. Its value is retrieved at the end of the rule and used to parse the last identifier; that parsing of course only succeeds if the two identifiers are identical.

It is interesting to see how this feature is implemented. Conceptually, the moment the assignment is made, a new grammar rule is created in which the non-terminal variable is substituted out. For example, once the **arg2** has been recognized in the input segment **arg2){print(arg1+arg2**, a new rule

```
S ::= "arg2" Any_Sequence "arg2";
```

is created, which then enforces the head-tail match. Note that this makes non-terminal variable assignment in §-calculus similar to logic variable binding in Prolog.

In addition to local variables there are global variables, and functions to manipulate and use them. This opens up a gamut of possibilities that could fill a book, which is exactly what the inventor did [291], and we refer the reader to it.

For completeness, the §-grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ is:

```
grammar AnBnCn {
    S ::= (('[abc]+')<A>)<B>;
    A ::= X 'c+';
    X ::= "a" [X] "b";
    B ::= 'a+' Y;
    Y ::= "b" [Y] "c";
};
```

Although §-grammars are defined as generating devices, they can equally well be seen as recognition devices. Jackson [291] describes a parser for them based on a push-down automaton the stack elements of which are a restricted form of trees; this implements the variable substitution mechanism. In addition, it uses many optimizations. The time complexity is unknown; in practice it is almost always less than $O(n^2)$ and always less than $O(n^3)$.

15.9 Conclusion

Three non-Chomsky systems stand out in the landscape at the moment: attribute grammars, as the most practical; VW grammars, as the most elegant and mathematically satisfying; and Boolean systems like PEG, Boolean grammars, and §-calculus, as the most promising. Coupled grammars are an interesting fourth.

Parsing of the non-Chomsky systems is generally based on top-down depth-first search, sometimes with some guidance against left recursion. Only TAGs are habitually parsed with a bottom-up method, and recognition systems can be conveniently handled with both methods.

Problems

Problem 15.1: Design a systematic way to write CS grammars for languages $t_1^n t_2^n \cdots t_k^n$ for any set of k symbols.

Problem 15.2: *a*. Write a VW grammar for the language *ww* where *w* is any string of `a`s and `b`s. *b*. Convert it to a working Prolog program. What is its time dependency?

Problem 15.3: How is negation (`where M not equals N`) expressed in a VW grammar? How about `where X symbol not equals Y symbol`?

Problem 15.4: Refer to the grammar in Figure 15.10. *a*. Explain how it produces `i symbol`, `i symbol`, `i symbol`, `ii symbol`, `ii symbol`, `ii symbol`, `iii symbol`, `iii symbol`, `iii symbol`. *b*. If `N` is chosen to be ε in the right hand side of the rule for $\texttt{text}_S$, what does the grammar produce?

Problem 15.5: Design a TAG for the language $\texttt{a}^n\texttt{b}^n\texttt{c}^n\texttt{d}^n$. Hint: create an auxiliary tree that inserts `ab` between the `a` and the `b` in the elementary tree and at the same time inserts `cd` between the `c` and the `d`.

Problem 15.6: Design a TAG for the language *ww* where *w* is any string of `a`s and `b`s. Hint: It's all cross-dependencies.

Problem 15.7: If your native language is not Dutch or Swiss German, try to find examples of cross-dependencies in your native language. If you are Dutch or Swiss German, try to find examples of multiple cross-dependencies, for example of the type $A_1A_2A_3B_1B_2B_3$.

Problem 15.8: Expand the parsing algorithm sketched in Section 15.4.2 into a complete algorithm, and compare the result to Vijay-Shankar and Joshi's version [264].

Problem 15.9: Research Project: Devise a reasonable error reporting and recovery technique for TAGs.

Problem 15.10: Using the obvious coupled grammar for the language *ww* where *w* is any string of `a`s and `b`s, simulate the top-down parsing of the input string `aaaaaa`. Why does the system backtrack even on obvious parsings like `abbabb`?

Problem 15.11: The official definition of coupled grammars demands that when a grammar rule uses one component of a non-terminal, it has to use all components, and use them in the right order. There seems to be little reason to demand this. The full backtracking top-down parsing algorithm is not affected by it, for example. On the other hand, the gain from dropping it is not obvious either. Examine the consequences of lifting this restriction.

Problem 15.12: Take the rhetorical questions at the beginning of Section 15.7 at face value and give reasons why we should prefer grammars over recognition systems.

Problem 15.13: *a*. At the end of Section 15.7.1 we use the expression `!.` to check for the absence of spurious characters. Construct a string $\texttt{a}^p\texttt{b}^q\texttt{c}^r$ not in $\texttt{a}^n\texttt{b}^n\texttt{c}^n$ that would be recognized by `S` if we had left the check out. *b*. Give a simpler recognition expression for `S`.

Problem 15.14: Given a CF grammar `G` with a rule `P` and a token `a`, devise a technique to obtain a grammar rule `Q` in `G` which produces `a&P`, that is, all the strings which `P` produces that start with an `a`.

Problem 15.15: *a*. Draw up the precise criteria for the order of subexpressions in bottom-up PEG recognition. *b*. Design an algorithm for obtaining the desired order.

Problem 15.16: Design a way to derive a root set (Section 12.2.1.1) for a recognition system like PEG, and use it to a create the corresponding suffix parser.

Problem 15.17: *History of Recognition Systems:* Birman and Ullman [246, p. 21] give a very clever recognition program for the language a^{n^2}, sequences of *a*s whose lengths are square numbers. The program is specified in a formalism that differs considerably from the one discussed here. Rewrite the program in a more PEG-like formalism, and convince yourself that it works.