
H4

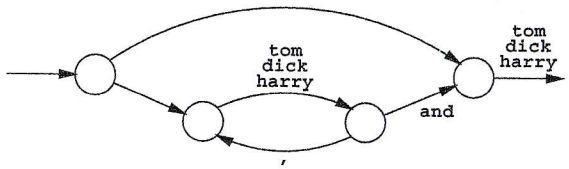

Fig. 2.24. A transition graph for the `[tdh]` language

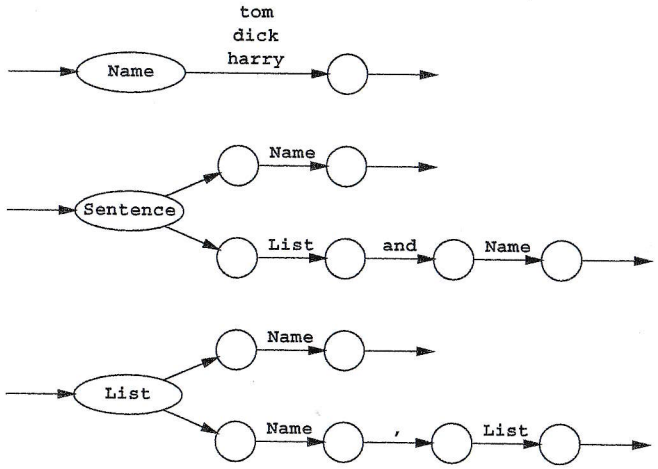

Fig. 2.25. A recursive transition network for the sample grammar on page 23

Rather than producing N by appending it to the output, we push node n_2 on a stack, and continue our walk at the entrance to the transition graph for N. And when we are leaving the transition graph for N, we pop n_2 from the stack and continue at node n_2. This is the *recursive transition network* interpretation of context-free grammars: the set of graphs is the transition network, and the stacking mechanism provides the recursion.

Figure 2.26 shows the right-regular rules of the FS grammar Figure 2.14(a) as transition graphs. Here we have left out the unmarked arrows at the exits of the graphs and the corresponding nodes; we could have done the same in Figure 2.25, but doing so would have complicated the stacking mechanism.

We see that we have to produce a non-terminal only when we are just leaving another, so we do not need to stack anything, and can interpret an arrow marked with a non-terminal N as a jump to the transition graph for N. So a regular grammar corresponds to a (non-recursive) transition network.

If we connect each exit marked N in such a network to the entrance of the graph for N we can ignore the non-terminals, and obtain a transition graph for the corresponding language. When we apply this short-circuiting to the transition network of Figure 2.26 and rearrange the nodes a bit, we get the transition graph of Figure 2.24.

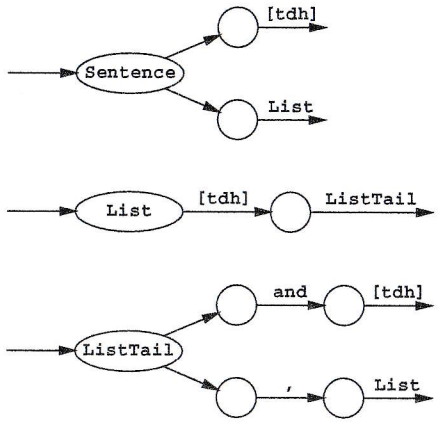

Fig. 2.26. The FS grammar of Figure 2.14(a) as transition graphs

2.9 Hygiene in Context-Free Grammars

All types of grammars can contain *useless rules*, rules that cannot play a role in any successful production process. A production process is *successful* when it results in a terminal string. Production attempts can be unsuccessful by getting stuck (no further substitution possible) or by entering a situation in which no substitution sequence will ever remove all non-terminals. An example of a Type 0 grammar that can get stuck is

1. $S_S \rightarrow$ `A B`
2. `S` $\rightarrow$ `B A`
3. `S` $\rightarrow$ `C`
4. `A B` $\rightarrow$ `x`
5. `C` $\rightarrow$ `C C`

When we start with the first rule for `S`, all goes well and we produce the terminal string `x`. But when we start with rule 2 for `S` we get stuck, and when we start with rule 3, we get ourselves in an infinite loop, producing more and more `C`s. Rules 2, 3 and 5 can never occur in a successful production process: they are useless rules, and can be removed from the grammar without affecting the language produced.

Useless rules are not a fundamental problem: they do not obstruct the normal production process. Still, they are dead wood in the grammar, and one would like to remove them. Also, when they occur in a grammar specified by a programmer, they probably point at some error, and one would like to detect them and give warning or error messages.

The problems with the above grammar were easy to understand, but it can be shown that in general it is undecidable whether a rule in a Type 0 or 1 grammar is useless: there cannot be an algorithm that does it correctly in all cases. For context-free grammars the situation is different, however, and the problem is rather easily solved.

Rules in a context-free grammar can be useless through three causes: they may contain undefined non-terminals, they may not be reachable from the start symbol, and they may fail to produce anything. We will now discuss each of these ailments in more detail; an algorithm to rid a grammar of them is given in Section 2.9.5.

2.9.1 Undefined Non-Terminals

The right-hand side of some rule may contain a non-terminal for which no production rule is given. Such a rule will never have issue and can be removed from the grammar. If we do this, we may of course remove the last definition of another non-terminal, which will then in turn become undefined, etc.

We will see further on (for example in Section 4.1.3) that it is occasionally useful to also recognize undefined terminals. Rules featuring them in their right-hand sides can again be removed.

2.9.2 Unreachable Non-Terminals

If a non-terminal cannot be reached from the start symbol, its defining rules will never be used, and it cannot contribute to the production of any sentence. Unreachable non-terminals are sometimes called "unused non-terminals". But this term is a bit misleading, because an unreachable non-terminal A may still occur in some right-hand side $B \rightarrow \cdots A \cdots$, making it look useful, provided B is unreachable; the same applies of course to B, etc.

2.9.3 Non-Productive Rules and Non-Terminals

Suppose X has as its only rule $X \rightarrow aX$ and suppose X can be reached from the start symbol. Now X will still not contribute anything to the sentences of the language of the grammar, since once X is introduced, there is no way to get rid of it: X is a non-productive non-terminal. In addition, any rule which has X in its right-hand side is non-productive. In short, any rule that does not in itself produce a non-empty sublanguage is non-productive. If all rules for a non-terminal are non-productive, the non-terminal is non-productive.

In an extreme case *all* non-terminals in a grammar are non-productive. This happens when all right-hand sides in the grammar contain at least one non-terminal. Then there is just no way to get rid of the non-terminals, and the grammar itself is non-productive.

These three groups together are called *useless non-terminals*.

2.9.4 Loops

The above definition makes "non-useless" all rules that can be involved in the production of a sentence, but there still is a class of rules that are not really useful: rules of the form $A \rightarrow A$. Such rules are called *loops*. Loops can also be indirect: $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$; and they can be hidden: $A \rightarrow PAQ$, $P \xrightarrow{*} \varepsilon$, $Q \xrightarrow{*} \varepsilon$, so a production sequence $A \rightarrow PAQ \rightarrow \ldots A \ldots \rightarrow A$ is possible.

A loop can legitimately occur in the production of a sentence, and if it does, there is also a production of that sentence without the loop. Loops do not contribute to the language and any sentence the production of which involves a loop is *infinitely ambiguous*, meaning that there are infinitely many production trees for it. Algorithms for loop detection are given in Section 4.1.2.

Different parsers react differently to grammars with loops. Some (most of the general parsers) faithfully attempt to construct an infinite number of derivation trees, some (for example, the CYK parser) collapse the loop as described above and some (most deterministic parsers) reject the grammar. The problem is aggravated by the fact that loops can be concealed by ε-rules: a loop may only become visible when certain non-terminals produce ε.

A grammar without useless non-terminals and loops is called a *proper grammar*.

2.9.5 Cleaning up a Context-Free Grammar

Normally, grammars supplied by people do not contain undefined, unreachable or non-productive non-terminals. If they do, it is almost certainly a mistake (or a test!), and we would like to detect and report them. Such anomalies can, however, occur normally in generated grammars or be introduced by some grammar transformations, in which case we wish to detect them to "clean up" the grammar. Cleaning the grammar is also very important when we obtain the result of parsing as a parse-forest grammar (Section 3.7.4, Chapter 13, and many other places).

The algorithm to detect and remove useless non-terminals and rules from a context-free grammar consists of two steps: remove the non-productive rules and remove the unreachable non-terminals. Surprisingly it is not necessary to remove the useless rules due to undefined non-terminals: the first step does this for us automatically.

```
Ss -> A B | D E
A  -> a
B  -> b C
C  -> c
D  -> d F
E  -> e
F  -> f D
```

Fig. 2.27. A demo grammar for grammar cleaning

We will use the grammar of Figure 2.27 for our demonstration. It looks fairly innocent: all its non-terminals are defined and it does not exhibit any suspicious constructions.

2.9.5.1 Removing Non-Productive Rules

We find the non-productive rules by finding the productive ones. Our algorithm hinges on the observation that a rule is productive if its right-hand side consists of symbols all of which are productive. Terminal symbols are productive since they produce terminals and empty is productive since it produces the empty string. A non-terminal is productive if there is a productive rule for it, but the problem is that initially we do not know which rules are productive, since that is exactly the thing we are trying to find out.

We solve this dilemma by first marking all rules and non-terminals as "Don't know". We now go through the grammar of Figure 2.27 and for each rule for which we do know that all its right-hand side members are productive, we mark the rule and the non-terminal it defines as "Productive". This yields markings for the rules `A→a`, `C→c`, and `E→e`, and for the non-terminals `A`, `C` and `E`.

Now we know more and apply this knowledge in a second round through the grammar. This allows us to mark the rule `B→bC` and the non-terminal `B`, since now `C` is known to be productive. A third round gives us `S→AB` and `S`. A fourth round yields nothing new, so there is no point in a fifth round.

We now know that `S`, `A`, `B`, `C`, and `E` are productive, but `D` and `F` and the rule `S→DE` are still marked "Don't know". However, now we know more: we know that we have pursued all possible avenues for productivity, and have not found any possibilities for `D`, `F` and the second rule for `S`. That means that we can now upgrade our knowledge "Don't know" to "Non-productive". The rules for `D`, `F` and the second rule for `S` can be removed from the grammar; the result is shown in Figure 2.28. This makes `D` and `F` undefined, but `S` stays in the grammar since it is productive, in spite of having a non-productive rule.

```
Ss → A B
A  → a
B  → b C
C  → c
E  → e
```

Fig. 2.28. The demo grammar after removing non-productive rules

It is interesting to see what happens when the grammar contains an undefined non-terminal, say U. U will first be marked "Don't know", and since there is no rule defining it, it will stay "Don't know". As a result, any rule R featuring U in its right-hand side will also stay "Don't know". Eventually both will be recognized as "Non-productive", and all rules R will be removed. We see that an "undefined non-terminal" is just a special case of a "non-productive" non-terminal: it is non-productive because there is no rule for it.

The above knowledge-improving algorithm is our first example of a *closure algorithm*. Closure algorithms are characterized by two components: an *initialization*, which is an assessment of what we know initially, partly derived from the situation and partly "Don't know"; and an inference rule, which is a rule telling how knowledge from several places is to be combined. The inference rule for our problem was:

> For each rule for which we do know that all its right-hand side members are productive, mark the rule and the non-terminal it defines as "Productive'.'

It is implicit in a closure algorithm that the inference rule(s) are repeated until nothing changes any more. Then the preliminary "Don't know" can be changed into a more definitive "Not X", where "X" was the property the algorithm was designed to detect.

Since it is known beforehand that in the end all remaining "Don't know" indications are going to be changed into "Not X", many descriptions and implementations of closure algorithms skip the whole "Don't know" stage and initialize everything to "Not X". In an implementation this does not make much difference, since the meaning of the bits in computer memory is not in the computer but in the mind of the programmer, but especially in text-book descriptions this practice is unelegant and can be confusing, since it just is not true that initially all the non-terminals in our grammar are "Non-productive".

We will see many examples of closure algorithms in this book; they are discussed in more detail in Section 3.9.

2.9.5.2 Removing Unreachable Non-Terminals

A non-terminal is called *reachable* or *accessible* if there exists at least one sentential form, derivable from the start symbol, in which it occurs. So a non-terminal A is reachable if $S \xrightarrow{*} \alpha A \beta$ for some α and β.

We found the non-productive rules and non-terminals by finding the "productive" ones. Likewise, we find the unreachable non-terminals by finding the reachable ones. For this, we can use the following closure algorithm. First, the start symbol is marked "reachable"; this is the initialization. Then, for each rule in the grammar of the form $A \rightarrow \alpha$ with A marked, all non-terminals in α are marked; this is the inference rule. We continue applying the inference rule until nothing changes any more. Now the unmarked non-terminals are not reachable and their rules can be removed.

The first round marks `A` and `B`; the second marks `C`, and the third produces no change. The result — a clean grammar — is in Figure 2.29. We see that rule `E→e`, which was reachable and productive in Figure 2.27 became isolated by removing the non-productive rules, and is then removed by the second step of the cleaning algorithm.

```
Ss → A B
A  → a
B  → b C
C  → c
```

Fig. 2.29. The demo grammar after removing all useless rules and non-terminals

Removing the unreachable rules cannot cause a non-terminal N used in a reachable rule to become undefined, since N can only become undefined by removing all its defining rules but since N is reachable, the above process will not remove any rule for it. A slight modification of the same argument shows that removing the unreachable rules cannot cause a non-terminal N used in a reachable rule to become non-productive: N, which was productive or it would not have survived the previous clean-up step, can only become non-productive by removing some of its defining rules but since N is reachable, the above process will not remove any rule for it. This shows conclusively that after removing non-productive non-terminals and then removing unreachable non-terminals we do not need to run the step for removing non-productive non-terminals again.

It is interesting to note, however, that first removing unreachable non-terminals and then removing non-productive rules may produce a grammar which again contains unreachable non-terminals. The grammar of Figure 2.27 is an example in point.

Furthermore it should be noted that cleaning a grammar may remove *all* rules, including those for the start symbol, in which case the grammar describes the empty language; see Section 2.6.

Removing the non-productive rules is a bottom-up process: only the bottom level, where the terminal symbols live, can know what is productive. Removing unreachable non-terminals is a top-down process: only the top level, where the start symbol(s) live(s), can know what is reachable.

2.10 Set Properties of Context-Free and Regular Languages

Since languages are sets, it is natural to ask if the standard operations on sets — union, intersection, and negation (complement) — can be performed on them, and if so, how.

The *union* of two sets S_1 and S_2 contains the elements that are in either set; it is written $S_1 \cup S_2$.. The *intersection* contains the elements that are in both sets; it is written $S_1 \cap S_2$. And the *negation* of a set S contains those in Σ^* but not in S; it is written $\neg S$. In the context of formal languages the sets are defined through grammars, so actually we want to do the operations on the grammars rather than on the languages.

Constructing the grammar for the union of two languages is trivial for context-free and regular languages (and in fact for all Chomsky types): just construct a new start symbol $S' \rightarrow S_1|S_2$, where S_1 and S_2 are the start symbols of the two grammars that describe the two languages. (Of course, if we want to combine the two grammars into one we must make sure that the names in them differ, but that is easy to do.)

Intersection is a different matter, though, since the intersection of two context-free languages need not be context-free, as the following example shows. Consider the two CF languages $L_1 = \mathbf{a}^n\mathbf{b}^n\mathbf{c}^m$ and $L_2 = \mathbf{a}^m\mathbf{b}^n\mathbf{c}^n$ described by the CF grammars

```
L1s -> A P                          L2s -> Q C
A   -> a A b | ε      and           Q   -> a Q | ε
P   -> c P | ε                      C   -> b C c | ε
```


When we take a string that occurs in both languages and thus in their intersection, it will have the form $\mathbf{a}^p\mathbf{b}^q\mathbf{c}^r$ where $p = q$ because of L_1 and $q = r$ because of L_2. So the intersection language consists of strings of the form $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ and we know that that language is not context-free (Section 2.7.1).

The intersection of CF languages has weird properties. First, the intersection of two CF languages always has a Type 1 grammar — but this grammar is not easy to construct. More remarkably, the intersection of three CF languages is more powerful than the intersection of two of them: Liu and Weiner [390] show that there are languages that can be obtained as the intersection of k CF languages, but not of $k-1$. In spite of that, *any* Type 1 language, and even any Type 0 language, can be constructed by intersecting just two CF languages, provided we are allowed to erase all symbols in the resulting strings that belong to a set of *erasable symbols*.

The CS language we will use to demonstrate this remarkable phenomenon is the set of all strings that consist of two identical parts: ww, where w is any string over the given alphabet; examples are **aa** and **abbababbab**. The two languages to be intersected are defined by

```
L3s -> A P                                  L4s -> Q C
A   -> a A x | b A y | ε      and           Q   -> a Q | b Q | ε
P   -> a P | b P | ε                        C   -> x C a | y C b | ε
```

where **x** and **y** are the erasable symbols. The first grammar produces strings consisting of three parts, a sequence A_1 of **a**s and **b**s, followed by its "dark mirror" image M_1, in which **a** corresponds to **x** and **b** to **y**, followed by an arbitrary sequence G_1 of **a**s and **b**s. The second grammar produces strings consisting of an arbitrary sequence G_2 of **a**s and **b**s, a "dark" sequence M_2, and its mirror image A_2, in which again **a** corresponds to **x** and **b** to **y**. The intersection forces $A_1 = G_2$, $M_1 = M_2$, and $G_1 = A_2$. This makes A_2 the mirror image of the mirror image of A_1, in other words equal to A_1. An example of a string in the intersection is **abbabyxyyxabbab**, where we see the mirror images **abbab** and **yxyyx**. We now erase the erasable symbols **x** and **y** and obtain our result **abbababbab**.

Using a massive application of the above mirror-mirror trick, one can relatively easily prove that any Type 0 language can be constructed as the intersection of two CF languages, plus a set of erasable symbols. For details see, for example, Révész [394].

Remarkably the intersection of a context-free and a *regular* language is always a context-free language, and, what's more, there is a relatively simple algorithm to construct a grammar for that intersection language. This gives rise to a set of unusual parsing algorithms, which are discussed in Chapter 13.

If we cannot have intersection of two CF languages and stay inside the CF languages, we certainly cannot have negation of a CF language and stay inside the CF languages. If we could, we could negate two languages, take their union, negate the result, and so obtain their intersection. In a formula: $L_1 \cap L_2 = \neg((\neg L_1) \cup (\neg L_2))$; this formula is known as De Morgan's Law.

In Section 5.4 we shall see that union, intersection and negation of regular (Type 3) languages yield regular languages.