3

Introduction to Parsing

To parse a string according to a grammar means to reconstruct the production tree (or trees) that indicate how the given string can be produced from the given grammar. It is significant in this respect that one of the first publications on parsing (Greibach's 1963 doctoral thesis [6]), was titled "Inverses of Phrase Structure Generators", where a phrase structure generator is to be understood as a system for producing phrases from a phrase structure (actually context-free) grammar.

Although production of a sentence based on a Type 0 or Type 1 grammar gives rise to a production graph rather than a production tree, and consequently parsing yields a parse graph, we shall concentrate on parsing using a Type 2, context-free grammar, and the resulting parse trees. Occasionally we will touch upon parsing with Type 0 or Type 1 grammars, as for example in Section 3.2, just to show that it *is* a meaningful concept.

3.1 The Parse Tree

There are two important questions on reconstructing the production tree: why do we need it; and how do we do it.

The requirement to recover the production tree is not natural. After all, a grammar is a condensed description of a set of strings, i.e., a language, and our input string either belongs or does not belong to that language; no internal structure or production path is involved. If we adhere to this formal view, the only meaningful question we can ask is if a given string can be recognized according to a grammar; any question as to how would be a sign of senseless, even morbid curiosity. In practice, however, grammars have semantics attached to them: specific semantics is attached to specific rules, and in order to determine the semantics of a string we need to find out which rules were involved in its production and how. In short, recognition is not enough, and we need to recover the production tree to get the full benefit of the syntactic approach.

The recovered production tree is called the *parse tree*. The fact that it is next to impossible to attach semantics to specific rules in Type 0 and Type 1 grammars explains their relative unimportance in parsing, compared to Types 2 and 3.

How we can reconstruct the production tree is the main subject of the rest of this book.

3.1.1 The Size of a Parse Tree

A parse tree for a string of n tokens consists of n nodes belonging to the terminals, plus a number of nodes belonging to the non-terminals. Surprisingly, there cannot be more than $C_G n$ nodes belonging to non-terminals in a parse tree with n token nodes, where C_G is a constant that depends on the grammar, provided the grammar has no loops. This means that the size of any parse tree is linear in the length of the input.

Showing that this is true has to be done in a number of steps. We prove it first for grammars in which all right-hand sides have length 2; these result in *binary trees*, trees in which each node either has two children or is a leaf (a node with no children). Binary trees have the remarkable property that all binary trees with a given number of leaves have the same number of nodes, regardless of their shapes. Next we allow grammars with right-hand sides with lengths > 2, then grammars with unit rules, and finally grammars with nullable rules.

As we said, an input string of length n consists of n token nodes. When the parse tree is not there yet, these nodes are parentless leaves. We are now going to build an arbitrary binary tree to give each of these nodes a parent, labeled with a non-terminal from the grammar. The first parent node P_1 we add lowers the number of parentless nodes by 2, but now P_1 is itself a parentless node; so we now have $n+1$ nodes of which $n-2+1=n-1$ are parentless. The same happens with the second added parent node P_2, regardless of whether one of its children is P_1; so now we have $n+2$ nodes of which $n-2$ are parentless. After j steps we have $n+j$ nodes of which $n-j$ are parentless and after $n-1$ steps we have $2n-1$ nodes of which 1 is parentless. The 1 parentless node is the top node, and the parse tree is complete. So we see that when all right-hand sides have length 2, the parse tree for an input of length n contains $2n-1$ nodes, which is linear in n.

If some of the right-hand sides have length > 2, fewer parent nodes may be required to construct the tree. So the total tree size may be smaller than $2n-1$, which is certainly smaller than $2n$.

If the grammar contains unit rules — rules of the form $A \rightarrow B$ — it is no longer true that adding a parent node reduces the number of parentless nodes: when a parentless node B gets a parent A through the rule $A \rightarrow B$, it is no longer parentless, but the node for A now is, and, what is worse, the number of nodes has gone up by one. And it may be necessary to repeat the process, say with a rule $Z \rightarrow A$, etc. But eventually the chain of unit rules must come to an end, say at P (so we have $P \rightarrow Q \cdots Z \rightarrow A \rightarrow B$), or there would be a loop in the grammar. This means that P gets a parent node with more than one child node and the number of parentless nodes is reduced (or P is the top node). So the worst thing the unit rules can do is to "lengthen" each node by a constant factor C_u, the maximum length of a unit rule chain, and so the size of the parse tree is smaller than $2C_u n$.

If the grammar contains rules of the form $A \rightarrow \varepsilon$, only a finite number of εs can be recognized between each pair of adjacent tokens in the input, or there would again be a loop in the grammar. So the worst thing nullable rules can do is to "lengthen" the input by a constant factor C_n, the maximum number of εs recognized between two tokens, and the size of the parse tree is smaller than $2C_nC_un$, which is linear in n.

If, on the other hand, the grammar is allowed to contain loops, both the above processes can introduce unbounded stretches of nodes in the parse tree, which can then reach any size.

3.1.2 Various Kinds of Ambiguity

A sentence from a grammar can easily have more than one production tree, i.e., there can easily be more than one way to produce the sentence. From a formal point of view this is a non-issue (a set does not count how many times it contains an element), but as soon as we are interested in the semantics, the difference becomes significant. Not surprisingly, a sentence with more than one production tree is called *ambiguous*, but we must immediately distinguish between *essential ambiguity* and *spurious ambiguity*. The difference comes from the fact that we are not interested in the production trees per se, but rather in the semantics they describe. An ambiguous sentence is spuriously ambiguous if all its production trees describe the same semantics; if some of them differ in their semantics, the ambiguity is essential. The notion of "ambiguity" can also be defined for grammars: a grammar is essentially ambiguous if it can produce an essentially ambiguous sentence, spuriously ambiguous if it can produce a spuriously ambiguous sentence (but not an essentially ambiguous one) and unambiguous if it cannot do either. For testing the possible ambiguity of a grammar, see Section 9.14.

A simple ambiguous grammar is given in Figure 3.1. Note that rule 2 differs

1.	`Sum`$_S$	`-->`	`Digit`	$\{\ A_0 := A_1\ \}$
2.	`Sum`	`-->`	`Sum + Sum`	$\{\ A_0 := A_1 + A_3\ \}$
3a.	`Digit`	`-->`	`0`	$\{\ A_0 := 0\ \}$
	...			
3j.	`Digit`	`-->`	`9`	$\{\ A_0 := 9\ \}$

Fig. 3.1. A simple ambiguous grammar

from that in Figure 2.30. Now **`3+5+1`** has two production trees (Figure 3.2) but the semantics is the same in both cases: 9. The ambiguity is spurious. If we change the `+` into a `-`, however, the ambiguity becomes essential, as seen in Figure 3.3. The unambiguous grammar in Figure 2.30 remains unambiguous and retains the correct semantics if `+` is changed into `-`.

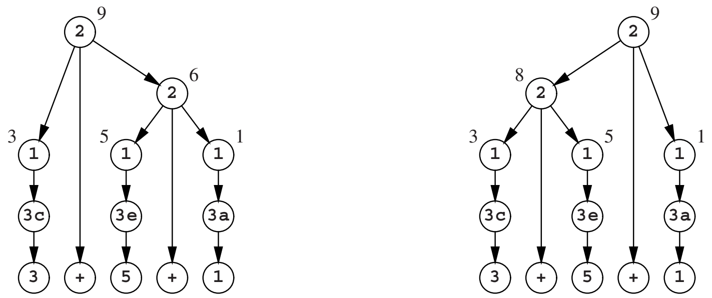


Fig. 3.2. Spurious ambiguity: no change in semantics

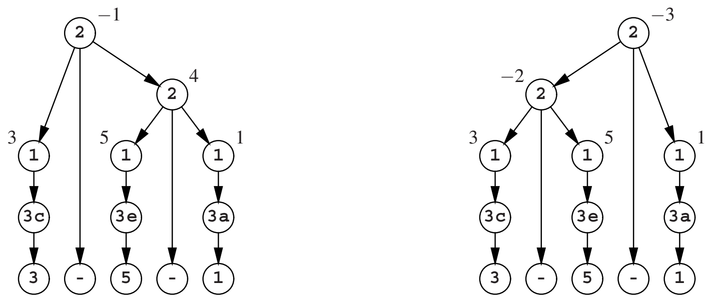


Fig. 3.3. Essential ambiguity: the semantics differ

Strangely enough, languages can also be ambiguous: there are (context-free) languages for which there is no unambiguous grammar. Such languages are *inherently ambiguous*. An example is the language $L = \mathtt{a}^m\mathtt{b}^n\mathtt{c}^n \cup \mathtt{a}^p\mathtt{b}^p\mathtt{c}^q$. Sentences in L consist either of a number of **a**s followed by a nested sequence of **b**s and **c**s, or of a nested sequence of **a**s and **b**s followed by a number of **c**s. Example sentences are: **abcc**, **aabbc**, and **aabbcc**; **abbc** is an example of a non-sentence. L is produced by the grammar of Figure 3.4.

S_s	→	AB | DC
A	→	a | aA
B	→	bc | bBc
D	→	ab | aDb
C	→	c | cC

Fig. 3.4. Grammar for an inherently ambiguous language

Intuitively, it is reasonably clear why L is inherently ambiguous: any part of the grammar that produces $\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n$ cannot avoid producing $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, and any part of the grammar that produces $\mathtt{a}^p\mathtt{b}^p\mathtt{c}^q$ cannot avoid producing $\mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. So whatever we do, forms with equal numbers of **a**s, **b**s, and **c**s will always be produced twice. Formally proving that there is really no way to get around this is beyond the scope of this book.

3.1.3 Linearization of the Parse Tree

Often it is inconvenient and unnecessary to construct the actual parse tree: a parser can produce a list of rule numbers instead, which means that it *linearizes* the parse tree. There are three main ways to linearize a tree, prefix, postfix and infix. In *prefix notation*, each node is listed by listing its number followed by prefix listings of the subnodes in left-to-right order; this gives us the leftmost derivation (for the right tree in Figure 3.2):

leftmost: 2 2 1 3c 1 3e 1 3a

If a parse tree is constructed according to this scheme, it is constructed in *pre-order*. In *postfix notation*, each node is listed by listing in postfix notation all the subnodes in left-to-right order, followed by the number of the rule in the node itself; this gives us the rightmost derivation (for the same tree):

rightmost: 3c 1 3e 1 2 3a 1 2

This constructs the parse tree in *post-order*. In *infix notation*, each node is listed by first giving an infix listing between parentheses of the first n subnodes, followed by the rule number in the node, followed by an infix listing between parentheses of the remainder of the subnodes; n can be chosen freely and can even differ from rule to rule, but $n = 1$ is normal. Infix notation is not common for derivations, but is occasionally useful. The case with $n = 1$ is called the *left-corner derivation*; in our example we get:

left-corner: (((3c)1) 2 ((3e)1)) 2 ((3a)1)

The infix notation requires parentheses to enable us to reconstruct the production tree from it. The leftmost and rightmost derivations can do without them, provided we have the grammar ready to find the number of subnodes for each node.

It is easy to tell if a derivation is leftmost or rightmost: a leftmost derivation starts with a rule for the start symbol, while a rightmost derivation starts with a rule that produces terminal symbols only. (If both conditions hold, there is only one rule, which is both a leftmost and a rightmost derivation.)

The existence of several different derivations should not be confused with ambiguity. The different derivations are just notational variants for one and the same production tree. No semantic significance can be attached to their differences.

3.2 Two Ways to Parse a Sentence

The basic connection between a sentence and the grammar it derives from is the parse tree, which describes how the grammar was used to produce the sentence. For the reconstruction of this connection we need a parsing technique. When we consult the extensive literature on parsing techniques, we seem to find dozens of them, yet there are only two techniques to do parsing; all the rest is technical detail and embellishment.

The first method tries to imitate the original production process by rederiving the sentence from the start symbol. This method is called *top-down*, because the parse tree is reconstructed from the top downwards.[1]

The second method tries to roll back the production process and to reduce the sentence back to the start symbol. Quite naturally this technique is called *bottom-up*.

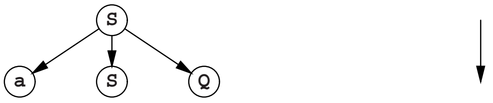
3.2.1 Top-Down Parsing

Suppose we have the monotonic grammar for the language **a**^{n}**b**^{n}**c**^{n} from Figure 2.7, which we repeat here:

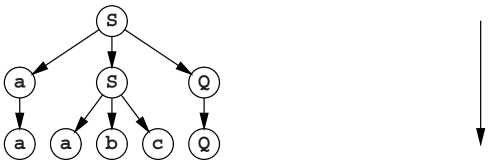
```
  Ss  -->  aSQ
   S  -->  abc
 bQc  -->  bbcc
  cQ  -->  Qc
```

and suppose the (input) sentence is **aabbcc**. First we try the top-down parsing method. We know that the production tree must start with the start symbol:

Now what could the second step be? We have two rules for **S**: **S-->aSQ** and **S-->abc**. The second rule would require the sentence to start with **ab**, which it does not. This leaves us **S-->aSQ**:

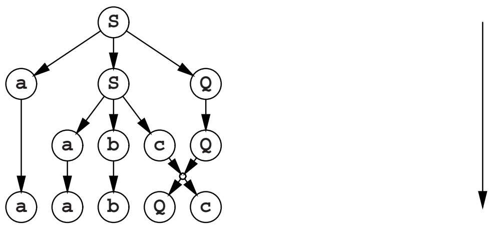


This gives us a good explanation of the first **a** in our sentence. Again two rules apply: **S-->aSQ** and **S-->abc**. Some reflection will reveal that the first rule would be a bad choice here: all production rules of **S** start with an **a**, and if we would advance to the stage **aaSQQ**, the next step would inevitably lead to **aaa...**, which contradicts the input string. The second rule, however, is not without problems either:

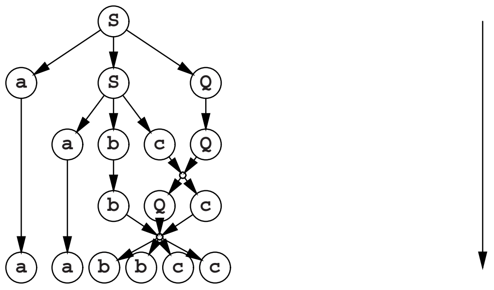


since now the sentence starts with **aabc...**, which also contradicts the input sentence. Here, however, there is a way out: **cQ-->Qc**:

[1] Trees grow from their roots downwards in computer science; this is comparable to electrons having a negative charge in physics.



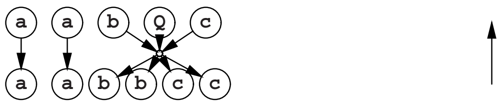
Now only one rule applies: **bQc**$\rightarrow$**bbcc**, and we obtain our input sentence (together with the parse tree):



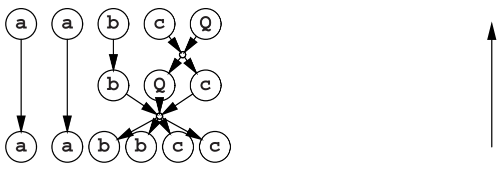
Top-down parsing identifies the production rules (and thus characterizes the parse tree) in prefix order.

3.2.2 Bottom-Up Parsing

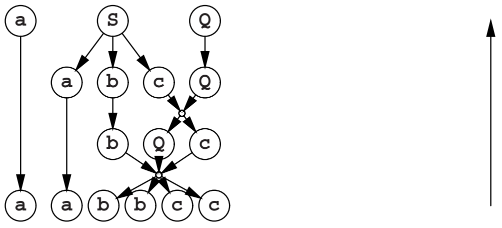
Using the bottom-up technique, we proceed as follows. One production step must have been the last and its result must still be visible in the string. We recognize the right-hand side of **bQc**$\rightarrow$**bbcc** in **aabbcc**. This gives us the final step in the production (and the first in the reduction):



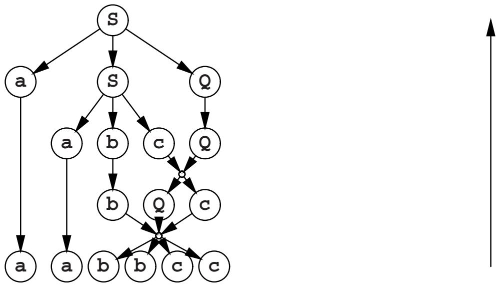
Now we recognize the **Qc** as derived by **cQ**$\rightarrow$**Qc**:



Again we find only one right-hand side: **abc**:



and also our last reduction step leaves us no choice:



Bottom-up parsing tends to identify the production rules in postfix order.

It is interesting to note that bottom-up parsing turns the parsing process into a production process. The above reduction can be viewed as a production with the reversed grammar:

```
 aSQ  -->  S
 abc  -->  S
bbcc  -->  bQc
  Qc  -->  cQ
```

augmented with a rule that turns the start symbol into a new terminal symbol:

```
S  -->  !
```

and a rule which introduces a new start symbol, the original sentence:

$$\mathtt{I}_s \rightarrow \mathtt{aabbcc}$$

If, starting from **I**, we can produce **!** we have recognized the input string, and if we have kept records of what we did, we also have obtained the parse tree.

This duality of production and reduction is used by Deussen [21] as a basis for a very fundamental approach to formal languages.

3.2.3 Applicability

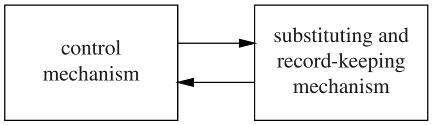
The above examples show that both the top-down and the bottom-up method will work under certain circumstances, but also that sometimes quite subtle considerations are involved, of which it is not at all clear how we can teach them to a computer.

Almost the entire body of parser literature is concerned with formalizing these subtle considerations, and there has been considerable success.

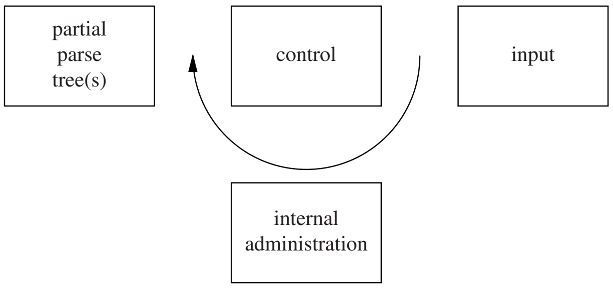
3.3 Non-Deterministic Automata

Both examples above feature two components: a machine that can make substitutions and record a parse tree, and a control mechanism that decides which moves the machine should make. The machine is relatively simple since its substitutions are restricted to those allowed by the grammar, but the control mechanism can be made arbitrarily complex and may incorporate extensive knowledge of the grammar.

This structure can be discerned in all parsing methods: there is always a substituting and record-keeping machine, and a guiding control mechanism:



The substituting machine is called a *non-deterministic automaton* or *NDA*; it is called "non-deterministic" because it often has several possible moves and the particular choice is not predetermined, and an "automaton" because it automatically performs actions in response to stimuli. It manages three components: the input string (actually a copy of it), the partial parse tree and some internal administration. Every move of the NDA transfers some information from the input string through the administration to the partial parse tree. Each of the three components may be modified in the process:



The great strength of an NDA, and the main source of its usefulness, is that it can easily be constructed so that it can only make "correct" moves, that is, moves that keep the system of partially processed input, internal administration and partial parse tree consistent. This has the consequence that we may move the NDA any way we choose: it may move in circles, it may even get stuck, but if it ever gives us an answer, in the form of a finished parse tree, that answer will be correct. It is also essential that the NDA *can* make all correct moves, so that it can produce all parsings if the control mechanism is clever enough to guide the NDA there. This property of the NDA is also easily arranged.

The inherent correctness of the NDA allows great freedom to the control mechanism, the "control" for short. It may be naive or sophisticated, it may be cumbersome or it may be efficient, it may even be wrong, but it can never cause the NDA to produce an incorrect parsing; and that is a comforting thought. If it is wrong it may, however, cause the NDA to miss a correct parsing, to loop infinitely, or to get stuck in a place where it should not.

3.3.1 Constructing the NDA

The NDA derives directly from the grammar. For a top-down parser its moves consist essentially of the production rules of the grammar and the internal administration is initially the start symbol. The control moves the machine until the internal administration is equal to the input string; then a parsing has been found. For a bottom-up parser the moves consist essentially of the reverse of the production rules of the grammar (see Section 3.2.2) and the internal administration is initially the input string. The control moves the machine until the internal administration is equal to the start symbol; then a parsing has been found. A left-corner parser works like a top-down parser in which a carefully chosen set of production rules has been reversed and which has special moves to undo this reversion when needed.

3.3.2 Constructing the Control Mechanism

Constructing the control of a parser is quite a different affair. Some controls are independent of the grammar, some consult the grammar regularly, some use large tables precomputed from the grammar and some even use tables computed from the input string. We shall see examples of each of these: the "hand control" that was demonstrated at the beginning of this section comes in the category "consults the grammar regularly", backtracking parsers often use a grammar-independent control, LL and LR parsers use precomputed grammar-derived tables, the CYK parser uses a table derived from the input string and Earley's and GLR parsers use several tables derived from the grammar and the input string.

Constructing the control mechanism, including the tables, from the grammar is almost always done by a program. Such a program is called a *parser generator*; it is fed the grammar and perhaps a description of the terminal symbols and produces a program which is a parser. The parser often consists of a driver and one or more tables, in which case it is called *table-driven*. The tables can be of considerable size and of extreme complexity.

The tables that derive from the input string must of course be computed by a routine that is part of the parser. It should be noted that this reflects the traditional setting in which a large number of different input strings is parsed according to a relatively static and unchanging grammar. The inverse situation is not at all unthinkable: many grammars are tried to explain a given input string, for example an observed sequence of events.

3.4 Recognition and Parsing for Type 0 to Type 4 Grammars

Parsing a sentence according to a grammar is in principle always possible provided we know in advance that the string indeed derives from the grammar. If we cannot think of anything better, we can just run the general production process of Section 2.4.1 on the grammar and sit back and wait until the sentence turns up (and we know it will). This by itself is not exactly enough: we must extend the production process a little, so that each sentential form carries its own partial production tree, which must be updated at the appropriate moments, but it is clear that this can be done with some programming effort. We may have to wait a little while (say a couple of million years) for the sentence to show up, but in the end we will surely obtain the parse tree. All this is of course totally impractical, but it still shows us that at least theoretically any string can be parsed *if* we know it is parsable, regardless of the grammar type.

3.4.1 Time Requirements

When parsing strings consisting of more than a few symbols, it is important to have some idea of the *time requirements* of the parser, i.e., the dependency of the time required to finish the parsing on the number of symbols in the input string. Expected lengths of input range from some tens (sentences in natural languages) to some tens of thousands (large computer programs); the length of some input strings may even be virtually infinite (the sequence of buttons pushed on a coffee vending machine over its life-time). The dependency of the time requirements on the input length is also called *time complexity*.

Several characteristic time dependencies can be recognized. A time dependency is *exponential* if each following input symbol multiplies the required time by a constant factor, say 2: each additional input symbol doubles the parsing time. Exponential time dependency is written $O(C^n)$ where C is the constant multiplication factor. Exponential dependency occurs in the number of grains doubled on each field of the famous chess board; this way lies bankruptcy.

A time dependency is *linear* if each following input symbol takes a constant amount of time to process; doubling the input length doubles the processing time. This is the kind of behavior we like to see in a parser; the time needed for parsing is proportional to the time spent on reading the input. So-called *real-time parsers* behave even better: they can produce the parse tree within a constant time after the last input symbol was read; given a fast enough computer they can keep up indefinitely with an input stream of constant speed. Note that this is not necessarily true of linear-time parsers: they can in principle read the entire input of n symbols and then take a time proportional to n to produce the parse tree.

Linear time dependency is written $O(n)$. A time dependency is called *quadratic* if the processing time is proportional to the square of the input length (written $O(n^2)$) and *cubic* if it is proportional to the third power (written $O(n^3)$). In general, a dependency that is proportional to any power of n is called *polynomial* (written $O(n^p)$).

3.4.2 Type 0 and Type 1 Grammars

It is a remarkable result in formal linguistics that the *recognition* problem for a arbitrary Type 0 grammar is unsolvable. This means that there cannot be an algorithm that accepts an arbitrary Type 0 grammar and an arbitrary string and tells us in finite time whether the grammar can produce the string or not. This statement can be proven, but the proof is very intimidating and, what is worse, it does not provide any insight into the cause of the phenomenon. It is a proof by contradiction: we can prove that, if such an algorithm existed, we could construct a second algorithm of which we can prove that it only terminates if it never terminates. Since this is a logical impossibility and since all other premises that went into the intermediate proof are logically sound we are forced to conclude that our initial premise, the existence of a recognizer for Type 0 grammars, is a logical impossibility. Convincing, but not food for the soul. For the full proof see Hopcroft and Ullman [391, pp. 182-183], or Révész [394, p. 98].

It is quite possible to construct a recognizer that works for a certain number of Type 0 grammars, using a certain technique. This technique, however, will not work for all Type 0 grammars. In fact, however many techniques we collect, there will always be grammars for which they do not work. In a sense we just cannot make our recognizer complicated enough.

For Type 1 grammars, the situation is completely different. The seemingly inconsequential property that Type 1 production rules cannot make a sentential form shrink allows us to construct a control mechanism for a bottom-up NDA that will at least work in principle, regardless of the grammar. The internal administration of this control consists of a set of sentential forms that could have played a role in the production of the input sentence; it starts off containing only the input sentence. Each move of the NDA is a reduction according to the grammar. Now the control applies all possible moves of the NDA to all sentential forms in the internal administration in an arbitrary order, and adds each result to the internal administration if it is not already there. It continues doing so until each move on each sentential form results in a sentential form that has already been found. Since no move of the NDA can make a sentential form longer (because all right-hand sides are at least as long as their left-hand sides) and since there are only a finite number of sentential forms as long as or shorter than the input string, this must eventually happen. Now we search the sentential forms in the internal administration for one that consists solely of the start symbol. If it is there, we have recognized the input string; if it is not, the input string does not belong to the language of the grammar. And if we still remember, in some additional administration, how we got this start symbol sentential form, we have obtained the parsing. All this requires a lot of book-keeping, which we are not going to discuss, since nobody does it this way anyway.

To summarize the above, we cannot always construct a parser for a Type 0 grammar, but for a Type 1 grammar we always can. The construction of a practical and reasonably efficient parser for these types of grammars is a very difficult subject on which slow but steady progress has been made during the last 40 years (see (Web)Section 18.1.1). It is not a hot research topic, mainly because Type 0 and Type

1 grammars are well-known to be human-unfriendly and will never see wide application. Yet it is not completely devoid of usefulness, since a good parser for Type 0 grammars would probably make a good starting point for a theorem prover.[2]

The human-unfriendliness consideration does not apply to two-level grammars. Having a practical parser for two-level grammars would be marvelous, since it would allow parsing techniques (with all their built-in automation) to be applied in many more areas than today, especially where context conditions are important. The present possibilities for two-level grammar parsing are discussed in Section 15.2.3.

All known parsing algorithms for Type 0, Type 1 and unrestricted two-level grammars have exponential time dependency.

3.4.3 Type 2 Grammars

Fortunately, much better parsing algorithms are known for CF (Type 2) grammars than for Type 0 and Type 1. Almost all practical parsing is done using CF and FS grammars, and almost all problems in context-free parsing have been solved. The cause of this large difference can be found in the locality of the CF production process: the evolution of one non-terminal in the sentential form is totally independent of the evolution of any other non-terminal, and, conversely, during parsing we can combine partial parse trees regardless of their histories. Neither is true in a context-sensitive grammar.

Both the top-down and the bottom-up parsing processes are readily applicable to CF grammars. In the examples below we shall use the simple grammar

```
SentenceS    -->  Subject Verb Object
Subject      -->  the Noun | a Noun | ProperName
Object       -->  the Noun | a Noun | ProperName
Verb         -->  bit | chased
Noun         -->  cat | dog
ProperName   -->  ···
```

3.4.3.1 Top-Down CF Parsing

In top-down CF parsing we start with the start symbol and try to produce the input. The keywords here are *predict* and *match*. At any time there is a leftmost non-terminal *A* in the sentential form and the parser tries systematically to predict a fitting alternative for *A*, as far as compatible with the symbols found in the input at the position where the result of *A* should start. This leftmost non-terminal is also called the *goal* of the prediction process.

Consider the example of Figure 3.5, where `Object` is the leftmost non-terminal, the "goal". In this situation, the parser will first predict `the Noun` for `Object`, but will immediately reject this alternative since it requires `the` where the input has `a`. Next, it will try `a Noun`, which is temporarily accepted. The `a` is matched and

[2] A theorem prover is a program that, given a set of axioms and a theorem, proves or disproves the theorem without or with minimal human intervention.

```
Input:                           the  cat  bit  a dog
Sentential form:                 the  cat  bit  Object
(the internal administration)
```

Fig. 3.5. Top-down parsing as the imitation of the production process

the new leftmost non-terminal is **`Noun`**. This parse will succeed when **`Noun`** eventually produces **`dog`**. The parser will then attempt a third prediction for **`Object`**, **`ProperName`**; this alternative is not immediately rejected as the parser cannot see that **`ProperName`** cannot start with **`a`**. It will fail at a later stage.

There are two serious problems with this approach. Although it can, in principle, handle arbitrary CF grammars, it will loop on some grammars if implemented naively. This can be avoided by using some special techniques, which result in general top-down parsers; these are treated in detail in Chapter 6. The second problem is that the algorithm requires exponential time since any of the predictions may turn out wrong and may have to be corrected by trial and error. The above example shows that some efficiency can be gained by preprocessing the grammar: it is advantageous to know in advance what tokens can start **`ProperName`**, to avoid predicting an alternative that is doomed in advance. This is true for most non-terminals in the grammar and this kind of information can be easily computed from the grammar and stored in a table for use during parsing. For a reasonable set of grammars, linear time dependency can be achieved, as explained in Chapter 8.

3.4.3.2 Bottom-Up CF Parsing

In bottom-up CF parsing we start with the input and try to reduce it to the start symbol. Here the keywords are *shift* and *reduce*. When we are in the middle of the process, we have in our hands a sentential form reduced from the input. Somewhere in this sentential form there must be a segment (a substring) that was the result of the last production step that produced this sentential form. This segment corresponds to the right-hand side α of a production rule $A \rightarrow \alpha$ and must now be reduced to A. The segment and the production rule together are called the *handle* of the sentential form, a quite fitting expression; see Figure 3.6. (When the production rule is obvious

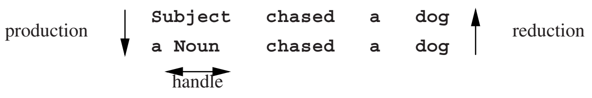


Fig. 3.6. Bottom-up parsing as the inversion of the production process

from the way the segment was found, the matching segment alone is often called the "handle". We will usually follow this custom, but call the matching segment the *handle segment* when we feel that that is clearer.)

The trick is to find the handle. It must be the right-hand side of a rule, so we start looking for such a right-hand side by shifting symbols from the sentential form into the internal administration. When we find a right-hand side we reduce it to its left-hand side and repeat the process, until only the start symbol is left. We will not always find the correct handle this way; if we err, we will get stuck further on, will have to undo some steps, shift in more symbols and try again. In Figure 3.6 we could have reduced the **`a Noun`** to **`Object`**, thereby boldly heading for a dead end.

There are essentially the same two problems with this approach as with the top-down technique. It may loop, and will do so on grammars with ε-rules: it will continue to find empty productions all over the place. This can be remedied by touching up the grammar. And it can take exponential time, since the correct identification of the handle may have to be done by trial and error. Again, doing preprocessing on the grammar often helps: it is easy to see from the grammar that **`Subject`** can be followed by **`chased`**, but **`Object`** cannot. So it is unprofitable to reduce a handle to **`Object`** if the next symbol is **`chased`**.

3.4.4 Type 3 Grammars

A right-hand side in a regular grammar contains at most one non-terminal, so there is no difference between leftmost and rightmost derivation. Top-down methods are much more efficient for right-regular grammars; bottom-up methods work better for left-regular grammars. When we take the production tree of Figure 2.15 and if we turn it 45° counterclockwise, we get the production chain of Figure 3.7. The se-

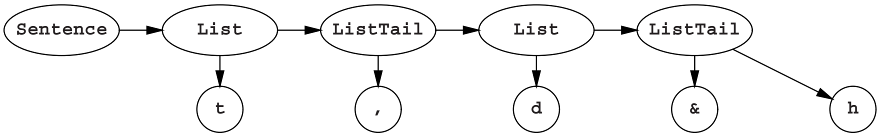


Fig. 3.7. The production tree of Figure 2.15 as a production chain

quence of non-terminals rolls on to the right, producing terminal symbols as they go. In parsing, we are given the terminal symbols and are supposed to construct the sequence of non-terminals. The first one is given, the start symbol (hence the preference for top-down). If only one rule for the start symbol starts with the first symbol of the input we are lucky and know which way to go. Very often, however, there are many rules starting with the same symbol and then we are in need of more wisdom. As with Type 2 grammars, we can of course find the correct continuation by trial and error, but far more efficient methods exist that can handle any regular grammar. Since they form the basis of some advanced parsing techniques, they are treated separately, in Chapter 5.

3.4.5 Type 4 Grammars

Finite-choice (FC) grammars do not involve production trees, and membership of a given input string in the language of the FC grammar can be determined by simple look-up. This look-up is generally not considered to be "parsing", but is still mentioned here for two reasons. First, it can benefit from parsing techniques, and second, it is often required in a parsing environment. Natural languages have some categories of words that have only a very limited number of members; examples are the pronouns, the prepositions and the conjunctions. It is often important to decide quickly if a given word belongs to one of these finite-choice categories or will have to be analysed further. The same applies to reserved words in a programming language.

One approach is to consider the FC grammar as a regular grammar and apply the techniques of Chapter 5. This is often amazingly efficient.

Another often-used approach is to use a hash table. See any book on algorithms, for example Cormen et al. [415], or Goodrich and Tamassia [416].

3.5 An Overview of Context-Free Parsing Methods

Among the Chomsky grammar types the context-free (Type 2) grammars occupy the most prominent position. This has three reasons: 1. CF parsing results in trees, which allow semantics to be expressed and combined easily; 2. CF languages cover a large part of the languages one would like to process automatically; 3. efficient CF parsing is possible – though sometimes with great difficulty. The context-free grammars are followed immediately by the finite-state grammars in importance. This is because the world and especially equipment is finite; vending machines, remote controls, virus detectors, all exhibit finite-state behavior. The rest of the chapters in this book will therefore be mainly concerned with CF parsing, with the exception of Chapter 5 (finite-state grammars) and Chapter 15 (non-Chomsky systems). We shall now give an overview of the context-free parsing methods.

The reader of literature about parsing is confronted with a large number of techniques with often unclear interrelationships. Yet all techniques can be placed in a single framework, according to some simple criteria; they are summarized in Figure 3.11.

We have already seen that a parsing technique is either top-down, reproducing the input string from the start symbol, or bottom-up, reducing the input to the start symbol. The next division is that between directional and non-directional parsing methods.

3.5.1 Directionality

Non-directional methods construct the parse tree while accessing the input in any order they see fit. This of course requires the entire input to be in memory before parsing can start. There is a top-down and a bottom-up version. *Directional* parsers access the input tokens one by one in order, all the while updating the partial parse tree(s). There is again a top-down and a bottom-up version.

3.5.1.1 Non-Directional Methods

The non-directional top-down method is simple and straightforward and has probably been invented independently by many people. To the best of our knowledge it was first described by Unger [12] in 1968, but in his article he gives the impression that the method already existed. The method has not received much attention in the literature but is more important than one might think, since it is used anonymously in a number of other parsers. We shall call it Unger's method; it is described in Section 4.1.

The non-directional bottom-up method has also been discovered independently by a number of people, among whom Cocke (in Hays [3, Sect. 17.3.1]), Younger [10], and Kasami [13]; an earlier description is by Sakai [5]. It is named CYK (or sometimes CKY) after the three best-known inventors. It has received considerable attention since its naive implementation is much more efficient than that of Unger's method. The efficiency of both methods can be improved, however, arriving at roughly the same performance; see Sheil [20]. The CYK method is described in Section 4.2.

Non-directional methods usually first construct a data structure which summarizes the grammatical structure of the input sentence. Parse trees can then be derived from this data structure in a second stage.

3.5.1.2 Directional Methods

The directional methods process the input symbol by symbol, from left to right. (It is also possible to parse from right to left, using a mirror image of the grammar; this is occasionally useful.) This has the advantage that parsing can start, and indeed progress, considerably before the last symbol of the input is seen. The directional methods are all based explicitly or implicitly on the parsing automaton described in Section 3.4.3, where the top-down method performs predictions and matches and the bottom-up method performs shifts and reduces.

Directional methods can usually construct the (partial) parse tree as they proceed through the input string, unless the grammar is ambiguous and some postprocessing may be required.

3.5.2 Search Techniques

A third way to classify parsing techniques concerns the search technique used to guide the (non-deterministic!) parsing automaton through all its possibilities to find one or all parsings.

There are in general two methods for solving problems in which there are several alternatives in well-determined points: *depth-first search*, and *breadth-first search*.

- In depth-first search we concentrate on one half-solved problem. If the problem bifurcates at a given point P, we store one alternative for later processing and keep concentrating on the other alternative. If this alternative turns out to be a failure (or even a success, but we want all solutions), we roll back our actions to point *P* and continue with the stored alternative. This is called *backtracking*.

- In breadth-first search we keep a set of half-solved problems. From this set we compute a new set of (better) half-solved problems by examining each old half-solved problem; for each alternative, we create a copy in the new set. Eventually, the set will come to contain all solutions.

Depth-first search has the advantage that it requires an amount of memory that is proportional to the size of the problem, unlike breadth-first search, which may require exponential memory. Breadth-first search has the advantage that it will find the simplest solution first. Both methods require in principle exponential time. If we want more efficiency (and exponential requirements are virtually unacceptable), we need some means to restrict the search. See any book on algorithms, for example Sedgewick [417] or Goodrich and Tamassia [416], for more information on search techniques.

These search techniques are not at all restricted to parsing and can be used in a wide array of contexts. A traditional one is that of finding an exit from a maze. Figure 3.8(*a*) shows a simple maze with one entrance and two exits. Figure 3.8(*b*) depicts the path a depth-first search will take; this is the only option for the human maze-walker: he cannot duplicate himself and the maze. Dead ends make the depth-first search backtrack to the most recent untried alternative. If the searcher will also backtrack at each exit, he will find all exits. Figure 3.8(*c*) shows which rooms are

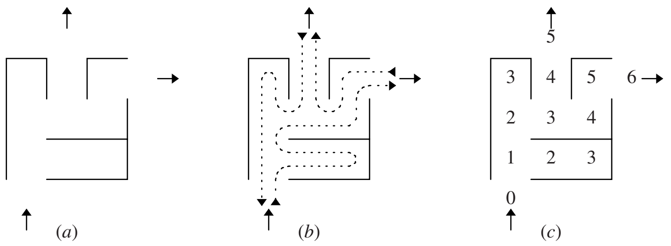


Fig. 3.8. A simple maze with depth-first and breadth-first visits

examined in each stage of the breadth-first search. Dead ends (in stage 3) cause the search branches in question to be discarded. Breadth-first search will find the shortest way to an exit (the shortest solution) first. If it continues until there are no branches left, it will find all exits (all solutions).

3.5.3 General Directional Methods

The idea that parsing is the reconstruction of the production process is especially clear when using a directional method. It is summarized in the following two sound bites.

> A directional top-down (left-to-right) CF parser identifies leftmost productions in production order.

and

> A directional bottom-up (left-to-right) CF parser identifies rightmost productions in reverse production order.

We will use the very simple grammar

$$\begin{array}{lll}
\mathtt{S}_\mathrm{s} & \to & \mathtt{P\ Q\ R} \\
\mathtt{P} & \to & \mathtt{p} \\
\mathtt{Q} & \to & \mathtt{q} \\
\mathtt{R} & \to & \mathtt{r}
\end{array}$$

to demonstrate this. The grammar produces only one string, **`pqr`**.

The leftmost production process for **`pqr`** proceeds as follows:

```
   |S
1
   |P Q R
2
   p |Q R
3
   p q |R
4
   p q r |
```

where the | indicates how far the production process has proceeded. The top-down analysis mimics this process by first identifying the rule that produced the **`p`**, **`P-->p`**, then the one for **`q`**, etc.:

$$|\mathtt{S} \ \overset{\mathtt{S\to PQR}}{\underset{(1)}{\Rightarrow}}\ |\mathtt{PQR} \ \overset{\mathtt{P\to p}}{\underset{(2)}{\Rightarrow}}\ \mathtt{p}|\mathtt{QR} \ \overset{\mathtt{Q\to q}}{\underset{(3)}{\Rightarrow}}\ \mathtt{pq}|\mathtt{R} \ \overset{\mathtt{R\to r}}{\underset{(4)}{\Rightarrow}}\ \mathtt{pqr}|$$

The rightmost production process for **`pqr`** proceeds as follows:

```
   S|
1
   P Q R|
2
   P Q| r
3
   P| q r
4
   | p q r
```

where the | again indicates how far the production process has proceeded. The bottom-up analysis rolls back this process. To do this, it must first identify the rule in production step 4, **`P-->p`** and use it as a reduction, then step 3, **`Q-->q`**, etc. Fortunately the parser can easily do this, because the rightmost production process makes the boundary between the unprocessed and the processed part of the sentential form creep to the left, so the last production brings it to the left end of the result, as we see above. The parsing process can then start picking it up there:

$$|\mathtt{pqr} \ \overset{\mathtt{P\to p}}{\underset{(4)}{\Rightarrow}}\ \mathtt{P}|\mathtt{qr} \ \overset{\mathtt{Q\to q}}{\underset{(3)}{\Rightarrow}}\ \mathtt{PQ}|\mathtt{r} \ \overset{\mathtt{R\to r}}{\underset{(2)}{\Rightarrow}}\ \mathtt{pqr}| \ \overset{\mathtt{S\to pqr}}{\underset{(4)}{\Rightarrow}}\ \mathtt{S}|$$

This double reversal is inherent in directional bottom-up parsing.

The connection between parse trees under construction and sentential forms is shown in Figure 3.9, where the dotted lines indicate the sentential forms. On the left

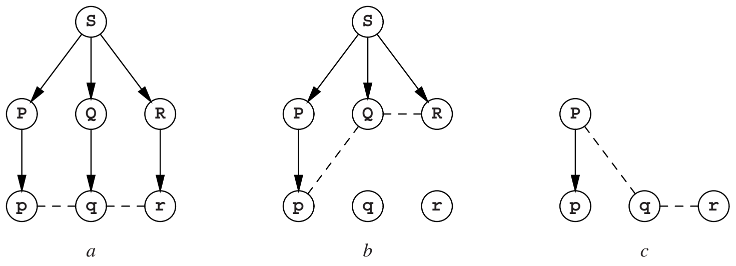


Fig. 3.9. Sentential forms in full parse tree (*a*), during top-down (*b*), and during bottom-up (*c*)

we have the complete parse tree; the corresponding sentential form is the string of terminals. The middle diagram shows the partial parse tree in a top-down parser after the `p` has been processed. The sentential form corresponding to this situation is `pQR`. It resulted from the two productions `S` $\Rightarrow$ `PQR` $\Rightarrow$ `pQR`, which gave rise to the partial parse tree. The diagram on the right shows the partial parse tree after the `p` has been processed in a bottom-up parser. The corresponding sentential form is `Pqr`, resulting from `pqr` $\Leftarrow$ `Pqr`; the single reduction gave rise to a partial parse tree of only one node.

Combining depth-first or breadth-first with top-down or bottom-up gives four classes of parsing techniques. The top-down techniques are treated in Chapter 6. The depth-first top-down technique allows a very simple implementation called recursive descent; this technique, which is explained in Section 6.6, is very suitable for writing parsers by hand. Since depth-first search is built into the Prolog language, recursive descent parsers for a large number of grammars can be formulated very elegantly in that language, using a formalism called “Definite Clause Grammars” (Section 6.7). The applicability of this technique can be extended to cover all grammars by using a device called “cancellation” (Section 6.8).

The bottom-up techniques are treated in Chapter 7. The combination of breadth-first and bottom-up leads to the class of Earley parsers, which have among them some very effective and popular parsers for general CF grammars (Section 7.2). A formally similar but implementationwise quite different approach leads to “chart parsing” (Section 7.3).

Sudkamp [397, Chapter 4] gives a full formal explanation of [breadth-first | depth-first][top-down | bottom-up] context-free parsing.

3.5.4 Linear Methods

Most of the general search methods indicated in the previous section have exponential time dependency in the worst case: each additional symbol in the input multiplies the parsing time by a constant factor. Such methods are unusable except for very small input length, where 20 symbols is about the maximum. Even the best variants of the above methods require cubic time in the worst case: for 10 tokens they perform 1000 actions, for 100 tokens 1 000 000 actions and for 10 000 tokens (a fair-sized computer program file) 10^{12} actions, which even at 10 nanoseconds per action will already take almost 3 hours. It is clear that for real speed we should like to have a linear-time general parsing method. Unfortunately no such method has been discovered to date, and although there is no proof that such a method could not exist, there are strong indications that that is the case; see Section 3.10 for details. Compare this to the situation around unrestricted phrase structure parsing, where it has been proved that no algorithm for it can exist (see Section 3.4.2).

So, in the meantime, and probably forever, we shall have to drop one of the two adjectives from our goal, a linear-time general parser. We can have a general parser, which will need cubic time at best, or we can have a linear-time parser, which will not be able to handle all CF grammars, but not both. Fortunately there are linear-time parsing methods (in particular LR parsing) that can handle very large classes of grammars but still, a grammar that is designed without regard for a parsing method and just describes the intended language in the most natural way has a small chance of allowing linear parsing automatically. In practice, grammars are often first designed for naturalness and then adjusted by hand to conform to the requirements of an existing parsing method. Such an adjustment is usually relatively simple, depending on the parsing method chosen. In short, making a linear-time parser for an arbitrary given grammar is 10% hard work; the other 90% can be done by computer.

We can achieve linear parsing time by restricting the number of possible moves of our non-deterministic parsing automaton to one in each situation. Since the moves of such an automaton involve no choice, it is called a "deterministic automaton".

The moves of a deterministic automaton are determined unambiguously by the input stream (we can speak of a stream now, since the automaton operates from left to right). A consequence of this is that a deterministic automaton can give only one parsing for a sentence. This is all right if the grammar is unambiguous, but if it is not, the act of making the automaton deterministic has pinned us down to one specific parsing. We shall say more about this in Sections 8.2.5.3 and 9.9.

All that remains is to explain how a deterministic control mechanism for a parsing automaton can be derived from a grammar. Since there is no single good solution to the problem, it is not surprising that quite a number of sub-optimal solutions have been found. From a very global point of view they all use the same technique: they analyse the grammar in depth to bring to the surface information that can be used to identify dead ends. These are then closed. If the method, applied to a grammar, closes enough dead ends so that no choices remain, the method succeeds for that grammar and gives us a linear-time parser. Otherwise it fails and we either have to look for a different method or adapt our grammar to the method.

A (limited) analogy with the maze problem can perhaps make this clearer. If we are allowed to do preprocessing on the maze (unlikely but instructive) the following method will often make our search through it deterministic. We assume that the maze consists of a grid of square rooms, as shown in Figure 3.10(*a*). Depth-first search would find a passage through the maze in 13 moves (Figure 3.10(*b*)). Now we preprocess the maze as follows: if there is a room with three walls, add the fourth wall, and continue with this process until no rooms with three walls are left. If all rooms now have either two or four walls, there are no choices left and our method

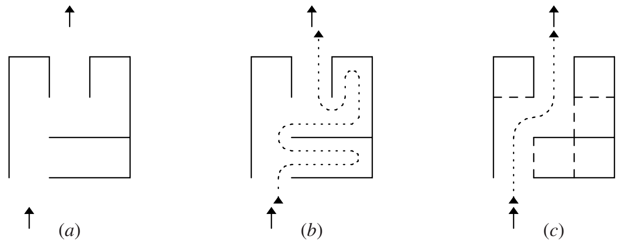


Fig. 3.10. A single-exit maze made deterministic by preprocessing

has succeeded; see Figure 3.10(c), where the passage now takes 5 moves, with no searching involved. We see how this method brings information about dead ends to the surface, to help restrict the choice.

It should be pointed out that the above analogy is a limited one. It is concerned with only one object, the maze, which is preprocessed. In parsing we are concerned with two objects, the grammar, which is static and can be preprocessed, and the input, which varies. (But see Problem 3.6 for a way to extend the analogy.)

Returning to the parsing automaton, we can state the fact that it is deterministic more precisely: a parsing automaton is *deterministic* with *look-ahead k* if its control mechanism can, given the internal administration and the next k symbols of the input, decide unambiguously what to do next — to either match or predict and what to predict in the top-down case, and to either shift or reduce and how to reduce in the bottom-up case.

It stands to reason that a deterministic automaton creates a linear-time parser, but this is not completely obvious. The parser may know in finite time what to do in each step, but many steps may have to be executed for a given input token. More specifically, some deterministic techniques can require k steps for a given position k, which suggests that quadratic behavior is possible (see Problem 3.5). But each parsing step either creates a node (predict and reduce) or consumes an input token (match and shift). Both actions can only be performed $O(n)$ times where n is the length of the input: the first because the size of the parse tree is only $O(n)$ and the second because there are only n input tokens. So however the actions of the various tasks are distributed, their total cannot exceed $O(n)$.

Like grammar types, deterministic parsing methods are indicated by initials, like LL, LALR, etc. If a method X uses a look-ahead of k symbols it is called $X(k)$. All deterministic methods require some form of grammar preprocessing to derive the parsing automaton, plus a parsing algorithm or driver to process the input using that automaton.

3.5.5 Deterministic Top-Down and Bottom-Up Methods

There is only one deterministic top-down method; it is called *LL*. The first L stands for Left-to-right, the second for "identifying the Leftmost production", as directional top-down parsers do. LL parsing is treated in Chapter 8. LL parsing, especially LL(1) is very popular. LL(1) parsers are often generated by a parser generator but a simple variant can, with some effort, be written by hand, using recursive-descent techniques; see Section 8.2.6. Occasionally, the LL(1) method is used starting from the last token of the input backwards; it is then called *RR(1)*.

There are quite a variety of deterministic bottom-up methods, the most powerful being called *LR*, where again the L stands for Left-to-right, and the R stands for "identifying the Rightmost production". Linear bottom-up methods are treated in Chapter 9. Their parsers are invariably generated by a parser generator: the control mechanism of such a parser is so complicated that it is not humanly possible to construct it by hand. Some of the deterministic bottom-up methods are very popular and are perhaps used even more widely than the LL(1) method.

LR(1) parsing is more powerful than LL(1) parsing, but also more difficult to understand and less convenient. The other methods cannot be compared easily to the LL(1) method. See Chapter 17.1 for a comparison of practical parsing methods. The LR(1) method can also be applied backwards and is then called *RL(1)*.

Both methods use look-ahead to determine which actions to take. Usually this look-ahead is restricted to one token (LL(1), LR(1), etc.) or a few tokens at most, but it is occasionally helpful to allow unbounded look-ahead. This requires different parsing techniques, which results in a subdivision of the class of deterministic parsers; see Figure 3.11.

The great difference in variety between top-down and bottom-up methods is easily understood when we look more closely at the choices the corresponding parsers face. A top-down parser has by nature little choice: if a terminal symbol is predicted, it has no choice and can only ascertain that a match is present; only if a non-terminal is predicted does it have a choice in the production of that non-terminal. A bottom-up parser can always shift the next input symbol, even if a reduction is also possible (and it often has to do so). If, in addition, a reduction is possible, it may have a choice between a number of right-hand sides. In general it has more choice than a top-down parser and more powerful methods are needed to make it deterministic.

3.5.6 Non-Canonical Methods

For many practical grammars the above methods still do not yield a linear-time deterministic parser. One course of action that is often taken is to modify the grammar slightly so as to fit it to the chosen method. But this is unfortunate because the resulting parser then yields parse trees that do not correspond to the original grammar, and patching up is needed afterwards. Another alternative is to design a parser so it postpones the decisions it cannot take for lack of information and continue parsing "at half power" until the information becomes available. Such parsers are called *non-canonical* because they identify the nodes in the parse trees in non-standard, "non-canonical" order. Needless to say this requires treading carefully, and some of the most powerful, clever, and complicated deterministic parsing algorithms come in this category. They are treated in Chapter 10.

3.5.7 Generalized Linear Methods

When our attempt to construct a deterministic control mechanism fails and leaves us with a non-deterministic but almost deterministic one, we need not despair yet: we can fall back on breadth-first search to solve the remnants of non-determinism at run-time. The better our original method was, the less non-determinism will be left, the less often breadth-first search will be needed, and the more efficient our parser will be. Such parsers are called "generalized parsers"; generalized parsers have been designed for most of the deterministic methods, both top-down and bottom-up. They are described in Chapter 11. Generalized LR (or GLR) (Tomita [162]) is one of the best general CF parsers available today.

Of course, by reintroducing breadth-first search we are taking chances. The grammar and the input could conspire so that the non-determinism gets hit by each input symbol and our parser will again have exponential time dependency. In practice, however, they never do so and such parsers are very useful.

3.5.8 Conclusion

Figure 3.11 summarizes parsing techniques as they are treated in this book. Nijholt [154] paints a more abstract view of the parsing landscape, based on left-corner parsing. See Deussen [22] for an even more abstracted overview. An early systematic survey was given by Griffiths and Petrick [9].

3.6 The "Strength" of a Parsing Technique

Formally a parsing technique T_1 is stronger (more powerful) than a parsing technique T_2 if T_1 can handle all grammars T_2 can handle but not the other way around. Informally one calls one parsing technique stronger than another if to the speaker it appears to handle a larger set of grammars. Formally this is of course nonsense, since all parsing techniques can handle infinite sets of grammars, and the notion of "larger" is moot. Also, a user grammar designed without explicit aim at a particular parsing technique has an almost zero chance of being amenable to any existing technique anyway. What counts from a user point of view is the effort required to modify the "average" practical grammar so it can be handled by method T, and to undo the damage this modification causes to the parse tree. The *strength* (*power*) of parsing technique T is inversely proportional to that effort.

Almost invariably a strong parser is more complicated and takes more effort to write than a weak one. But since a parser or parser generator (see Section 17.2) needs to be written only once and then can be used as often as needed, a strong parser saves effort in the long run.

Although the notion of a "strong" parser is intuitively clear, confusion can arise when it is applied to the combination of parser and grammar. The stronger the parser is, the fewer restrictions the grammars need to obey and the "weaker" they can afford to be. Usually methods are named after the grammar and it is here where the

	Top-down	Bottom-up
Non-directional methods	Unger parser	CYK parser
Directional Methods, depth-first or breadth-first	The predict/match automaton recursive descent DCG (Definite Clause Grammars) cancellation parsing	The shift/reduce automaton Breadth-first, top-down restricted (Earley) chart parsing

Deterministic directional methods: breadth-first search, with breadth restricted to 1, bounded look-ahead	LL(k)	precedence
		bounded-right-context
		LR(k)
		LALR(k)
		SLR(k)

unbounded look-ahead	LL-regular	DR (Discriminating-Reverse) LR-regular LAR(m)

Deterministic with postponed ("non-canonical") node identification	LC(k)
	deterministic cancellation
	Partitioned LL(k)
		total precedence
		NSLR(k)
		LR(k,∞)
		Partitioned LR(k)

Generalized deterministic: maximally restricted breadth-first search	Generalized LL Generalized cancellation Generalized LC	Generalized LR

Fig. 3.11. An overview of context-free parsing techniques

confusion starts. A "strong LL(1) grammar" is more restricted than an "LL(1) grammar"; one can also say that it is more strongly LL(1). The parser for such grammars is simpler than one for (full) LL(1) grammars, and is — can afford to be — weaker. So actually a strong-LL(1) parser is weaker than an LL(1) parser. We have tried to consistently use the hyphen between "strong" and "LL(1)" to show that "strong" applies to "LL(1)" and not to "parser", but not all publications follow that convention, and the reader must be aware. The reverse occurs with "weak-precedence parsers" which are stronger than "precedence parsers" (although in that case there are other differences as well).

3.7 Representations of Parse Trees

The purpose of parsing is to obtain one or more parse trees, but many parsing techniques do not tell you in advance if there will be zero, one, several or even infinitely many parse trees, so it is a little difficult to prepare for the incoming answers. There are two things we want to avoid: being under-prepared and miss parse trees, and being over-prepared and pre-allocate an excessive amount of memory. Not much published research has gone into this problem, but the techniques encountered in the literature can be grouped into two models: the producer-consumer model and the data structure model.

3.7.1 Parse Trees in the Producer-Consumer Model

In the producer-consumer model the parser is the producer and the program using the parse trees is the consumer. As in all producer-consumer situations in computer science, the immediate question is, which is the main program and which is the subroutine.

The most esthetically pleasing answer is to have them both as equal partners, which can be done using *coroutine*s. Coroutines are explained in some books on principles of programming languages and programming techniques, for example *Advanced Programming Language Design* by R.A. Finkel (Addison-Wesley). There are also good explanations on the Internet.

In the coroutine model, the request for a new parse tree by the user and the offer of a parse tree by the parser are paired up automatically by the coroutine mechanism. The problem with coroutines is that they must be built into the programming language, and no major programming language features them. So coroutines are not a practical solution to parse tree representation.

The coroutine's modern manifestation, the thread, in which the pairing up is done by the operating system or by a light-weight version of it inside the program, is available in some major languages, but introduces the notion of parallelism which is not inherently present in parsing. The UNIX pipe has similar communication properties but is even more alien to the parsing problem.

Usually the parser is the main program and the consumer is the subroutine. Each time the parser has finished constructing a parse tree, it calls the consumer routine with a pointer to the tree as a parameter. The consumer can then decide what to do with this tree: reject it, accept it, store it for future comparison, etc. In this setup the parser can just happily produce trees, but the consumer will probably have to save state data between being called, to be able to choose between parse trees. This is the usual setup in compiler design, where there is only one parse tree and the user state saving is less of a problem.

It is also possible to have the user as the main program, but this places a heavy burden on the parser, which is now forced to keep all state data of its half-finished parsing process when delivering a parse tree. Since data on the stack cannot be saved as state data (except by draconian means) this setup is feasible only with parsing methods that do not use a stack.

With any of these setups the user still has two problems in the general case. First, when the parser produces more than one parse tree the user receives them as separate trees and may have to do considerable comparison to find the differences on which to base further decisions. Second, if the grammar is infinitely ambiguous and the parser produces infinitely many parse trees, the process does not terminate.

So the producer-consumer model is satisfactory for unambiguous grammars, but is problematic for the general case.

3.7.2 Parse Trees in the Data Structure Model

In the data structure model the parser constructs a single data structure which represents all parse trees simultaneously. Surprisingly, this can be done even for infinitely ambiguous grammars; and what is more, it can be done in a space whose size is at most proportional to the third power of the length of the input string. One says that the data structure has *cubic space dependency*.

There are two such representations: parse forests and parse-forest grammars. Although the two are fundamentally the same, they are very different conceptually and practically, and it is useful to treat them as separate entities.

3.7.3 Parse Forests

Since a forest is just a collection of trees, the naive form of a *parse forest* consists of a single node from which all trees in the parse forest are directly reachable. The two parse trees from Figure 3.2 then combine into the parse forest from Figure 3.12, where the numbers in the nodes refer to rule numbers in the grammar of Figure 3.1.

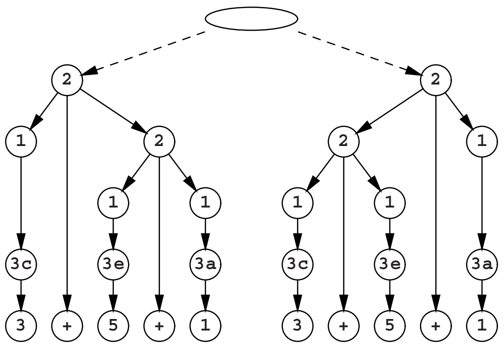


Fig. 3.12. Naive parse forest from the trees in Figure 3.2

When we look at this drawing, we notice two things: the meaning of the dashed arrows differs from that of the drawn arrows; and the resulting tree contains a lot of duplicate subtrees. One also wonders what the label in the empty top node should be.

The meaning of the dashed arrow is "or-or": the empty top node points to *either* the left node marked 2 *or* to the right node, whereas the drawn arrows mean "and-and": the left node marked 2 consists of a node marked `Sum` *and* a node marked `+` *and* a node marked `Sum`. More in particular, the empty top node, which should be labeled `Sum` points to two applications of rule 2, each of which produces `Sum + Sum`; the leftmost `Sum` points to an application of rule 1, the second `Sum` points to an

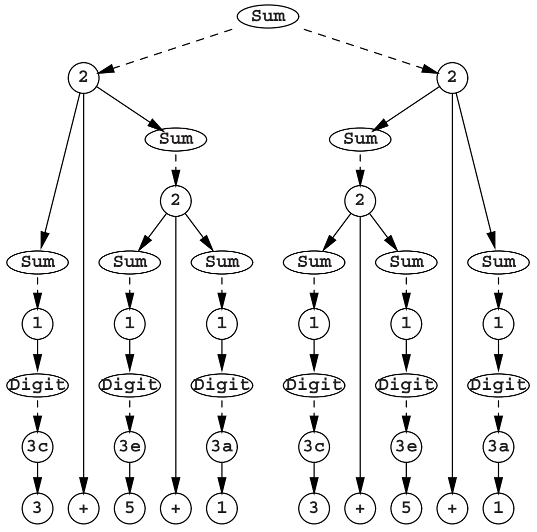


Fig. 3.13. The naive parse forest as an AND-OR tree

application of rule 2, etc. The whole *AND-OR tree* is presented in Figure 3.13, where we see an alteration of nodes labeled with non-terminals, the *OR-nodes* and nodes labeled with rule numbers, the *AND-nodes*. An OR-node for a non-terminal *A* has the rule numbers of the alternatives for *A* as children; an AND-node for a rule number has the components of the right-hand side of the rule as children.

3.7.3.1 Combining Duplicate Subtrees

We are now in a position to combine all the duplicate subtrees in the forest. We do this by having only one copy of a node labeled with a non-terminal *A* and spanning a given substring of the input. If *A* produces that substring in more than one way, more than one or-arrow will emanate from the OR-node labeled *A*, each pointing to an AND-node labeled with a rule number. In this way the AND-OR tree turns into a directed acyclic graph, a dag, which by rights should be called a *parse dag*, although the term “parse forest” is much more usual. The result of our example is shown in Figure 3.14.

It is important to note that two OR-nodes (which represent right-hand sides of rules) can only be combined if all members of the one node are the same as the corresponding members of the other node. It would not do to combine the two nodes marked 2 in Figure 3.14 right under the top; although they both read **`Sum+Sum`**, the **`Sum`**s and even the **`+`**s are not the same. If they were combined, the parse forest would represent more parse trees than correspond with the input; see Problem 3.8.

It is possible to do the combining of duplicate subtrees *during* parsing rather than afterwards, when all trees have been generated. This is of course more efficient, and has the additional advantage that it allows infinitely ambiguous parsings to be

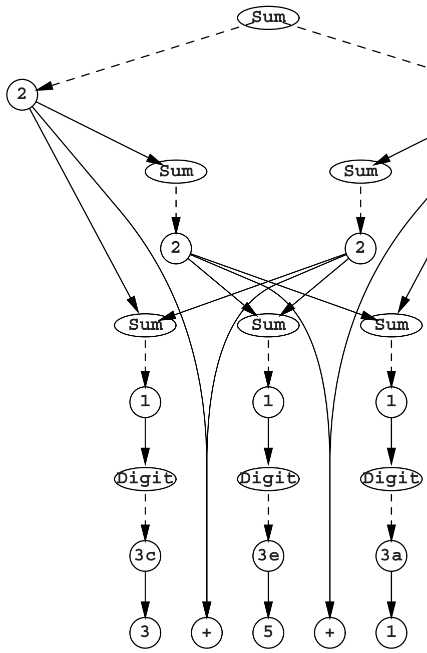


Fig. 3.14. The parse trees of Figure 3.2 as a parse forest

represented in a finite data structure. The resulting parse forest then contains loops (cycles), and is actually a *parse graph*.

Figure 3.15 summarizes the situation for the various Chomsky grammar types in relation to ambiguity. Note that finite-state and context-sensitive grammars cannot

	Most complicated data structure		
Grammar type	unambiguous	ambiguous	infinitely ambiguous
PS	dag	dag	graph
CS	dag	dag	—
CF	tree	dag	graph
FS	list	dag	—

Fig. 3.15. The data structures obtained when parsing with the Chomsky grammar types

be infinitely ambiguous because they cannot contain nullable rules. See Figure 2.16 for a similar summary of production data structures.

3.7.3.2 Retrieving Parse Trees from a Parse Forest

The receiver of a parse forest has several options. For example, a sequence of parse trees can be generated from it, or, perhaps more likely, the data structure can be pruned to eliminate parse trees on various grounds.

Generating parse trees from the parse forest is basically simple: each combination of choices for the or-or arrows is a parse tree. The implementation could be top-down and will be sketched here briefly. We do a depth-first visit of the graph, and for each OR-node we turn one of the outgoing dashed arrows into a solid arrow; we record each of these choices in a backtrack chain. When we have finished our depth-first visit we have fixed one parse tree. When we are done with this tree, we examine the most recent choice point, as provided by the last element of the backtrack chain, and make a different choice there, if available; otherwise we backtrack one step more, etc. When we have exhausted the entire backtrack chain we know we have generated all parse trees. One actualization of a parse tree is shown in Figure 3.16.

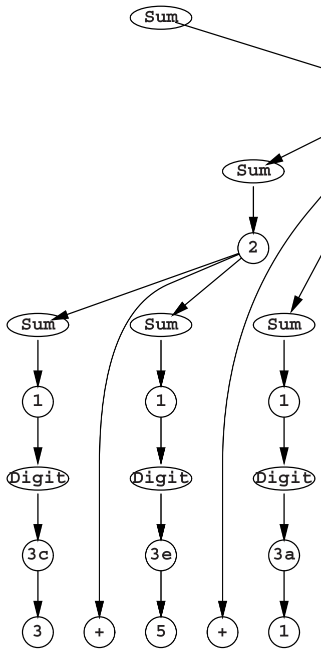


Fig. 3.16. A tree identified in the parse forest of Figure 3.14

It is usually more profitable to first prune the parse forest. How this is done depends on the pruning criteria, but the general technique is as follows. Information is attached to each node in the parse forest, in a way similar to that in attribute grammars (see Section 2.11.1). Whenever the information in a node is contradictory to the criteria for that type of node, the node is removed from the parse forest. This will often make other nodes inaccessible from the top, and these can then be removed as well.

Useful pruning of the parse forest of Figure 3.14 could be based on the fact that the + operator is left-associative, which means that **`a+b+c`** should be parsed as **`((a+b)+c)`** rather than as **`(a+(b+c))`**. The criterion would then be that for each node that has a **+** operator, its right operand cannot be a non-terminal that has a node with a **+** operator. We see that the top-left node marked 2 in Figure 3.14 violates this criterion: it has a **+** operator (in position 2) and a right operand which is a non-terminal (**Sum**) which has a node which has a node (2) which has a **+** operator (in position 4). So this node can be removed, and so can two further nodes. Again the parse tree from Figure 3.16 remains.

The above criterion is a (very) special case of an operator-precedence criterion for arithmetic expressions; see Problem 3.10 for a more general one.

3.7.4 Parse-Forest Grammars

Representing the result of parsing as a *grammar* may seem weird, far-fetched and even somewhat disappointing; after all, should one start with a grammar and a string and do all the work of parsing, just to end up with another grammar? We will, however, see that parse-forest grammars have quite a number of advantages. But none of these advantages is immediately obvious, which is probably why parse-forest grammars were not described in the literature until the late 1980s, when they were introduced by Lang [210, 220, 31], and by Billot and Lang [164]. The term "parse-forest grammar" seems to be used first by van Noord [221].

Figure 3.17 presents the parse trees of Figure 3.2 as a parse-forest grammar, and it is interesting to see how it does that. For every non-terminal *A* in the original

```
     Sum_S  -->  Sum_1_5
   Sum_1_5  -->  Sum_1_1 + Sum_3_3
   Sum_1_5  -->  Sum_1_3 + Sum_5_1
   Sum_1_3  -->  Sum_1_1 + Sum_3_1
   Sum_3_3  -->  Sum_3_1 + Sum_5_1
   Sum_1_1  -->  Digit_1_1
 Digit_1_1  -->  3
   Sum_3_1  -->  Digit_3_1
 Digit_3_1  -->  5
   Sum_5_1  -->  Digit_5_1
 Digit_5_1  -->  1
```

Fig. 3.17. The parse trees of Figure 3.2 as a parse-forest grammar

grammar that produces an input segment of length *l* starting at position *i*, there is a non-terminal *A_i_l* in the parse-forest grammar, with rules that show how *A_i_l* produces that segment. For example, the existence of **Sum_1_5** in the parse-tree grammar shows that **Sum** produces the whole input string (starting at position 1, with length 5); the fact that there is more than one rule for **Sum_1_5** shows that the parsing was ambiguous; and the two rules show the two possible ways **Sum_1_5** produces the whole input string. When we use this grammar to generate strings, it generates just the input sentence **3+5+1**, but is generates it twice, in accordance with the ambiguity.

We write *A_i_l* rather than $A_{i,l}$ because *A_i_l* represents the name of a grammar symbol, not a subscripted element of an entity *A*: there is no table or matrix *A*. Nor is there any relationship between *A_i_l* and say *A_i_m*: each *A_i_l* is a separate name of a grammar symbol.

Now for the advantages. First, parse-forest grammars implement in a graphical way the concept, already expressed less directly in the previous section, that there should be exactly one entity that describes how a given non-terminal produces a given substring of the input.

Second, it is mathematically beautiful: parsing a string can now be viewed as a function which maps a grammar onto a more specific grammar or an error value. Rather than three concepts — grammars, input strings, and parse forests — we now need only two: grammars and input strings. More practically, all software used in handling the original grammar is also available for application to the parse-forest grammar.

Third, parse-forest grammars are easy to clean up after pruning, using the algorithms from Section 2.9.5. For example, applying the disambiguation criterion used in the previous section to the rules of the grammar in Figure 3.17 identifies the first rule for **Sum_1_5** as being in violation. Removing this rule and applying the grammar clean-up algorithm yields the unambiguous grammar of Figure 3.18, which corresponds to the tree in Figure 3.16.

```
      Sum_S  -->  Sum_1_5
    Sum_1_5  -->  Sum_1_3 + Sum_5_1
    Sum_1_3  -->  Sum_1_1 + Sum_3_1
    Sum_1_1  -->  Digit_1_1
  Digit_1_1  -->  3
    Sum_3_1  -->  Digit_3_1
  Digit_3_1  -->  5
    Sum_5_1  -->  Digit_5_1
  Digit_5_1  -->  1
```

Fig. 3.18. The disambiguated and cleaned-up parse-forest grammar for Figure 3.2

Fourth, representing infinitely ambiguous parsings is trivial: the parse-forest grammar just produces infinitely many (identical) strings. And producing infinitely many strings is exactly what grammars normally do.

And last but probably not least, it fits in very well with the interpretation of parsing as intersection, an emerging and promising approach, further discussed in Chapter 13.

Now it could be argued that parse forests and parse-forest grammars are actually the same and that the pointers in the first have just been replaced by names in the second, but that would not be fair. Names are more powerful than pointers, since a pointer can point only to one object, whereas a name can identify several objects, through overloading or non-determinism: names are multi-way pointers. More in particular, the name **Sum_1_5** in Figure 3.17 identifies *two* rules, thus playing the role of the top OR-node in Figure 3.14. We see that in parse-forest grammars we get the AND-OR tree mechanism free of charge, since it is built into the production mechanism of grammars.

3.8 When are we done Parsing?

Since non-directional parsers process the entire input at once and summarize it into a single data structure, from which parse trees can then be extracted, the question of when the parsing is done does not really arise. The first stage is done when the data structure is finished; extracting the parse trees is done when they are exhausted or the user is satisfied.

In principle, a directional parser is finished when it is in an accepting state and the entire input has been consumed. But this is a double criterion, and sometimes one of these conditions implies the other; also other considerations often play a role. As a result, for directional parsers the question has a complex answer, depending on a number of factors:

- Is the parser at the end of the input? That is, has it processed completely the last token of the input?
- Is the parser in an accepting state?
- Can the parser continue, i.e, is there a next token and can the parser process it?
- Is the parser used to produce a parse tree or is just recognition enough? In the first case several situations can arise; in the second case we just get a yes/no answer.
- If we want parsings, do we want them all or is one parsing enough?
- Does the parser have to accept the entire input or is it used to isolate a prefix of the input that conforms to the grammar? (A string x is a *prefix* of a string y if y begins with x.)

The answers to the question whether we have finished the parsing are combined in the following table, where EOI stands for "end of input" and the yes/no answer for recognition is supplied between parentheses.

at end of	can	in an accepting state?	
input?	continue?	yes	no
yes	yes	prefix identified / continue	continue
yes	no	OK (yes)	premature EOI (no)
no	yes	prefix identified / continue	continue
no	no	prefix identified & trailing text (no)	error in input (no)

Some answers are intuitively reasonable: if the parser can continue in a non-accepting state, it should do so; if the parser cannot continue in a non-accepting state, there was an error in the input; and if the parser is in an accepting state at the end of the input and cannot continue, parsing was successful. But others are more complicated: if the parser is in an accepting state, we have isolated a prefix, even if the parser could continue and/or is at EOI. If that is what we want we can stop, but usually we want to continue if we can: with the grammar **S-->a | ab** and the input **ab** we could stop after the **a** and declare the **a** a prefix, but it is very likely we want to continue and get the whole **ab** parsed. This could be true even if we are at EOI: with the grammar **S-->a | aB** where **B** produces ε and the input **a** we need to continue and parse the **B**, if we want to obtain all parsings. And if the parser cannot, we have recognized a string in the language with what error messages usually call "trailing garbage".

Note that "premature EOI" (the input is a prefix of a string in the language) is the dual of "prefix isolated" (a prefix of the input is a string in the language). If we are looking for a prefix we usually want the longest possible prefix. This can be implemented by recording the most recent position *P* in which a prefix was recognized and continuing parsing until we get stuck, at the end of the input or at an error. *P* is then the end of the longest prefix.

Many directional parsers use look-ahead, which means that there must always be enough tokens for the look-ahead, even at the end of the input. This is implemented by introducing an *end-of-input* token, for example **#** or any other token that does not occur elsewhere in the grammar. For a parser that uses *k* tokens of look-ahead, *k* copies of **#** are appended to the input string; the look-ahead mechanism of the parser is modified accordingly; see for example Section 9.6. The only accepting state is then the state in which the first **#** is about to be accepted, and it always indicates that the parsing is finished.

This simplifies the situation and the above table considerably since now the parser cannot be in an accepting state when not at the end of the input. This eliminates the two prefix answers from the table above. We can then superimpose the top half of the table on the bottom half, after which the leftmost column becomes redundant. This results in the following table:

can	in an accepting state?	
continue?	yes	no
yes	—	continue
no	OK (yes)	error in input / premature EOI (no)

where we leave the check to distinguish between "error in input" and "premature EOI" to the error reporting mechanism.

Since there is no clear-cut general criterion for termination in directional parsers, each parser comes with its own stopping criterion, a somewhat undesirable state of affairs. In this book we will use end-of-input markers whenever it is helpful for termination, and, of course, for parsers that use look-ahead.

3.9 Transitive Closure

Many algorithms in parsing (and in other branches of computer science) have the property that they start with some initial information and then continue to draw conclusions from it based on some inference rules until no more conclusions can be drawn. We have seen two examples already, with their inference rules, in Sections 2.9.5.1 and 2.9.5.2. These inference rules were quite different, and in general inference rules can be arbitrarily complex. To get a clear look at the algorithm for drawing conclusions, the closure algorithm, we shall now consider one of the simplest possible inference rules: transitivity. Such rules have the form

if $A \otimes B$ and $B \otimes C$ then $A \otimes C$

where $\otimes$ is any operator that obeys the rule. The most obvious one is $=$, but $<$, $\leq$ and many others also do. But note that, for example, $\neq$ (not equal) does not.

As an example we shall consider the computation of the "left-corner set" of a non-terminal. A non-terminal B is in the *left-corner* set of a non-terminal A if there is a derivation $A \overset{*}{\rightarrow} B \cdots$; it is sometimes useful to know this, because among other things it means that A can begin with anything B can begin with.

Given the grammar

$$\begin{array}{rcl}
\mathtt{S}_s & \rightarrow & \mathtt{S\ T} \\
\mathtt{S} & \rightarrow & \mathtt{A\ a} \\
\mathtt{T} & \rightarrow & \mathtt{A\ t} \\
\mathtt{A} & \rightarrow & \mathtt{B\ b} \\
\mathtt{B} & \rightarrow & \mathtt{C\ c} \\
\mathtt{C} & \rightarrow & \mathtt{x}
\end{array}$$

how can we find out that $\mathtt{C}$ is in the left-corner set of $\mathtt{S}$? The rules $\mathtt{S} \rightarrow \mathtt{ST}$ and $\mathtt{S} \rightarrow \mathtt{Aa}$ in the grammar tell us immediately that $\mathtt{S}$ and $\mathtt{A}$ are in the left-corner set of $\mathtt{S}$. We write this as $\mathtt{S}\angle\mathtt{S}$ and $\mathtt{A}\angle\mathtt{S}$, where $\angle$ symbolizes the left corner. It also tells us $\mathtt{A}\angle\mathtt{T}$, $\mathtt{B}\angle\mathtt{A}$, and $\mathtt{C}\angle\mathtt{B}$. This is our initial information (Figure 3.19(*a*)).

$\mathtt{S}\angle\mathtt{S}$	$\mathtt{S}\angle\mathtt{S}$ ✓	$\mathtt{B}\angle\mathtt{S}$ ✓	$\mathtt{C}\angle\mathtt{S}$ ✓
$\mathtt{A}\angle\mathtt{S}$	$\mathtt{A}\angle\mathtt{S}$ ✓	$\mathtt{C}\angle\mathtt{S}$	
$\mathtt{A}\angle\mathtt{T}$	$\mathtt{A}\angle\mathtt{S}$ ✓	$\mathtt{C}\angle\mathtt{T}$	
$\mathtt{B}\angle\mathtt{A}$	$\mathtt{B}\angle\mathtt{S}$	$\mathtt{C}\angle\mathtt{S}$ ✓	
$\mathtt{C}\angle\mathtt{B}$	$\mathtt{B}\angle\mathtt{T}$	$\mathtt{C}\angle\mathtt{T}$ ✓	
	$\mathtt{B}\angle\mathtt{S}$ ✓		
	$\mathtt{B}\angle\mathtt{T}$ ✓		
	$\mathtt{C}\angle\mathtt{A}$		
	$\mathtt{C}\angle\mathtt{A}$ ✓		
(*a*)	(*b*)	(*c*)	(*d*)

Fig. 3.19. The results of the naive transitive closure algorithm

Now it is easy to see that if A is in the left-corner set of B and B is in the left-corner set of C, then A is also in the left-corner of C. In a formula:

$A \angle B \wedge B \angle C \Rightarrow A \angle C$

This is our *inference rule*, and we will use it for drawing new conclusions, or "inferences", by pairwise combining known facts to produce more known facts. The *transitive closure* is then obtained by applying the inference rules until no more new facts are produced. The facts are also called "relations" in the transitive-closure context, although formally $\angle$ is the (binary) relation, and $A \angle B$ and $B \angle C$ are "instances" of that relation.

Going through the list in Figure 3.19(*a*) we first combine **S**∠**S** and **S**∠**S**. This yields **S**∠**S**, which is rather disappointing since we knew that already; it is in Figure 3.19(*b*), marked with a ✔ to show that it is not new. The combination (**S**∠**S**, **A**∠**S**) yields **A**∠**S**, but we already knew that too. No other facts combine with **S**∠**S**, so we continue with **A**∠**S**, which yields **A**∠**S** and **B**∠**S**; the first is old, the second our first new discovery. Then (**A**∠**T**, **B**∠**A**) yields **B**∠**T**, etc., and the rest of the results of the first round can be seen in Figure 3.19(*b*).

The second round combines the three new facts with the old and new ones. The first new discovery is **C**∠**S** from **A**∠**S** and **C**∠**A** (*c*); **C**∠**T** follows.

The third round combines the two new facts in (*c*) with those in (*a*), (*b*), and (*c*), but finds no new facts; so the algorithm terminates with 10 facts.

Note that we have already implemented an optimization in this naive algorithm: the basic algorithm would start the second and subsequent rounds by pairing up *all* known facts with all known facts, rather than just the new ones.

It is often useful to represent the facts or relations in a graph, in which they are arcs. The initial situation is shown in Figure 3.20(*a*), the final one in (*b*). The numbers

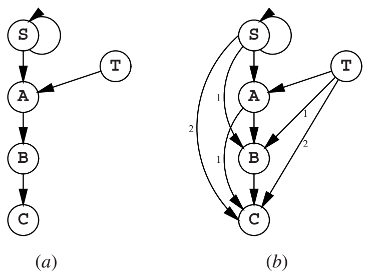


Fig. 3.20. The left-corner relation as a graph

next to the arrows indicate the rounds in which they were added.

The efficiency of the closure algorithm of course depends greatly on the inference rule it uses, but the case for the transitive rule has been studied extensively. There are three main ways to do transitive closure: naive, traditional, and advanced; we will discuss each of them briefly. The naive algorithm, sketched above, is usually quite efficient in normal cases but may require a large number of rounds to converge in exceptional cases on very large graphs. Also it recomputes old results several times, as we see in Figure 3.19: of the 15 results 10 were old. But given the size of "normal" grammars, the naive algorithm is satisfactory in almost all situations in parsing.

The traditional method to do transitive closure is to use Warshall's algorithm [409]. It has the advantage that it is very simple to implement and that the time it requires depends only on the number of nodes N in the graph and not on the number of arcs, but it has the disadvantage that it always requires $O(N^3)$ time. It always loses in any comparison to any other closure algorithm.

The advanced algorithms avoid the inefficiencies of the naive algorithm: 1. cycles in the graph are contracted as "strongly connected components"; 2. the arcs are combined in an order which avoids duplicate conclusions and allows sets of arcs to be copied rather than recomputed; 3. efficient data representations are used. For example, an advanced algorithm would first compute all outgoing arcs at `A` and then copy them to `T` rather than recomputing them for `T`. The first advanced transitive closure algorithm was described by Tarjan [334]. They are covered extensively in many other publications; see Nuutila [412] and the Internet. They require time proportional to the number of conclusions they draw.

Advanced transitive closure algorithms are very useful in large applications (databases, etc.) but their place in parsing is doubtful. Some authors recommend their use in LALR parser generators but the grammars used would have to be very large for the algorithmic complexity to pay off.

The advantage of emphasizing the closure nature of algorithms is that one can concentrate on the inference rules and take the underlying closure algorithm for granted; this can be a great help in designing algorithms. Most algorithms in parsing are, however, simple enough as to not require decomposition into inference rules and closure for their explanation. We will therefore use inference rules only where they are helpful in understanding (Section 9.7.1.3) and where they are part of the culture (Section 7.3, chart parsing). For the rest we will present the algorithms in narrative form, and point out in passing that they are transitive-closure algorithms.

3.10 The Relation between Parsing and Boolean Matrix Multiplication

There is a remarkable and somewhat mysterious relationship between parsing and Boolean matrix multiplication, in that it is possible to turn one into the other and vice versa, with a lot of ifs and buts. This has interesting implications.

A Boolean matrix is a matrix in which all entries are either 0 or 1. If the indexes of a matrix T represent towns, the element $T_{i,j}$ could, for example, indicate the existence of a direct railroad connection from town i to town j. Such a matrix can be multiplied by another Boolean matrix $U_{j,k}$, which could, for example, indicate the existence of a direct bus connection from town j to town k. The result $V_{i,k}$ (the product of T and U) is a Boolean matrix which indicates if there is a connection from town i to town k by first using a train and then a bus. This immediately shows how $V_{i,k}$ must be computed: it should have a 1 if there is a j for which both $T_{i,j}$ and $U_{j,k}$ hold a 1, and a 0 otherwise. In a formula:

$$V_{i,k} = (T_{i,1} \wedge U_{1,k}) \vee (T_{i,2} \wedge U_{2,k}) \vee \cdots \vee (T_{i,n} \wedge U_{n,k})$$

where $\wedge$ is the Boolean AND, $\vee$ is the Boolean OR, and n is the size of the matrices. This means that $O(n)$ actions are required for each entry in V, of which there are n^2; so the time dependency of this algorithm is $O(n^3)$.

Figure 3.21 shows an example; the boxed row $T_{2,*}$ combines with the boxed column $U_{*,2}$ to produce the boxed entry $V_{2,2}$. Boolean matrix multiplication is not

$$\overset{T}{\begin{vmatrix} 0&0&0&0&0\\ \boxed{0\ \ 0\ \ 0\ \ 0\ \ 1}\\ 0&0&0&1&0\\ 0&0&0&0&0\\ 1&0&0&0&0 \end{vmatrix}} \times \overset{U}{\begin{vmatrix} 0&0&1&0&0\\ 0&0&0&1&0\\ 1&0&0&0&0\\ 0&0&0&0&0\\ 0&1&0&0&0 \end{vmatrix}} = \overset{V}{\begin{vmatrix} 0&0&0&0&0\\ 0&\boxed{1}&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&1&0&0 \end{vmatrix}}$$

Fig. 3.21. Boolean matrix multiplication

commutative: it is quite possible that there is a train-bus connection but no bus-train connection from one town to another, so $T \times U$ will in general not be equal to $U \times T$. Note also that this differs from transitive closure: in transitive closure a single relation is followed an unbounded number of times, whereas in Boolean matrix multiplication first one relation is followed and then a second.

The above is a trivial application of *Boolean matrix multiplication* (*BMM*), but BMM is very important in many branches of mathematics and industry, and there is a complete science on how to perform it efficiently.[3] Decades of concentrated effort have resulted in a series of increasingly more efficient and complicated algorithms. V. Strassen[4] was the first to break the $O(n^3)$ barrier with an $O(n^{2.81\cdots})$ algorithm, and the present record stands at $O(n^{2.376\cdots})$; it dates from 1987. It is clear that at least $O(n^2)$ actions are required, but it is unlikely that that efficiency can be achieved.

More important from our viewpoint is the fact that in 1975 Valiant [18] showed how a CF parsing problem can be converted into a BMM problem. In particular, if you can multiply two Boolean matrices of size $n \times n$ in $O(n^k)$ actions, you can parse a string of length n in $O(n^k) + O(n^2)$ actions, where the $O(n^2)$ is the cost of the conversion. So we can do general CF parsing in $O(n^{2.376\cdots})$, which is indeed better than the cubic time dependency of the CYK algorithm. But the actions of both Valiant's algorithm and the fast BMM are extremely complicated and time-consuming, so this approach would only be better for inputs of millions of symbols or more. On top of that it requires all these symbols to be in memory, as it is a non-directional method, and the size of the data structures it uses is $O(n^2)$, which means that it can only be run profitably on a machine with terabytes of main memory. In short, its significance is theoretical only.

[3] For a survey see V. Strassen, "Algebraic complexity theory", *in* Handbook of Theoretical Computer Science, vol. A, Jan van Leeuwen, Ed. Elsevier Science Publishers, Amsterdam, The Netherlands, pp. 633-672, 1990.

[4] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, 13:354-356, 1969.

In 2002 Lee [39] showed how a BMM problem can be converted into a CF parsing problem. More in particular, if you can do general CF parsing of a string of length n in $O(n^{3-\delta})$ actions, you can multiply two Boolean matrices of size $n \times n$ in $O(n^{3-\delta/3})$ actions. There is again a conversion cost of $O(n^2)$, but since δ can be at most 2 (in which unlikely case parsing could be done in $O(n)$), $O(n^{3-\delta/3})$ is at least $O(n^{2\frac{1}{3}})$, which dominates the $O(n^2)$; note that for $\delta = 0$ the usual $O(n^3)$ bounds for both problems result. The computational efforts involved in Lee's conversion are much smaller than those in Valiant's technique, so a really fast general CF parsing algorithm would likely supply a fast practical BMM algorithm. Such a fast general CF parsing algorithm would have to be non-BMM-dependent and have a time complexity better than $O(n^3)$; unfortunately no such algorithm is known.

General CF parsing and Boolean matrix multiplication have in common that the efficiencies of the best algorithms for them are unknown. Figure 3.22 summarizes the possibilities. The horizontal axis plots the efficiency of the best possible general

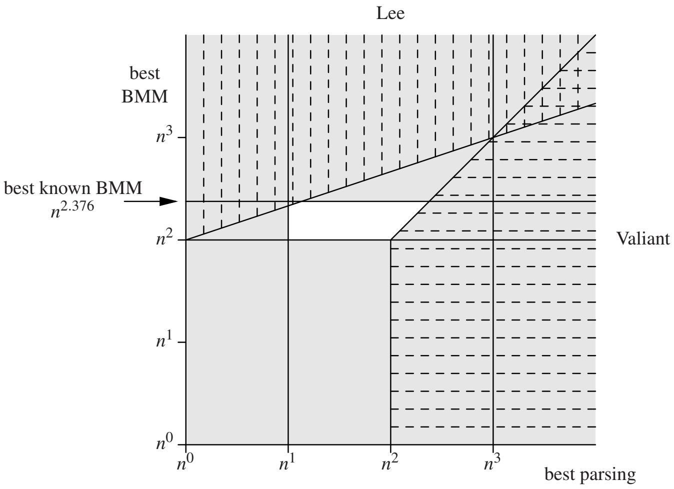


Fig. 3.22. Map of the best parser versus best BMM terrain

CF parsing algorithm; the vertical axis plots the efficiency of the best possible BMM algorithm. A position in the graph represents a combination of these values. Since these values are unknown, we do not know which point in the graph corresponds to reality, but we can exclude several areas.

The grey areas are excluded on the grounds of existing algorithms. For example, the grey area on the right of the vertical line at n^3 is excluded because we have the CYK algorithm, which does general CF parsing in $O(n^3)$; so the pair (best parser, best BMM) cannot have a first component which is larger than $O(n^3)$. Likewise the area left of the vertical line at n^1 represents parsing algorithms that work in less than $O(n)$, which is impossible since the parser must touch each token. BMM requires at least $O(n^2)$ actions, but an algorithm for $O(n^{2.376\cdots})$ is available; this yields two horizontal forbidden areas.

The shading marks the areas that are excluded by the Valiant and Lee conversion algorithms. Valiant's result excludes the horizontally shaded area on the right; Lee's result excludes the vertically shaded area at the top. The combination of the efficiencies of the true best parsing and BMM algorithms can only be situated in the white unshaded area in the middle.

Extensive research on the BMM problem has not yielded a usable algorithm that is substantially better than $O(n^3)$; since BMM can be converted to parsing this could explain why the admittedly less extensive research on general CF parsing has not yielded a better than $O(n^3)$ algorithm, except through BMM. On the other hand Figure 3.22 shows that it is still possible for general CF parsing to be linear ($O(n^1)$) and BMM to be worse than $O(n^2)$.

Rytter [34] has linked general CF parsing to a specific form of shortest-path computation in a lattice, with comparable implications.

Greibach [389] describes the "hardest context-free language", a language such that if we can parse it in time $O(n^x)$, we can parse any CF language in time $O(n^x)$. Needless to say, it's hard to parse. The paper implicitly uses a parsing technique which has received little attention; see Problem 3.7.

3.11 Conclusion

Grammars allow sentences to be produced through a well-defined process, and the details of this process determines the structure of the sentence. Parsing recovers this structure either by imitating the production process (top-down parsing) or by rolling it back (bottom-up parsing). The real work goes into gathering information to guide the structure recovery process efficiently.

There is a completely different and — surprisingly — grammarless way to do CF parsing, "data-oriented parsing", which is outside the scope of this book. See Bod [348] and the Internet.

Problems

Problem 3.1: Suppose all terminal symbols in a given grammar are different. Is that grammar unambiguous?

Problem 3.2: Write a program that, given a grammar G and a number n, computes the number of different parse trees with n leaves (terminals) G allows.

Problem 3.3: If you are familiar with an existing parser (generator), identify its parser components, as described on page 69.

Problem 3.4: The maze preprocessing algorithm in Section 3.5.4 eliminates all rooms with three walls; rules with two or four walls are acceptable in a deterministic maze. What about rooms with zero or one wall? How do they affect the algorithm and the result? Is it possible/useful to eliminate them too?

Problem 3.5: Construct an example in which a deterministic bottom-up parser will have to perform k actions at position k, for a certain k.

Problem 3.6: *Project:* There are several possible paths through the maze in Figure 3.10(b), so a maze defines a set of paths. It is easy to see that these paths form a regular set. This equates a maze to a regular grammar. Develop this analogy, for example: 1. Derive the regular grammar from some description of the maze. 2. How does the subset algorithm (Section 5.3.1) transform the maze? 3. Is it possible to generate a set of mazes so that together they define a given CF set?

Problem 3.7: *Project:* Study the "translate and cross out matching parentheses" parsing method of Greibach [389].

Problem 3.8: Show that a version of Figure 3.14 in which the nodes marked 2 near the top are combined represents parse trees that are not supported by the input.

Problem 3.9: Implement the backtracking algorithm sketched in Section 3.7.3.

Problem 3.10: Assume arithmetic expressions are parsed with the highly ambiguous grammar

```
    Expr_S  -->  Number
    Expr    -->  Expr Operator Expr | ( Expr )
Operator    -->  + | - | × | / | ↑
```

with an appropriate definition of **`Number`**. Design a criterion that will help prune the resulting parse forest to obtain the parse tree that obeys the usual precedences for the operators. For example, $4+5\times6+8$ should come out as $((4+(5\times6))+8)$. Take into account that the first four operators are left-associative, but the exponentiation operator $\uparrow$ is right-associative: $6/6/6$ is $((6/6)/6)$ but $6\uparrow6\uparrow6$ is $(6\uparrow(6\uparrow6))$.

Problem 3.11: *Research project:* Some parsing problems involve extremely large CF grammar, with millions of rules. Such a grammar is generated by program and results from incorporating finite context conditions into the grammar. It is usually very redundant, containing many very similar rules, and very ambiguous. Many general CF parsers are quadratic in the size of the grammar, which for ten million rules brings in a factor of 10^{14}. Can parsing techniques be designed that work well on such grammars? (See also Problem 4.5.)

Problem 3.12: *Extensible Project:* 1. A string S is *balanced* for a token pair (t_1,t_2) if $\#t_1 = \#t_2$ for S and $\#t_1 \geq \#t_2$ for all prefixes of S, where $\#t$ is the number of occurrences of t in S or a prefix of it. A token pair (t_1,t_2) is a *parentheses pair* for a grammar G if all strings in $\mathcal{L}(G)$ are balanced for (t_1,t_2). Design an algorithm to check if a token pair (t_1,t_2) is a parentheses pair for a given grammar G: a) under the simplifying but reasonable assumption that parentheses pairs occur together in the right hand side of a rule (for example, as in **`F-->(E)`**), and b) in the general case.

2. A token t_1 in position i in a string *matches* a token t_2 in a position j if the string segment $i+1\cdots j-1$ between them is balanced for (t_1,t_2). A parentheses pair (t_1,t_2) is *compatible with* a parentheses pair (u_1,u_2) if every segment between a t_1 and its matching t_2 in every string in $\mathcal{L}(G)$ is balanced for (u_1,u_2). Show that if (t_1,t_2) is compatible with (u_1,u_2), (u_1,u_2) is compatible with (t_1,t_2).

3. Design an algorithm to find a largest set of compatible parentheses pairs for a given grammar.

4. Use the set of parentheses pairs to structure sentences in $\mathcal{L}(G)$ in linear time.

5. Derive information from G about the segments of strings in $\mathcal{L}(G)$ that are not structured in that process, for example regular expressions.

6. Devise further techniques to exploit the parentheses skeleton of CF languages.