Lecture notes for the

Introduction to Symbolic Computational Linguistics (684.01)

Detmar Meurers, Winter 2001
Department of Linguistics
The Ohio State University

This is a modified version of the module workbook for “Techniques in Natural Language Processing 1” by Chris Mellish, Pete Whitelock and Graeme Ritchie, 1994, Department of Artificial Intelligence, University of Edinburgh. I would like to thank them for permitting me to adapt their material for this course.

Contents

Chapter 1

Finite State Techniques

1.1 Aims of this Chapter

- To introduce the notions of finite state machine and finite state transducer.

1.2 Finite State Machines

Suppose we have some set of symbols (such as letters or words in a language) and we wish to design a machine that is going to scan sequences (strings) of these symbols. The simplest possible kind of machine for this task would be a finite state machine.

Finite state machines were originally developed in the 1950s. A Finite State Machine (FSM) (also known as a "finite-state automaton" (FSA)) has a finite set of possible *states,* (e.g. "red", "amber" and "green") that it can be in and a set of possible *transitions* that enable it to change from one state to another. The machine processes an input sequence of symbols, *terminal symbols*, on a tape sequentially and at any point in time is looking at a particular current symbol. Each possible transition specifies a rule of the form:

> If the machine is in state X and the current symbol is Y
> then the machine can move to state Z
> with the tape moved on one position.

One or more states are defined to be *initial states* , and it is determined that the machine must always start in one of these states. In addition, one or more states are defined to be *final states* . An FSM is said to have "successfully" processed an input tape if, starting in an initial state and changing only as allowed by the possible transitions, it ends up in a final state with the tape exhausted.

FSMs have been explored extensively in computer science, and there are many mathematical results concerning them. There are three main notations used for specifying FSMs: finite state transition networks, transition tables and regular expressions.

1.2.1 Finite state transition networks

One possible way to describe what a FSM does is to indicate what steps are taken as each symbol is processed, including what tests are made on that symbol and what actions are carred out in the various possible cases. That is, to specify some form of “flow chart” of the possible options and actions.

A Finite State Transition Network (FSTN) specifies an FSM by means of a graph, in which the nodes represent the possible states of the machine and the arcs the possible transitions. If there is a possible transition from state X to state Z, consuming symbol Y, then the graph contains a directed arc from the node representing X to the node representing Z and this arc is labelled with Y. FSTNs also provide a notation for indicating an initial state (which we shall indicate with an arrow), and a final state (which we shall indicate with a double circle), as in Figure 1.1.

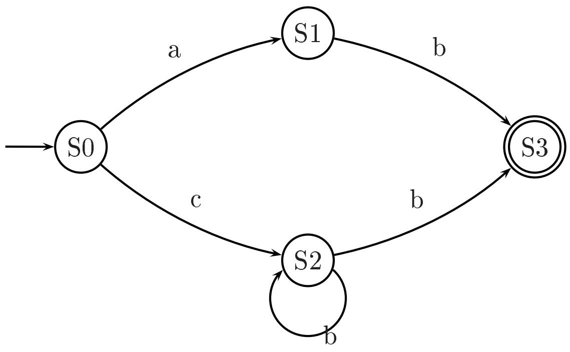

Figure 1.1: A simple finite-state machine

There is a very natural algorithm for using an FSTN to tell whether a string of symbols is recognised by the corresponding FSM—start in an initial state, and scan the string one symbol at a time, following arcs which are labelled with the current symbol. If the current input symbol (i.e. the one being scanned) does not correspond to any of the arcs from the current state, then the scan is said to *block*, and the string of symbols is not generated (accepted, recognised) by the FSM. For example, the FSM in Figure 1.1 would accept the sequences:

ab
cb
cbb
cbbb
...

This algorithm has to be complicated if there are several initial states (which is unusual), or if there is *non-determinism* (see below).

1.2.2 Procedural and declarative readings

As we have described it, an FSTN is a way of specifying an FSM that successfully processes some strings and not others. If we associate "successful processing" with "recognition" then the corresponding FSM *recognises* some strings and doesn't recognise others.

Clearly, an FSTN can also be viewed as a specification of a machine that *produces* strings, by starting from the initial state and *outputting* symbols as arcs are followed (transitions are made). In this case, the possible sequences generated by the machine are exactly the same as those that would be recognised by the first machine.

The fact that there are these two different views tells us that there is a more abstract way of viewing an FSTN. An FSTN can be thought of as a description of a set of strings (sequences of symbols)—those that would be recognised by the first machine or (equivalently) generated by the second. We call this set the *language generated by* the description, with a particular technical use of the word "generated".

This *declarative* interpretation of the FSTN contrasts with the previous *procedural* interpretations as giving the instructions for machines. The same possibilities apply to some extent to the other two notations.

1.2.3 Transition tables

The possible transitions of a FSM can also be described by a "transition table", where each row lists the options from the state named at the start of the row. Here initial states are marked with a dot "." after their name, and final states with a colon ":". The FSM shown above has the following transition table:

	a	b	c	d
S0.	S1		S2	
S1		S3:		
S2		S2,S3:		
S3:				

In general, there may be several possible states that can be reached using the same symbol from the same state (this is *non-determinism*—see below) and so there may be multiple entries in a given position of the table.

1.2.4 Regular expressions

Regular expressions provide a notation for describing FSMs in a way closer to the languages generated. In particular, the notation abstracts away from the details of particular states and transitions.

A *regular expression* is a way of describing a language which can be generated by a FSM. The asterisk (*), often referred to as Kleene star, is used to indicate “zero or more occurrences”, the plus sign (+) is used to indicate “one or more occurrences”, the question mark (?) indicates “zero or one occurrence”, and the “|” indicates disjunction. Finally, parentheses are used to group sequences together where necessary. For example, the expression

$$a*|(ba)+$$

denotes a language in which a valid sequence could be either zero or more “a”s, or one or more repetitions of the sequence “b a”.

Any set of strings which can be exactly defined by a regular expression is the language of some FSM.

This notation can be very useful in describing the behaviour of FSMs, and there is extensive literature on how these expressions can be manipulated mathematically. (Many pieces of computer software, such as text editors, define some of their possible commands in terms of regular expressions.)

1.3 Enhancements

There are some enhancements that can be made to the basic notations for FSMs to make them easier to specify. All of these can be thought of as simply more convenient notations which abbreviate more complex (but standard) specifications.

Abbreviations allow us to specify more compactly a set of similar arcs. For instance, if there is to be an arc from node a to node b corresponding to scanning each possible English vowel, then we can specify this by a single arc labelled “VOWEL” rather than 5 individual arcs for “a”, “e”, “i”, “o” and “u”. As an extreme case, if there is a finite set of possible symbols then the dot symbol “.” is generally used to label an arc that can be traversed regardless of what the current symbol is.

Secondly, we can allow *empty transitions*, which are arcs without labels, or labelled with a special symbol ϵ, and which may be followed regardless of the current symbol, and without moving the input scanner.

Any specification of an FSM using abbreviations and/or empty transitions can be translated mechanically into one that does not use these facilities. Thus we do not sacrifice any distinctions by thinking of these enhancements as being an inherent part of what FSMs allow – they do not have any affect on the languages generatable using the notation.

1.4 Determinism

In a FSM, there may be states where more than one outgoing arc is labelled with the same symbol. If the FSM was being used for recognition, this would present the scanning mechanism with a choice—which arc should be followed? An automaton in which this does not occur (i.e. the currently scanned symbol will always uniquely determine which arc to follow), is said to be *deterministic,* and an automaton where choices may occur is *non-deterministic.* If empty transitions are allowed then this can create non-determinism as well. (Notice that the issue of determinism/non-determinism relies to a large extent on a *processing* viewpoint for FSMs).

We say that a non-deterministic FSM recognises a sequence of symbols if there is *any* path from an initial state to a final state that corresponds to that sequence. In an implementation, it would be necessary to *search* for such a path amongst the possible ones, using some AI search technique.

Given a non-deterministic FSM, it is straightforward to construct an FSM which recognises exactly the same set of strings (language), but is deterministic. This is done more or less by brute force—group states in the original FSM into clusters according to whether they can be reached by the same sequence of symbols, then construct a new FSM in which each cluster is represented by a state. For example, suppose the states of the original machine are S0, S1, S2, S3, S4 and S5. Then states of the new machine will be labelled with names such as {S0}, {S1, S2}, {S1, S4}, etc; that is, any *set* of original state-names constitutes a possible name for a state in the new machine, and being "in" such a composite state is equivalent to being "in" some one of the originals. In other words, we are labelling the new machine with *disjunctions* of state-names from the old machine.

For example, consider the FSM given in Figure 1.2 with the following transition table:

	a	b	c	d	e
S1.	S2	S3			
S2			S3,S5	S4	
S3	S5				
S4			S6:		S4,S5
S5	S6:				
S6:					

States S1 and S2 are clustered together, since they are reached from S0 by symbol a. From that state-cluster {S1, S2}, the allowable symbols are b, which causes a transition back to the same state-cluster {S1, S2}, and c or d which go to S3. This gives a "collapsed" machine as in Figure 1.3.

1.5 Combining FSMs

It is possible to combine different FSMs together to make machines that recognise more complex sets of strings.

For instance, if we had two FSMs which recognised all the sentences in English and all the sentences in French respectively (with the terminal symbols being words in the two languages) then those FSMs could be combined into a single FSM that accepted only those sentences which were legal in both English

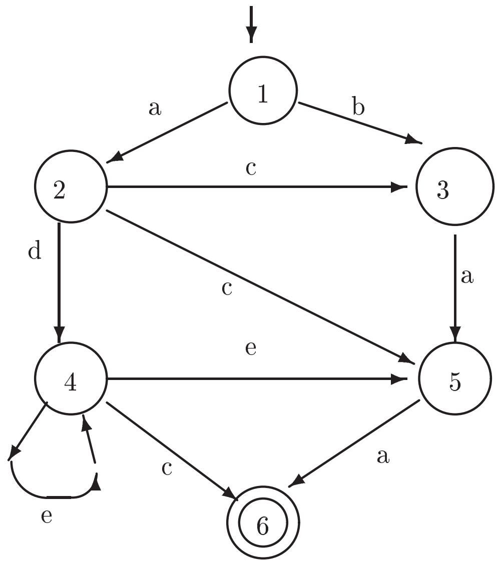

Figure 1.2: A non-deterministic FSM

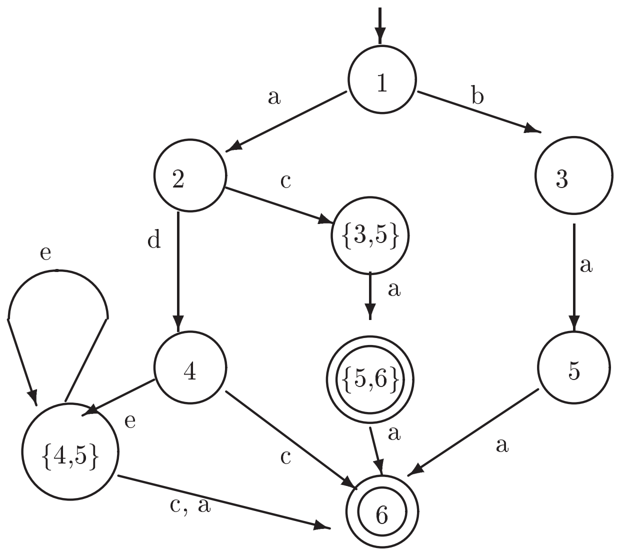

Figure 1.3: Determinised Finite-State Machine

and French. Similarly (and more simply) we could construct an FSM that recognised any string that was either legal in English or in French.

1.6 Transducers

A *transducer* is an abstract machine which transforms or translates sequences of symbols into other sequences of symbols. A *finite state transducer* (FST) is like a FSM, except that it is augmented to perform output of symbols while scanning the input symbols. This is done by having associated with each transition (arc) *two* symbols—one to be output when that arc is followed as well as the symbol to be tested against the input data. We use the notation $a : b$ to denote such a pair of symbols, where a is an input symbol and b the corresponding output.

The usual way of depicting a finite automaton using a table of transitions can be adapted to transducers, by using symbol-pairs in place of single elements of the alphabet. The columns of the table are labelled with symbol-pairs, the rows are labelled with state-identifiers (integers are used here) and the entries in the table indicate the state to be entered on encountering each symbol-pair. If no state is indicated, then the automaton or transducer *blocks* in that situation—it fails to accept the sequence of symbol-pairs. For example, the following transition table specifies a transducer for translating simple English sentences to the corresponding French:

	`Who:Qui`	`John:Jean`	`is:est`	`sees:voit`	`Mary:Marie`
`S1.`	`S2`	`S2`			
`S2`			`S3`	`S3`	
`S3`					`S4:`
`S4:`					

This would, for instance, transduce the input "Who is Mary" to "Qui est Marie".

1.6.1 Determinism for transducers

In the case of a transducer, determinism is normally defined in terms of whether the *input* symbol would be sufficient to determine the choice of transition. (It is reasonable to assume that the *output* symbol on an arc is not guiding the progress of the transducer).

Unfortunately the determinising process sketched above is not guaranteed to work for genuine FS *transducers*, where a distinction is made between input and output, and output is supposed to be determined on the basis solely of symbols already scanned, The reason for this can be illustrated using the FST given in Figure 1.4.

The correct output symbol for the earlier arcs cannot be determined until the final input symbol in the string is read, so the choice between the sequence of "b"s and the sequence of "c"s cannot be resolved deterministically. (This argument depends crucially on the assumption that symbols have to be output as the corresponding inputs are scanned, which is reasonable.)

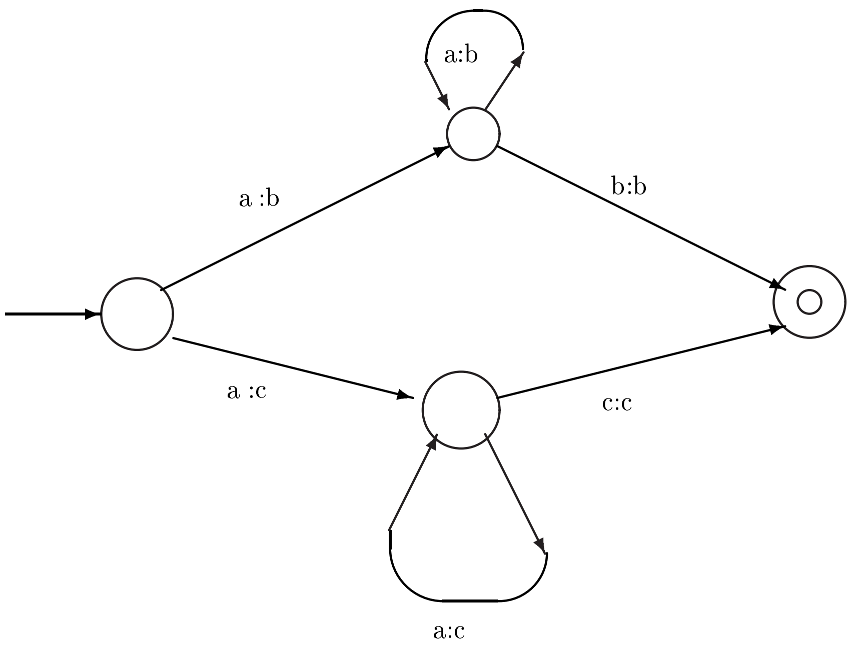

Figure 1.4: An inherently non-deterministic transducer

1.6.2 Other ways of viewing transducers

In fact, there are various ways one could interpret an automaton with two symbols per transition. What has just been described is a genuine transducer—one symbol for testing against the scanned string, one symbol to provide the output. However, one could also regard an FSM with two symbols on each arc as defining a process for recognising *pairs* of input sequences, by interpreting the two symbols marked on each arc as *both* being tests to be made against input, assuming two simultaneously scanned sequences of symbols. Then the automaton is essentially a *two-tape recogniser*, and not a true transducer.

Abstractly, an automaton with pairs of symbols on its arcs defines a correspondence between sequences of symbols; it becomes a transducer only when we specify a direction of translation, by indicating which of the symbols is regarded as input and which as output.

If the same automaton is to be used to translate in two different directions (i.e. the question of which of the symbol-pair is the input and which is the output one may vary from one use of the automaton to the other), then the question of whether or not it is deterministic could be different for each direction. Although the determinising process will not work for a genuine transducer, it would work for a two-tape FS recogniser (where the choice of the next state is dependent on *both* the symbols being scanned).

1.7 References

1.7.1 Introductory

- Jurafsky and Martin (2000, Ch. 2) is a general introduction to regular expressions and automata.
- Gazdar and Mellish (1989, Ch. 2) provides a general introduction to finite state techniques.
- Winograd (1983, Ch. 2) describes words, patterns, and finite state machines, though his notation may be confusing to some.
  - T. Winograd (1983): *Language as a Cognitive Process. Volume 1: Syntax*, Addison-Wesley.

1.7.2 Advanced

- Roche and Shabes (1997, Ch. 1) is a detailed discussion of the basic notions of finite-state automata and finite-state transducers.
  - E. Roche and Y. Shabes (1997): *Finite-State Language Processing.* MIT Press.

1.8 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- What is the difference between an initial state and a final state in an FSM?
- What causes non-determinism in an FSM?
- What is the difference between an FSM and an FST?
- What is the relation between regular expressions and FSMs?

What you should be able to do now:

- Translate between FSTNs, transition tables and regular expressions.
- Explain what determinism means for FSMs and FSTs.
- Specify simple FSMs and FSTs.
- Give two procedural, and one declarative, interpretation for a given FSTN.

Chapter 2

Basic Formal Language Theory

2.1 Aims of this Chapter

- To introduce the idea that it is possible to formally describe languages and that according to what description language you choose it may or may not be possible to describe a given language in it.
- To introduce the major formal framework for describing languages that is used in mathematics and theoretical computer science: formal language theory.
- In particular, to cover the basics of set theory, the definition of what a *language* is, grammars and automata and the Chomsky hierarchy.

In the next chapter we will consider how these formal notions can be used to reason about the characteristics of natural languages.

2.2 Some very basic background on sets

A *set* is simply a collection of objects that have been grouped together for some purpose. The objects in a set are called its *elements* or *members*. A set can be shown by listing its elements inside curly brackets. Thus:

> S_{week} = {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}

is the set of names of days of the week in English. We use the symbol $\in$ to denote "is an element of". Thus, the following are all true:

Monday $\in S_{week}$
Tuesday $\in S_{week}$
Wednesday $\in S_{week}$
...

Notice that it makes no sense to ask "how many times is x an element of S?". Given an object and a set, either the object is an element, or it is not—there are no further possibilities. It also makes no sense to ask "does x appear earlier in the set than y?". Two sets are the same just when they have exactly the same elements. Thus, the following are simply four different ways of indicating the same set:

$\{1, 2, 3, 3, 3, 3, 3\}$
$\{2, 2, 1, 1, 3, 3, 1, 2, 3\}$
$\{1, 2, 3\}$
$\{3, 2, 1\}$

When we implement sets computationally, perhaps by using lists, we often remove any duplicates from the lists and keep the elements in a special sequence (e.g., in alphabetical order), though this is not necessary. Such moves make it easier to detect sets that are the same, remove elements from sets, determine whether two sets have common elements, and so on.

Instead of specify a set by giving its elements directly, we can do so by saying that it consists of all objects satisfying a given property. Thus we could have written above:

$$S_{week} = \{x | x \text{ is the English name of a day of the week}\}$$

("the set of all x such that x is the English ..."). This is just as exact a way of specifying what the set is, but given this description a foreigner would need to consult a dictionary in order to find out what the actual elements were.

Two important operations on sets are computing their *intersection* and *union*. Given two sets S_1 and S_2, their intersection, $S_1 \cap S_2$ is simply the set of objects that belong to both sets:

$$S_1 \cap S_2 = \{x \mid x \in S_1 \text{ and } x \in S_2\}$$

Thus:

$$\{1, 2, 3, 3, 5\} \cap \{2, 4, 6, 8\} = \{2\}$$

Similarly, given two sets S_1 and S_2, their union, $S_1 \cup S_2$ is simply the set of objects that belong to either of the two sets (or both):

$$S_1 \cup S_2 = \{x \mid x \in S_1 \text{ or } x \in S_2\}$$

Thus:

$$\{1, 2, 3, 3, 5\} \cup \{2, 4, 6, 8\} = \{1, 2, 3, 4, 5, 6, 8\}$$

One set S_1 is a *subset* of another set S_2, $S_1 \subseteq S_2$, if every element of S_1 is also an element of S_2. Notice that this definition includes every set as a subset of itself. If we wanted to exclude this possibility, then we would use the notion of *strict subset*, written $S_1 \subset S_2$.

2.3 Languages

A *vocabulary* is a set of symbols that can be strung together to make phrases. For instance, if we regard English sentences as sequences of words, then the vocabulary of English V_{Eng} can be defined something like the following:

$$V_{Eng} = \{\text{A, a, aback, abacus, } \ldots \text{, zygoma}\}.$$

If S is a vocabulary, then $S*$ is the set containing all the possible finite sequences of elements of S. $S*$ will always be an infinite set (unless S is empty). Thus:

$$V_{Eng}* = \left\{\begin{array}{l}\text{a,}\\ \text{a a a a a a a a,}\\ \text{a dog,}\\ \text{aback zygoma a a he,}\\ \text{abacus aback a zygoma,}\\ \text{barks,}\\ \text{this sounds nice,}\\ \ldots\end{array}\right\}$$

Note that the empty string conventionally denoted by ϵ (epsilon) is also in this set. It is the only sequence of zero vocabulary elements.

Not every element of $V_{Eng}*$ is in any way a sensible phrase of the language English. Mathematically, we define a *language* with respect to a vocabulary S to be some subset of $S*$. Thus English is a subset of $V_{Eng}*$, i.e., something like:

$$\left\{\begin{array}{l}\text{a dog barks,}\\ \text{this sounds nice,}\\ \ldots\end{array}\right\}$$

If α and β denote vocabulary elements, or sequences of such, we shall use the convention that writing them next to one another

$$\alpha\ \beta$$

denotes the concatenation of the relevant phrases (the string obtained by taking first α and then β). Also we will use the notation

$$\alpha^n$$

to indicate n copies of α put side to side (n being a number). In written English we use devices such as adding spaces to show where one word ends and the next one begins—it is standard to assume that from a sequence of symbols (here, English words) one can easily determine the component symbols and that the devices that make this possible are not interesting in themselves. If this assumption does not hold, then one can usually split up the symbols into smaller units (here, for instance, letters) and talk about the language in terms of sequences of these.

2.4 Formal Systems for Defining Languages

A formal system that can be used to define languages consists of two components.

1. A characterisation of those knowledge sources whose content will vary with the particular language being defined.

2. A specification of how any instance of such knowledge determines the membership of a given sentence in the defined language. It should be possible to further develop this specification into a procedure, a finite sequence of explicit instructions that can be carried out mechanically. Ideally, we would like such a procedure to be an algorithm, a procedure that terminates on all inputs, and is thus decidable.

We will look at two alternative ways of defining languages, one by means of grammars which generate the strings (i.e., sentences) of the languages concerned, and the other by means of automata which are machines that recognise sentences of the languages.

In particular, we will look at four types of grammar, and four corresponding automata, that form a hierarchy, in which the languages that are describable by a type n device can be shown to properly include those definable by a type $n+1$ device. That is, the lower the number, the more powerful the device. This is called the *Chomsky hierarchy*.

The import of this for natural language processing is as follows. We can tentatively identify certain constructions in natural languages that we know require a certain type of device for their description—a grammar of a weaker type will just not do. We can argue about the data—whether a particular construction is indeed of a certain form—but once we agree on the linguistics, we have mathematical certainty about the sort of formal device required.

2.5 Grammars

A grammar is defined as having four components (N, Σ, S, P), where

1. N is a finite set of nonterminal symbols (corresponding to syntactic categories).
2. Σ is a finite set of terminal symbols disjoint from N (corresponding to words or perhaps lexical categories).
3. S is a distinguished symbol, a member of N, called the *start symbol*.
4. P is a finite set of rules of the form $\alpha \to \beta$, where α and β are elements of $(N \cup \Sigma)*$, α containing at least one non-terminal symbol.

The elements of P are simply the familiar *rewrite rules* of linguistics.

Although we will shortly look at some different classes of grammars, here is a quick example to illustrate the above terminology. The following is a grammar for decimal numbers, using a vocabulary of digits:

$$
\begin{aligned}
N &= \{\text{digit, pos\_digit, pos\_number, digits, number}\} \\
\Sigma &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
S &= \text{number} \\
P &= \left\{
\begin{array}{l}
\text{digits} \to \epsilon, \\
\text{digits} \to \text{digit digits}, \\
\text{pos\_number} \to \text{pos\_digit digits}, \\
\text{number} \to 0, \\
\text{number} \to \text{pos\_number}, \\
\text{digit} \to 0, \\
\text{digit} \to \text{pos\_digit}, \\
\text{pos\_digit} \to 1, \\
\vdots \\
\text{pos\_digit} \to 9
\end{array}
\right\}
\end{aligned}
$$

In practice, when a grammar is specified it is often obvious from the context which symbols are nonterminals and which are terminals. There is also often a convention about what the start symbol is (in linguistics, often "S"). Thus the elements N, Σ and S are often not specified, even though for complete accuracy they should be.

How is a grammar G a definition of a language? First of all, we can define the notion of a *sentential form* for a language:

- S is a sentential form, where S is the start symbol.
- If x is a sentential form of the form $\alpha\beta\gamma$, and there is a rewrite rule of the form $\beta \to \delta$, then $\alpha\delta\gamma$ is a sentential form.

For our grammar of decimal numbers, the following are sentential forms:

number
pos_digit 3 4 digits
0
1 2 digits
2 3 4 5

Intuitively, a sentential form is a description of a possible format for a sentence. It may be totally vague (e.g., simply S), more precise than this but still partly vague (e.g., 'the man verb NP'), or totally precise (e.g., 'an armadillo smiled'). A sentential form of the last kind, which contains no elements of N (the set of nonterminal symbols), is called a *sentence*, and the language $L(G)$ *generated by* the grammar G is defined to be the set of sentences for that grammar. Thus G defines the language $L(G)$ because, together with the above definitions, it tells us precisely what $L(G)$ is.

We can use the definition of sentential form to justify the fact that a given string is a sentence of the language generated by a grammar. For instance, the string "1 2" is a sentence of the language defined above because each of the following is:

number
pos_number
pos_digit digits
1 digits
1 digit digits
1 pos_digit digits
1 2 digits
1 2

The first is a sentential form by the first case in the definition, and each subsequent one is a sentential form by the second case and the fact that the preceding one is a sentential form.

It is important to realise that the mathematical meaning of the word "generate" as specified here (and as used generally in formal language theory) is different from other meanings of that word that you may encounter. The definition here says nothing about how in practice one might, given a grammar, generate example sentences. Thus any connection with phrases like "natural language generation systems" is very remote. It might perhaps be better if a word like "define" was used instead, but the word "generate" is already firmly established in the literature.

The position of a grammar on the Chomsky hierarchy can be determined by examination of the rules of P (Figure 2.1). That is, the more constrained the rules are, the higher in the hierarchy is the grammar, and the smaller is the set of languages that can be defined.

2.6 Automata

An automaton can be pictured as having three components:

1. An input tape, divided into squares, each containing a symbol from an input alphabet. There is a tape head, positioned on one of the squares, initially the leftmost.

2. An auxiliary memory, whose behaviour is characterised by two functions:

   (a) A fetch function, which maps from the set of memory configurations to a set of symbols.

   (b) A store function, which maps a memory configuration and a control string to a memory configuration

3. A finite state control, which mediates between the above two components.

A recogniser operates by making a series of moves. Each move consists of reading the input symbol under the tape head, and probing the auxiliary memory by means of the fetch function. These two items of information, together with the current state of the finite state control, then determine the rest of the move. This consists of shifting the tape head one square left or right, or keeping it stationary; storing information into the memory; and changing the state of the control.

Moves are made until the input string has been read (the tape head is at the right hand end), and the control is in one of a designated set of final states. There may also be a condition on the state of the memory. If these requirements are satisfied, then the input string is in the language defined by the recogniser. If not, then it is not.

The position on the Chomsky hierarchy to which a particular recogniser corresponds is determined by the type of auxiliary memory that it incorporates. As shown in figure 2.1, the more restricted the memory functions are, the smaller is the set of languages that can be defined. The rule format is shown in that figure assuming that A and B are nonterminals, x is any string of terminals, α, β, and γ are any strings of terminals and nonterminals, and δ is any non-empty string of terminals and nonterminals.

Type	Automaton		Grammar	
	Memory	Name	Rule	Name
0	Unbounded	Turing Machine	$\alpha \rightarrow \beta$	General rewrite
1	Bounded	LBA	$\beta\ A\ \gamma \rightarrow \beta\ \delta\ \gamma$	Context-sensitive
2	Stack	PDA	$A \rightarrow \beta$	Context-free
3	None	FSA	$A \rightarrow xB$, $A \rightarrow x$	Right linear

Figure 2.1: The Chomsky Hierarchy

2.6.1 Type 3 devices

Chapter 1 introduced the type 3 automaton (*finite state automaton*), which can be described by a *finite state transition network*. A FSTN describes an automaton by having nodes corresponding to the possible states of the machine and arcs corresponding to possible symbols that can be read on the tape. If the machine, starting in state s_1 and reading symbol x, changes to state s_2, then the network has an arc labelled x from node s_1 to node s_2. A type 3 automaton has no memory to consult in deciding which state to change to.

The equivalent grammars are those whose rewrite rules have a maximum of one non-terminal symbol, which must be the rightmost symbol, hence the name *right linear grammar*. Formally equivalent are the left linear grammars, in which the single non-terminal must be leftmost.

The languages describable are also precisely those that can be expressed as regular expressions, in which symbols may be combined by concatenation, alternation and repetition. Hence they are often known as *regular languages*. An example regular language is described by:

(det adj* noun) | pronoun

which can be glossed as those languages consisting of the single symbol 'pronoun', or 'det' followed by any number of 'adj's followed by a 'noun'. One equivalent right linear grammar has the rules:

$$\left\{\begin{array}{l} \mathrm{S} \rightarrow \mathrm{pronoun}, \\ \mathrm{S} \rightarrow \mathrm{det\ N}, \\ \mathrm{N} \rightarrow \mathrm{adj\ N}, \\ \mathrm{N} \rightarrow \mathrm{noun} \end{array}\right\}$$

An equivalent FSA is described in Figure 2.2 with initial state 1 and final state 3.

Start state	Input symbol	End state
1	pronoun	3
1	det	2
2	adj	2
2	noun	3

Figure 2.2: Specification of FSA

2.6.2 Type 2 devices

The type 2 grammars are the well-known *Context Free Grammars* (CFG), in which a single non-terminal symbol appears on the left hand side of a rule. The name "context-free" refers to the fact that the expansion of a non-terminal does not depend on the context in which it appears.

The type 2 automaton is the *Push Down Automaton* (PDA), which is a finite state automaton enriched with a stack. In making a transition, a PDA can push a symbol onto the stack, and in determining which transitions are legal, the symbol on the top of the stack may be examined and popped.

A typical context-free language which is not a right linear grammar would be the language $a^n b^n$, that is the language containing those strings which consist of some number (n) or as followed by the same number (n) of bs. The rules for one CFG for this language are:

$$\left\{\begin{array}{l} \mathrm{S} \rightarrow \mathrm{a\ S\ b}, \\ \mathrm{S} \rightarrow \epsilon \end{array}\right\}$$

and an equivalent PDA is described in Figure 2.3.

start state	input symbol	stack before	end state	stack after
0	a	Z	1	0Z
1	a	0...	1	00...
1	b	0...	2	...
2	b	0...	2	...
2	ϵ	Z	0	

Figure 2.3: Specification of PDA

2.6.3 Type 1 devices

The type 1 grammars are called the *Context Sensitive Grammars* (CSG). A rule in such a grammar must rewrite at most one non-terminal from the left-hand side, but contextual restrictions may be imposed on this, hence the name. The non-terminal must not rewrite as the empty string ϵ.

The same constraint is in evidence in the definition of the type 1 automaton, the *Linear Bounded Automaton* (LBA), which says that the amount of auxiliary memory must not exceed the length of the input string.

Some typical context-sensitive languages that are not context-free languages are $a^n b^n c^n$, the language consisting of all strings with a number (n) of as folowed by the same number of bs and then the same number of cs; and ww, the language consisting of all strings whose first half is the same as their second half.

One grammar for the first of these has the following rules:

$$\left\{\begin{array}{l} \mathrm{S} \rightarrow \mathrm{a\ S\ B\ C}, \\ \mathrm{S} \rightarrow \mathrm{a\ b\ C}, \\ \mathrm{b\ B} \rightarrow \mathrm{b\ b}, \\ \mathrm{b\ C} \rightarrow \mathrm{b\ c}, \\ \mathrm{c\ C} \rightarrow \mathrm{c\ c}, \\ \mathrm{C\ B} \rightarrow \mathrm{B\ C} \end{array}\right\}$$

Note that the last rule in this grammar as such does not fit the format $\beta\ A\ \gamma \rightarrow \beta\ \delta\ \gamma$ we mentioned above for context-sensitive rules. An alternative characterization of context-sensitive grammars is provided in H. R. Lewis and C. H. Papadimitriou (1981, p. 302):

> A grammar is said to be context-sensitive if and only if each rule is of the form $u \rightarrow v$ where $|v| \geq |u|$.

2.6.4 Type 0 devices

Finally, the type 0 devices are the *General Rewrite Grammar*, in which there are no restrictions on the form of the rules, and the *Turing Machine*, which has an unbounded auxiliary memory. Any language for which there is a recognition procedure can be defined using these devices, but in general, the recognition problem is not decidable.

2.7 Closure Properties

Since languages as we have defined them above are just sets, we can perform operations such as union and intersection upon them. It is often easier to define a language by such operations on other languages than directly. Hence it is of interest to know whether a given class of languages is closed under various operations, e.g.,

- Is the union of two context-free languages a context-free language?
- What class of language is obtained by intersecting a context-sensitive language with a regular set?

Proofs exist for almost all such questions, the details of which we will not be concerned with here. We merely note that:

1. All classes are closed under union with themselves. So, for instance, if we take two Type 2 languages then the set of strings that belong to at least one of the languages is itself a Type 2 language.
2. All classes are closed under intersection with regular sets (languages definable by Type 3 devices). So, for instance, if we take a Type 2 language and a Type 3 language, then the set of strings that belong to *both* languages is a Type 2 language.
3. The class of context-free languages is not closed under intersection with itself.

The proof of the last of these is as follows. Let:

$L_1 = \left\{a^n b^n c^i | n \geq 1 \text{ and } i \geq 0\right\}$
$L_2 = \left\{a^j b^n c^n | n \geq 1 \text{ and } j \geq 0\right\}$

L_1 and L_2 are both context-free languages. But:

$$L_1 \cap L_2 = \{a^n b^n c^n | n \geq 1\}$$

which is not context-free.

2.8 References

It is hard to find an easy introduction to this material. Hopcroft and Ullman (1979) is a detailed formal discussion of the classes of grammar and automata discussed above, and also contains chapters on closure properties and complexity of recognition.

Do not be misled by the terminology often used in texts on mathematical linguistics, which calls the vocabulary an *alphabet* and each sentence in the language under consideration a *word*. We will always refer to the members of the vocabulary as words or possibly morphemes.

- Hopcroft, J. E. and Ullman, J. D., "Introduction to Automata Theory, Languages and Computation", Addison-Wesley, 1979.

- Lewis, H. R. and C. H. Papadimitriou (1981, p. 302): *Elements of the theory of computation*. Englewood Cliffs: Prentice Hall.

2.9 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- According to the formal definitions, how is a language (in general) related to its vocabulary?

- What has to be specified in order to define a grammar?

- Why are different kinds of automata associated with different classes of languages?

- What are the main components of an automaton?

- What does the word "generate" mean, in the technical sense used here?

- What is a type 3 device, what is the corresponding type of language and what kinds of rules does one have to use in defining such a language?

- Name a typical example of a context free language that is not a regular language.

- Name a typical example of a context sensitive language that is not context free.

- What sort of input can formal language theory give into the study of languages?

What you should be able to do now:

- Explain the relations between automata and grammars.
- Cite examples of languages of different types and give grammars for them.
- Parse and generate with grammars of types 1, 2, and 3 by hand.

Chapter 3

Formal Languages and Natural Languages

3.1 Aims of this Chapter

- To show what is known about where natural languages fall in the classification of languages introduced in the last chapter.
- To give some intuitions about why this is of interest to Computational Linguistics.

3.2 Placing Natural Languages in the Chomsky Hierarchy

In trying to place natural languages in the hierarchy of formal languages described in the last chapter, we will be subject to two opposing considerations. On the one hand, we would like to be able to place them as high (weak) as possible, for several reasons:

- The weaker the type of grammar, the stronger the claims we make about possible languages.
- The weaker the type of grammar, the greater the efficiency of the parsing procedure.

On the other hand, we may be forced to use a stronger grammar for various reasons:

- To capture the bare facts about actual languages.
- To provide for more perspicuous or elegant analyses, or to make for more compact grammars.

Of course, perspicuity and elegance are in the eye of the beholder, but we will introduce some concepts here to clarify the idea.

Of itself, a formal description of a language, like a grammar or automaton, defines only a property of strings, i.e. set membership. However, the use of non-terminal symbols in a grammar that is sufficiently constrained as to rewrite only a single symbol imposes a hierarchical structure on strings that reflects the course of derivation.

The capacity of a generative grammar to describe a set of strings is called its *weak generative capacity*. For any given language, there are an infinite number of grammars that characterise it, i.e. of equivalent weak generative capacity.

The capacity of a generative grammar to associate structures with strings is called its *strong generative capacity* This is not a notion that has a precise mathematical definition, but rather a linguistic notion that is motivated by considerations of ambiguity, paraphrase and the grammatical relations that hold between parts of a sentence. An elegant grammar is one whose strong generative capacity corresponds to our linguistic intuitions about such questions.

3.3 Finite State Languages

It is fairly obvious that finite state languages are inadequate in strong generative capacity to describe natural languages. A regular expression for simple subject-verb-object sentences in English, viz:

((det adj* noun) | pn) verb ((det adj* noun) | pn)

obviously fails to capture the common structure of the subject and object noun phrases. What is less obvious (and more contentious) is that they are also weakly inadequate, as demonstrated by an argument that goes as follows (reported in Pullum and Gazdar, 1982):

The set (1):

(1) {A white male (whom a white male)^n (hired)^n hired another white male. $| n \geq 0\}$

is the intersection of English with the regular set (2):

(2) {A white male (whom a white male)^i (hired)^j another white male. $| i, j \geq 0\}$

But (1) is not regular (because the set $a^n b^n$ is not). Now since the regular sets are closed under intersection, if English was a regular language, since (2) is a regular language then (1) would be also. Hence English is not a regular language.

It is centre-embedding that causes a language to be non-regular, and centre-embedded constructions are notoriously difficult for humans to parse. Contrast the acceptability of the centre-embedded (3) with the right branching (4) and left-branching (5) structures:

(3) The bath the plumber the firm your mother recommended sent put in is cracked.

(4) On the table by the cupboard under the stairs in my house in London.

(5) My best friend's mother's hairdresser's Afghan hound's fur coat.

Nevertheless, there are good reasons for treating natural languages as of greater than regular power. First, there are several Central Sudanic languages in which centre-embedding appears to be commoner and more acceptable than in English. Secondly, it is difficult to justify imposing any strict bounds on the depth of centre-embedding, as acceptability seems to trail off, rather than cut-off, as it increases. Thirdly, if we were to impose an upper limit, say 4, on the depth of embedding, and to write a finite state grammar that would weakly characterise the string set thus obtained, such a grammar would be much larger, and less strongly adequate, than the equivalent context-free grammar. One approach that has been advocated is to explain the difficulty of processing by assuming that humans have a context-free grammar that is interpreted by a machine with a limited stack, and hence provably equivalent to a finite state grammar. The limit on the size of the stack is explicable in terms of limited human memory. Hence we have a distinction between a *competence* grammar, and its behaviour in *performance*.

3.4 Deterministic Context Free Languages

Finite state languages are not the only languages that can be recognised in linear time. A larger class of such languages is one that has been greatly studied in Computer Science, the Deterministic Context Free Languages (DCFLs). The DCFLs are those languages that are accepted by a deterministic PDA. The standard argument that natural languages are not DCFLs is that they are ambiguous. However, in terms of string sets, the crucial question is not whether NLs are ambiguous, but whether they are *inherently ambiguous*. An inherently ambiguous language is one for which there is no unambiguous grammar. Note that a grammar for a language may be ambiguous, that is, associate more than one distinct derivation tree with a string in the language, without that language being inherently ambiguous:

S → if b then S else S
S → if b then S
S → a

gives for the sentence:

if b then if b then a else a p

the two derivation trees shown in Fig 1 and Fig 2.

However, it is a simple matter to give an unambiguous grammar that generates the same language:

S1 → if b then S1

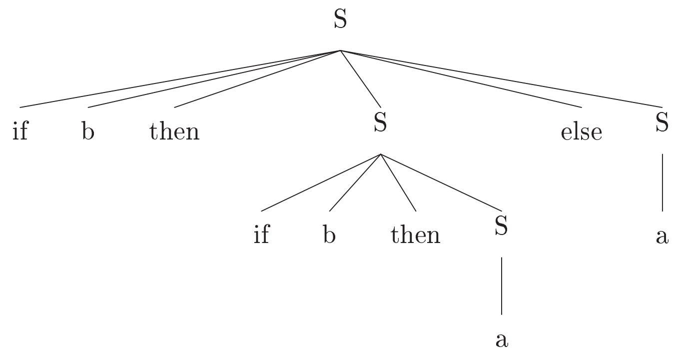

Figure 3.1: First derivation

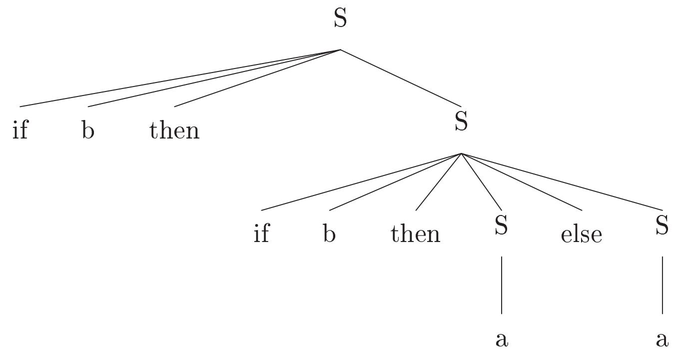

Figure 3.2: Second derivation

S1 → if b then S2 else S1
S1 → a
S2 → if b then S2 else S2
S2 → a

Note the following result. There cannot be a general algorithm to determine whether an arbitrary CFG is ambiguous, or to determine whether a given ambiguous CFG generates an inherently ambiguous language.

It is possible that a large subset of NL could be defined by means of an unambiguous grammar. The main problem with this is, as so often, a linguistic one - changing a grammar changes its strong generative capacity, and we would like an English string that is ambiguous in meaning to be associated with alternative derivations.

But again, this is not an absolute consideration. Consider compound nominals such as (6) and (7):

(6) plastic baby pants

(7) left engine fuel pump suction line

A grammar that will give all possible structures is:

CN → CN CN
CN → N

The number of parses associated with a compound nominal by this grammar grows as the so-called Catalan series (Catalan(n) is the number of ways of binarily bracketing a string of n elements):

1 2 5 14 42 132 469 1430

giving (7), for instance, 132 parses. We might very well consider using a canonical grammar of the form:

CN → N*

or, more precisely, something like:

CN →
CN → N CN

for recognition purposes, and leave the precise structure to be determined by some semantic component.

3.5 Context Free Languages

So, arguments from ambiguity are at best a demonstration that DCFGs are not strongly adequate for describing NLs. Are there any phenomena that would demonstrate that CFGs are not weakly adequate?

Notice that the standard dismissal on the basis of subject-verb agreement is not an argument against even strong adequacy. Consider the following schema, which expresses the facts of subject-verb agreement:

S → NP[num=α] VP[num=α]

Since for any sentence α could only have the value *singular* or *plural*, the same could be expressed by the following two rules:

S → NP[num=singular] VP[num=singular]
S → NP[num=plural] VP[num=plural]

which, given an appropriately extended (large, but still finite) set of nonterminals, could be expressed quite adequately by the two conventional rules:

S → NP_singular VP_singular
S → NP_plural VP_plural

The same argument can be made about other phenomena that are readily described schematically using features. As long as a finite set of features is used and each has only a finite number of possible values, then the result can always be stated (perhaps with a very large number of rules and nonterminals) by a standard CFG.

Other constructions that have been claimed to render NLs weakly greater than context-free power include the "respectively" construction in English, as in

(8) Harry, Rod and David love Bev, Sandra and Mary respectively.

Arguments about this construction are generally based on the assumption that there must be two conjunctions of NPs, each of which has the same number of basic components. Thus the following is illegal:

(9) Harry and Rod love Bev, Sandra and Mary respectively.

However, the agreement of the number of NPs is a matter of semantics, and this is shown by the fact that the following is quite acceptable:

(10) The three boys love the two girls and the gerbil respectively.

Thus the deviance of (9) is of the same type as that in:

(11) Our three main weapons are fear, surprise, ruthlessness and a fanatical devotion to the Pope.

and attributable to semantic, not syntactic, considerations.

Recently, however, certain constructions have been discovered that do appear to show that certain NLs are not even weakly context-free. The first example is that of subordinate infinitivals in Dutch and Swiss German. In Dutch, the translation of the English (12) is as in (13)

(12) ...that John saw Pete help Mary make the children swim.

(13) ...dat Jan Piet Marie de kinderen zag helpen laten zwemmen.
("that John Pete Mary the children saw help make swim")

Now strictly, the grammatical sentences of Dutch displaying this phenomenon are of the form $a^n b^n$, and thus can be weakly generated by a context-free grammar. However, in Swiss German (but not in Dutch), certain verbs demand dative rather than accusative objects, and therefore a CFG is not even weakly adequate. We can demonstrate this by arranging for all accusative noun phrases to come before all dative noun phrases, (by intersection with the regular set of strings of the form ${NP_a}^i {NP_d}^j {V_a}^k {V_d}^l$). Then the grammatical clauses have the structure ${NP_a}^n {NP_d}^m {V_a}^n {V_d}^m$. The language has the form $a^n b^m c^n d^m$, which cannot be generated by a context-free grammar.

Another example is from Bambara, a language spoken in Senegal. In Bambara, compound nouns are formed by concatenation of noun stems, as in (14) - (16)

(14) wulu
dog

(15) wulu-filela
dog watcher

(16) wulu-filela-nyinila
dog watcher hunter

One can also form a noun "N - o - N" meaning "whatever N", so the Bambara vocabulary also contains words of the form (17) - (19)

(17) wulu-o-wulu
whatever dog

(18) wulu-filela-o-wulu-filela
whatever dog watcher

(19) wulu-filela-nyinila-o-wulu-filela-nyinila
whatever dog watcher hunter

That is, Bambara has constructions of the form $\{w\text{-o-}w \mid w \epsilon \Sigma *\}$, which is not a context-free language.

3.6 References

Section 4.6 of Gazdar and Mellish is a short summary of the relation between natural languages and the classes in the Chomsky hierarchy. The structure of this chapter is, however, based on Pullum and Gazdar (1982). See this paper for many of the original references.

- Pullum, G. and G. Gazdar (1982), "Natural Languages and Context-Free Languages", *Linguistics and Philosophy*, vol. 4, 471-504.
- Gazdar, G. (1985), "Applicability of Indexed Grammars to Natural Languages", Report CSLI-85-34, Center for the Study of Language and Information, Stanford University.

3.7 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- What is the difference between the weak and strong generative capacities of grammars?
- What is centre embedding, and why can it not be described in finite state languages?
- What are the two examples given of natural language phenomena that are not context free, and how do the natural language phenomena relate to simple formal languages that are not context-free?

What you should be able to do now:

- Explain how formal language theory has been applied to natural language syntax, the basic results and limitations of this approach.
- Cite examples of phenomena that have been claimed to occur in NLs that would make them non-context-free.

Chapter 4

DCGs as a Grammar Formalism

4.1 Aims of this Chapter

- To introduce the definite clause grammar (DCG) formalism.
- To show how a DCG can be regarded as a shorthand for a description of the language written in logic.
- To show the basis of a context-free grammar of English, using X-bar theory and the DCG notation.
- To show how a feature-based notation for grammars can capture appropriate generalisations but not prevent the grammar from being context-free (because the features all take a finite number of possible values).
- To discuss the distinction between *complementation* and *modification*.

In this chapter, we are looking at definite clause grammars (DCGs) as a descriptive formalism, independently of how they may be used for recognition or parsing. In later chapters we will see that DCGs can be given different procedural interpretations.

4.2 The DCG Notation

DCGs provide a grammar notation that extends the basic notation of context-free grammars. Thus, for instance, to write the equivalent to the context-free rules:

$$\text{a} \rightarrow \text{b c d}$$
$$\text{a} \rightarrow \epsilon$$

where ‘a’, ‘b’ and ‘d’ are nonterminals and ‘c’ a terminal, in DCGs one writes:

```
a --> b, [c], d.
a --> [].
```

In DCGs one can also annotate nonterminals with information about features, using a positional notation, and one can specify that certain feature values should always be the same, as in number agreement:

```
s --> np(Num), vp(Num).
```

Features can be arbitrary Prolog terms, and can also be used to build up structural information, as in the traditional way of constructing parse trees:

```
s(s(NP,VP)) --> np(NP), vp(VP).
```

So much for the basic DCG notation. In addition, the following extensions are often used:

- Items on the right hand side of rules can be disjunctions (;) as well as conjunctions, with round brackets being used where the relative scopes are unclear:

  ```
  adjectives --> []; (adjective, adjectives).
  ```

- Variables (and, in fact, arbitrary terms) can be used as terminals and nonterminals (but not on the left hand side of a rule). This allows various “meta-level” constructions to be defined and the values of features to be terminal symbols, e.g.

  ```
  zero_or_more(X) --> [].
  zero_or_more(X) --> X, zero_or_more(X).

  vp --> verb(Particle), np, [Particle].    % pick the ball up
  ```

- A sequence of terminals can appear after the nonterminal on the left hand side. This can be used to express directly rewrite rules of the form:

  nt1 t1 t2 t3 $\rightarrow \ldots$

  For instance, the following says intuitively “if you are looking for a positive verb followed by the word ‘not’, then you can be satisfied by finding a negative verb”:

  ```
  verb(positive), [not] --> verb(negative). % did not --> didn't
  ```

- Arbitrary Prolog goals can be included in rules, being placed within curly brackets. This can be used to carry our arbitrary computations, but if one wishes to use DCGs as a serious grammar-writing framework then resorting to lots of Prolog code is counter-productive. The best use of this device is to express extra conditions on feature values, hence capturing the effect of many rules in one. For instance:

```
s --> np(Num), vp(Num), {legal_number(Num)}.
```

- The Prolog "cut" can appear in the right hand side of rules. Although this can improve the *procedural* behaviour of the rules, it has no *declarative* reading and indeed often destroys any declarative reading of the rules. Its use is therefore discouraged; the same procedural effect can often be obtained by constructions like the Prolog conditional (`P -> Q; R`), which is sometimes allowed in DCGs and whose reading is more declarative.

4.3 Translation of DCGs into Logic

How is it that Prolog, which is a theorem proving interpreter, can be used directly as a parser for CFGs? To understand this, we need to reconstrue context-free rules as logical axioms. Then proving a sentence to be a theorem that follows from these axioms will be equivalent in all respects to parsing. This view of the relation between logic and grammar is a very powerful and attractive one, that is normally referred to by the slogan *Parsing as Deduction*.

Context-free grammars and DCGs can be viewed as logical axioms by the simple step of considering the symbols of the grammar (terminals and nonterminals) as predicates that hold of strings of words. The rule:

$$\mathrm{s} \rightarrow \mathrm{np}, \mathrm{vp}.$$

can be considered as an axiom of the form:

$$\forall NP \forall VP \forall S\ np(NP) \wedge vp(VP) \wedge concat(NP, VP, S) \supset s(S).$$

We can read this as: for all strings of words NP, VP and S, if the predicate *np* is true of the string of words NP, and *vp* of the string VP, and S is the concatenation of NP and VP, then the predicate *s* is true of S.

The predicate *concat*, also known as *append*, should be very familiar to you from your Prolog programming. In its compilation of DCGs, however, Prolog does not use the standard encoding of strings of words as lists, but represents lists in terms of differences. Thus, whereas above the nonterminal 's' was translated into a logic predicate with arity 1, Prolog uses a different translation, where each nonterminal is interpreted as a predicate with two (+ however many features are mentioned) arguments. Here is how 's' appears in this interpretation:

`s(S1,S2)` - there is a sentence in the portion of the string between S1 and S2.

We will investigate what sorts of things `S1` and `S2` might be below, but for now it suffices that between them they demarcate a portion of string, `S1` indicating somehow where the portion starts and `S2` where it ends. Given a similar translation for all the other nonterminals, here is the alternative encoding of the above context-free rule:

$$\forall S0 \forall S1 \forall S\ np(S0, S1) \wedge vp(S1, S) \supset s(S0, S).$$

or in standard Prolog notation:

```
s(S0,S) :- np(S0,S1), vp(S1,S).
```

A DCG rule which introduces a lexical item, such as:

```
det --> [the]
```

could correspond to an axiom of the form

$$\forall S \; det(S) \subset the(S).$$

but we don't want to have a predicate for each lexical item. An auxiliary predicate `connects` or `'C'` is used. In Prolog:

```
det(S0,S) :- connects(S0,the,S).
```

Now, to define the predicate `connects` we need to take into account how the string to be analysed is represented. One possibility is to have words in the string identified by their numerical positions; in this case the string to be parsed is represented as a set of `connects` facts in the Prolog database:

```
connects(1,john,2).
connects(2,saw,3).
connects(3,her,4).
connects(4,duck,5).
```

On the other hand, we can represent a portion of the string by a pair of Prolog lists, the first of which holds the desired portion and everything coming after it and the second of which holds everything that comes after it in the whole string. The portion of string represented is then the *difference* between these lists (hence the pair is often called a *difference list*). For instance, the portion `[a,b]` can be represented by the pair of lists `[a,b,c|X]` and `[c|X]`. You could think of each of the two lists as "pointing" to the position in the string corresponding to its first element, in the same way as was done with the numerical positions, but in this representation it is easy to find out what word appears in that position without going to the database. Given this representation of portions of string, the definition of `connects` has a single clause:

```
connects([Head|Tail],Head,Tail).
```

In actual fact, most Prolog systems assume that the programmer wants to use the list notation and, whenever a call to `connects` notionally appears in the translation of a DCG rule, this is replaced by the result of running this one-clause definition. Thus the above `det` clause would actually come out as:

```
det([the|S],S).
```

When a DCG has been compiled into Definite Clauses as described in this section, the parsing problem for a string such as all cats chase Fido may be represented as the goal:

```
:- s([all,cats,chase,fido],[]).
```

4.4 Basics of a DCG Grammar for English

In this section, we show how DCGs can be used to express a context-free grammar for a reasonable fragment of English. We saw in Chapter 3 that adding features to the categories in a context-free grammar adds to expressivity but does not change the context-freeness of the result, as long as there are only finitely many features with a finite number of possible values. The grammar presented here will be of this kind, and thus we are essentially using the DCG notation to abbreviate a rather large context-free grammar. Although we saw in Chapter 3 that natural languages are not always context free, this grammar (and its extensions with unbounded dependencies in the next chapter) demonstrate that substantial portions of natural languages can be described in this way. The presentation here assumes a basic knowledge of English syntax.

Notice that in this Chapter we are only concerned with DCGs as a way of giving correct descriptions of natural language sentences. We are not concerned with whether these DCGs will actually do anything interesting when "run" with a Prolog system. Indeed, some of the DCGs that we present are not very useful as programs when loaded into a bare Prolog system. We will look further into what sorts of operations we might perform with DCG grammars in a later chapter.

4.4.1 Basic Features

For nouns, we treat only count nouns, so the only feature we need here is *num*ber.

> *num* = {*sing*,*plur*}

For verbs, we need to capture the number of different forms that can arise. For this, we use the feature *vform*:

> *vform* = {*fin*,*bse*,*inf*,*prp*,*pas*,*psp*}

Prepositional phrases may be required (for instance, by a verb that it comes after) to contain a particular preposition (see below on subcategorisation). Thus it is convenient to introduce a feature *pform* that takes the set of surface forms of prepositions as possible values:

> *pform* = {*of, to, with, about, ...*}

4.4.2 X-bar Theory

The grammar is based around the categories noun, verb, adjective and preposition because each of these generally forms the focus of a larger *phrasal category* having a distinctive structure. Thus, from nouns we can build noun phrases, from verbs, verb phrases, and so on. We can often recognise categories intermediate between the lexical categories and the phrasal categories (*maximal projections*) they correspond to. To describe these phenomena, many syntactic theories use the notion of *bar level.* Lexical categories (e.g. noun) have bar level 0, phrasal categories (e.g. noun phrase) have bar level 2 or sometimes 3, and intermediate categories are notated accordingly (this approach is called "X-bar theory"). We will assume a maximum of two bars.

$$bar = \{0,1,2\}$$

One advantage of this sort of notation is that it makes clear the regularities in the structure of different phrases. For instance, the following patterns seem to be appropriate, whether X is noun, verb, adjective or preposition:

```
X(2) --> specifier, X(1).
X(1) --> X(1), modifier.
X(1) --> modifier, X(1).
X(1) --> X(0), complement.
```

A specifier is a part of a phrase which (if it appears at all) only appears once, at the very beginning. As one works inwards from the outside of the phrase, one next encounters optional modifiers (of which there may be several), and then finally the lexical category and the complements it requires. Modifiers and complements are discussed in more detail later.

Notice that in a rule that has X on its left hand side, there is an X in the right hand side, with its bar level the same or one less than that of the LHS. If we were to flesh out these rules with other features, then for certain features, for instance *num*, *vform* and *pform*, they would show that the X on the RHS always had the same value as the X on the LHS. The X on the RHS is called the *head* of the phrase, and the shared features are called *head features*.

4.4.3 Noun Phrases

Noun phrases are built by rules such as:

```
n(2,Num) --> pronoun(Num).
n(2,Num) --> proper_noun(Num).
n(2,Num) --> det(Num), n(1,Num).
n(2,plur) --> n(1,plur).
n(1,Num) --> pre_mod, n(1,Num).
n(1,Num) --> n(1,Num), post_mod.
n(1,Num) --> n(0,Num).
```

The first two of these introduce two classes of words, pronouns and proper nouns, that make noun phrases on their own.

4.4.4 Verb Phrases

Verb phrases are described by rules such as:

```
v(2,Vform,Num) --> v(1,Vform,Num).
v(1,Vform,Num) --> adv v(1,Vform,Num).
v(1,Vform,Num) --> v(1,Vform,Num), verb_postmods.
v(1,Vform,Num) --> v(0,Vform,Num).
```

Verb phrases have no obvious specifiers. As modifiers they can have adverbs and prepositional phrases.

Notice that verb phrases, not just verbs, have values for *vform*. *vform* is a head feature, and so the rules show the verb phrase inheriting its *vform* from the main verb. Having *vform* as a feature of VPs enables the grammar to specify that in certain contexts a VP is required and that its main verb must be of a particular form.

4.4.5 Prepositional Phrases

Prepositional phrases are described by rules such as:

```
p(2,Pform) --> p(1,Pform).
p(1,Pform) --> adv p(1,Pform).
p(1,Pform) --> p(0,Pform), n(2,_).
```

There are not many possible forms for PPs in English, though adverbs can act as modifiers, as in 'slowly past the window'.

4.4.6 Adjective Phrases

Adjective phrases are described by rules such as:

```
a(2) --> deg, a(1).
a(1) --> adv, a(1).
a(1) --> a(0).
```

Adjective phrases usually consist of single adjectives, but it is possible for these to be accompanied by an indication of degree and some number of adverbs as modifiers, as in 'very commonly used'.

4.4.7 Sentences

We can now present the basic rule for a sentence:

```
s(Vform) --> n(2,Num), v(2,Vform,Num).
```

4.4.8 Complementation and Modification

The details of what gets introduced at what bar level may vary from grammar to grammar. However, the general intuition is quite clear. A particular lexical item licenses certain complements, which are more tightly bound to it than modifiers. The latter can typically occur in any number with any word of the class. We say that a certain item *sub-categorises* (for) certain complements, and these have to be associated with that item in the lexicon. A standard way to do this is to associate a feature with a word, called its *subcat* feature, whose value is referred to in a specific phrase structure rule[1].

$$subcat = \{intrans, trans, emotion, exchange\_of\_views, ...\}$$

The first two of these are named after traditional categories of verbs - an 'intransitive' verb (e.g. 'dream') expects no complements, whereas a 'transitive' verb (e.g. 'hit') expects a single object noun phrase. But there are many more possibilities for what a verb, noun, adjective or preposition can require to follow it.

Given the *subcat* feature, instead of the `X(1) --> X(0)` rules above, we will have a set of rules, e.g.

```
n(1,Num) -->
   n(exchange_of_views,Num), p(2,with), p(2,about).   % "argument"
n(1,Num) -->
   n(plan,Num), v(2,inf,_).                            % "plan"
a(1) -->
   a(prop), s(fin).                                    % "afraid"
a(1) -->
   a(emotion), p(2,of).                                % "fond"
v(1,Vform,Num) -->
   v(ditrans,Vform,Num), n(2,_), n(2,_).               % "give"
v(1,Vform,Num) -->
   v(obj_raising,Vform,Num), n(2,_), v(2,inf,_).       % "prefer"
v(1,Vform,Num) -->
   v(perfect,Vform,Num), v(2,psp,_).                   % "have"
v(1,inf,_) -->
   v(to,inf,_), v(2,bse,_).                            % "to"
p(1,Pform) -->
   p(takes_pp,_), p(2,of).                             % "out"
```

Notice how the complements of an item are always maximal projections (phrasal categories with bar level 2). Another important characteristic of complements is that their head imposes *selectional restrictions* upon them. That is, a particular head only makes sense with a particular type of complement, semantically speaking. For example, 'kill' selects for a living object, but is not particular about its subject. 'Murder' requires an entity that can be considered responsible for its actions as its subject, while 'assassinate' requires the rather specialised semantic property "political figure" to be true of its object.

Notice the way that the verb 'have' has been analysed above as a verb which subcategorises for a following VP with *vform psp*. Other auxilliary verbs can be handled similarly.

[1]Note that in the term structures that we are using to represent categories, the features *bar* and *subcat* occupy the same position - the different values of *subcat* replace the *bar* value 0. This is just a notational convenience.

A given lexical item may occur in several different subcategorisation patterns. For instance, there is a class of verbs like 'give' that occur in the patterns "give y x" and "give x to y". It is a simple matter to enter these alternatives in the lexicon, or at least to have some process going on in the lexicon that produces them both from some single specification of the properties of the word.

GPSG

Generalised Phrase Structure Grammar (GPSG) was devised in the early 1980's. It represented a reaction to the prevailing assumption (generally assumed within the framework of Transformational Grammar in Linguistics at the time) that context-free languages were clearly inadequate to describe NLs. Although we have seen that in fact there do seem to be some NL phenomena that are not context-free, nevertheless significant grammars for different languages were written in GPSG.

GPSG was unusual for the formal precision with which it was specified. It was also unusual in the way it was stated. A context-free grammar of anything as complex as a natural language would be huge and incomprehensible. A major feature of GPSG was the fact that the CFG was defined *implicitly*, by the interaction of a number of components in the grammar, for instance feature instantiation principles (which determine how features in rules can be instantiated), metarules (which specify that certain extra rules must exist as a consequence of others) and a separate definition of ordering and dominance principles.

A number of the ideas in the grammar of this chapter are inspired by approaches taken in GPSG.

- Gazdar, G., Klein, E., Pullum, G. and Sag, I. (1985), *Generalized Phrase Structure Grammar*, Blackwell/ Harvard University Press.

In the next chapter we will go on to consider how DCGs can allow us to describe some phenomena that are awkward to describe with context-free grammars - *unbounded dependency constructions*. Then we will look at procedural interpretations of DCGs.

4.5 References

Tutorial introductions to DCGs are given in Clocksin and Mellish (1981), Pereira and Shieber (1987) and Gazdar and Mellish (1989). DCGs were developed from the earlier work of Colmerauer and were originally presented in Pereira and Warren (1980).

- Clocksin, W. and Mellish, C., *Programming in Prolog*, Springer Verlag, 1981.

- Gazdar, G. and Mellish, C., *Natural Language Processing in Prolog*, Addison-Wesley, 1989.
- Pereira, F. and Shieber, S., *Prolog and Natural Language Analysis*, CSLI/ University of Chicago Press, 1987.
- Pereira, F. and Warren, D. H. D., "Definite Clause Grammars for Language Analysis - A survey of the formalism and a comparison with augmented transition networks", *Artificial Intelligence* Vol 13, pp. 231–271, 1980 (also appears in Grosz et al (eds) *Readings in Natural Language Processing*.

4.6 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- What are the five main extensions to basic DCGs that are described?
- Give three ways of representing strings and substrings in translating a DCG into logic.
- In X-bar theory, how many bars are assigned to a noun? a noun phrase? a verb? a verb phrase? a preposition? a prepositional phrase? an adjective? an adjectival phrase?
- State some possible values of the $vform$ feature and what they represent.
- Give an example of where a verb subcategorises for a given complement. Do the same for a noun and an adjective.

What you should be able to do now:

- Use DCGs to write grammars for context-free languages.
- Say whether a given DCG is equivalent to a context-free grammar.
- Explain how grammar description can be viewed as writing axioms in logic.
- Explain and exemplify the use of finite-valued features in a grammar for English, in particular the use of the *subcat* feature to mediate between lexical entries and grammar rules.
- Distinguish between complementation and modification and explain how this might be represented in phrase structure.
- Complete the grammar above by writing appropriate lexical entries and modifier rules, and extend its coverage to new lexical items.

Chapter 5

Unbounded Dependencies in DCGs

5.1 Aims of this Chapter

- To describe what "unbounded dependencies" are (and how they differ from "bounded dependencies") in natural language constructions.
- To show how gap threading can be used within a DCG elegantly to describe examples of unbounded dependency constructions.

Unbounded dependency constructions seem to be very common in natural languages, and yet they are awkward to describe in context-free grammars.

5.2 Unbounded Dependencies

In Chapter 4, we saw how the complementation properties of lexical items such as verbs could be encoded by means of a feature SUBCAT. This allowed such an item only to appear in the RHS of a rule when the other items in the RHS were appropriate. In this way, we can rule out unacceptable sentences such as (20)—(22):

(20) * Mary sneezed the dog.

(21) * Fido chased.

(22) * John persuades to eat a bone.

In an *unbounded* or *long-distance* dependency construction (UDC), a word's complementation requirements are not satisfied within the constituent immediately containing the verb. Such phenomena are called unbounded because there is no limit to the number of clause or phrase boundaries over which the dependency operates. Consider, for instance, the sentences (23)—(25):

(23) This book, I read.

(24) This book, I managed to read.

(25) This book, I believe Mary managed to read.

In each case, the object of the verb read does not immediately follow it, as it usually would, but appears at the beginning of the sentence. It is common to indicate that the topic this book is in fact the object of read by writing 't' (for trace) in the post-verbal position, and subscripting the 't' and the topic with the same letter, thus:

(26) This book$_i$, I read t$_i$.

If we annotate (23)—(25) to show relevant structure, as in (27)—(29):

(27) This book$_i$, [$_s$ I read t$_i$].

(28) This book$_i$, [$_s$ I managed [$_{vp}$ to read t$_i$]].

(29) This book$_i$, [$_s$ I believe that [$_s$ Mary managed [$_{vp}$ to read t$_i$]]].

we see that the sentences stay fine as the number of boundaries between the two subscripted elements increases.

(30)—(32) show the same annotations for some other unbounded dependency phenomena:

(30) The college$_i$ that [$_s$ I expected John [$_{vp}$ to want Mary [$_{vp}$ to attend t$_i$]]].

(31) What$_i$ [$_s$ do you believe that [$_s$ Mary told Bill that [$_s$ John had said t$_i$]]]?

(32) This film$_i$ is [$_{ap}$ easy for me [$_{vp}$ to persuade the children [$_{vp}$ not to see t$_i$]]].

These are examples of so-called *relative clauses*, *wh-questions* and *tough constructions* respectively.

5.3 Describing UDCs

Without theoretical commitment I will talk about the t$_i$ as a *gap* and the constituent with which it shares an index as the *filler*. We can look at an unbounded dependency in terms of three components:

1. The top of the dependency, i.e. the construction where the gap is introduced.

2. The bottom of the dependency, where the gap is discharged.

3. The middle, through which the gap information flows.

As the examples show, gaps may be introduced in a variety of constructions. Gaps are all discharged in the same way, though, as the non-appearance of a constituent where the rules require one. Sentences such as (21) and (22) would be fine if they discharged a gap:

(33) This is the cat that Fido chased.

(34) This is the cat that John persuades to eat a bone.

Unbounded dependencies are difficult to describe in context-free grammars because it is necessary to keep track of the requirement of a gap between the top and the bottom of the dependency. Since in CFGs categories are atomic, this means having separate categories (and rules) for notions like "a VP with no gap inside", "a VP with one gap inside", "a VP with two gaps inside", etc. To handle the syntax of UDCs in DCGs, we can augment our categories with a boolean feature *gap* which takes values *gap* and *nogap*. Then a noun-phrase with a *gap* value for *gap* will be able to appear as the empty string.

```
n(2,gap) --> []
```

We can introduce a gap in a sentence modifying a noun, i.e., a relative clause.

```
n(1) --> n(1), s(gap).
```

or

```
n(1) --> n(1), rel.
rel --> [that],s(gap).
```

How does the information flow between these? In particular, how is the value of *gap* partitioned between the RHS's of a grammar rule? Essentially, we want the gap to go to only one of the daughters.

(35) * This is the cat$_i$ that Fido persuades t$_i$ to chase t$_i$.

This daughter must not be the lexical head of the rule, hence *gap* is sometimes called a foot feature. In order to pass the gap to only a single daughter, we could add an auxiliary procedure, say `legal_gaps`, and call it from each appropriate rule:

```
legal_gaps(X,nogap,X).
legal_gaps(nogap,X,X).

s(X) --> n(2,Y), v(2,Z), {legal_gaps(Y,Z,X)}.
```

5.4 Gap Threading

We can eliminate calls to the auxiliary procedure by choosing a different representation for gap information[1]. This representation uses a pair of boolean arguments, one to represent gap information passed into a constituent, and one to represent gap information passed out. This gives us the following equivalences:

[1] In the same way as the call to `subcat` was eliminated in the translation from DCGs to Prolog.

Single Argument	*Pair of Arguments*
gap = gap	gapin = gap , gapout = nogap
gap = nogap	gapin = X , gapout = X

Of course, there are no constituents that have gapin = nogap; gapout=gap. (36) is rubbish:

(36) * The man that Mary loved John was.

Given this new representation, we can write rules directly, such as:

```
s(GapIn,GapOut) --> n(2,GapIn,GapMid), v(2,GapMid,GapOut).
```

Our gap introduction and gap elimination rules become:

```
n(1) --> n(1), s(gap,nogap).
n(2,gap,nogap) --> [].
```

5.5 Constraints on UDCs

Can a gap be passed into any of the RHS constituents that is not a lexical head? Consider (37) and (38)[2].

(37) Fido chased [$_{np}$ the dog$_i$ that [$_s$ t$_i$ bit a cat]].

(38) * The cat$_i$ that [$_s$ Fido chased [$_{np}$ the dog$_j$ that [$_s$ t bit t]]] was Tigger.

In (38) 'the cat' cannot be related to either of the gaps in the noun phrase 'the dog that t bit t'. This cannot be reduced to a problem of multiple gaps, because the same effect is seen when there is only a single gap. (39) and (40) are equally good, but if we try to make questions out of them, (41) is fine and (42) is bad.

(39) Bill believes that [$_s$ John claimed that [$_s$ Mary kissed someone]].

(40) Bill believes [$_{np}$ John's claim that [$_s$ Mary kissed someone]].

(41) Who$_i$ does Bill believe that [$_s$ John claimed that [$_s$ Mary kissed t$_i$]]?

(42) * Who$_i$ does Bill believe [$_{np}$ John's claim that [$_s$ Mary kissed t$_i$]]?

In these sentences, 'that' serves to introduce sentential complements (of 'claim' and 'believe'), not relative clauses, and such complements do not contain any gaps. The gaps in (41) and (42) correspond to the Wh (question) word. (42), with the gap inside an np, is much worse than (41) where it is inside an s.

In the grammar listed below a simplified treatment of this phenomenon has been included. All noun phrases which are not directly realised as gaps have their gap-threading arguments coinstantiated. This simple step prevents a gap being discharged within the np, because any constituent that contains a gap must have the arguments (gap,nogap). In what way is it an oversimplification to treat all nps as islands?

[2]To understand what is going on in (38), i.e. to see what this sentence would mean if it was grammatical, try replacing the offending 'that' and gap by 'such that' and 'it' respectively.

Not all languages share the constraint illustrated by (37) and (38). In Japanese, (43) is fine.

(43) $[[[[\mathrm{t}_i\ \mathrm{t}_j$ kawaigatteita $]_s$ $\mathrm{inu}_j]_{np}$ ga shinda$]_s$ kodomo_i wa$]_{np}$
fond-of *dog* *died* *child*

'the child such that the dog that he was fond of died'

To handle such a language, we would allow a gap to be passed into an np. In addition we would have to replace the atomic-valued gap feature with a list of gaps. There are a few cases in English that also show multiple gaps, e.g. (44):

(44) which violin_i is this sonata_j easy to play t_j on t_i

We will not deal with these.

Finally note the arguments to a top-level sentence:

```
top_s --> s(fin,nogap,nogap).
```

The grammar listed below summarises the discussion above. Number agreement has been omitted for clarity.

5.6 Extensions

Not all gaps are noun phrases. As well as (45), we have (the rather formal) (46):

(45) the man who I spoke to

(46) the man to whom I spoke

We can elaborate the gap-threading features to include information about the category of the gap, and have the following rules for discharging gaps.

```
np(gap(np),nogap) --> [].
pp(gap(pp),nogap) --> [].
```

As our grammar only allows relative clauses without relative pronouns, we would need to make further changes. These are left as an exercise for the interested reader, who should consider the syntax of forms such as (47):

(47) Those books, the wording on the covers of which everybody disagreed with, have been burned.

5.7 DCG Grammar

The following simple DCG grammar based on the discussions of this chapter is available in the file `udc_example.pl`. Note, however, that the grammar in this file won't necessarily execute properly under the standard DCG interpreter.

```
top_s --> s(fin,nogap,nogap).

s(Vform,G0,G) --> n(2,G0,G1), v(2,Vform,G1,G).

n(2,G,G) --> proper_noun.
n(2,G,G) --> det, n(1).
n(1) --> n(1), s(fin,gap,nogap).
n(1) --> n(0).

n(2,gap,nogap) --> [].

v(2,Vform,G0,G) --> v(1,Vform,G0,G).

v(1,Vform,G,G) --> v(intrans,Vform).
v(1,Vform,G0,G) --> v(trans,Vform), n(2,G0,G).
v(1,Vform,G0,G) --> v(obj_equi,Vform), n(2,G0,G1), v(2,inf,G1,G).
v(1,Vform,G0,G) --> v(scomp,Vform),[that],s(fin,G0,G).
v(1,inf,G0,G) --> v(to,inf), v(2,bse,G0,G).

v(intrans,bse) --> [run].         v(intrans,fin) --> [runs].
v(trans,bse) --> [chase].         v(trans,fin) --> [chases].
v(obj_equi,bse) --> [persuade].   v(obj_equi,fin) --> [persuades].
v(scomp,bse) --> [believe].       v(scomp,fin) --> [believes].
v(to,inf) --> [to].

det --> [a].                det --> [the].

n(0) --> [cat].             n(0) --> [dog].

proper_noun --> [fido].     proper_noun --> [tigger].

test1 :-
  top_s([the,dog,fido,chases,persuades,a,cat,to,chase,tigger],[]).
test2 :-
  top_s([a,dog,chases,the,cat,fido,persuades,to,run],[]).
test3 :-
  top_s([tigger,chases,the,dog,fido,believes,that,a,cat,chases],[]).
```

5.8 References

See Pereira and Shieber (1987) for a treatment of gap-threading and its origins. GPSG uses a feature called slash in a similar way as part of a large coverage grammar of English.

5.9 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- Why is it that relative clauses introduce *unbounded* dependencies?
- Where are the top and bottom of the dependency in "This is the cat that John persuades to eat a bone"?
- If gap threading is being used as described and X and Y are the two gap threading arguments of a phrase, what possible values can X and Y get?

What you should be able to do now:

- Explain what makes UDCs unbounded.
- Give examples of different UDCs.
- Give example gap-threading grammars, describe how they account for the data that they do, and show the patterns of gap arguments in example derivations.

Chapter 6

Computability and Complexity

6.1 Aims of this Chapter

- To introduce standard terminology and concepts that are used to analyse whether a given task can be performed by a computer and, if so, how efficiently. These come from the areas of Computability Theory and Complexity Theory.
- To show what results this can give us about the complexity of tasks involving languages of the different Chomsky types.
- To show that efficiency considerations hence lead one to prefer grammars with types high in the Chomsky hierarchy – a preference that pulls in the opposite direction to the requirements of expressivity investigated in the last chapter.

6.2 Problems and Algorithms

Most problems in Computer Science that can be stated precisely can be stated in the form of *algorithmic problems*. An algorithmic problem is a characterisation of a problem in terms of inputs and outputs. Inputs are for information that will be provided to or available to the problem solver, apart from the solution instructions themselves. The instructions will have to ensure that the problem is solved whatever the inputs are (within some specified set of possibilities). The outputs are for what is produced in solving the problem - how these are to relate to the inputs needs to be specified as part of the description.

An *algorithm* is the best possible type of specification as to how to solve an algorithmic problem. Traditionally, an algorithm specifies what to do in terms of a sequence of steps and must satisfy the conditions of finiteness, definiteness and effectiveness (Knuth 1976).

- Finiteness - a problem solver using the algorithm should always terminate after a finite number of steps, regardless of the input.
- Definiteness - each step of the algorithm must be precisely defined (and it must include a precise definition of how the next step is to be determined).
- Effectiveness - each step is simple enough that it can always be performed in a finite amount of time.

In addition, it is necessary to specify that an algorithm at each stage can only use information that is either given in the inputs or made available by previous steps, and that an algorithm specifies what the output is to be at the end.

6.3 Computability

An algorithmic problem is said to be *computable* if there is an algorithm which for every possible allowed input computes the correct output (and of course always terminates, as an algorithm must). Otherwise the problem is said to be *non-computable*.

This might seem a bit vague, as the definition given above of what can be a "step" of an algorithm is not precise. A precise definition can be given, in terms of a program for a particular kind of hypothetical, extremely simple, computer called a *Turing machine*. For a Turing machine, a "step" means executing a single instruction, which involves inspecting the single symbol at a fixed position on an input tape, possibly moving the tape one way or the other and changing what is recorded at that position, and then, according to what the inspected symbol is, selecting the next instruction for execution. This definition of what an algorithm can be is, however, not very useful, as actually writing programs for Turing machines is extremely laborious and it is believed that every non-trivial computing model (assuming that it is allowed arbitrary amounts of memory) must be equivalent to a Turing machine in terms of the definition of computability that arises (this is called the **Church-Turing hypothesis**). Hence we can be satisfied with algorithm steps that are effective in virtue of always succeeding in finite time when performed by the computers we are familiar with, when performed by competent human beings, etc. Conversely, if some problem has been proved to be non-computable then there is no point in searching for a solution algorithm aimed at any computing device yet conceived of.

An important class of algorithmic problem is the kind of problem that takes a single input and requires an output of either "yes" or "no". Such a problem is called a *decision problem*. A decision problem is a problem involving testing whether its input has a certain property, or, equivalently, whether its input is a member of a particular set. A decision problem which is computable is called *decidable*.

A set is called *recursive* if membership of that set is decidable [1]. Thus, for instance, the set of decimal notations of even numbers is recursive. It is easy to tell whether a given number in decimal notation is even, because one simply has to see whether the last digit is '0', '2', '4', '6' or '8'. However, not all sets are recursive. For instance there is no algorithm which, presented with two formulae of logic as inputs, determines whether the second is a logical consequence of the first (logical consequence is undecidable for first order logic).

If a set is not recursive, it may still be *recursively enumerable*, that is, there may be an enumeration of the elements of the set and an algorithm that, given any positive integer n, can compute the first n elements in that enumeration. If a set is recursively enumerable then we can approximate an algorithm for testing whether an item x is an element as follows. We simply invoke the enumeration algorithm for $n = 1$, $n = 2$, $n = 3$ etc. and at each stage look to see whether x is the last element in the sequence produced. This will not be an algorithm, because although it will terminate with the answer "yes" if x is an element, if x is not then it will never terminate (unless the set is finite, in which case it is recursive anyway). The set of consequences of a formula of first order logic is recursively enumerable.

6.4 Complexity

Analysis of computability is aimed at telling us whether a problem can be solved at all. Complexity analysis is aimed at giving us more information about how efficiently a given problem can be solved, and how efficient a given algorithm is.

6.4.1 Algorithm complexity

For convenience, we will now restrict attention to algorithmic problems with single inputs, although these notions extend straightforwardly to multiple inputs. Now the number of steps taken in executing the algorithm will usually depend on the complexity of the input. For instance, the number of steps taken to determine whether a string is in a given language might be expected to depend on the length of the string. Complexity theory deals in terms of how the amount of work done by an algorithm depends on some measure of the complexity of the input (usually called n). Sometimes it is possible to get an exact formula expressing the work in terms of a number of primitive operations (whose individual complexities do not depend on n) which is a function of n (for instance $25n^2 + 368$), but this amount of detail tends to be of little use as it fails to take into account the complexity of the individual steps (and this would depend dramatically on the peculiarities of the machine running the algorithm). In order to attempt to get at the inherent complexity of an algorithm, irrespective of where it might be run, complexity theory looks at how the amount of work increases as n increases. This gives us an idea of how the algorithm will perform if we

[1] Be careful that this meaning of "recursive" is quite different from that used in Linguistics and most of Computer Science.

$log_e(n)$	0	0.7	1.6	2.3	3	4.6	6.9
n	1	2	5	1×10	2×10	1×10^2	1×10^3
n^2	1	4	2×10	1×10^2	4×10^2	1×10^4	1×10^6
n^3	1	8	1×10^2	1×10^3	8×10^3	1×10^6	1×10^9
$100 \times n^3$	1×10^2	8×10^2	1×10^4	1×10^5	8×10^5	1×10^8	1×10^{11}
n^4	1	2×10	6×10^2	1×10^4	2×10^5	1×10^8	1×10^{12}
n^{25}	1	3×10^7	3×10^{17}	1×10^{25}	3×10^{32}	1×10^{50}	1×10^{75}
$1000 \times n^{25}$	1×10^3	3×10^{10}	3×10^{20}	1×10^{28}	3×10^{35}	1×10^{53}	1×10^{78}
2^n	2	4	3×10	1×10^3	1×10^6	1×10^{30}	1×10^{301}

Table 6.1: Some functions of n

give it bigger and bigger problems, as well as abstracting away from exactly how long the primitive operations take. Complexity theory thus deals in *order of magnitude reasoning* about the amount of work done as a function of n. Here are some possible answers that we might get for the complexity of an algorithm:

- *constant* - the amount of work does not depend on n.
- *logarithmic* - the amount of work behaves like $log_k(n)$, for some constant k.
- *polynomial* - the amount of work behaves like n^k, for some constant k. This is sometimes subdivided into the cases *linear* ($k = 1$), *quadratic* ($k = 2$), *cubic* ($k = 3$), etc.
- *exponential* - the amount of work behaves like k^n, for some constant k.

These are loose answers - for instance, two algorithms with complexity $n/4000$ and $50000n$ are both simply called "linear". The fact that this makes sense comes down to what we mean by "behaves like". As n gets larger, small differences like constant factors become less significant than questions like "what is the largest power of n involved?" Thus for every linear function of n and every quadratic function of n there is a value of n beyond which the quadratic function is always greater than the linear one. Similarly, for every polynomial function and exponential function there is a value of n beyond which the latter is always larger. This is illustrated in Table 1. The table shows that once n gets to 500 these functions are already ranked as one would expect from the complexity classes even though, for instance, $1000 \times n^{25}$ starts off competing quite well with 2^n and $100 \times n^3$ starts off competing quite well with n^4. The table illustrates the fact that there is a ranking of functions of n, as follows:

$\ldots$
2^n
$\ldots$
n^3
n^2
n
$log_e(n)$
k

If one takes a function high up in this ranking and combines it with functions further down by addition and multiplication, then the behaviour of that function will still be like that of the original function, in that, for large enough n, it will dwarf functions further down and be dwarfed by functions further up. So, for instance, $25n^3 + 3887n^2 + 100$ would simply be classed as "cubic". This means that for the kind of order of magnitude reasoning needed for complexity analysis we can forget everything except the most significant part (by this ranking) of each function, which makes calculating much easier.

In fact, the subdivision of the class "polynomial" into subcases is not always appropriate, as different classes of Turing machine-equivalent machines have different characteristics when viewed at this level of detail. As a result, for instance, an algorithm that runs with complexity n^3 on a "random access machine" (a machine that can access every element of its memory in the same constant time, as can current digital computers) might run with complexity n^6 on a Turing machine, which only has immediate access to a single position on the tape. So, although the distinction between polynomial and exponential complexity is always meaningful, it only makes sense to be more precise about the kind of polynomial complexity that an algorithm has if one has a particular class of machines in mind.

6.4.2 Determining Complexity

To determine the complexity of a given algorithm, we need to decide with respect to which input(s) the complexity is to be measured, how n is to be determined and how the amount of work depends on n. We will now give three examples to show how this can be done. In this section we will assume that we are only talking about random access machines.

Some deterministic Prolog programs

First of all, consider some deterministic Prolog programs (programs that don't do deep backtracking). The following programs implement "naive" and "good" versions of list reversal:

```
naive_rev([],[]) :- !.
naive_rev([H|T],Rev) :- naive_rev(T,Rev1), append(Rev1,[H],Rev).

   append([],X,X) :- !.
   append([X|Y],Z,[X|Y1]) :- append(Y,Z,Y1).

good_rev(L,Rev) :- rev1(L,[],Rev).

   rev1([],L,L) :- !.
   rev1([X|Y],L,Rev) :- rev1(Y,[X|L],Rev).
```

First of all consider the complexity of `append`. It will always be used here to concatenate two particular lists to yield a third. Let us assume that n will be the length of the input list (the first argument) - the procedure never looks at the second argument, apart from to instantiate the end of the result list to it. `append` works by recursing down its first argument, each time calling itself once more. Each time it extracts from the first argument the head `X` and the tail `Y`, instantiating the third argument to a list consisting of `X` followed by a new variable `Y1`. The complexity of these operations does not depend on n (though this requires making some weak assumptions about the Prolog implementation). Similarly, the operation of making a call to `append` and backtracking to the second clause if the first argument is not `[]` does not depend on the length of the first list. So we can say that the complexity of `append` is linear in the length of its first argument (it involves n steps, each of whose complexity does not depend on n).

Now for `naive_rev`. It also recurses down its first argument, and each time calls `append` with the result of reversing the rest of the list. Thus if the length of the original list is n then we will be concatenating lists of length $n-1$, $n-2$, $\ldots$, 2, 1. For each of these, the `append` complexity is linear in the length. Thus the total work done in `naive_rev` is (ignoring constants):

$$n - 1 + n - 2 + \ldots + 2 + 1$$

This sum is actually $n^2/2 - n/2$, which gives quadratic complexity. Another way to get this result would be to argue that the sum involves $n-1$ terms, and that the average term is about $(n-1)/2$; hence the result is about $(n-1)(n-1)/2$, i.e. quadratic. Notice that we don't need to be very precise, as long as we get the right power of n at each point.

The predicate `good_rev` can be dealt with similarly to `append`, again yielding linear complexity. This shows the superiority of `good_rev` (linear) over `naive_rev` (quadratic), at least for long lists.

A non-deterministic Prolog program

Our second example concerns a non-deterministic Prolog program (or, rather, a program where we initiate backtracking to enumerate all solutions).

```
p(X) :- p1(X).
p(X) :- p2(X).
p([]).

p1([a|L]) :- p(L).
p2([b|L]) :- p(L).
```

The idea is that the predicate `p` is given a list of Prolog variables and instantiates each one to either `a` or `b`:

```
?- p([A,B,C,D,E]).
A=a, B=a, C=a, D=a, E=a;
A=a, B=a, C=a, D=a, E=b;
A=a, B=a, C=a, D=b, E=a;
...
```

What is the complexity in terms of the length of the list provided? Since we are enumerating all solutions, whenever `p` is called *both* of the clauses must be used (at some point). Each of these involves some constant operations (instantiating the head of the list) and then enumerating all possible instantiations of the rest of the list. Thus we enumerate all possible instantiations of the tail of the list twice. In each of these, we enumerate all possible instantiations of the tail of *that* list twice. And so on. So each extra element of the list involves doubling the amount of work. Thus we have something like $2 \times 2 \times 2 \ldots \times 2$ (n times), which is 2^n operations (of calling `p`, say). The complexity is hence exponential. We have here just counted the "positive" work done by the Prolog system, ignoring the work needed to store the information about where to backtrack to and to actually return to a backtrack point (undoing variable bindings). However the number of backtrackings is 2^n and the maximum number of variables needing resetting never exceeds n, and so the complexity of that part of the work will not exceed $n \times 2^n$, which can be assimilated into the class labelled "exponential".

A conventional program

As the final example of determining complexity, here is a description of a sorting procedure, expressed in a way similar to what one would say in a conventional programming language (it is adapted from (Harel 87)). It assumes that a set of n numbers are placed in successive locations of an array, the ith element being denoted by $\mathrm{V}(i)$. The effect of this procedure is to leave the numbers sorted into ascending order.

```
for x going from n down to 1 do
   for y going from 1 up to x-1 do
      if V(y+1) < V(y) then exchange them
   endfor
endfor
```

It is not necessary to understand why this works in order to determine its complexity. If we assume that finding $\mathrm{V}(i)$ for any i, comparing two numbers and exchanging two numbers in the array are all constant complexity operations with respect to n (this requires that the array is stored sensibly), then the instructions inside the second loop are of constant complexity. The second loop involves doing $x-1$ of these, and x varies from n to 1. Thus, ignoring constant factors, we have

$$n - 1 + n - 2 + \ldots + 2 + 1$$

which, as we saw above, is quadratic.

6.4.3 Problem complexity

There may be different algorithms for solving a given problem, and these may have different complexities. We can classify problems according to what algorithms there are for solving them.

Looking at Table 1, we can see that the function 2^n increases dramatically more quickly than the polynomial or constant functions. This indicates the unattactiveness of exponential complexity algorithms in practice. Problems that have only exponential complexity algorithms are called *intractable*.

There are a number of complexity classes that problems can lie in, including the following:

- P - the class of problems that can be solved by a conventional Turing machine in polynomial time.
- EXP (or EXPTIME) - the class of problems that can be solved by a conventional Turing machine in exponential time
- NP - the class of problems that can be solved by a nondeterministic Turing machine in polynomial time

As we argued above, "conventional Turing machine" can be replaced by most other types of machine in these definitions. A nondeterministic Turing machine is essentially a machine that is able to take a nondeterministic program (one where there arise choices as to how to proceed) and always finds a solution in the same time it would take a conventional machine if that machine was told at each decision point in the program what choice to make. Thus a nondeterministic machine can "guess" an answer (for instance, can guess the factors of a number that it is trying to show is not a prime), though it still has to verify that its guesses are correct (of course, nothing like this exists in reality).

Since obviously the set P is contained in the set EXP, the above does not give us a simple way to talk about the problems that are "really exponential", rather than just "no worse than exponential". A problem Pr can be specially significant for a complexity class X as follows:

> Pr is X *hard* if for every problem P_1 in X there is an algorithm for solving P_1 by translating the input to a legal input of Pr, running an algorithm for Pr and then translating the output into a legal output of P_1 (and the associated translation problems themselves belong to X).

If Pr is X hard, then essentially Pr provides a way to solve *any* problem in X with complexity the same as the complexity of Pr itself. Thus Pr is as difficult a problem as any problem in the class X (but possibly more difficult still).

> Pr is X *complete* if it is X hard and also an element of X.

For instance, the problem of determining whether a formula of the Propositional Calculus is satisfiable is NP complete. This means that that every problem in NP can be reduced to solving this problem, and moreover that this problem is itself in NP. Thus it is a kind of "true representative" of the hardest problems in the class NP. It is still unknown whether the classes P and NP might be the same, though it is generally believed that the classes are different and hence that problems that are NP-complete are intractable.

6.5 Complexity of Recognition

Determining whether or not a string belongs to a given language is one example of a decision problem, and we can now state the complexities of the standard (more or less best known) random access machine algorithms for recognition for languages of the various types.

Type	*Complexity*
0	undecidable
1	e^n (exponential)
2	n^3 (cubic)
3	n (linear)

where n is the length of the string. It is important to realise that these are *worst case* results, e.g. there are type 2 grammars such that for some strings the complexity is cubic. This does not mean that every grammar of the type shows cubic complexity on the set of all strings. In particular, every type 3 grammar is also type 2 and so, by the above results, can recognise a string in at worst linear time.

That recognition for type 3 grammars can be done in linear time can be seen intuitively from the following algorithm for traversing a finite state transition network[2], given a string of length n whose *ith* element is given by w_i:

set S_0 to the set of initial states in the network
for i going up from 1 to n do
  set S_i to $\{s \mid$ there is a state s_1 in S_{i-1} and an arc labelled w_i
 from s_1 to $s\}$
endfor
if S_n contains a final state, return "yes"
otherwise return "no"

The algorithm maintains in S_i the set of all the possible states that a network traverser might be in after reading the first i words. Each S_i has size at most the number of nodes in the network (which does not depend on n) and the job of computing S_i involves at worst taking each possible pair of nodes and looking for an arc labelled w_i between them, together with removing duplicates from the collection of states obtained (numbering at most the square of the number of nodes) in order to get a set representation. Computing S_i is therefore a constant operation (as far as n is concerned), and so the algorithm as a whole is linear.

We will consider the complexity of type 2 recognition in a later chapter As regards type 1 grammars, it can be shown that every context-sensitive language is a recursive set. Therefore the recognition problem for these languages is decidable (though intractable). Languages of type 0 are recursively enumerable

[2]This algorithm does not handle networks with "jump" arcs, but that would be a fairly simple extension

but not in general recursive, which means that the best we can do in general for a language of this type is to have a procedure which, given a string, if the string is in the language answers "yes" but otherwise may not terminate.

These results give some weight to the Computer Scientist's reluctance to consider grammars of type less than 2 for practical applications.

6.6 Complexity of Parsing

Enumerating all the analyses of a sentence is always more complex than the task of recognition, because, even for languages of type 3, it is possible for strings to have a number of analyses which grows exponentially with the length of the string. Consider, for instance, the type 3 language with the following rules:

S → X
S → Y
S →
X → a S
Y → a S

where a is a terminal symbol. It is easy to show that a string of n as has 2^n distinct analyses by this grammar. For instance, the string aa has the analyses:

```
[S [X a [S [X a [S ]]]]]
[S [X a [S [Y a [S ]]]]]
[S [Y a [S [X a [S ]]]]]
[S [Y a [S [Y a [S ]]]]]
```

Thus, if "parsing" is taken to involve enumerating all the analyses then the worst case complexity of parsing is always at best exponential for all types of grammars.

6.7 Idealisations

Computability and complexity theory make a number of idealisations that make it difficult to adopt their conclusions uncritically:

1. *Choice of parameters.* It is assumed that one can select a set of parameters with respect to which the complexity can be usefully measured. A bad choice of parameters can lead to misleading results. For instance, when we presented a simple algorithm to indicate that type 3 recognition is linear, we only considered the length of the string as a parameter. If we had also considered the size of the grammar (the number of nodes in the FSTN) then we would have observed that the amount of work done depended on the square of that parameter. Whether our original idealisation makes sense depends on whether in practice it is likely that we will encounter large grammars. This might happen if we started using grammars that were generated automatically, for instance.

2. *The role of arbitrarily large problems.* It is assumed that the performance of an algorithm on small problems gives little indication of its usefulness, and that one has to consider arbitrarily large n in order to see the true complexity. But in practice, we will only want to deal with sentences of bounded lengths - how often will we want to deal with a sentence longer than 500 words? To counter this, one might say that there is no principled bound that can be placed on the length of an English sentence and that therefore the human parser (whatever that is) and artificial parsers must act as if they can be presented with arbitrarily long sentences, and choose algorithms that are appropriate to this. If only short sentences seem to used in practice, then that is to do with performance limitations of human beings, not with the inherent form of the language itself.

3. *The role of worst case analysis.* The recognition results are for the worst possible grammars in the various classes, and the worst possible strings. Yet one might argue that natural languages are not likely to be typical of any class like the class of type 2 languages, let alone worst examples, as they have evolved partly in order to support algorithms for tasks like parsing. However, a more refined analysis will have to wait until linguistics has provided us with a better characterisation of exactly what natural languages are. There seem to be infinitely many possible natural languages, and yet all are devised by animals with the same mental apparatus and for similar purposes. Nevertheless we seem to have very few ideas about their general characteristics. In the meantime, complexity analysis can yield advance warning that particular extensions to grammar formalisms are suspect because of the complexity implications.

6.8 References

Harel (1987) is an excellent introduction to issues of computability and complexity and Knuth (1976) is a standard reference book on the properties of algorithms. Garey and Johnson (1979) is the standard work on intractable problems. Hopcroft and Ullman (1979) gives a detailed formal discussion of the classes of grammar and automata and also contains chapters on the complexity of recognition. Barton *et al* (1987) argue for the relevance of complexity theory to Computational Linguistics and present examples of the complexity analysis of grammar frameworks.

- Barton, G. E., Berwick, R. C. and Ristad, E. S., *Computational Complexity and Natural Language*, MIT Press, 1987.
- Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to NP-Completeness*, W. H. Freeman, 1979.
- Harel, D., *Algorithmics: The Spirit of Computing*, Addison-Wesley, 1987.
- Hopcroft, J. E. and Ullman, J. D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

- Knuth, D. E., *Fundamental Algorithms*, Addison-Wesley, 1976.

6.9 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- Why is the idea of a Turing Machine introduced in computability theory?
- What is a decidable problem?
- Is logarithmic complexity better than linear complexity?
- What is an intractable algorithm?
- For which classes of languages is recognition in the worst case intractable?
- Why is the complexity of parsing (for some senses of the word) worse than that of recognition?

What you should be able to do now:

- Understand and use terms like "decidable", "intractable" and "polynomial complexity".
- Appreciate the relevance of these terms to Computational Linguistics and also the limits of their usefulness.
- Analyse simple programs in terms of their complexity.

Chapter 7

Introduction to Parsing

7.1 Aims of this Chapter

- To describe the conditions that a good parser must satisfy.
- To introduce the basic distinctions between parsing strategies:
  - top-down vs. bottom-up
  - left-right vs. right-left
  - depth-first vs. breadth-first
- To locate the standard DCG parsing strategy in the space of possible parsing strategies.

7.2 Criteria under which to evaluate a parser

A parser is provided with a grammar and a string, and it returns possible analyses of that string. Here are the main criteria for evaluating parsers:

1. *Correctness.* A parser is *correct* if all the analyses it returns are indeed valid analyses for the string, given the grammar provided. In practice, we almost always require correctness. In some cases, however, we might allow the parser to produce some analyses that are incorrect, and we would then filter out the bad analyses subsequently. This might be useful if some of the constraints imposed by the grammar were very expensive to test whilst parsing was in progress but very few possible analyses would actually be rejected by them.

2. *Completeness.* A parser is *complete* if it returns every possible analysis of every string, given the grammar provided. If a particular grammar assigns infinitely many analyses to some strings, then completeness in this context means that there is no analysis that the parser will not eventually find. In

such a situation, a complete parser will, of course, not terminate, but it will print out (or otherwise make available) analyses as it finds them. In some circumstances, completeness may not be desirable. For instance, in some applications there may not be time to enumerate all analyses and there may be good heuristics to determine what the "best" analysis is without considering all possibilities. Nevertheless, we will generally assume that the parsing problem entails returning all valid analyses.

3. *Efficiency* A parser should not be unnecessarily inefficient. For instance, it should not repeat work that only needs to be done once.

7.3 Parsing Strategies

In turning from mathematics to computation, there are choices that have to be made. This is particularly true in the case of rewrite grammars, like CFGs, which seem, as mathematical devices, to abstract much more than the equivalent automata from the details of implementation on conventional von Neumann architectures.

One fundamental dimension of parsing strategy is whether we operate in a *goal-driven* or *data-driven* manner. Our goal is to prove that a string of words is an S, and our data is the string of words. A goal-driven strategy thus starts with S. This is matched against the LHS of a rule and replaced by the RHS. Some symbol is chosen, and matched against a rule with that as its LHS, and the process continues. If the string of words is a sentence, then we will eventually arrive at the string by rewriting.

A data-driven strategy starts with the string of words, and matches each against the RHS of a rule of the form PT → w, where PT is a non-terminal symbol representing a lexical category, often called a pre-terminal, and w is an actual word. Strings of pre-terminals will then be matched against the RHS's of further rules, introducing non-terminal symbols. The process continues until the whole string has been written as an S (if it is in the language).

Since the application of context-free rules imposes a heirarchical structure on the string, i.e. a syntax tree, and since syntax trees are normally written with their root at the top, we call goal-driven parsing *top-down* and data-driven parsing *bottom-up*.

Another dimension of choice in parsing strategy is the order in which we consider symbols or strings of symbols for matching against rules. Do we work left-to-right, or according to some other pattern, e.g. *island-driven*? The latter might be more appropriate in a speech-understanding application, where we wish to move out from the words that have been understood clearly to those parts of the utterance that are less well-defined.

Finally, how do we handle the non-determinism inherent in the parsing process? Faced with multiple possibilities for rewriting, do we choose one and work on that exclusively, backtracking to an earlier choice point if we get into a blind alley; or do we work on alternative pathways simultaneously? The first of these is called *depth-first search* and the second *breadth-first search*. Since we currently have only sequential machines at our disposal, what this choice amounts to is whether we encode the nondeterminism behind the scenes, in the control regime, or 'up front' as a complex data structure which stores simultaneous possibilities.

To realise the theoretical lower bound on recognition (time) complexity for CFGs — n^3 — we must use a complex data structure known as the *well-formed substring table* or *chart* which are discussed in later chapters.

7.4 Top-down Parsing

The state of a left-to-right top-down recogniser can be characterised by:

1. A sequence of *goals*, the type of phrases it is looking for.
2. The sequence of words in which these phrases are to be found.

The way that such a recogniser works is to take the first goal from its list, find a rule in the grammar which has this as its left hand side and replace that goal by the sequence of items on the rule's right hand side. If the first item in the sequence of goals is the same terminal symbol as the first in the sequence of words, that symbol can be removed from both sides. The aim of the game is to get both sides (simultaneously) empty. Thus here are the stages in a possible successful recognition of the sentence 'Phoebe likes Mars':

Goals	*Words*
s	Phoebe likes Mars
np vp	Phoebe likes Mars
Phoebe vp	Phoebe likes Mars
vp	likes Mars
verb np	likes Mars
likes np	likes Mars
np	Mars
Mars	Mars

In such a sequence, one has to constantly make choices about which rule to use to expand the first goal, as not all possibilities will lead to successful derivations. What we have described here is an extreme case of a top-down recogniser, one that even expands preterminal symbols top-down. Thus, given the first goal 'verb' such a recogniser would have to consider which of all the possible verbs to replace it by (without having the benefit of being able to look at the next word). In practice, most top-down recognisers and parsers are only strictly top-down as far as the preterminal level.

For a right-left recogniser, one would have to change this scheme so that items are removed and inserted at the ends of the two sequences, rather than at the beginnings. For an island-driven scheme, one would have to generalise this scheme to allow the recogniser to be working on several parts of the string at once. To make a parser from this recogniser, we need to augment it to keep track of which rules it has used in satisfying the original S goal and its subgoals.

Top-down parsing can be quite inappropriate for some grammars. Consider what would happen if we used the a top-down, left-right control strategy to parse with a grammar containing rules such as:

v(1,Vform,Num) → v(1,Vform,Num) verb_mods

At the point where the parser is looking for a verb as its first goal, the following could happen:

Goals	*Words*
...	...
v(1,...) ...	...
v(1,...) verb_mods ...	...
v(1,...) verb_mods verb_mods ...	...
v(1,...) verb_mods verb_mods verb_mods ...	...
...	...

and so on *ad infinitum*. Even if the search strategy means that this rule is not chosen first every time, nevertheless the parser has to try all these possibilities eventually (e.g. there might really by 534 verb modifiers, in which case it has to go through this iteration 534 times) in order to cover all possible solutions. As a top-down parser, it cannot realise that infinitely many of these are in fact impossible (e.g. there can't be 534 verb modifiers if there are only 3 words left in the sentence). The rules that cause problems for this strategy are those that are *left-recursive*. Note that a pair of rules neither of which is left-recursive on its own may be so in combination. For instance, we might write a rule to allow a det, previously a lexical category only, to expand as a genitive noun phrase:

det → n(2) 's

Since the grammar also contains a rule which expands an n(2) as a det (followed by an n(1)), the two rules are left-recursive as a pair.

More generally, a top-down parser will do badly when there are many ways of realising some abstract category. For instance, if there are 600 rules for the category 'S' then a top-down parser will have to consider these, even if 599 of these require the sentence to start with an NP and the actual string starts with a verb.

7.5 Bottom-up Parsing

Whereas a top-down recogniser works from goals that it wants to find, gradually refining these until it gets to words that it can compare with the string, a bottom-up recogniser is sensitive to what is actually in the string, building up progressively bigger phrases until (hopefully) an S is obtained.

At each point, a bottom-up recogniser will have a record of a sequence of terminal and nonterminal symbols expressing what it has found in the string, for instance, the sequence "NP V Petra". It must scan along this sequence looking for a subsequence that is the same as the RHS of a rule. It needs to decide both how far along the string to scan and also which rule to use. One way of organising this is to have a *shift-reduce* recogniser. The state of a shift-reduce recogniser can be summarised by the following:

1. A sequence of symbols representing completed phrases found at the beginning of the string.
2. A sequence of symbols representing the words that have not yet been looked at

In a shift-reduce recogniser, the sequence of symbols for completed phrases is normally known as the *stack*. At each stage, the recogniser either *shifts*, that is removes and looks up the next untouched word and puts its category at the end of the sequence of completed phrases, or it *reduces*, that is, finds a sequence of symbols at the end of the completed sequence (stack) which matches the right hand side of a rule and replaces that sequence by the left hand side of the rule. The aim of the game is to get a single completed S at the same time as an empty sequence of remaining untouched words. Here are the stages that such a recogniser might go through with the sentence we saw above:

Completed	*Untouched*
	Phoebe likes Mars
np	likes Mars
np verb	Mars
np verb np	
np vp	
s	

In a shift-reduce recogniser, search arises because of two kinds of choices. First, at each stage, there is the choice whether to shift or to reduce. Secondly, there is the choice of which rule to use, either to find a category for a lexical item or to replace a sequence of categories by a single one. To make such a recogniser into a parser, it is necessary to have the machine remember the rules used in a derivation. This can be done by having the completed categories actually represented by parse trees headed by the given categories. When a reduction takes place, the new element of the stack is then a parse tree labelled with the LHS of the rule and with the sequence of parse trees used forming its subtrees.

Another form of bottom-up parsing, *left corner parsing* starts considering rules as soon as an instance of the first category of their RHS has been found. This will be discussed further in the next Chapter.

Bottom-up parsing is not suitable when there are grammar rules with empty right hand sides ("ϵ-productions"). This is for the obvious reason that if there is a rule like 'C $\rightarrow$', then bottom-up parsing can hypothesise a possible 'C' at any point in the string, which leads to many false hypotheses. In general, bottom-up parsing is also less attractive than top-down parsing when there is a great deal of lexical ambiguity. For instance, if the first word in a string can belong to 10 different syntactic categories, then a bottom-up parser will have to try all of these (and also look for larger phrases that could contain these categories), even if only one of them could actually start an S. Combinations of top-down and bottom-up parsing can reduce some of these problems, and we will consider an improvement of left-corner parsing to introduce top-down guidance in the following chapter.

7.6 Depth-first vs Breadth-first

Depth-first search means, given a choice of several alternative next states, picking one of them and pursuing all possible continuations from that state (including making all possible choices) before coming back to try the next possible alternative. This strategy, known also as *backtracking* is familiar to us from the operation of the Prolog interpreter. Backtracking can be implemented efficiently, because the interpreter needs only keep a stack of previous choice points to return to (whose size will be proportional to the depth of the search tree). In breadth-first search, the interpreter simulates trying all possibilities in parallel (in fact, spending a bit of time on the first one, then moving to the second one, then the third one, ..., then back to the first one ..., and so on). In such a regime, it is necessary to store a representation of the whole fringe of the search tree, which will usually be considerably more bulky than the sequence of choice points needed for a depth-first system.

It is important to realise that if the search space is infinite (e.g. for a top-down parser with a grammar with left recursion) then a depth-first system will not be complete. A breadth-first regime will, however, always find all possible answers (though it will not terminate if the search space is infinite).

7.7 Grammar Translation

One way to tackle problems of left recursion or ϵ-productions is to change the grammar, rather than the parser. It is possible automatically to convert a context-free grammar into a weakly equivalent one that accepts the same strings but does not have any left recursion, and it is also possible automatically to produce a weakly-equivalent grammar with no ϵ-productions. See the hints for exercises 5.2 and 5.4 in Gazdar and Mellish for more information.

Unfortunately, although such translations produce weakly equivalent grammars, the grammars are never strongly equivalent (unless the original grammar is already in the desired form). This approach is therefore not always very attractive, unless one is prepared to further transform the parse trees after parsing has finished.

7.8 The Standard DCG Parser

The notation used for Definite Clause Grammars (DCGs) is conceptually independent of the strategies embodied for parsing or recognition using it. The problems that arise in interpreting DCGs computationally are not a function of the notation, and one can easily build a parser (perhaps in Prolog) to interpret the DCG notation according to alternative strategies (see next chapter). Nevertheless, most Prolog implementations come equipped with an automatic translation from DCGs into Prolog (see Chapter 4). This, together with the standard mode of operation of the Prolog interpreter, means that there is a default manner in which DCGs are used for recognition (or parsing).

How does Prolog operate on standard programs? Prolog's search strategy is top-down, left-right, depth-first with backtracking. In proving a goal, a Prolog clause whose head matches the goal is chosen, and the predicates in the body become sub-goals whose simultaneous satisfaction is sufficient condition for the top-level goal to be proven. Prolog tries to prove these goals in left-right order. Prolog only investigates one possible way of satisfying a goal at a time; if a possibility fails, it backtracks to the last choice point and tries another alternative. Hence search is depth-first.

Given the way that DCGs translate into Prolog, exactly the same description can be applied to the standard mode of recognition using DCGs. All we need to do is substitute "recognise a phrase of a given category" for "prove a goal", "grammar rule LHS" for "clause head", "recognising a sequence of subphrases" for "satisfying a set of subgoals", and so on. Thus the standard DCG recogniser is top-down, left-right and depth-first. Notice that the features of the standard DCG recogniser mean that it is not complete for all grammars.

7.9 References

Chapter 5 of Gazdar and Mellish provides a slightly different presentation of the basic parsing strategies than has been given here.

7.10 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- Why would it be trivial to produce a parser that was correct but not complete?
- Which kind of parsing has trouble with left-recursion?

- Is shift-reduce parsing top-down or bottom-up?
- If a parsing search space is infinite but completeness if required, would it be better to use depth-first or breadth-first?
- Is standard DCG parsing top-down or bottom-up?
- Does standard DCG parsing have trouble with epsilon productions?

What you should be able to do now:

- List the dimensions along which Context Free parsing algorithms may be classified.
- Explain the differences between top-down and bottom-up, depth-first and breadth-first.
- Explain what kind of parser results from the standard compilation from DCGs into Prolog.
- Give the steps taken by Prolog in parsing a sentence with a given compiled DCG.
- Describe the reasons for non-termination of parsing with certain grammars and exemplify such grammars.

Chapter 8

Interpreted Parsing

8.1 Aims of this Chapter

- To show that the same DCG grammar can be the basis of many different types of parsers.
- To introduce the two main ways in which grammars can give rise to parsers - compilation and interpretation, as well as showing that partial execution can be used to produce a compiler from an interpreter.
- To show DCG interpreters yielding top-down, shift-reduce and left-corner recognisers.

8.2 Compilers and Interpreters

When DCGs are translated into Prolog, a notation that cannot be directly executed is translated into one that can. This is the notion of *compilation*. A *compiler* is a program that translates a program expressed in a high-level language into one expressed in a low-level language that some machine can understand directly. Usually the high-level language is something like LISP or Prolog and the low-level language is the machine code of something like a SUN workstation. We can consider DCG compilation as the same kind of process - it translates DCGs into a notation that a Prolog machine can interpret directly as a set of instructions. The actual Prolog machine comes to exist when we load and run a Prolog system. It may be able to execute Prolog directly either by itself compiling Prolog into the code of a “real” machine, or by including an *interpreter* for Prolog.

Interpretation is another way of executing a program written in a high-level language. An interpreter is a program that treats the high-level program as a complex piece of data and moves around that piece of data simulating what would happen if the program was being executed by an appropriate machine.

In general, compiling a high-level notation yields a more efficient result than interpreting it, because, whereas the result of compilation is a piece of code that says directly what to do next at each point, the operation of an interpreter introduces overheads by having to explicitly consult the original notation all the time. On the other hand, an interpreter can be a lot simpler to understand and show to be correct than the result of a compilation. For instance, it would be rather painful to have to check that all the difference list arguments of the Prolog translation of a DCG were actually correct. It is much easier to do this with the interpreters we show below.

We have seen that the standard Prolog execution of DCGs leads to problems for certain kinds of grammars, notably those with left-recursive rules. So it is appropriate to investigate alternative parsing strategies for this notation. For clarity, we will do this by writing (in Prolog) interpreters for the DCG notation. We will look at two bottom-up interpreters, a shift-reduce recogniser and a left-corner recogniser. These can be straightforwardly converted into parsers, but the principles are easier to follow if we just look at recognisers. First, to illustrate the basic technique of writing an interpreter, we will look at a top-down interpreter that includes the same search technique as the underlying Prolog interpreter (which is usually written in a low-level language like C).

We will make two changes to the DCG notation. First, we will use `--->` as the rewrite arrow, rather than `-->`. The only reason for this change is to prevent the DCG being compiled as it is loaded. We will have to define this operator.

```
:- op(1100,xfx,'--->').
```

Second, we will assume a maximum of two symbols on the RHS of a DCG rule. More particularly, we will use rules that have either two non-terminals or a single terminal as the RHS. This form is called Chomsky Normal Form (CNF), and every CFG has a weakly equivalent CNF grammar. This will make the discussion clearer. It would be tricky to give a shift-reduce interpreter for a DCG with arbitrary RHS's. The others can be changed with little trouble.

8.3 Top-down recogniser

Here is a DCG interpreter written in Prolog, which implements the standard DCG recognition strategy:

```
recognise(NT,P0,P) :-
    (NT ---> Body),
    recognise_body(Body,P0,P).

recognise_body((Body1,Body2), P0,P) :-
    recognise_body(Body1,P0,P1),
    recognise_body(Body2,P1,P).
recognise_body([Word],P0,P) :-
```

```
    connects(P0,Word,P).
recognise_body(Body,P0,P) :-
    \+ (Body = (_,_)),
    \+ (Body = [_]),
    recognise(Body,P0,P).
```

Notice that the top-down interpreter for DCGs above is itself in the form of a DCG translated into Prolog. Therefore we could write it more simply as a DCG itself, thus:

```
recognise(X) -->
   {X ---> Body},
   recognise_body(Body).

recognise_body((B1,B2)) -->
   recognise_body(B1), recognise_body(B2).
recognise_body([W]) --> [W].
recognise_body(B) --> {\+ (B = (_,_)), \+(B = [_])}, recognise(B).

:- recognise(s,String,[]).
```

8.4 Shift-Reduce Recogniser

Of course, the top-down recogniser above suffers from exactly the same problem as Prolog itself. This is partly because a DCG rule is chosen as a prediction, on the basis of its LHS. A bottom-up interpreter chooses rules on the basis of the symbols in the RHS, ultimately the words that occur in the string.

As we saw in Chapter 7, a shift-reduce recogniser makes use of an auxiliary data structure, a *stack*, which is initially empty. The categories of the words of the input string are pushed in turn onto the stack. The recogniser is so-called because at any point, there are two options. The top symbols of the stack may be matched against the RHS of a DCG rule and replaced by (reduced to) the LHS of the rule. Alternatively, the next word from the input may be looked up and shifted onto the stack. The recognition is successful if the stack contains a single distinguished symbol and the string has been consumed. Here is an interpreter that implements this strategy. The single argument to the predicate `recognise` (apart from the implicit string arguments) is the stack, represented by a list of categories in reverse order.

```
recognise([s],[],[]).

recognise([Y,X|Rest]) -->            % reduce
    [],
    {LHS ---> X,Y},
    recognise([LHS|Rest]).
```

```
recognise(Stack) -->                % shift
    [Word],
    {Cat ---> [Word]},
    recognise([Cat|Stack]).

:- recognise([],String,[]).
```

8.5 Left-Corner Recogniser

The shift-reduce recogniser finds a match for a complete RHS of a rule before using the rule. A left-corner recogniser uses just the first constituent of the RHS (the left-corner) to determine applicable rules. If `LC` has been found, and there is a rule `M ---> LC,RC`, then `M` will be recognised if `RC` can be found. An auxiliary procedure called `lc` is used. This computes a relation that is the reflexive, transitive closure of the left-corner relation (e.g. if a determiner is a possible left corner of an NP and an NP is a possible corner of an S, then a determiner is a possible left corner of an S). Then a string can be recognised as a category C if its first element's category is a `lc` of C.

```
recognise(Phrase) -->
    [Word],
    {Cat ---> [Word]},
    lc(Cat,Phrase).

lc(Phrase,Phrase) --> [].

lc(SubPhrase,SuperPhrase) -->
    {Phrase ---> SubPhrase,Right},
    recognise(Right),
    lc(Phrase,SuperPhrase).

:- recognise(s,String,[]).
```

The left-corner recogniser has some advantages over the shift-reduce recogniser. The latter computes every constituent it can from the given input string, even those that could not be used within a complete recognition. The left-corner offers the possibility of making predictions (introducing a top-down element into a basically bottom-up strategy) and thus reducing the search space. The `lc` relation may be true in infinite number of ways (if the grammar is left-recursive), but it is true of only a finite number of pairs of categories. These can be determined by inspection of the grammar. Before trying to prove `lc` for a pair of categories and an input string, we can check that the categories are possible candidates by 'table-lookup', and thus prune necessarily dead-branches from the search tree.

The table of possible category-pairs is often called an *oracle*. For the grammar:

```
s ---> np, vp.
np ---> det, n.
det ---> np, gen.
vp ---> v, np.
```

we would have the oracle:

```
link(s,s).      link(np,s).
link(np,np).    link(det,det).
link(n,n).      link(gen,gen).
link(det,s).    link(det,np).
link(vp,vp).    link(v,vp).
link(v,v).
```

The modified recogniser now looks like this:

```
recognise(Phrase) -->
    [Word],
    {Cat ---> [Word]},
    {link(Cat,Phrase)},
    lc(Cat,Phrase).

lc(Phrase,Phrase) --> [].

lc(SubPhrase,SuperPhrase) -->
    {Phrase ---> SubPhrase,Right},
    {link(Phrase,SuperPhrase)},
    recognise(Right),
    lc(Phrase,SuperPhrase).
```

The use of an oracle can lead to marked improvements in efficiency. For small-scale testing, a left-corner recogniser is a very easy way of avoiding the problems with left-recursivity.

8.6 Compiling a Recogniser

Having constructed interpreted recognisers, such as the above, it is natural for us to wish to produce more efficient compiled recognisers which embody the same recognition strategies. This is in fact often possible through the operation of *partial execution*.

Imagine a Prolog interpreter with a standard tracing mechanism that provides you with one extra option. When the system is about to invoke a goal, instead of causing it to fail, skipping over the goal's execution, or whatever, you can tell the system not to bother satisfying the goal just now. When the Prolog system tells you the answers to your queries, it then displays these in the form of *clauses*, where the body goals of the clauses are the goals that you told it to postpone looking at. So, given the program:

```
foo(X,Y) :- baz(X), zap(X,Y).
baz(a).
baz(b).
zap(X,Y) :- .....
```

here is what an interaction with this modified Prolog system might look like ('c' being the user response indicating "continue executing (and tracing)"):

```
?- foo(X,Y).
CALL foo(_1,_2)? c
CALL baz(_1)? c
EXIT baz(a)? c
CALL zap(a,_2)? POSTPONE
EXIT zap(a,_2)? c
EXIT foo(a,_2)? c

SOLUTIONS:
  foo(a,_2) :- zap(a,_2).
MORE? y

RETRY foo(a,_2)? c
RETRY zap(a,_2)? c
FAIL zap(a,_2)? c
RETRY baz(a)? c
EXIT baz(b)? c
CALL zap(b,_2)? POSTPONE
EXIT zap(b,_2)? c

SOLUTIONS:
  foo(b,_2) :- zap(b,_2).
MORE? y

RETRY foo(b,_2)? c
RETRY zap(b,_2)? c
FAIL zap(b,_2)? c
RETRY baz(b)? c
FAIL baz(b)? c
FAIL foo(_1,_2)? c
no
```

Now these clauses that this hypothetical Prolog system returns:

```
foo(a,_2) :- zap(a,_2).
foo(b,_2) :- zap(b,_2).
```

are actually useful things. They represent the results of *partially executing* the query `?- foo(X,Y).` If we ever wanted to present that query again (or a query that was more instantiated than that one), then these clauses would support a more efficient execution than the original program. That is the basic idea behind partial execution. Note that in this case the goals that we *have* satisfied in the partial execution (`baz(X)`) have yielded information (the possible values for `X`) that appears explicitly in the final program.

Now consider using this technique with DCG interpreters. Imagine that we have the following grammar:

```
s ---> np, vp.
vp ---> verb, np.
vp ---> vp, pp.
pp ---> prep, np.
```

(together with some lexical rules) and that we partially execute the query `?- recognise(X,Y,Z).` for our first, top-down, interpreter. In this partial execution, we will execute normally (*unfold* the definitions of, in the standard terminology) `recognise_body` and `--->`, but we will postpone evaluation of `recognise`. The result will be as follows:

```
recognise(s, _5, _6) :-
   recognise(np, _5, _7),
   recognise(vp, _7, _6).
recognise(vp, _8, _9) :-
   recognise(verb, _8, _10),
   recognise(np, _10, _9).
recognise(vp, _11, _12) :-
   recognise(vp, _11, _13),
   recognise(pp, _13, _12).
recognise(pp, _14, _15) :-
   recognise(prep, _14, _16),
   recognise(np, _16, _15).
```

Now, in this program every use of `recognise` has an instantiated first argument. So we can make the program more efficient by introducing a new predicate for each possible combination of `recognise` with some first argument. The easiest way is to name the new predicates after the first arguments themselves. Thus `recognise(Cat,W1,W2)` will be transformed to `Cat(W1,W2)`. Here is the result of performing this transformation on the partially executed program:

```
s(_5, _6) :-
   np(_5, _7),
   vp(_7, _6).
vp(_8, _9) :-
   verb(_8, _4),
   np(_4, _9).
vp(_11, _12) :-
   vp(_11, _13),
   pp(_13, _12).
pp(_14, _15) :-
```

```
      prep(_14, _16),
      np(_16, _15).
```

This is, of course, the standard translation of the DCG into Prolog. This shows that the standard DCG translation can be viewed as the result of partially executing our top-down interpreter on the grammar.

What happens if we try this on our other interpreters? For the left-corner interpreter, we need to change slightly the first clause for `lc`, to the following:

```
   lc(P,P) --> [], {category(P)}.
```

where `category(P)` succeeds if `P` is a syntactic category used in the grammar (and is capable of generating such categories on backtracking). With this change made and given the same grammar, the result of partially executing the goal `?- lc(X,Y,W1,W2).` for the left-corner interpreter, unfolding `--->` and `category`, but not `lc` or `recognise`, yields the following clauses:

```
   lc(np, np, _6, _6).
   lc(pp, pp, _6, _6).
   lc(prep, prep, _6, _6).
   lc(s, s, _6, _6).
   lc(verb, verb, _6, _6).
   lc(vp, vp, _6, _6).
   lc(np, _7, _8, _9) :-
      recognise(vp, _8, _10),
      lc(s, _7, _10, _9).
   lc(verb, _11, _12, _13) :-
      recognise(np, _12, _14),
      lc(vp, _11, _14, _13).
   lc(vp, _15, _16, _17) :-
      recognise(pp, _16, _18),
      lc(vp, _15, _18, _17).
   lc(prep, _19, _20, _21) :-
      recognise(np, _20, _22),
      lc(pp, _19, _22, _21).
```

We can then transform the clauses in the following way:

From:	*To:*
`lc(Sub,Super,W1,W2)`	`Sub(Super,W1,W2)`
`recognise(Goal,W1,W2)`	`goal(Goal,W1,W2)`

and the result is as follows:

```
   np(np, _23, _23).
   pp(pp, _23, _23).
   prep(prep, _23, _23).
   s(s, _23, _23).
```

```
verb(verb, _23, _23).
vp(vp, _23, _23).
np(_7, _8, _9) :-
   goal(vp, _8, _10),
   s(_7, _10, _9).
verb(_11, _12, _13) :-
   goal(np, _12, _14),
   vp(_11, _14, _13).
vp(_15, _16, _17) :-
   goal(pp, _16, _18),
   vp(_15, _18, _17).
prep(_5, _20, _21) :-
   goal(np, _20, _22),
   pp(_5, _22, _21).
```

This is exactly the result that would be obtained by the bottom-up DCG compilation discussed by Gazdar and Mellish.

8.7 References

The left-corner recogniser is from Pereira and Shieber (1987), who reconstructed it from the work of Matsumoto et al. We effectively do the reverse in the above. Pereira and Shieber also discuss the idea of partial evaluation in more detail. Gazdar and Mellish (1990) discuss bottom-up compilation of DCGs (in their Chapter 5).

The programs in this chapter are in the files `topdown`, `shiftreduce` and `leftcorner` available from the course web page under lesson 12. It may be useful for you to trace their operation on example grammars (e.g. `dcg_basis.pl` under code link for lesson 10) if you have difficulties seeing what the code is doing.

8.8 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- What are the main differences between a compiler and an interpreter?
- What do the shift and reduce operations do in a shift-reduce recogniser?
- Why is it useful to add an “oracle” to a left-corner interpreter?
- What are the inputs and outputs of a partial evaluator?

What you should be able to do now:

- Write interpreters in Prolog for left-corner and shift-reduce recognition.
- Explain how a left-corner parser can be equipped with an oracle, and why this is useful.
- Explain the notions of compiler, interpreter and partial execution and their relevance to parsing.

In this chapter we have shown how to obtain different kinds of DCG recognisers by writing interpreters in Prolog. Although these different recognisers are indeed quite different, nevertheless they all inherit one thing from Prolog - a depth-first search strategy with no memory. In the next chapters we will see that there are real problems with this characteristic, which motivate the development of other classes of recognisers.

Chapter 9

Well-Formed Substring Tables

9.1 Aims of this Chapter

- To explain the inefficiencies of backtrack parsing and the need for the parser to have a memory of what it has already done. This motivates the use of both well-formed substring tables introduced in this chapter and charts discussed in the next chapter.
- To describe the CKY algorithm, the standard use of a well-formed substring table.

9.2 Problems with Backtrack Parsing

There is a serious problem with all of the interpreters given in the last chapter, which arises from their use of backtracking as a strategy for coping with the inherent non-determinism of parsing. The fact that well-formed constituents discovered on one branch of the search are lost if that branch eventually fails and may subsequently be recomputed leads to great inefficiency. A simple example will illustrate this point. Suppose a grammar contained the following two verb phrase expansions:

VP → V[ditrans] NP PP
VP → V[ditrans] NP NP

Now suppose we are trying to recognise a sentence such as (48).

(48) I sent the very pleasant double-glazing salesman that I met on holiday in Marbella last year a postcard.

The first rule will be chosen, and the rather long object noun phrase will be parsed as such. Then a prepositional phrase will be looked for, and of course there isn't one. Backtracking, the second verb phrase expansion will be chosen. All the work of parsing the object noun phrase will be repeated, and then finally 'a postcard' will be parsed as another noun phrase, and parsing will be successful. Because the results of an unsuccessful search are not stored, they must be recomputed. Although Prolog is very efficient, this failure to store well-formed constituents leads to a complexity of recognition that is exponential in the length of the input string, and for long sentences this is totally unacceptable. The complexity of *parsing*, rather than *recognition* is at worst exponential for context-free grammars. Thus we know that if our English sentences have exponentially many analyses then there is nothing that can make parsing tractable. Here, however, we have an example where there is only one analysis but a search behavior that is still pathological. The problem is that complexity of the Prolog strategy is at worst exponential *even for recognition*. In order to be efficient where there are not large numbers of analyses, we need to base our parsers on better recognisers.

The solution to this problem is to move some of the computational complexity from the control to the data structure. The space complexity of backtracking parsers is linear in the length of the string, but by using a more complex data structure with a size bounded by n^2 we can reduce time complexity to polynomial. We will illustrate this with a bottom-up tabular parser, in which we ensure that we build no constituent more than once, at the expense of building some constituents that we do not need. Then we will go on to generalise the algorithm so that we build only what we need.

To avoid the repeated work involved in re-analysing the same constituents in different parts of the search space, it is necessary for a parser to store successfully parsed constituents, and use these stored results in preference to recomputing them. The abstract data structure used to store results is called a *well-formed substring table*, or *chart*. Since a tractable parsing algorithm (i.e. with polynomial complexity) must use a chart, there has been a considerable literature devoted to the question of parsing with a chart.

In presenting backtracking algorithms, we used (pure) Prolog, and let the latter's search algorithm take care of the housekeeping associated with backtracking. A chart parser can be implemented in Prolog, even pure Prolog, but since the programmer is forced to take over the housekeeping, there are not such advantages to using Prolog. In what follows, we will be less programming language-specific, and we will give algorithms in pseudo-code similar to imperative programming languages like PASCAL.

Again, to keep the presentation conceptually simple, we shall look first at an algorithm which requires grammars in Chomsky Normal Form. This will allow us to consider the major issues of correctness and complexity in their simplest form. Then we will see how the CNF restriction may be lifted in a completely general way.

9.3 The Cocke Kasami Younger algorithm

The following is a rational reconstruction of work by various people that is usually referred to as the Cocke Kasami and Younger (CKY) algorithm. If a string to be parsed has n words, then the chart will be stored as a matrix size $(n+1)^2$, indexed from 0 to n. The indices represent string positions (between words and at either end), and the entries in the chart represent constituents of certain categories that have been found between those positions. The entries are therefore sets of nonterminal symbols. Here we will just consider the recognition problem. If we wished to extract parse trees from the chart, the entries would need to be more complicated.

The process of recognition is one of systematically filling in this matrix, so that the set $chart(i, j)$ is the set of all categories of constituents spanning from position i to position j. The matrix is triangular since no constituent ends before it starts. Obviously then, recognition is successful if the final chart has the distinguished symbol S in the set $chart(0, n)$.

We must take some care about how we systematically fill in such a matrix. In particular, we must guarantee that a matrix entry is complete - that it contains all possible constituents over the string it covers - before we use it to compute a further entry. If this is not the case, a larger constituent looking for a certain phrase-type at a particular point will not know whether to look in the chart or try to parse the phrase from the beginning, and we will gain no benefit from the chart.

To ensure completeness, we will build all constituents ending at a certain point before we build any that end at a later point; and at a given point, we will build smaller constituents before we build larger ones. These two decisions make our version of the algorithm depth-first and bottom-up. It is a simple matter to change it to a breadth-first strategy.

To compute an entry in the chart we use the equation:

$$chart(i, j) = \bigcup_{i<k<j} chart(i, k) * chart(k, j)$$

The symbol $*$ (multiply) is an infix function that takes two sets of symbols $\{a_1, \ldots a_m\}$ and $\{b_1, \ldots b_p\}$ and returns the set $\{g_1, \ldots g_s\}$, containing all symbols g for which there is a rule $g \rightarrow \alpha\beta$, $\alpha \in \{a_1, \ldots a_m\}$, $\beta \in \{b_1, \ldots b_p\}$. Thus:

{np, n, verb, prep} * {np, s} = {vp, pp}

(assuming some appropriate grammar).

The CKY algorithm can be performed in cubic time by choosing all combinations of i, j and k, each of which has no more than n possible values. The complexity of the action performed in the innermost loop is constant in the length of the input string. (It is bounded by the square of the number of nonterminal symbols.) Since it depends only on the size of the grammar, it is known as the grammar constant.

One formulation of the algorithm is as follows:

> for j from 1 going up to n do
>   set $chart(j-1, j)$ to $\{A \mid A \rightarrow \text{word}_j\}$
>   for i from $j-2$ going down to 0 do
>   for k from $i+1$ going up to $j-1$ do
>   set $chart(i, j)$ to $chart(i, j) \cup (chart(i, k) * chart(k, j))$
> if S $\in chart(0, n)$ then accept else reject

Intuitively,

- j is the point in the string up to which all complete phrases are being computed.
- i is a chosen starting point less than j - all complete phrases between i and j need to be computed.
- k is a chosen point between i and j - a phrase between i and j will be found if there are appropriate subphrases between i and k and between k and j.

Obviously one can envisage alternative enumeration orders, but as long as we don't try to use a chart entry before we have completed it, this makes little difference to the efficiency of the algorithm.

An interesting point to notice in passing is the strong similarity between this algorithm and matrix multiplication, which originates in the similarity of the actions in the inner loops, viz:

$$\begin{aligned} chart(i, j) &= \textstyle\bigcup_{i<k<j} chart(i, k) * chart(k, j) \\ c_{ij} &= \textstyle\sum_k a_{ik} b_{kj} \end{aligned}$$

Also note that nothing hinges on any particular way that we may care to represent the chart pictorially. Typically, computer scientists draw the matrix as such; computational linguists use a graph notation, in which the spaces between words are the vertices of the graph, and an edge labeled with C spans from vertex i to vertex j just in case $C \in chart(i, j)$.

9.4 Ambiguity

The Appendix of this chapter shows an example of the CKY algorithm running on the following grammar:

np $\rightarrow$ tigger	v $\rightarrow$ chases
n $\rightarrow$ dog	n $\rightarrow$ bone
n $\rightarrow$ garden	det $\rightarrow$ a
p $\rightarrow$ with	p $\rightarrow$ round

s → np vp	vp → v np
vp → vp pp	np → det n
np → np pp	pp → p np

A grammar like this assigns a Catalan($N+1$) number of parses to a sentence ending with N prepositional phrases, due to the ambiguity of attachment of a pp to a vp or np. (You should demonstrate this yourself by drawing the 5 parse trees for the sentence like 'Tigger chases a dog with a bone round a garden.'). Importantly, even though a constituent such as 'chases a dog with a bone' has two parses as a vp, with the CKY algorithm a higher constituent which uses that vp is not found or represented twice.

9.5 Making a Parser

So far, the algorithm described is for a recogniser that does not keep enough information for analyses to be reconstructed. To store this extra information but retain polynomial complexity, we need to represent the possible analyses in a factored form rather than by multiplying them out. We can do this by changing an element of a chart entry to be, instead of a simple category, a triple of the form:

$$< Category, Rule, Pos >$$

where each *Rule* is a rule of the grammar that justifies there being a phrase of the given category in this place and *Pos* is the position in the string where the two subphrases on the RHS meet. This, together with the information about the portion of the string covered by the chart entry is then enough information for a program to find the chart entries corresponding to the subphrases, recursively find their possible internal structures and so generate all possible analyses.

For such a modified system, the * operation needs to be redefined so as to ignore the extra structure information attached to its input categories, but to include extra information in its output:

$$\begin{aligned}
&chart(i,k) * chart(k,j) = \\
&\quad \{< LHS, R, k > | \\
&\qquad \text{rule } R \text{ of the grammar is } LHS \rightarrow RHS_1\ RHS_2, \\
&\qquad\ \text{there is an element of the form } < RHS1, _, _ > \text{ in } chart(i,k) \\
&\qquad \text{and} \\
&\qquad\ \text{there is an element of the form } < RHS2, _, _ > \text{ in } chart(k,j)\}
\end{aligned}$$

Note that if there are several analyses of one of the subphrases, e.g. RHS_1, this is not reflected in a multiplication in the number of entries for the phrases containing it (in the construction of the set, multiple occurrences of $< LHS, R, k >$ will simply disappear). So, although the containing phrase may be found several times, it will only be represented once and so the multiple analyses of the subphrases cannot contribute to the work done in finding phrases containing that. In particular, when we consider non CNF grammars, this kind of chart representation will allow us to handle a grammar with a rule like:

S → S

(as well as conventional rules for S and other categories). Such a grammar provides an infinite number of analyses for any string that is an S, and so an enumeration of the possible parses would never terminate. Nevertheless, such an enhanced recogniser will always terminate and will be able to represent the infinite set of parses in a finite form (that can be subsequently enumerated for as long as one wishes).

9.6 Drawbacks of Bottom-up Parsing

One of the drawbacks of a bottom-up strategy is that all constituents that are licensed by the grammar are built, regardless of whether they could be incorporated into a complete parse, that is, the algorithm is insufficiently goal-driven. For example, suppose our sentence was ‘the search for Spock was successful’. The phrase ‘search for spock’ would be found not only as part of a noun phrase, but also a present-tensed verb phrase. But a verb phrase can never directly follow a determiner such as ‘the’, so top-down prediction would allow us to prune the search tree. In the next chapter we will see how the use of a chart allows a parse to be driven by top-down prediction, while avoiding Prolog’s problems with non-termination.

9.7 Appendix: Trace of CKY algorithm

The following shows a trace of the CKY algorithm recognising the string:

	Tigger	chases	a	dog	with	a	bone	round	a	garden.
0	1	2	3	4	5	6	7	8	9	10

We show the values of the counters i, j and k given in the algorithm description, as well as the entries made to the chart as they are added. When for some value of k no additions are made to the chart, that value of k is not shown.

```
j = 1
   np from 0 to 1
j = 2
   v from 1 to 2
   i = 0
j = 3
   det from 2 to 3
   i = 1
   i = 0
```

```
j = 4
   n from 3 to 4
   i = 2
      k = 3
         np from 2 to 4
   i = 1
      k = 2
         vp from 1 to 4
   i = 0
      k = 1
         s from 0 to 4
j = 5
   p from 4 to 5
   i = 3
   i = 2
   i = 1
   i = 0
j = 6
   det from 5 to 6
   i = 4
   i = 3
   i = 2
   i = 1
   i = 0
j = 7
   n from 6 to 7
   i = 5
      k = 6
         np from 5 to 7
   i = 4
      k = 5
         pp from 4 to 7
   i = 3
   i = 2
      k = 4
         np from 2 to 7
   i = 1
      k = 2
         vp from 1 to 7
   i = 0
      k = 1
         s from 0 to 7
j = 8
   p from 7 to 8
   i = 6
   i = 5
```

```
   i = 4
   i = 3
   i = 2
   i = 1
   i = 0
j = 9
   det from 8 to 9
   i = 7
   i = 6
   i = 5
   i = 4
   i = 3
   i = 2
   i = 1
   i = 0
j = 10
   n from 9 to 10
   i = 8
      k = 9
         np from 8 to 10
   i = 7
      k = 8
         pp from 7 to 10
   i = 6
   i = 5
      k = 7
         np from 5 to 10
   i = 4
      k = 5
         pp from 4 to 10
   i = 3
   i = 2
      k = 4
         np from 2 to 10
   i = 1
      k = 2
         vp from 1 to 10
   i = 0
      k = 1
         s from 0 to 10
```

9.8 References

The formulation of the CKY algorithm is due to Martin, Church and Patil (1981), a very clear exposition of issues in chart parsing. See this paper for references to the original work.

- Martin, W.A., K.W.Church and R.S.Patil (1981) "Preliminary Analysis of a Breadth-First Parsing Algorithm" MIT Technical Report, MIT/LCS/TR-261

9.9 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- Give an example where a backtrack recogniser will duplicate work unnecessarily.
- Is the CKY algorithm left-to-right or right-to-left?
- Is the CKY algorithm top-down or bottom-up?
- What is the computational complexity of recognition by the CKY algorithm?
- If in the CKY algorithm categories stored in the chart were replaced by parse trees, why would the complexity then be at worst exponential?

Chapter 10

The Active Chart

10.1 Aims of this Chapter

- To introduce dotted rules as the key to generalising the CKY algorithm.
- To show Earley's algorithm for recognition using a chart.
- To show how variations on Earley's algorithm can be made.

10.2 Dotted Rules

To generalise the CKY algorithm to work with grammars that have more than two symbols on their right hand sides, we introduce the notion of a *dotted rule*. The dotted rule is an entry in the chart, i.e. a type of edge, that embodies the idea of a partially found constituent. Instead of the multiplication step combining complete constituents according to the grammar, it combines one partial and one complete constituent according to information in the edges themselves.

The dotted rule is so-called because it comprises a rule of the grammar with a dot between the symbols that have matched a portion of the input and those that have yet to be matched. Examples are:

vp → .v np pp
vp → v. np pp
vp → v np. pp
vp → v np pp.

The first of these represents a vp of which nothing has yet been found, and the last a vp which is complete. The other two represent vps in various stages of completion.

The full specification of an edge includes a dotted rule or *state* with its start and finish vertices. If an edge v → v np . pp spans from vertex 3 to vertex 8, we can write:

$$_3[\text{v} \rightarrow \text{v np}\ _8\ \text{np}]$$

in which the position of the end vertex's number represents the dot.

The multiplication rule for sets of edges is now independent of the grammar and takes the following form:

$$\begin{aligned} &S_1 * S_2 = \\ &\quad \{_i[\text{L} \rightarrow \text{A X}\ _j\ \text{B}] \mid \\ &\qquad _i[\text{L} \rightarrow \text{A}\ _k\ \text{X B}] \in S_1 \text{ and} \\ &\qquad _k[\text{X} \rightarrow \ldots _j] \in S_2\} \end{aligned}$$

Since we will more often need to talk about the combination of two single edges that are of the right form than of the combination of two sets, for simplicity we will from now on use the multiplication symbol for that purpose. Thus:

$$_i[\text{L} \rightarrow \text{A}\ _k\ \text{X B}] * {}_k[\text{X} \rightarrow \ldots _j] = {}_i[\text{L} \rightarrow \text{A X}\ _j\ \text{B}]$$

where L is a non-terminal, X is a terminal or non-terminal, and A and B are any string of terminals and non-terminals, possibly empty[1]. This is called the *fundamental rule* of chart parsing. Possible instances of the rule are:

$$\begin{aligned} &_2[\text{vp} \rightarrow \text{v}\ _3\ \text{np pp}] * {}_3[\text{np} \rightarrow \ldots _5] = {}_2[\text{vp} \rightarrow \text{v np}\ _5\ \text{pp}] \\ &_2[\text{vp} \rightarrow \text{v np}\ _5\ \text{pp}] * {}_5[\text{pp} \rightarrow \ldots _8] = {}_5[\text{vp} \rightarrow \text{v np pp}\ _8] \end{aligned}$$

Since an edge which does not have the dot in a final position contains a specification of what needs to occur after it, and thus can be given an active role in the parsing process, it is known as an *active edge*. An edge with the dot in the final position is called *complete*, *inactive*, or *passive*.. A parser that uses a chart including active edges is known as an *active chart parser*.

The problem we have now is how edges get into the chart in the first place, and we will discuss alternative approaches to this below.

10.3 Bottom-Up Invocation

We can incorporate the dotted rule idea into a bottom-up parser. As in the left-corner backtracking interpreter in Prolog, we use left-corners to index the rules. When we enter a word into the chart, we also add a dotted rule spanning that word for each rule that the word may be a left-corner of. For instance, suppose we find a verb from position 4 to position 5, and our grammar contains the rule:

$$\text{vp} \rightarrow \text{v np pp}$$

Then we also add a dotted rule:

$$_4[\text{vp} \rightarrow \text{v}\ _5\ \text{np pp}]$$

[1]Note that for this to work unchanged with terminals as well as non-terminals, we need to ensure that terminals are entered into the chart so that they look like dotted rules. For instance, terminal α between i and j could be entered as $_i[\alpha \rightarrow {}_j]$.

This embodies the hypothesis that there is a vp starting at 4, which includes a verb from 4 to 5, and needs an np and a pp to be completely recognised.

We will not give the bottom-up active chart parsing algorithm here. Instead we will turn to a top-down active chart parser. The algorithm was essentially discovered by a computer scientist, Jay Earley, in 1970, and recreated within the CL community by Martin Kay. It is the most efficient parsing algorithm known for unrestricted CFGs. As usual, we actually state the algorithm for the corresponding *recogniser*.

10.4 Top-Down Invocation: Earley's Algorithm

As mentioned in Chapter 9, a bottom-up invocation strategy will find every phrase, even those that cannot contribute to a complete parse of the sentence. A top-down predictive element can be introduced into active chart parsing as follows. An entry in the diagonal of the chart represents a zero length phrase, one that spans from vertex i to vertex i for any i. It represents a hypothesis that there is a constituent of the type of the LHS starting at that point, though none of the sub-constituents have yet been found. The only reasonable hypothesis that we can make at the start of parsing is that there is a sentence starting at position 0. Therefore we need to add to the chart an active edge of the form:

$$_0[\text{s} \rightarrow {_0}\ \text{np vp}]$$

for each rule that expands an s.

Since it will be the active edges that dictate the course of parsing, and the only active edges that are in the initial state of the chart are those that we know will be useful, the number of useless constituents that are built will be greatly reduced from those built by the CKY algorithm.

Earley defined his algorithm in terms of three cases that can obtain when an edge is put into the chart. These are as follows:

1. *Predicting (Proposing)*: The edge that is being entered is active, but there is no edge for it to combine with already in the chart. In this case, the required edge is predicted, by entering an initial dotted rule for each way that it can be built.

2. *Scanning*: The edge that is being entered is active, and there are edges already in the chart with which it can combine. For each of these edges the multiplication rule is applied. This gives rise to further edges.

3. *Completing*: The edge that is being entered is complete. For each of the active edges that it can combine with, the multiplication rule is applied, giving rise to further edges. Note that there must be such active edges, for the complete edge to have been predicted in the first place.

When a number of new edges are created by one of these operations, there is some choice about how we deal with them. We can add each of them before we add any other edge. Alternatively, we can add the first and then any new edges that that operation gives rise to, before considering the rest. The first of these approaches gives rise to a breadth first strategy and the second to a depth first one. This point is developed further below.

A related choice is how we start the chart off. Do we predict the distinguished symbol or do we add the lexical edges first? With the former strategy, we will only enter a complete edge after we have made all predictions up to the start of that edge. This means that the conditions for scanning will never arise (though scanning will be appropriate in the latter case). If we always add predictions first, then, nondiagonal edges will only arise as a result of completing, because the left edges will always be in place before the right ones they need to combine with arrive. Hence we will only need to access the chart by end vertex, retrieving the set of edges that end at a certain point. This is the approach we will adopt.

For the sake of exposition we will assume that:

chart(s,i,j)

indicates that there is an entry in the chart, between vertices i and j, for the state s. In a grammar with rules:

s → np vp
np → det n
vp → v np np

the possible states are:

s → . np vp	s → np . vp	s → np vp .	np → . det n
np → det . n	np → det n .	vp → . v np np	vp → v . np np
vp → v np . np	vp → v np np .		

Here now is an algorithm for chart parsing (actually recognition):

```
procedure chartparse:

   for state in predictions(... --> . s) do
      enter_edge(state,0,0)
   for j from 1 going up to n do
      for state in {(A --> .) | A --> wordj} do
         enter_edge(state,j-1,j)

procedure enter_edge(state1,i,j):

   if chart(state1,i,j) is not true, then
      add (state1,i,j) to the chart

      % predict
      for state2 in predictions(state1) do
         enter_edge(state2,j,j)
```

```
% complete
for each k, state2 such that chart(state2,k,i) is true do
   if left_sister(state2,state1) then
      enter_edge(state2*state1,k,j)
% scan
for each k, state2 such that chart(state2,j,k) is true do
   if right_sister(state2,state1) then
      enter_edge(state1*state2,i,k)
```

where we have included the scan case for completeness – it will not actually have anything to do, given the order in which edges are added here. We first enter the initial dotted edge(s) for the distinguished symbol, and all edges that this gives rise to. Then we read each word in turn and enter the final dotted edges for their preterminal categories, doing everything that results from a given word in the string before reading the next word.

Note that before we enter an edge, we must check that an identical edge has not been entered before. Although this algorithm is top-down, doing only what is necessary, this check avoids the problem that the Prolog interpreter has with left-recursive grammars. If the edge is already present, `enter_edge` just returns with no effect.

The algorithm assumes a function `predictions`, from states to sets of states, and two predicates on pairs of states, `left_sister` and `right_sister`. `predictions` may be computed off-line (once and for all, in advance of any parsing). It takes a state and returns the set of states corresponding to all the possible grammar rules that could find the next needed phrase. For the above grammar:

`predictions`(s → np . vp) = {vp → . v np np}
`predictions`(vp → v . np np) = {np → . det n}

If the symbol after the dot exists only as a lexical category, then the set of predictions will be the empty set. In `enter_edge`, an active edge with the dot immediately before a constituent A will have a state in its `predictions` for each rule with A as its LHS. Each of these will be a dotted rule with the dot on the left of the RHS, and each will be entered (on the diagonal of the chart) by a recursive call to `enter_edge` (during the prediction stage).

`left_sister(x,y)`, "x is a left sister of y", takes two states and returns 'true' if the first state can combine to the left of the second. For instance,

`left_sister`((s → . np vp), (np → det n .)) = true
`left_sister`((np → det . n), (np → det n .)) = false
`left_sister`((s → . np vp), (np → det . n)) = false

A complete edge of category B will have a left sister state corresponding to each way in which B can appear on the RHS of a rule (i.e. the states in which a dot appears immediately before a B). For each edge with such a state, and which ends where the complete edge begins, a new edge spanning the two is built by the fundamental rule (during the completion stage).

`right_sister(x,y)`, "x is a right sister of y", acts similarly for combinations to the right. For instance,

right_sister((np → det n .), (s → . np vp)) = true
right_sister((np → det n .), (np → det . n)) = false
right_sister((np → det . n), (s → . np vp)) = false

An active edge with dot before category B will have a right sister state corresponding to each way in which B can appear on the LHS of a rule, with a dot at the end of the RHS. For each edge with such a state, and which begins where the active edge ends, a new edge spanning the two is built by the fundamental rule (during the scanning stage).

left_sister and right_sister could also be computed off-line, but in practice they can be implemented by simple tests on the structure of the states. For left_sister to be true, the first category after the dot of the first state must be the same as the LHS category of the second and the second state must be complete; for right_sister to be true, the first state must be complete and its LHS category must be the same as the first category after the dot of the second state.

Every state only has left sisters or right sisters. An active state has no left sisters, an inactive (complete, passive) one no right sisters. Also, only an active state has non-empty predictions.

10.5 Expressing the Earley algorithm in Prolog

In this section, we briefly outline a Prolog implementation of the algorithm discussed in the last section. The Prolog program is available as `library(chart)`.

First of all, we need to have a way of representing the rules of the grammar. We choose to use a DCG-like notation (with `--->`, rather than `-->`, as before), though it makes the program simpler if the right hand sides of rules are lists. We will also have a separate predicate `lex` for lexical entries. Thus here is a sample piece of a grammar in the notation used:

```
vp ---> [vp,pp].
pp ---> [prep,np].

lex(man,n).
lex(dog,n).
```

Secondly, we need to have a way of representing a state (dotted rule). A state has three main parts – the category on the left hand side, the categories on the right before the "dot", and the categories on the right after the "dot". A natural way would be to use a Prolog structure of the form `state(_,_,_)` to hold these three parts. In fact, however, a chart recogniser does not really need to remember the categories on the left of the "dot", and so we can use a two-place representation, as follows:

```
state(s,[vp]).     % s --> np . vp  (or s --> pp . vp, etc.)
state(np,[det,n]). % np --> . det n (etc.)
```

The second element of a state structure will be [] for an inactive edge and a non-empty list for an active edge. It is useful to read state(X,Y) as "an X needing Y", e.g. "an NP needing a det and an N".

Finally, the chart will be represented in the Prolog database, using the predicate chart(S,I,J), where S is a state (represented as above) and I and J are numbers indicating the starting and finishing positions in the chart.

With these preliminaries over, we can present the top-level structure of the Earley algorithm:

```
chartparse(String) :-
   foreach(s ---> RHS,
           enter_edge(state(s,RHS),0,0)),
   foreach((element(Word,Pos,String),Pos1 is Pos+1,lex(Word,Cat)),
           enter_edge(state(Cat,[]),Pos,Pos1)).
```

Now foreach(X,Y) is a predicate (defined in the library file) that backtracks over all solutions to the goal X, for each solution calling the goal Y. So the first goal in this clause looks for each possible way to satisfy s ---> RHS (i.e. all rules with s as the LHS), and for each one it calls enter_edge to add an appropriate edge to the chart (here, an active edge attempting to create an s edge). The second goal in chartparse has a similar structure. element(X,N,L) is just like member(X,L), except that it returns the position of the element in N. It is used to enumerate the words of the string, one by one. For each of these, the category is looked up, using lex. For each combination of position and category, enter_edge is called to add an appropriate inactive edge to the chart. This is exactly what is done in the algorithm of the last section.

Of course, the real key to chart parsing is the enter_edge procedure. Here is the Prolog version of this:

```
enter_edge(State,I,J) :-
   chart(State,I,J), !.
enter_edge(State,I,J) :-
   assertz(chart(State,I,J)),
   predict(State,I,J),
   complete(State,I,J),
   scan(State,I,J).
```

Note how the procedure does nothing if the edge is already in the chart. The definition has been decomposed in terms of the three operations of the Earley algorithm. Here are the definitions of the Prolog predicates:

```
predict(state(_,[Cat|_]),_,J) :- !,
   foreach(Cat ---> RHS,
           enter_edge(state(Cat,RHS),J,J)).
predict(_,_,_).
```

```
complete(state(Cat,[]),I,J) :- !,
   foreach(chart(state(LHS,[Cat|Needed]),K,I),
           enter_edge(state(LHS,Needed),K,J)).
complete(_,_,_).

scan(state(Cat,[Need|Needs]),I,J) :- !,
   foreach(chart(state(Need,[]),J,K),
           enter_edge(state(Cat,Needs),I,K)).
scan(_,_,_).
```

Notice that `predict` and `scan` do nothing unless they are given information specifying an active edge. Similarly `complete` does nothing unless it is given an inactive edge. `predict` looks for predictions from the original state simply by looking for the grammar rules for the first "needed" category. `complete`'s search for left-sisters does not need an explicit `left_sister` predicate – our representation for states means that it can just look for a state whose first "needed" element is the category of the original edge. Similarly for `scan` and right-sisters.

10.6 The Agenda

The Earley algorithm adds new edges to the chart in a specific kind of left-to-right order. Although this facilitates the implementation somewhat (for instance, no scanning is actually ever required), this rigidness is not really required. All that we need to do is check that whenever a new edge is added to the chart all necessary consequences of that edge (by prediction, completion and scanning) are also added (if they are not already present).

We can make the algorithm somewhat more flexible by the use of a data structure called an *agenda*. Instead of entering an edge directly into the chart when we have constructed it (by completing) or determined that it is useful (in the prediction step), we add it to the agenda. We start the algorithm off by putting the initial edges (the lexical edges and the predictions for the possible ways of finding an 's') on the agenda. Then we continue to perform the loop:

> while there are edges on the agenda do
>   choose an edge from the agenda
>   enter the edge into the chart

The rest of the algorithm is the same as before, except that the recursive calls to `enter_edge` are replaced by calls to add the edge to the agenda.

Now we can change the order in which things get done merely by adopting different strategies for choosing an edge from the agenda. If the agenda is treated as a stack, so that the last edge added to it is the first one to be chosen for entering in the chart, then we will be realising a depth-first parser. This is because if there are two alternative edges to be added (perhaps for a lexically ambiguous word) then in a stack-based regime the first of these and all new edges that arise as a consequence of it will be added before the second is considered. On the other hand, if the agenda is a queue, then the strategy will be breadth-first. We can even adopt more sophisticated criteria for choosing an edge from the agenda, such as choosing the edge which progresses furthest through the sentence (with the highest numbered end vertex). Notice that, in general, fancy scheduling regimes will require the system to do both completion and scanning.

As mentioned above, the active chart parsing algorithm is the most-efficient general CFG parsing algorithm. Programming languages generally have certain restricted properties, such as being able to be parsed deterministically with limited lookup, and assigning unambiguous structure to programs. For parsing with CFGs without such properties, as typically written to describe Natural Languages, chart parsing has become a standard framework. The flexibility of strategy given by the agenda has also made chart parsing popular in psycholinguistic models.

10.7 From Recognising to Parsing

We can make a chart recogniser into a chart parser in much the same way as we suggested with the CKY algorithm. Since we are no longer restricted to binary grammar rules, the representation used there has to be generalised, but in fact the dotted rule notation provides a good basis for this. One way of remembering enough information to reconstruct the possible analyses is for the positions in the chart to be inserted into the dotted rule to reflect the subphrases found. Thus the following might be the stages in recognising a vp:

vp $\rightarrow$ (2) . v np np
vp $\rightarrow$ (2) v (3) . np np
vp $\rightarrow$ (2) v (3) np (5) . np
vp $\rightarrow$ (2) v (3) np (5) np (7) .

In this example, we found a verb at position (2), then an np from position (3) to (5) and finally another np from (5) to (7). Together with the information stored about these subphrases, the final (annotated) dotted rule shown contains all the information necessary to reconstruct the possible analyses. On the other hand, if there are multiple possible np analyses between (5) and (7) then there is still only a single representation for the set of vp analyses using them - so no pathological multiplying out of possibilities.

10.8 References

Gazdar and Mellish (1989) gives a thorough discussion of chart parsing and shows example chart parsers written in Prolog. See also Winograd (1983) section 3.6 and Pereira and Shieber (1987). Grosz et al. (1986) contains Earley's original paper (1970) and a useful paper by Kay (1980).

- Gazdar, G. and Mellish, C., *Natural Language Processing in Prolog*, Addison-Wesley, 1989.
- Grosz, B., Sparck Jones, K. and Webber, B. L., Eds., *Readings in Natural Language Processing*, Morgan Kaufmann, 1986.
- Pereira, F. and Shieber, S., *Prolog and Natural-Language Analysis*, CSLI Press, 1987.
- Winograd, T., *Language as a Cognitive Process. Volume 1: Syntax*, Addison-Wesley, 1983.

10.9 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- What is a dotted rule?
- Which kinds of dotted rules correspond to the entries made in the CKY algorithm?
- What is the difference between an active and an inactive (passive) edge?
- Does prediction apply to inactive or active edges?
- Does completion apply to active or inactive edges?
- What kind of existing edges are looked for in completion?
- Does scanning apply to active or inactive edges?
- What kind of existing edges are looked for in scanning?

What you should be able to do now:

- Explain the fundamental rule and the active chart parsing algorithm
- Exemplify the algorithm parsing a simple sentence

Chapter 11

Introduction to Unification

The next few chapters further develop the idea of using complex categories (rather than, say, the simple categories of context-free grammar) in order to write expressive grammars that capture generalisations. We have already seen the advantages of complex categories to some extent with DCGs. In this approach, categories are essentially partial objects, built from features with possibly complex values.

11.1 Aims of this Chapter

- To motivate the need for a *unification* operation for grammars with features.
- To describe how *term unification* works in Prolog and DCGs.
- To indicate that *graph unification* has some advantages over this.
- To introduce PATR-II, a DCG-like notation which uses graph unification, rather than term unification.

11.2 Features and Unification

In the fragments of grammars that we have encountered so far, we have often found it necessary or desirable to add features to categories. Now we must be more precise about what is going on in a grammar augmented with features.

We have already encountered a definition of how we expect a grammar with features to behave, when number agreement was discussed in the context of Definite Clause Grammars. In a rule of the form:

```
s(Vform) --> n(2,Num), v(2,Vform,Num).
```

we expect the rule to be used if and only if the same value is bound to both occurrences of the variable `Num`. In particular, it does not matter whether the final value of `Num` originated in the np, the vp, or both, as long as it has only a single value. We can think of a DCG rule with variables as a shorthand for the rules that arising for substituting every possible ground value[1] for every variable. When the same variable appears twice in the rule, this is naturally reflected by the fact that in all the ground instances the same value appears in both places. So the above rule actually means the same as:

```
s(fin) --> n(2,sing), v(2,fin,sing).   % Num=sing, Vform=fin
s(fin) --> n(2,plur), v(2,fin,plur).   % Num=plur, Vform=fin
s(bse) --> n(2,sing), v(2,bse,sing).   % Num=sing, Vform=bse
s(bse) --> n(2,plur), v(2,bse,plur).   % Num=plur, Vform=bse
s(inf) --> n(2,sing), v(2,inf,sing).   % Num=sing, Vform=inf
s(inf) --> n(2,plur), v(2,inf,plur).   % Num=plur, Vform=inf
s(prp) --> n(2,sing), v(2,prp,sing).   % Num=sing, Vform=prp
...
```

(here there are only a finite number of ground rules, but in general there could be infinitely many). This explains what a rule with variables *means*, but we must ensure that our operations on rules respect this. Again, if we have a parsing goal containing variables, this is simply a shorthand for the set of ground goals that would arise for every possible ground substitution being made for each variable. If we ask:

```
?- s(Vform).
```

we are really[2] presenting the set of simultaneous goals:

```
?-
  {
     s(fin),    % Vform=fin
     s(bse),    % Vform=bse
     s(inf),    % Vform=inf
     s(prp),    % Vform=prp
     ...
  }.
```

and we expect Prolog to tell us which of these has succeeded.

Now how should the above set of rules be used to satisfy the above set of queries? Clearly, now that we are just looking at ground goals and rules, the only rules applicable are those whose LHS are *identical to* one of the goals. This gives all the rules in this case, but if the query had been just:

```
?- s(fin).
```

[1] A value that does not itself contain any variables

[2] For simplicity, I am assuming that we can query DCG rules directly, and that we don't need to provide the string arguments. This is of course not the case, but including the extra string arguments here would complicate the presentation. I also assume below that we can think in terms of presenting Prolog with a set of of simultaneous goals - in reality we would have to present the elements of such a set one after the other.

then only the first two rules would have been applicable, and so we would have wanted to consider the two following sets of subgoals:

```
?-
  {
     n(2,sing), v(2,fin,sing)    % from 1st rule
     n(2,plur), v(2,fin,plur)    % from 2nd rule
  }.
```

which is equivalent to the single abstract goal

```
?- n(2,Num), v(2,fin,Num).
```

In general, if we have a set of ground goals:

$\{Goal_1, Goal_2, Goal_3, \ldots, Goal_m\}$

and a set of ground rules:

$\{ \quad LHS_1 \text{ -> } RHS_1., \quad LHS_2 \text{ -> } RHS_2., \quad \ldots, \quad LHS_n \text{ -> } RHS_n. \quad \}$

then we want to transform it into the subgoals:

?- $\{RHS_i \mid \text{for some } j,\ LHS_i = Goal_j\}$.

When we are parsing, we obviously don't want to consider a large number of rules explicitly, and so we retain variables in our goals and rules where we have not yet narrowed down the precise feature values. We have to ensure, however, that the operations we perform correctly reflect the semantics of these goals and rules as abbreviations for sets of ground instances. So, if we have a goal *Goal* and a rule *Rule* with LHS *LHS* (all possibly containing variables), the set of instances of *Rule* that are applicable for solving *Goal* are represented correctly by $Rule'$ just in case:

$$\begin{aligned}
&gi(Rule') = \\
&\quad \{r \mid \\
&\quad\quad r \text{ is a ground instance of } Rule \text{ and} \\
&\quad\quad \text{LHS}(r) \in gi(Goal) \cap gi(LHS)\}
\end{aligned}$$

where $gi(X)$ is the set of ground instances of X. $Rule'$ is a partially instantiated version of *Rule*, because it has less (or the same number of) ground instances. Intuitively, we need to find a way of partially instantiating *Rule* (yielding $Rule'$) just enough so that the LHS of the new rule, LHS', has the same ground instances as the intersection of those of *Goal* and *LHS*:

$$\text{gi}(\text{LHS}(Rule')) = gi(Goal) \cap gi(LHS)$$

In our example above:

$Goal$ = `s(fin)`
$Rule$ = `s(Vform) --> n(2,Num), v(2,Vform,Num).`
LHS = `s(Vform)`
$gi(Goal)$ = {`s(fin)`}
$gi(LHS)$ = {`s(fin),s(bse),s(pas),...`}
$gi(LHS')$ = {`s(fin)`}

and the correct value of $Rule'$ is given by:

$Rule'$ = `s(fin) --> n(2,Num), v(2,fin,Num).`

It is from this instantiated rule that we then take the next goals to be satisfied (which themselves may represent whole sets of ground instances).

In conclusion, when a DCG rule is used top-down, the way for us to correctly model the semantics of this rule is to further instantiate this rule in such a way that the new LHS has as ground instances exactly those formulae which are ground instances of both the goal and the old LHS (the common ground instances). We can then take the RHS of the instantiated rule as the correct subgoals to be attempted. A similar argument can be made about using a DCG rule for bottom-up parsing. The operation of determining how to instantiate the rule for the goal is known as *unification*.

11.3 Term Unification

A term may be an atom, a variable, or a compound term, the latter being a functor and some fixed number of arguments, each of which is a term. For unification, we take two terms and determine how to instantiate them so that they are both the same term. If we instantiate them as little as possible to achieve this, the result will be a term whose ground instances are the common ground instances of the two original terms. Thus the recipe for how to do this instantiation will serve as a recipe for how to instantiate a rule so as to solve for a given goal.

More precisely, two terms unify if there exists an assignment of values to variables, called a *substitution*, that when applied to the two terms makes them identical. A substitution can be thought of as a function that applies to a term and returns a term with each of the variables replaced by its value as given in the substitution. So for instance the terms

```
p(X,f(Y,b),c)   p(g(Z,c),f(a,W),V)
```

may be made identical by the substitution:

```
{g(Z,c)/X, a/Y, b/W, c/V}
```

In order not to exclude any possible solutions, we are interested in the *most general unifier* (unifying substitution) of two terms. Informally, the mgu is the unique substitution that does not contain any pair which is not strictly necessary to collapse the pair to a singleton. The mgu of a goal and the LHS of a rule determines how to instantiate the rule to solve for that goal.

unify is a function that takes two terms represented in list notation, e.g.

p(a,f(b,c)) = (p a (f b c))

and returns such a substitution if there is one, or 'fail' if there is not (if there is not, then there is no common instance and the rule is not applicable for this goal). For instance, using Prolog's notational conventions for variables/constants:

unify(a(X,b(c,Y),d(X)), a(e,b(Z,W),W)) = e/X,c/Z,Y/W,d(e)/Y

Here is an algorithm for term unification:

```
function unify(T1,T2:terms) -> substitution;

if either T1 or T2 is atomic(functor,constant or variable) then
    if T1 = T2
    then return the empty substitution
    else
        if T1 is a variable then
           if T1 occurs in T2 then return fail    % occurs check
           else return {T2/T1}
        else
           if T2 is a variable then
              if T2 occurs in T1 then return fail % occurs check
              else return {T1/T2}
           else return fail                     % different atoms
else
   S1 <- unify(first(T1),first(T2))             % unify heads
   if S1 = fail then return fail
   else
      G1 <- apply(S1,rest(T1))           % substitute in tails
      G2 <- apply(S1,rest(T2))
      S2 <- unify(G1,G2)                 % unify tails
      if S2 = fail then return fail
      else return compose(S1,S2)
```

What is going on here can be simply glossed as:

- A variable unifies with any term it does not occur in.
- An atom (functor or constant) unifies only with an identical atom.
- Compound terms unify if their functors are identical and their arguments unify pairwise, and the substitutions obtained as a result of each of these unifications are compatible.

The Prolog interpreter embodies an algorithm very similar to this one, except that the check for occurence of a variable in the term that substitutes for it is not made, as a concession to efficiency. To see the import of this, try typing `?- X = f(X).` to the Prolog interpreter.

Notice also that the Prolog interpreter not only computes substitutions but applies them to the terms involved (destructively) as part of its procedural semantics.

11.4 Graph Unification

When we wrote Definite Clause Grammars, we used a variety of features, such as:

```
number:   {sing,plur}
subcat:   {intrans,trans,ditrans,...}
vform:    {fin,bse,inf,...}
gap:      {gap,nogap}
```

As we attempt to extend the coverage of our grammar, we will soon come up against the limitations of term unification. In a term, the number of arguments that a functor may have (its arity) is fixed, and each argument is identified solely by its position within the term. This encoding of features gives rise to the following problems. First, we must remember the correspondence between positions and features. Secondly, if we want to add a new feature, we must change our grammar at every point that feature is relevant. Finally, if we want to refer to the value of a feature in a term, we must specify the rest of the elements in the term, perhaps by marking them as anonymous variables. To illustrate this, consider the handling of agreement. Suppose we had a term with functor `agr`, whose arguments were the values of various agreement features. Initially, we might just include number, so the term would have the form:

```
agr(Num)
```

If we then wanted to add agreement for `person`, we would have to change all instances of this term throughout our grammar to be of the form:

```
agr(Num,Per)
```

When we specify values for agreement features, e.g. on lexical entries, we must remember the order - the two terms:

```
agr(sing,3)                  agr(3,sing)
```

are not equivalent. And if we wished to specify a value only for `person`, not `number`, as in the lexical entry for 'fish', we must write:

```
agr(_,3)
```

not merely

```
agr(3)
```

All these problems are solved simultaneously by the adoption of an alternative representation for features. In this representation, the name of a feature is made explicit. Once this step is taken, features may be referred to by name, so that position in a term is no longer the means by which a feature is located. Thus we could notate a feature structure including agreement features corresponding to the term:

```
agr(sing,3)
```

as

$$\begin{bmatrix} num : sing \\ pers : 3 \end{bmatrix}$$

This is identical to the structure:

$$\begin{bmatrix} pers : 3 \\ num : sing \end{bmatrix}$$

and if the structure contains a value only for the feature person, then we can write

$$\begin{bmatrix} pers : 3 \end{bmatrix}$$

without loss of compatability with other more fully specified feature structures.

Representations of features in this form are called several different names in the linguistics literature. The name *feature structures* is self-explanatory. The name *functional structures* can be understood by considering these structures as partial functions from feature names to feature values. That is, the first feature structure above is a function that takes the feature name *pers* to the value 3, the feature name *num* to the value *sing*, and all other names to undefined (also known as "top" - $\top$). In set-theoretic terms, the set of name:value pairs (feature specifications) comprising a feature structure may contain at most one pair with a given name, thus conforming to the definition of a function.

Finally, the name DAG for such structures refers to the possibility of drawing them as *directed acyclic graphs*. Such a graph has nodes corresponding to feature structures, edges labelled with feature names, and leaves labelled with atomic feature values.

Graph Unification is an operation that determines compatability of pairs of graphs by merging them to become a graph that contains the union of their feature specifications. If the union contains two feature specifications with the same name but different values, unification will fail (evaluate to bottom - $\perp$).

11.5 Reentrancy

The notion of *re-entrancy*, or *structure-sharing*, that is notated in Prolog by the sharing of variable names, is what makes feature structures in the general case graphs and not trees. The same sub-structure can be reached by following different paths through the graph. Reentrancy is often notated by coindexation boxes, e.g.

$$\begin{bmatrix} f : \boxed{1} \begin{bmatrix} h : a \end{bmatrix} \\ g : \boxed{1} \end{bmatrix}$$

Note that this a different structure from:

$$\begin{bmatrix} f : \begin{bmatrix} h : a \end{bmatrix} \\ g : \begin{bmatrix} h : a \end{bmatrix} \end{bmatrix}$$

since if we unify them with

$$\begin{bmatrix} f : \begin{bmatrix} d : b \end{bmatrix} \end{bmatrix}$$

the first will now be:

$$\begin{bmatrix} f : \boxed{1} \begin{bmatrix} h : a \\ d : b \end{bmatrix} \\ g : \boxed{1} \end{bmatrix}$$

and the second will be:

$$\begin{bmatrix} f : \begin{bmatrix} h : a \\ d : b \end{bmatrix} \\ g : \begin{bmatrix} h : a \end{bmatrix} \end{bmatrix}$$

Figure 11.1 shows these two structures as DAGs.

11.6 The PATR-II Formalism

It would be possible to write graphs down directly, but it is often convenient to distinguish between a description of a graph, which we write down, and the graph which satisfies that description.

PATR-II is a widely-used notation for describing feature structure graphs and grammars for combining them. It consists of a skeleton of context-free rules, in which the symbols are not atoms or terms but feature structures; each rule is associated with a set of equations that describe the feature structures introduced in the skeleton. The DCG rule:

```
s --> np, vp.
```

would be equivalent to a PATR-II rule of the form:

```
X0 -> X1 X2

<X0 cat> = s
<X1 cat> = np
<X2 cat> = vp
```

Here, the category has been treated as just another feature. The angle-bracketed expressions describe paths through a feature structure. In this rule, all the equations are between a path and a feature value. However, equations between two paths may also be specified.

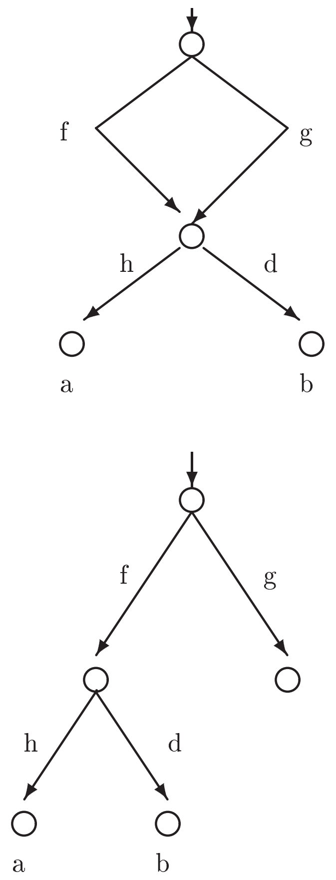

Figure 11.1: Directed Acyclic Graphs

```
s --> np(Agr), vp(Agr).
```

translates into:

```
X0 -> X1 X2

<X0 cat> = s
<X1 cat> = np
<X2 cat> = vp
<X1 agr> = <X2 agr>
```

To exemplify further, an alternative encoding of agreement could be given by the following rule and example lexical entries:

```
X0 -> X1 X2

<X0 cat> = s
<X1 cat> = np
<X2 cat> = vp
<X0 head> = <X2 head>
<X2 head subj> = <X1>

X0 -> fred

<X0 cat> = np
<X0 head agr num> = sing
<X0 head agr pers> =  3

X0 -> sleeps

<X0 cat> = vp
<X0 head vform> = fin
<X0 head subj head agr num> = sing
<X0 head subj head agr pers> = 3
```

Here `head` is a place where head features are kept. A sentence and its VP have the same values for all the head features. The head features for sentences and VPs divide into:

- `subj` - the form of the subject.
- `vform` - the form of the main verb.

For NPs, the head features include the agreement features (`agr`), which are person and number. Thus if a verb wants to say something about the person and number of its subject, it has to assign a value inside the `agr` of the `head` of the `subj` of its `head`.

In the next chapter we will look at how subcategorisation can be treated in a PATR-II grammar, and how graph unification may be implemented in Prolog.

11.7 References

Term unification was defined in Robinson's early work on theorem proving. Graph unification is originally due to Martin Kay, and has since become the basis for many modern linguistic theories. Gazdar and Mellish discuss PATR (chapters 4 and 7) and graph unification (sections 7.1 to 7.5). See also Shieber (1986) for an introduction to graph unification and a presentation of his own work on PATR-II.

- Shieber, S. M., *An Introduction to Unification-Based Approaches to Grammar*, CSLI/Chicago University Press, 1986.

11.8 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- What does term unification do when presented with two terms with the same functor but different number of arguments?
- What does term unification do when presented with `X` and `f(X)` (where `X` is the same in both inputs)? What should it do?
- How does one draw a complex feature structure as a DAG?
- What is reentrancy?
- How are categories like S, NP dealt with in the example PATR-II grammar?

What you should be able to do now:

- Describe the term unification algorithm and how it is used in Prolog.
- Discuss the advantages and disadvantages of term and graph unification.
- Explain the PATR-II notation and give simple example grammar rules using it.

Chapter 12

Implementing PATR-II

12.1 Aims of this Chapter

- To show how feature structures and graph unification can be implemented in Prolog.
- To show what is involved in constructing a simple Prolog implementation of PATR-II.
- To show a new approach to subcategorisation, similar in spirit to that used in Categorial Grammar, expressed in PATR-II.

12.2 PATR-II in Prolog

We saw in Chapter 11 how features on grammar rules could be interpreted by means of the operation of *unification*, but that representing feature structures as terms (i.e. suitable for direct interpretation by Prolog) gives rise to problems and inelegancies in grammar writing. Then we discussed an alternative representation as DAGs (sets of name:value pairs) which obviates such problems, and started to look at a formalism called PATR-II which supports grammar rules over structures of this form.

Actually expressing PATR-II rules in Prolog is simply a matter of defining some appropriate operators. Here are examples of a suggested notation in use:

```
X0 ---> X1, X2 :-
   X0:cat <=> s,
   X1:cat <=> np,
   X2:cat <=> vp,
   X1:num <=> X2:num.

X0 ---> [john] :-
   X0:cat <=> np,
```

```
X0:num <=> sing.
```

Note that `--->` and `<=>` are Prolog predicates, and that the functor `:` is used to construct paths. All of these need to be given appropriate operator declarations.

In order to construct a complete PATR-II interpreter, we just need to define the predicate `<=>`. `<=>` is given two arguments specifying feature structures. An argument may be a Prolog variable, an atom, or a path specifying a sequence of features whose values are to be found inside a feature structure specified by a variable. `<=>` needs to:

- "Evaluate" both arguments, yielding two feature structures/ constants (possibly instantiating categories, if the mentioned paths don't yet exist).
- Pass these two results to a predicate that "makes the two results equal" by graph unification.

This gives the following structure:

```
X <=> Y  :-
  denotes(X,Z1),
  denotes(Y,Z2),
  graph_unify(Z1,Z2).
```

where `denotes` is a predicate to evaluate paths and `graph_unification` is graph unification.

12.3 Representing Feature Structures

The first feature structure of Chapter 11 could be represented in Prolog as:

```
[num:sg,pers:3|_]
```

Note the variable tail, so that we can destructively add feature specifications without building a new structure. In general, we represent a feature structure by a list of `Feature:Value` pairs terminating in a variable. If a feature value is an atom, then it is just represented as a Prolog atom; if it is complex, then it in turn is represented as a list of pairs terminating in a variable.

Given this representation, it is quite straightforward to write the predicate `denotes` which, for instance, given the path `X:cat`, extracts the value of the feature `cat` stored in the value of the Prolog variable `X`, instantiating `X` to have a `cat` value if it does not already. In general, `denotes` will take a path description, i.e. one half of an equation, and return the graph that it denotes.

The predicate `denotes` has to be able to return the value denoted by any of the following expressions:

```
X
np
X:cat
X:subj:cat
```

Here is a Prolog definition that will do the trick:

```
denotes(X,X) :- var(X), !.
denotes(X:F,V) :- !, denotes(X,V1), member(F:V2,V1), !, V=V2.
denotes(X,X).
```

The first clause handles the case of an uninstantiated variable. The second handles the case of a non-trivial path. The program finds out what the first part of the path denotes (`V1`) and then looks for an entry for feature `F` in the resulting list. The first cut in this clause confirms the choice of clause if the first argument is of the form `X:F`, whereas the second ensures that only the first solution of the `member` goal is taken. Note that if `V1` is initially uninstantiated, the (standard) `member` predicate will instantiate it enough so that there is an (uninstantiated) entry for `F`. The last clause handles all other cases, for instance a value like `s` or something that is already a graph (i.e. a list of `F:V` pairs).

12.4 Implementing Graph Unification

Recall that Prolog's procedural semantics replaces two structures that are term unified with their unification. We want to do the same with graph unification - that is, when `X <=> Y` has been satisfied we want `X` and `Y` to be the same thing. Since two structures which may be graph-equal may not start off term-equal, they cannot be made term-equal. What we need to do in essence is redefine equality so that e.g.

```
[num:sg,pers:3|_] = [pers:3,num:sg|_]
[gen:m|_] = [num:sg|_]
```

etc. Since Prolog will not allow us to actually redefine `=`, and we wouldn't want to anyway, we will actually define a predicate `graph_unify` that expresses the desired relation.

```
graph_unify(X,X) :- !.
graph_unify([A:V1|R1],F2) :-
   del(A:V2,F2,R2),
   graph_unify(V1,V2),
   graph_unify(R1,R2).

del(F,[F|X],X) :- !.
del(F,[E|X],[E|Y]) :- del(F,X,Y).
```

The first clause for `graph_unify` just says that two terms `graph_unify` if they are `=`. This is used if two DAGs happen to be unifiable as Prolog terms (e.g. if one is a variable), and acts as the base case for non-f-structure values (atoms).

The second clause graph-unifies two arbitrary DAGs if possible, and fails otherwise. Its behaviour depends on that of `del`, which works as follows.

del(Element,List1,List2) is true if List2 is List1 with the first occurrence of Element removed[1],

Hence the name of the procedure. However, because of the variable tails, Element will always be in List1 after del has terminated. Since List1 is the second DAG, and since del is called for each feature spec. in the first DAG, this ensures that that the second DAG contains at least those feature specs. that are in the first. Meanwhile, a list containing all those feature specs. in the second DAG that weren't in the first has been constructed (R2). When each feature spec. in the first has been treated, this list becomes its tail. Thus the first DAG contains at least those feature specs. that were in the second. Therefore the two DAGs are the same.

Of course, graph_unify can also fail. This will happen if one list contains the feature spec. Name:Value1 and the other contains Name:Value2, where Value1 and Value2 are different atoms or non-unifiable DAGs (which ultimately reduces to the same thing).

Here is an example of graph_unify in action.

```
test(X,Y) :-
    X = [a1:v1, a2:v2, a3:[a11:v11, a21:v21|_]|_],
    Y = [a2:v2, a4:a4, a3:[a21:v21, a31:v31|_]|_],
    graph_unify(X,Y).

| ?- test(X,Y).
Y = [a2:v2, a4:a4, a3:[a21:v21, a31:v31, a11:v11|_112],
       a1:v1|_57]
X = [a1:v1, a2:v2, a3:[a11:v11, a21:v21, a31:v31|_112],
       a4:a4|_57]
yes
```

12.5 Subcategorisation

The grammars that we have considered so far have encoded the relation between a verb and its complements by means of an atomic-valued feature *subcat*, which mediates between a lexical entry and a phrase structure rule. A dominant trend in modern linguistics has been to make the connection between a lexical predicate and its complements much tighter, by having general phrase structure rules operate over more richly structured lexical entries. This lexicalist approach is characteristic of Head-Driven Phrase Structure Grammar (HPSG) and to an even greater extent of Categorial Grammar (CG).

In HPSG, the feature *subcat* takes as its value a list of categories. The graph representation of a list uses the features *first* and *rest*. For instance, a verb which takes np and vp complements would have the following description of subcat:

[1]If the arguments to del are feature structures, then there will only be one occurrence of Element at most.

```
<X subcat first cat> = np
<X subcat rest first cat> = vp
<X subcat rest rest> = nil
```

Each element of the list will unify in turn with one of the complement categories. Since the value on the *subcat* list contains the information about the category of the complement, the rule which performs the combination need not include this. Thus we can use a general rule of the form:

```
X0 -> X1 X2

<X0 cat> = vp
<X1 cat> = vp
<X0 head> = <X1 head>
<X1 subcat first> = <X2>
<X1 subcat rest> = <X0 subcat>
```

Notice how the first element of the head daughter's *subcat* list unifies with the complement daughter, and the rest of the list unifies with the subcat of the mother. This single rule, embodying what is called the *subcat feature principle*, may replace all the verb phrase rules of earlier grammars. To act as the basis for the recursive rule above, we merely need to enter vp's with the appropriate *subcat* values in the lexicon.

Subjects may either be the value of a specified (head) feature, or included on the *subcat* list, as the last element. In the latter case, they would be handled by the following rule:

```
X0 -> X1 X2

<X0 cat> = s
<X2 cat> = vp
<X0 head> = <X2 head>
<X2 subcat first> = <X1>
<X2 subcat rest> = nil
```

The verb 'persuades' would have the following lexical entry.

```
X0 -> persuades

<X0 cat> = vp
<X0 head vform> = fin
<X0 subcat first cat> = np
<X0 subcat first head nform> = norm
<X0 subcat rest first cat> = vp
<X0 subcat rest first head vform> = inf
<X0 subcat rest rest first cat> = np
<X0 subcat rest rest first agr pers> = 3
<X0 subcat rest rest first agr num> = sing
<X0 subcat rest rest rest> = nil
```

Of course, one would not store all this information with each lexical entry for a verb. PATR-II makes use of a template mechanism. A template is just a name for a set of equations. A verb will then be stored with the template name, which is replaced by the equations themselves during lookup. Templates may contain calls to other templates, so the expansion procedure must be recursive.

Categorial Grammar

Categorial Grammar is an approach to grammars which dramatically simplifies the grammar rules, at the expense of complicating the set of possible categories (and hence the lexicon). For instance, a transitive verb might be represented by a category such as $(s\backslash np)/np$ - something that combines with a following NP to produce $(s\backslash np)$ - something that combines with a preceding NP to produce an S. Given such categories, at its simplest CG needs only two schematic rules:

```
X --> X/Y, Y.
X --> Y, X\Y.
```

Simple CGs are weakly equivalent to context-free grammars, but many extensions to the basic idea are possible. Categorial grammars were originally developed by Ajdukiewicz in Mathematical Logic. They were used by Montague in his approach to the formal semantics of English ("Montague Grammar"), but his coverage of syntactic phenomena was not strong. Their serious use to describe the syntax of natural languages was probably pioneered by Steedman.

- Ades, A.E. and M.J. Steedman (1982), On the Order of Words, Linguistics and Philosophy, 4, 517-558.
- Dowty, D. R., Wall, R. E. and Peters, S., *Introduction to Montague Semantics*, Reidel, 1981 (chapter 7).

12.6 References

The implementation of the graph unification operator due to Eisele and Dörre (1986). The HPSG-style treatment of subcategorisation is from Shieber (1986). For further details of HPSG, see Pollard and Sag (1988). Gazdar and Mellish discuss the implementation of a PATR-II interpreter in Prolog, using the same representation for feature structures as is discussed here.

12.7 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- What are the main predicates that need to be defined for a PATR-II interpreter in Prolog, and what arguments do they take?
- Why are feature structures represented by lists terminating in variables?
- In the approach to subcategorisation shown, what is the format of the subcategorisation list associated with a word?

What you should be able to do now:

- Explain how representing feature-structures (graphs) as lists with a variable tail allows a graph unification operator to be defined in Prolog, and give that definition.
- Describe the HPSG-style encoding of the complementation properties of verbs and give example PATR-II grammars to illustrate this.

Chapter 13

Parsing with Unification Grammars

13.1 Aims of this Chapter

- To spell out the possible ways that a parser for unification grammars (e.g. PATR-II grammars) might be constructed.
- To show the ways in which the design of a parser that deals with UGs directly has to be more complex than a parser for context-free grammars.

13.2 Options for Parsing with UGs

We have presented chart-parsing algorithms using context-free phrase structure grammars, but from Chapter 11 onwards we have made use of grammars where the categories have more structure than the monadic categories of CFG in order to cover interesting natural language constructions. What exactly are the issues to be faced when implementing a parser (a chart parser, say) for a grammar that allows complex categories? There are really three basic options for how to parse when a grammar allows complex categories:

1. *Expand the grammar into a monadic CF-PSG.* As we have seen, the simplest use of features can be viewed as a way of specifying sets of rules in an abbreviated way. In this case, all we need to do is expand out the rule set and parse using the standard techniques. But this strategy is really only feasible for small toy grammars, since otherwise the set of rules will be quite vast. The size of the set of rules that deal only with coordination is likely to be at least as great as the size of the set of categories. The size of a set of categories based on n binary features is 3^n (it would only be 2^n if all features had to be fully specified), so even with a modest set of, say, 24 binary features, we have 3^{24} (282,429,536,481) categories. The size of

the grammar impacts directly on both the space (memory) and time requirements of parsing. In practice, therefore, even if the induced category set is finite, we have little choice but to employ the feature system directly while parsing, one way or another.

2. *Parse using a monadic CF-PSG backbone.* If there is a feature (or set of features) such that every description in every grammar rule specifies a definite value for that feature, an initial parsing can take place using the monadic CF-PSG derived by just considering this feature and forgetting the others. For instance, some grammars specify a specific value for the *cat* feature of each category and in such cases we could parse only using this feature. Of course, if other features are neglected in the parsing, the parser can be expected to waste time exploring hypotheses and formulating potential analyses that would be excluded by the full grammar. The parser will also produce some unsound analyses. A filtering process will then be required to determine which complete analyses found by the initial parsing were actually valid, and what the values of the other features were. Parsing with a context-free backbone suffers from the fact that a filtering process is needed after the main parsing process and also that the parsing search space is widened by ignoring other features (how much depends on the extent to which constraints on these features can actually reject parses, rather than enable structures based on the backbone features to be built). In addition, this approach will not be feasible when there is no single feature that is given a value in every description in the grammar.

3. *Incorporate special mechanisms into the parser.* The basic parsing problem is the same, whether categories are atoms or arbitrary feature structures. All that is required is that the basic operations of the parser (testing for equality, looking for rules, and so on) be redefined to accept a different data structure for categories. This simple fact, unfortunately, masks a number of special problems that arise in parsing directly with complex categories. Nevertheless, in the end this is probably the most effective approach, and we will survey here some of the special problems that arise and some standard ideas for their solution.

13.3 Parsing with Complex Categories

Let us start by surveying the main differences between parsing with monadic categories and with grammars involving complex categories. First of all, the raw material of the grammar, the rules and lexical entries, are different. Secondly, we must start using feature structures in appropriate places in the chart. Instead of storing dotted rules containing monadic categories, we now need to store dotted rules containing feature structures. In general this also means that we can't tabulate the relations `left_sister` and `right_sister` in advance, since there may be infinitely many possible dotted rules.

The categories in an arc will not in general be completely specified for all features, and, as with rules, we must take each arc to be a statement about all possible categories that extend the categories specified.

How does the operation of the fundamental rule look when we have complex categories? In general, the result of an application of the fundamental rule is a new arc similar the contributing active arc, except that it:

- Is further instantiated, by having in the place of the first required category the result of unifying that category with the category found, and
- Has the dot moved on one place to the right.

Figure 13.1 shows this in diagrammatic form. The active edge corresponds to an instance of the following grammar rule:

```
X0 -> X1 X2
<X0 cat> = s
<X1 cat> = np
<X2 cat> = vp
<X0 mood> = <X2 mood>
<X1 num> = <X2 num>
<X1 person> = <X2 person>
```

Here we are assuming that the noun phrase (`X1`) has already been found (in fact, it has been incorporated into the active edge by another application of the fundamental rule) and has person 3 and unknown number. The active edge represents the fact that a verb phrase needs to be found. The inactive edge contains a dotted rule indicating a completed VP (we ignore the RHS of whatever rule was used to find it). Assuming that the two edges are in the right places, they can combine by the fundamental rule to give the new edge. Notice how when the category for the complete verb phrase is unified with the category in the active edge indicating the needed verb phrase, the number of the noun phrase becomes instantiated to singular and the mood of the sentence becomes instantiated to declarative.

What about the prediction rule? Figure 13.2 shows a relatively simple case. To start off, we have an active edge which is looking for an NP with case nominative. If we are doing top-down parsing, this will cause the prediction rule to be used. For every rule in the grammar whose LHS unifies with the desired category, an active edge will be added looking for the RHS of that rule. Here we show the grammar rule that would arise from:

```
X0 -> X1 X2
<X0 cat> = np
<X1 cat> = det
<X2 cat> = noun
<X0 num> = <X1 num>
<X1 num> = <X2 num>
<X0 case> = <X1 case>
```

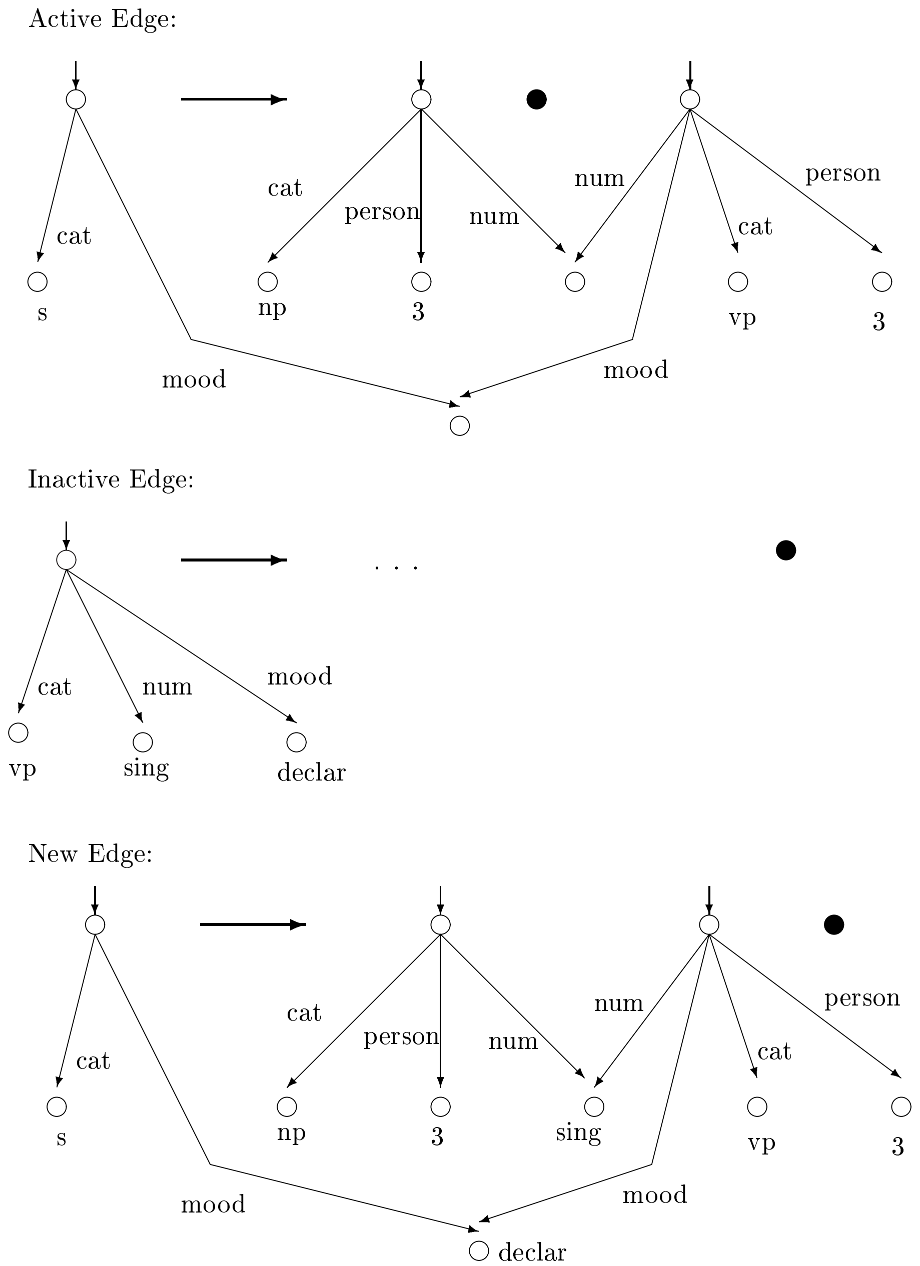

Figure 13.1: Action of the Fundamental Rule

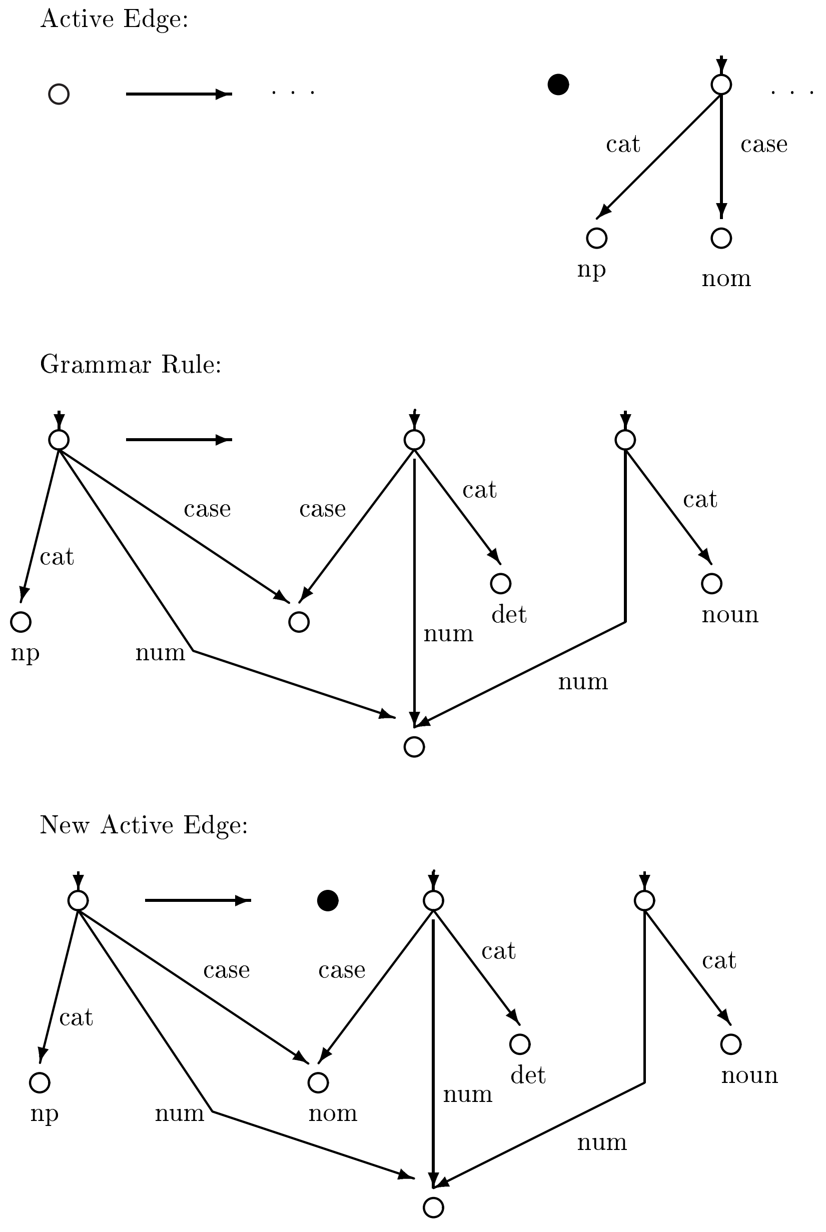

Figure 13.2: Prediction Rule in Operation

The result of prediction is a new active edge based on this rule, with the dot directly after the `->` and the rule instantiated to match the desired category (here, the case feature is instantiated).

13.4 Copying and Structure Sharing

Our sketch of how the fundamental rule can work with complex categories has glossed over important issues about the multiple use of edges. In a chart parser, every active and inactive arc must be preserved until parsing has finished, because it may need to combine together with an arc that has not yet entered the chart. Consider, for instance, the processing of the sentence

A woman with lots of children can still enjoy life.

and imagine that we are keeping track of the *num*ber (*sing* or *plur*) of phrases like NPs and VPs. Now, our chart parser might well find the VP 'can still enjoy life' fairly early; the *num* feature for this phrase will be unspecified, as the phrase could serve as either a singular or a plural VP. A bottom-up parser might well combine this phrase with the plural NP 'lots of children' to yield a complete sentence. The appropriate application of the fundamental rule will result in an edge where the NP and the VP will both be plural, as the subject and predicate of a sentence must share their *num* values. It would be wrong to keep any of the new information obtained about the VP in any place but the new edge, however, because the original VP edge will eventually have to combine with the incomplete singular NP 'a woman with lots of children' and hence be seen (in this context) as a singular phrase. In this example, the original VP edge must remain unscathed by its combination with the first NP if it is to correctly combine with the second.

What implications does this have for chart parser implementation? First of all, they mean that *destructive* unification cannot be used in the generation of new edges, as this is only safe if the graphs that are changed will never be needed again. Thus, in applying the fundamental rule, we cannot safely overwrite the active arc and reuse the data structures to build the new arc - the partial solution that it represents must be preserved for possible later use. same variables to represent both extensions at the same time. The obvious solution to these problems is to always take a copy of an edge before using it in unification [1]. In the new copies we can always ensure that fresh variables are used so as not to conflict with variables used elsewhere. Copying is, however, expensive, and a number of schemes for *structure sharing* have been devised in an attempt to reduce the amount of copying required. The new edge created by the fundamental rule will in general be quite similar, although not identical, to the active edge that participated. So, we can devise implementations of these data structures where the new structure is represented, not as a completely new object, but as the original active edge together with a record of the changes

[1] If in Prolog we use `assert` to store chart edges in the database, then we are effectively taking a new copy of an edge every time we retrieve it from the chart

(additions) necessary to convert that structure into the new one. In a Prolog implementation, this might be done by using substitutions to represent changes, and by representing an edge by a triple consisting of a grammar rule, a number (recording the position of the dot) and a substitution. Such a representation will make it harder to see exactly what the contents of a given edge are. On the other hand, it will make the job of building new edges much easier.

13.5 Termination

The crucial advantages of chart parsing come from the way in which duplication of effort is avoided. When categories are monadic, it is trivial to check whether an edge is already present in the chart, because either there is an edge mentioning the same categories or there is not. Checking that an edge is not already in the chart before doing anything with it also ensures termination, since with a CFG there are only finitely many edges that can be constructed for any input string. When categories have complex internal structure, however, there are more possible relations between a new edge and an edge already in the chart than simply being the same or different, and so our duplication tests must be more sophisticated. If we already have an active edge looking for a noun phrase of any *num* at some position in the string, for instance, we do not want to add a new edge that is looking for a plural noun phrase at that point, even though the edges are different. One might think that the appropriate test is for whether a new edge *unifies* with an edge already in the chart. However, if we already have an edge looking for a plural noun phrase, it is quite reasonable to add a new one that will accept any noun phrase, because otherwise possible solutions will be lost. So in fact, the test that a new edge must pass before being added to the chart is that it is not *subsumed* by any existing edge. One edge subsumes another if it is less than or equally instantiated than it[2].

The subsumption test is a way of preventing loops where different, but increasingly instantiated, edges are added to the chart. But it is still possible for an infinite number of edges to be added with none of them subsuming any other. Consider how the prediction rule will work with rather more pathological rules. For instance, in an implementation of Categorial Grammar, we might represent a complex category X/Y as a structure with two features – the "argument" (`arg`) Y and the "result" (`res`) X. We might then want to have a grammar rule

[2]In general, we can consider feature structures as forming an example of the mathematical concept of a *lattice*. They are partially ordered by the relation 'is at least as informative as' e.g.

`[pers:3, num:plu]` is at least as informative as `[num:plu]`

Instead of 'X is at least as informative as Y', we often say that 'X *extends* Y', or that 'Y *subsumes* X'. The *unification* of two elements a and b in this lattice is then the element that minimally extends both a and b, i.e. $a \sqcap b$. The dual of this, the element that minimally subsumes a and b, i.e $a \sqcup b$, is called the *generalisation* of the two feature structures. The unique maximal element of this lattice ($\top$) is the one which is maximally uninformative, i.e. the empty feature structure. The unique minimal member ($\bot$) is that element that is so informative it is contradictory.

expressed informally as follows:

```
X -> X/Y Y
```

which could be expressed in PATR-II as:

```
X0 -> X1 X2
<X1 res> = <X0>
<X1 arg> = <X2>
```

If at some point a constituent of category s is sought, prediction will use this rule to initiate a search for a phrase of category s/Y (for any Y). The prediction rule will then apply to this, causing a search for (s/Y)/Z, and so on indefinitely. Figure 13.3 shows how this looks for the PATR-II formulation. Notice that none of the new edges created subsumes any other.

The mechanism introduced to correct this is known as *restriction*. The idea is that when the prediction rule is used it should use a restricted copy of the goal category used to match against the LHS of grammar rules. In our example, we could have restriction only leave the top level res and arg features of the category. Restricting the categories that are used in prediction does lead to unnecesarily general edges being added to the chart but does not affect correctness. Its advantage is that there are fewer different edges added and the subsumption check thus applies to ensure termination. On the other hand, finding a good way to restrict categories used in the prediction rule can be a difficult task.

13.6 Indexing

Once we have a non-trivial grammar and non-trivial sentences, the efficient retrieval of edges from the chart and rules from the grammar becomes important. In the CFG case, it is fairly easy to devise ways of *indexing* the information on the categories that appear in particular positions. For instance, a top-down parser can productively index the grammar rules on the LHS categories and a left-corner parser can productively index them on the first categories on their RHS. The existence of complex categories gives us a greater choice on indexing strategies. Where there is a conventional *cat* component to each category, a simple strategy is to index on the *cat* values - this reduces to something essentially the same as the monadic category case. This strategy will not work straightforwardly, however, if rules do not always specify *cat* values for all constituents. For instance, it is very unclear how one would index the verb subcategorisation rule from Chapter 12:

```
X0 -> X1 X2
<X0 cat> = vp
<X1 cat> = vp
<X0 head> = <X1 head>
<X1 subcat first> = <X1>
<X1 subcat rest> = <X0 subcat>
```

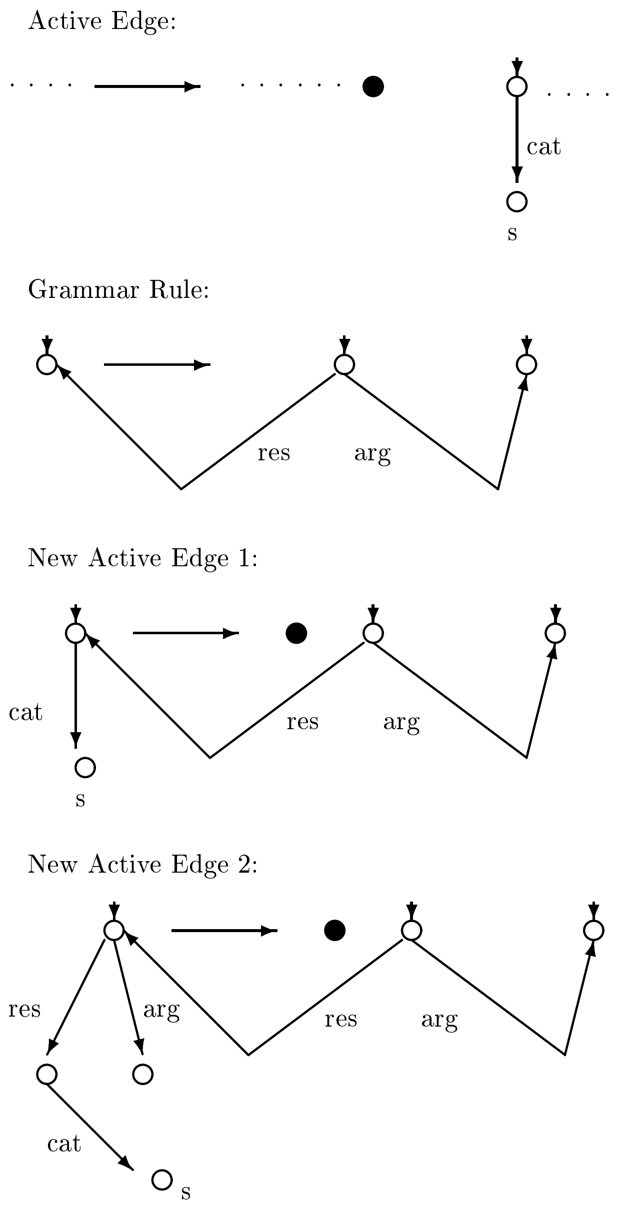

Figure 13.3: Nonterminating Applications of Prediction

for a *right*-corner parser, because the last category on the RHS could have any value for its *cat* feature (at least, this is true of the rule as we have presented it, which is something of an oversimplification). If indexing is on the basis of the *cat* feature, the only solution seems to be to lump together all rules of this kind in a bundle to be tried bottom up, whatever the category of a found phrase happens to be. If there are many such rules, or indeed many rules indexed by any given category, some form of secondary indexing (using other features) will be needed to cut down further the rules considered in any instance. In general, then, the designer of an efficient chart-parsing system needs to allow for indexing on a tunable set of features (such as *cat*, *subcat*, *slash*) that may well need to be changed as the grammar develops.

13.7 Conflating Similar Edges

As it is set up, a chart parser will eventually enumerate all possible analyses of a given string by the time it halts. Each of these will yield a, possibly complex, category describing the string, as well as a tree expressing its constituent structure, if we decide to build it. Now, if we are only interested in the category itself, and this will be the case where our task is only recognition or where the (possibly extensive) information in the topmost category is all we need for some task, we will be uninterested in alternative analyses that give rise to categories containing exactly the same information. Since the information in a category is obtained entirely from the grammar rule used to recognize it and the categories of the immediate subphrases, we will likewise be uninterested in alternative analyses of a smaller phrase that all yield essentially the same category. We are thus led to consider a strategy of conflating inactive edges for similar categories, even if the internal structure of the edges (how they were found) is not identical in every respect.

Consider, for example, a recognition task involving a grammar that describes categories entirely in terms of the values of *cat* and *num* features. A given noun phrase, occupying a given portion of the string, may well have three possible analyses, say, but if these are all *sing* in *num*, it is only necessary to consider one such analysis. Similarly, if there are three possible ways of finding a verb phrase after that noun phrase, all occupying the same portion of the string and being likewise *sing*, we only need to consider one of them. If we can avoid adding the extra edges to the chart in both cases, it will only be necessary to apply the fundamental rule once, rather than nine times, to recognize a sentence consisting of noun phrase followed by a verb phrase. This is obviously a considerable saving.

This idea of conflating similar edges can be extended to a technique of *packing*, which is useful even when we *are* interested in the complete internal structure of phrases. When there are two different analyses covering the same portion of the string and yielding equivalent categories, the technique is to conflate these into a single edge which indicates that there are several possible internal analyses for the phrase. When the resulting 'packed' edge is combined with another edge, it is left in its packed form - the two analyses are not expanded out. Only when a complete inactive edge covering the whole string has been found are all the possible analyses worked out (if this is required) by unpacking all the packed subphrases and combining together all the alternative analyses in all possible ways. If all possible analyses are eventually required, the packing and unpacking of the edges contributing to these analyses will not make much difference to the amount of work done. On the other hand, if there is a possible phrase with several analyses but which does not contribute to any complete analysis of the string, packing the relevant edges together can save an appreciable amount of work, because the (in the end futile) work done in combining this phrase with other phrases only has to be done once, rather than once for each possible internal analysis. This 'packing' technique is very similar to the technique we used to save possible analyses implicitly (rather than expanded out) in Chapter 10.

13.8 References

This chapter is based heavily on part of Chapter 7 of Gazdar and Mellish, which however contains useful extra information and examples in some places. Structure-sharing is described in more detail in Karttunen and Kay (1985) and Pereira (1985), whilst indexing and termination are discussed in Shieber (1985). "Packing" is introduced by Tomita (1986).

- Karttunen, L. and Kay, M., "Structure sharing with binary trees", Procs of the 23rd Annual Meeting of the ACL, 1985.
- Pereira, F. C. N., "A Structure-sharing representation for unification-based grammar formalisms", Procs of the 23rd Annual Meeting of the ACL, 1985.
- Shieber, S. M., "Using restriction to extend parsing algorithms for complex-feature-based formalisms", Procs of the 23rd Annual Meeting of the ACL, 1985.
- Tomita, M., *Efficient Parsing for Natural Language: A fast algorithm for practical systems*, Kluwer, 1986.

13.9 Comprehension check

After reading this chapter, you should be able to answer the following quick questions:

- What are the three possible ways discussed of implementing a parser for unification grammars?
- What does a dotted rule look like for a unification grammar?
- Where is unification used in the fundamental rule and in prediction?
- Why do chart edges have to be copied when they combine together?

- What is the appropriate test to use to see whether a new edge "is already in the chart"?
- What technique can handle loops involving repeated prediction?
- What is the purpose of indexing in a chart parser?

What you should be able to do now:

- Design the basic components and algorithms for a chart parser for PATR-II grammars.