Chapter 3

GRAMMARS AND LANGUAGES

Formal grammars are used to describe the syntax of languages, because they are more precise than descriptions in natural language and allow automatic processing. Both in Informatics and in Linguiastics, Context Free grammars have been widely accepted as a description formalism. For more ambitious language definitions, various forms of context sensitive grammars are used. Regular languages are used widely for the description of repetitive patterns, compiler makers use them for lexical analysis (recognition of lexemes).

This chapter is a recapitulation of some notations, facts and terminology from formal language theory that are of frequent use.

3.1 Phrase Structure Grammars

We begin by introducing a family of grammars, which includes the class of context-free grammars.

A **Phrase Structure Grammar** is a 4-tuple $G = (V_n, V_t, P, \Sigma)$, where

V_n	$=$	finite alphabet of **nonterminal symbols**
V_t	$=$	finite alphabet of **terminal symbols**
		$V_n \cap V_t = \emptyset$
		$V = V_n \cup V_t$
P	$\subseteq$	$V^+ \times V^*$, the set of **production rules**
Σ	$=$	The **initial symbol** $\in V_n$, the **root** of the grammar.

In a production (δ, τ) we call δ the **left-hand side** and τ the **right-hand side**. we will write $\delta \rightarrow \tau$ for $(\delta, \tau) \in P$. It is customary to generalize the relation $\rightarrow$ as follows:

Let $\pi, x \in V^*$. We define $\Rightarrow$ (**directly produces**) as:

$$\pi \Rightarrow x \quad \Longleftrightarrow_{def} \quad \exists \alpha, \beta, \delta, \tau \in V^* \mid \pi = \alpha\delta\beta, x = \alpha\tau\beta, \delta \rightarrow \tau$$

Let $\Rightarrow^+$ (resp. $\Rightarrow^*$) be the (reflexive-) transitive closure of $\Rightarrow$. Define $L(G)$, the **language** of G:

$$L(G) =_{def} \{x \in V_t^* \mid \Sigma \Rightarrow^+ x\}$$

Some notation and terminology:

$\delta \rightarrow \tau$	there is a production rule (δ, τ)
$\pi \Rightarrow x$	x is a **direct production** of π
$\pi \Rightarrow^+ x$	x is a **production** of π
$\pi \Rightarrow^+ x \wedge x \in V_t^*$	x is a **terminal production** of π
$x \in V^*$	x is a **sentential form**
$\Sigma \Rightarrow^+ x \wedge x \in V^*$	x is a sentential form of the grammar G
$\Sigma \Rightarrow^+ x \wedge x \in V_t^*$	x is a **sentence** of the grammar G

3.1.1 The Chomsky Hierarchy

In 1956, Noam Chomsky showed that by imposing restrictions on the set of productions P various classes of PSG's (and their corresponding languages) can be defined, which form an hierarchy:

type 0: $P \subseteq V^+ \times V^*$
No restrictions. **Semi-Thue System**, describing all recursive languages.

type 1: $P = \{(\delta, \tau) \in V^+ \times V^* \mid \delta = \alpha u \beta, \tau = \alpha v \beta, \alpha, \beta \in V^*, u \in V_n, v \in V^+\}$
context sensitive, no 'deleting' of nonterminals, corresponding to primitive recursion.

type 2: $P \subseteq V_n \times V^*$
context free, the most widely used member of the PSG family.

type 3: $P \subseteq V_n \times (V_t \cup V_t V_n) \cup \{\epsilon\}$
(right-)**regular**

an even more primitive class could be added:

type 4: $P \subseteq V_n \times V_t$
finite choice

The **Chomsky hierarchy** is strict. Of course all kind of further gradations can be defined.

Type	Grammar	Automaton	Language
Chomsky 0	Class 0	2-pushdown or Turing-machine	Recursive
Chomsky 1	Context Sensitive		Primitive recursive
Chomsky 2	Context Free	1-pushdown	Context Free
Chomsky 3	Regular	Finite State	Regular
Chomsky 4	Finite Choice		Finite.

Table 3.1: The Chomsky hierarchy (extended)

3.1.2 Examples

. Typical languages on various levels of the Chomsky hierarchy are:

- Regular: ab^*c
- Context Free: $a^n b^n$ or $w\ \overline{w}$, where $\overline{w}$ is the mirror image of w
- Context Sensitive: $a^n b^n c^n$.

3.2 Context-Free grammars

For the rest of this chapter we will restrict ourselves to **Context-Free Grammars**. Note that for a CF grammar, productions are elements of the set $V_n \times V^*$.

3.2.1 Notation

Rather than writing down a grammar as a tuple of sets, enumerating the elements of V_n, V_t, P and supplying Σ, there are more readable ways of denoting a CF-grammar (**CFG**).

3.2.1.1 Traditional linguistic notation

In the notation prevalent among linguists, the production (x, y) is denoted as
x → y
Several productions with the same left-hand side may be combined into one **rule**, like
a → b a | c
The sets V_n and V_t are defined implicitly by the rules. Σ is by convention the left-hand side of the first rule of the grammar. A small example (**G1**):

sentence → subject verb object
subject → the man | the dog
verb → bit | saw
object → the man | the dog | his wife

Notice that some convention has to be used to distinguish terminal and nonterminal symbols, like the use of different typefonts, and there is no sign indicating the end of the rule. This makes the formalism less suitable for automatic processing.

3.2.1.2 Backus-Naur form

In Informatics, the usual notation for CF grammars is **BNF**, introduced by Backus and Naur in the Lalgol 60 report. Nonterminal symbols are enclosed between sharp brackets, terminal symbols are given literally. In this notation, G1 looks as follows:

⟨sentence⟩ `::=` ⟨subject⟩ ⟨verb⟩ ⟨object⟩

⟨subject⟩ `::= the man $|$ the dog`

⟨verb⟩ `::= bit $|$ saw`

⟨object⟩ `::= the man $|$ the dog $|$ his wife`

Again there is no sign indicating the end of the rule.

3.2.1.3 One-level van Wijngaarden grammars

In the Lalgol68 report [?] a notation for two-level grammars was given (**VWG**) which is both compact and suitable for automatic processing, and from which CF grammars can be obtained by omitting the second level (**1VWG**).

A 1VWG is a collection of **rules** of the form:

`lhs: rhs.`

The left-hand side is (the name of) a nonterminal symbol. The right-hand side consists of one ore more **alternatives** separated by semicolons. An alternative consists of zero or more **members** separated by commas. A member is in the CF case (the name of) a single symbol. Each alternative corresponds with a production $\in P$: its left-hand side is the left-hand side of the rule, its right-hand side consists of the members of an alternative. In a slight deviation from the original notation, terminal symbols will be enclosed between quotes.

```
sentence:
   subject, verb, object.

subject:
   "the man"; "the dog".

verb:
   "bit"; "saw".

object:
   "the man"; "the dog"; "his wife".
```

The **language** of an L1vwg is defined as usual. Any sentence can be generated by starting with a sentential form consisting of the starting symbol, and repeatedly replacing in this sentential foem some nonterminal by one of its direct productions, until a terminal form is obtained.

sentence
$\Rightarrow$ subject, verb, object
$\Rightarrow$ the man, verb, object
$\Rightarrow$ the man, bit, object
$\Rightarrow$ the man, bit, his wife

Ignoring the comma's, we have obtained a sentence by repeated direct productions. Such a sequence of replacements is termed a **derivation**.

3.2.2 Weak equivalence

As the previous example shows, it is possible that two grammars describe the same language.

Two grammars G_1 and G_2 that describe the same language are termed **weakly equivalent**:

$$G_1 \approx G_2 \Longleftrightarrow_{def} L(G_1) = L(G_2)$$

Two grammars that not only generate the same language, but also assign the same parse tree to sentences are termed **strongly equivalent**.

A language may have any number of weakly equivalent grammars. The problem of finding a grammar to describe a given language (given e.g. by intuition, or in the form of another formalism or by enumeration) is called **grammar inference**. Automatic grammar inference is a very hard process, and due to the phenomenon of weak equivalence there is no such thing as *the* grammar of language.

3.2.3 Normal Forms

Various normal forms for CF grammars exist.

A CF grammar is in **Greibach normal form** (**GNF**) iff all its rules are of the form

$$V_n \to V_t V^*$$

(i.e. all alternatives start with a terminal symbol).

A CF grammar is in **Chomsky normal form** (**CNF**) iff all its rules are of one of the forms

$$V_n \to V_t$$

$$V_n \to V_n V_n$$

$$V_n \to \epsilon$$

(i.e. alternatives are of a simple form, and of at most length 2).

Every CF grammar can (through the application of certain transformations which preserve weak equivalence but may greatly lengthen the grammar) be brought into CNF form (which is what you would expect from a normal form). Every ϵ-free CF grammar can be transformed to GNF. In certain argumentations, it may be helpful to transform a CF grammar first into one of these normal forms.

In the next chapter we shall discuss such grammar transformations.

3.3 Parsing

Our task is usually not to generate a sentence of a language (**synthesis**) but to determine whether a particular sentence belongs to a given language (**analysis**).

A **parser** for a language L is an algorithm which determines whether a sentence ω is a sentence of L and, if so, determines the structure of this sentence. In other words, a parser is a program which constructs a parse tree for ω, if ω is an element of L, or produces an error message if ω is not an element of L.

In trying to parse a sentence, the input sentence will usually be read from left to right. Of course we could also read from right to left, but this is not essentially different (why?).

There exist two principally different methods to build a parse tree for a given sentence: Top-Down or Bottom-Up. As an example to show the difference we take a simplified form of G2:

```
E: E, "+", E ; "(", E, ")" ; id.
```

For the sentence `id + (id + id)` we want to derive the parse tree:

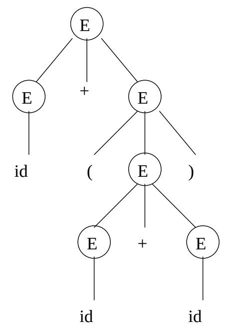

Figure 3.1: Parse tree for the input `id + (id + id)`

3.3.1 Order of derivation

A **parse tree** is a graphical representation of the derivation process. It shows the hierarchical syntactic structure of sentences implied by the grammar and abstracts from the order of the derivations.

Many different orders of derivation are possible. Two extremes:

- **left-most derivation**: always replace the left-most nonterminal symbol in a sentential form.
- **right-most derivation**: always replace the right-most nonterminal.

The parse trees do not depend on the derivation order (a **confluence** property of CF rewriting), so that we can take a specific order (e.g. left-most) without loss of generality.

3.3.2 Ambiguity

Every unambiguous sentence has a unique parse tree, whereas an ambiguous sentence has more than one. A grammar is **ambiguous** if it describes (at least) one ambiguous sentence. The corresponding automaton is then non-deterministic.

Since programming languages usually posess an unambiguous grammar, parsing techniques in Computer Science are mostly concerned with deterministic parsers. In natural languages however, ambiguity is a much more important issue. As a consequence, they create different problems and demand different parsers.

A language is ambiguous if it is impossible to give a deterministic (i.e. unambiguous) grammar for it. A grammar is ambiguous if its language contains one or more ambiguous sentences. A sentence is ambiguous if, according to the grammar, it has two or more different parse trees. As an example, the following grammar for arithmetic expressions (**G2**) is ambiguous:

```
expr:
   expr, "+", expr;
   "-", expr;
   "(", expr, ")";
   id.
```

As an example, $\mathsf{id} + \mathsf{id} + \mathsf{id}$ has two distinct left-most derivations:

$$
\begin{array}{lll}
\mathsf{E} & \rightarrow & \mathsf{E} + \mathsf{E} \\
 & \Rightarrow & \mathsf{id} + \mathsf{E} \\
 & \Rightarrow & \mathsf{id} + \mathsf{E} + \mathsf{E} \\
 & \Rightarrow & \mathsf{id} + \mathsf{id} + \mathsf{E} \\
 & \Rightarrow & \mathsf{id} + \mathsf{id} + \mathsf{id}
\end{array}
\qquad
\begin{array}{lll}
\mathsf{E} & \rightarrow & \mathsf{E} + \mathsf{E} \\
 & \Rightarrow & \mathsf{E} + \mathsf{E} + \mathsf{E} \\
 & \Rightarrow & \mathsf{id} + \mathsf{E} + \mathsf{E} \\
 & \Rightarrow & \mathsf{id} + \mathsf{id} + \mathsf{E} \\
 & \Rightarrow & \mathsf{id} + \mathsf{id} + \mathsf{id}
\end{array}
$$

In this case the grammar can be rewritten into a weakly equivalent but unambiguous form (**G2'**) by making it singly recursive:

```
expr:
  term, "+", expr;
  term.

term:
  "-", expr;
  "(", expr, ")";
  id.
```


This is possible because the language of arithmetic expressions posesses (also) a deterministic grammar. Two memorable grammars for languages which are **inherently ambiguous**:

- The **Fibonacci grammar** describing the set of all ways to partition a string of a's into parts of length one or two.

```
fib:
   "a";
   "a", "a";
   "a", fib;
   "a", "a", fib.
```

- The **partition grammar** describing the set of all ways to partition a string of a's into nonempty substrings.

```
part:
   "a";
   "a", part;
   part, part.
```

3.3.3 Bottom up parsing

A nondeterministic algorithm for **Bottom-Up parsing** is easy to formulate:

> Start with the sentence given. Repeatedly try to find the left-most substring in the sentence which matches a right-hand side of a production and then to apply that production by replacing the matched string by the left-hand side of the production (a **reduction**) until the start symbol has been derived.

The algorithm is nondeterministic because more than one rule may be applicable at any point.

$$
\begin{array}{lll}
\mathsf{id} + (\mathsf{id}) + \mathsf{id} & & \\
 & \Leftarrow & \{\mathsf{id} \leftarrow \mathsf{E}\} \\
\mathsf{E} + (\mathsf{id}) + \mathsf{id} & & \\
 & \Leftarrow & \{\mathsf{id} \leftarrow \mathsf{E}\} \\
\mathsf{E} + (\mathsf{E}) + \mathsf{id} & & \\
 & \Leftarrow & \{(\mathsf{E}) \leftarrow \mathsf{E}\} \\
\mathsf{E} + \mathsf{E} + \mathsf{id} & & \\
 & \Leftarrow & \{\mathsf{E} + \mathsf{E} \leftarrow \mathsf{E}\} \\
\mathsf{E} + \mathsf{id} & & \\
 & \Leftarrow & \{\mathsf{id} \leftarrow \mathsf{E}\} \\
\mathsf{E} + \mathsf{E} & & \\
 & \Leftarrow & \{\mathsf{E} + \mathsf{E} \leftarrow \mathsf{E}\} \\
\mathsf{E} & &
\end{array}
$$

If we look at the sequence of productions applied, we see that we have obtained a **right-most derivation** in reverse (always the right-most nonterminal has been replaced first).

3.3.4 Top down parsing

A nondeterministic algorithm for **Top-Down parsing**:

> Start with the startsymbol and repeatedly replace the left-most nonterminal by one of its direct production until the given sentence has been derived.

$$
\begin{array}{lll}
\mathsf{E} & & \\
 & \Rightarrow & \{\mathsf{E} \rightarrow \mathsf{E} + \mathsf{E}\} \\
\mathsf{E} + \mathsf{E} & & \\
 & \Rightarrow & \{\mathsf{E} \rightarrow \mathsf{E} + \mathsf{E}\} \\
\mathsf{E} + \mathsf{E} + \mathsf{E} & & \\
 & \Rightarrow & \{\mathsf{E} \rightarrow \mathsf{id}\} \\
\mathsf{id} + \mathsf{E} + \mathsf{E} & & \\
 & \Rightarrow & \{\mathsf{E} \rightarrow (\mathsf{E})\} \\
\mathsf{id} + (\mathsf{E}) + \mathsf{E} & & \\
 & \Rightarrow & \{\mathsf{E} \rightarrow \mathsf{id}\} \\
\mathsf{id} + (\mathsf{id}) + \mathsf{E} & & \\
 & \Rightarrow & \{\mathsf{E} \rightarrow \mathsf{id}\} \\
\mathsf{id} + (\mathsf{id}) + \mathsf{id} & &
\end{array}
$$

We have obtained a **left-most derivation**, because at every point the left-most nonterminal was replaced. The nondeterministic nature of the parsing algorithm should be obvious.

3.4 Exercises

1. Give a CF grammar for each of the languages introduced in 3.1.2.
2. Define two CF grammars G_1 and G_2 such that $\mathcal{L}(G_1) \bigcap \mathcal{L}(G_2) \; = \; a^n b^n c^n$.
3. Design an algorithm to transform an ϵ-free CFG into Greibach Normal Form. Prove that it terminates.
4. Design an algorithm to transform a given CFG into Chomsky Normal Form.
5. Give (in L1vwg form) a CF grammar for CF grammars in L1vwg form.
6. Derive from the Fibonacci grammar how many parse trees are given as a function of n, the length of the input a^n.
7. Derive from the partition grammar how many parse trees are given as a function of n, the length of the input a^n.

Chapter 4

ABOUT PARSING

In this chapter we consider methods for obtaining a recognizer or parser rom a given CF grammar (automatically or by hand) and the relevant properties of CF grammars. We emphasize the construction of deterministic recognizers, because this gives a good insight in the grammar properties that determine the necessity for nondeterministic recognizers.

Many examples in this chapter will be based on variants of the following CF grammar:

```
sentence:
  subject, verb, object.

subject:
  noun part.

noun part:
  "the cat"; "the dog"; "the man"; "his wife".

verb:
  "bit"; "saw".

object:
  noun part.
```

4.1 Recognizers and parsers

In this chapter we will focus on recognizers rather than parsers, because the same problems apply. A parser is a straightforward extension of a recognizer, building a syntax tree for a succesfully recognized sentence. In chapter 6 it will be made clear how this can be done using transduction.

> A **recognizer** for a nonterminal `n` is an algorithm that computes
>
> $$\exists g, w \in {V_T}^* (n \; w \Rightarrow^+ g \; w \wedge g \; w \; \# = input)$$

Notice that in general a prefix of the input is recognized (not necessarily the whole input). By adding an endmarker # to both the input and the language to be recognized, recognition of the complete input can be enforced.

Consider a CF rule of the form

```
n :
  alt$_1$;
  alt$_2$;
```

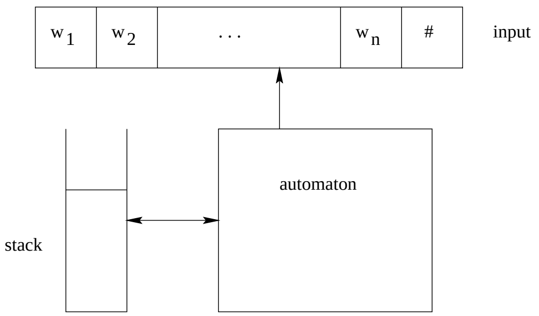

Figure 4.1: 1PDA

```
...
alt$_n$.
```

Let $input = w_1, w_2, ..., w_n, \#$ where n is its length and # is an endmarker.

A CF recognizer is to be implemented as **1PDA**, an automaton with one stack (its **state stack**) and a list of input symbols.

As we saw in the previous chapter, it may work either by applying reductions (Bottom-Up) or by applying productions (Top-Down). In order to be deterministic, the recognizer must be able to choose, on the basis of its state and the (remaining) input precisely which alternative is to be applied next. A recognizer which bases its choice on the first l symbols of the remaining input is said to use a **lookahead** (of size) l.

In the next sections we analyse the properties that a CF grammar must possess in order to be suitable for deterministic recognition. It will become obvious that realistic grammars for natural languages can not be deterministic, unlike those for programming languages.

4.2 Grammar properties

4.2.1 Well-formedness

We shall impose on the grammars certain **well-formedness rules**, which obviously do not restrict their expressive power but prevent undefined situations.

A **well-formed grammar** does not contain

- non-productive members ("dead ends")
- superfluous rules, or
- identical alternatives for the same rule.

In the sequel we shall consider only well-formed CF grammars.

4.2.2 Empty alternative

An empty alternative is always there to recognize. It turns out to be rather hard for recognizers to decide when it is applicable to recognize an empty alternative. More in general, the same holds for any alternative or rule that may produce the empty string.

Rules with an ϵ-producing alternative are called **epsilon rules** or **nonfalse rules**.

4.2.3 Left-recursion

A noun part like

`the man`

might, in a more ambitious grammar, be followed by any number of preposition phrases:

`the man with the hammer`

`the man with the hammer in the green shirt`

This fact can be expressed in the rule for `noun part` by means of a **left recursion**:

```
noun part :
   noun part, prep phrase;
  "the cat"; "the dog"; "the man"; "his wife".
```

A rule may also be **right recursive**, like

```
substantive part :
  adjective, substantive part ;
  substantive.
```

which expresses the fact that a substantive may be preceded by any number of adjectives.

A nonterminal a is called **left recursive** if there exists a string ω in V^* such that $a \Rightarrow^+ a\omega$. A grammar is left-recursive if it contains one or more left-recursive nonterminals.

Left-recursion is decideable and removeable (see section 5.1.2). A deterministic recognizer has to deal correctly and efficiently with both left- and right-recursive rules (e.g. without getting into endless loops).

4.2.4 Prefix sharing

Two alternatives of one rule have a common prefix **prefix sharing** if there exists a string of symbols that can be the beginning of a production of either alternative. A trivial example from the syntax of Lalgol60:

```
program:
  compound statement; block.
compound statement:
  "begin", statements, "end".
block:
  "begin", declarations, statements, "end".
```

[The terminal productions of] two alternatives a_1 and a_2 for a nonterminal n are said to share the **prefix** g if there are nonempty strings g, ω_1 and ω_2 in V^*, such that
both $n \Rightarrow a_1 \Rightarrow^* g\omega_1$
and $n \Rightarrow a_2 \Rightarrow^* g\omega_2$.

For arbitrarily sized prefixes, this is undecideable, but it is decideable for prefixes of bounded size, i.e. for $|g| < k$ for any given k.

Prefix sharing is also misleading to the human reader. English grammarians like to give examples of what they call **garden-path sentences**, such as

`the horse raced past the barn fell`

in which the reader is sent up the (wrong) garden path.

4.2.5 Local ambiguity

For deterministic recognizers, a complication (**local ambiguity**) arizes if a terminal production of one alternative of a rule is a prefix of a terminal production of another. We modify one rule of the example grammar:

```
object:   noun part ;
          noun part, "and", object.
```

Obviously, the choice between these two alternatives is not deterministic: whenever the second one applies the first one also applies.

`the dog bit the man and the cat`

Using a heuristic like "try the most complicated alternative first" does not help: to begin with, it is undecideable in general which alternative is "the most complicated". And even if it were, this would not solve the problem. To show this, we extend the grammar further

```
sentence :
  basicsentence ;
  basicsentence, "and", sentence.
basicsentence :
  noun part, verb, object.
```

Now the strategy to put the most complicated alternative first does not work any more for sentences like

`the dog bit the man and the cat bit his wife`

because the recognizer for `object` can not decide locally that in this case the less complicated alternative has to be taken.

A nonterminal n with alternatives $a1$ and $a2$ is said to have a **local ambiguity** if there exist nonempty strings g in V_t^* and ω in V^* such that
both $n \Rightarrow a1 \Rightarrow^* g$
and $n \Rightarrow a2 \Rightarrow^* g\omega$.

Again this property is undecideable.

4.2.6 Ambiguity

We extend the previous grammar still further

```
basicsentence: subject, verb, object.
subject: noun part;
         noun part, "and", subject.
```

Now the syntax describes obviously ambiguous sentences like

`the man bit the cat and the dog and his wife bit the man`

with the analyses

`(the man bit the cat) and (the dog and his wife bit the man)`
`(the man bit the cat and the dog) and (his wife bit the man)`

A nonterminal n with alternatives $a1$ and $a2$ is said to be ambiguous (has an **ambiguity**) if there exists a nonempty string g in V_t^* such that
both $n \Rightarrow a1 \Rightarrow^* g$
and $n \Rightarrow a2 \Rightarrow^* g$.

Ambiguity of grammars is again undecideable. Some interesting examples of ambiguous grammars were already given in the previous chapter (see 3.3.2). A useful lemma:

Any nonterminal that is both left- and right-recursive is ambiguous.

This is a sufficient but certainly not a necessary condition.

4.2.7 The LL(1) property

The intuition behind LL(1) grammars is simple: in order for the recognizer for some nonterminal n to be deterministic, the alternatives of n may not have even a prefix of size one in common.

In order to express this formally, the notion of starter set is introduced.

The **starter set** of the nonterminal n is defined as

$$S(n) = \{x \in V_t \mid n \Rightarrow^+ x\omega, \omega \in V^*\}$$

The starter set $S(a)$ of a sentential form $a = a_1, a_2, ..., a_n$ can be computed recursively by the following algorithm:

- $n = 0 \rightarrow \emptyset$
- $a_1 \in V_t \rightarrow a_1$
- $a_1 \in V_n \rightarrow nonfalse(a_1) \rightarrow S(a_1) \cup S(a_2, \ldots, a_n), otherwise S(a_1)$.

The **follower set** $F(n)$ of a nonterminal symbol n is defined as

$$F(n) \; = \; \{x \in V_t | \Sigma \Rightarrow^* \alpha n x \omega, \alpha, \omega \in V^*\}$$

An alternative formulation:

$$F(n) \; = \; \{x | \Sigma \Rightarrow^* \alpha n \delta, \alpha, \delta \in V^* \, x \in S(\delta)\}$$

A grammar is said to be of type **LL(1)** iff for every nonterminal n

1. the starter sets of the alternatives of n are disjoint
2. at most one alternative of n is nonfalse
3. if n is nonfalse then the starter set $S(n)$ has no element in common with the follower set $F(n)$.

It is decideable whether a grammar is LL(1). A language is of type LL(1) if an LL(1) grammar for it can be found.

A grammar that is LL(1) is free of left-recursion, prefix sharing, local ambiguity and ambiguity. We shall show this for direct left-recursion.

If the rule for n is directly left-recursive and not vicious it has at least two alternatives, a left-recursive alternative $alt1$ (starting with n) and another alternative $alt2$. All the starters of n are also starters for $alt1$, including the starters for $alt2$. Therefore

$$S(alt2) \cap S(alt1) \neq \emptyset$$

so the grammar is not LL(1).

The acronym LL(1) comes from

- Left-to-right parsing,
- using Leftmost productions
- with lookahead 1.

4.3 Deterministic recognizers

In this section we discuss various types of deterministic recognizers. The first three are of the Top-Down type, the last is Bottom-Up.

4.3.1 Recursive descent

In any programming language with recursion it is a simple matter to write a **recursive descent** (**RD**) recognizer. For each nonterminal of the grammar a corresponding recognition procedure is constructed. For recognizing terminal symbols from the input, a special recognition primitive is called:

```
BOOL PROC is (SYMB s):
  IF input[input pointer] = s
  THEN input pointer +:= 1; true
  ELSE false FI
ENDPROC is;
```

A RD recognizer for the example grammar, written in Lelan, should make the principle of RD recognition clear.

```
BOOL PROC sentence:
  IF subject
  THEN IF verb
       THEN object
       ELSE FALSE
  ELSE FALSE
  FI
ENDPROC sentence;

BOOL PROC subject:
  noun part
ENDPROC subject;

BOOL PROC noun part:
  IF is("the cat") THEN TRUE
  ELIF is("the dog") THEN TRUE
  ELIF is("the man") THEN TRUE
  ELSE is("his wife") FI
ENDPROC noun part;

BOOL PROC verb:
  IF is("bit") THEN TRUE
  ELSE is("saw") FI
ENDPROC verb;

BOOL PROC object:
  noun part
ENDPROC object;
```

Note that in a recursive descent recognizer or parser, alternatives are tried in textual order, whereas for CF grammars the ordering of alternatives carries no meaning. An eventual empty (or non-false) alternative must therefore come last (why?).

Recognizers and parsers in the recursive descent style are easy to construct without the aid of a specialized parser generator, and are quite adequate as long as an LL(1) CFG

is available.

4.3.2 LL(1) recognizers

The original LL(1) recognizers (e.g. Foster [FOS67, GRI74]) are slightly different from LL(1) parsers. They make use of the director set, which combines the information from the starter set and the follower set.

The **director set** $D(n, \alpha)$ for an alternative α of the nonterminal **n** is defined as

$$nonfalse(n) \rightarrow S(\alpha) \cup F(n), otherwise \rightarrow S(\alpha)$$

For a rule of the form

```
nont: alt$_1$;
      alt$_2$;
       ...
      alt$_n$.
```

the corresponding recognizer has the form

```
PROC nont:
   IF   next inputsymbol IN $D (nont, alt_1)$
   THEN alt$_1$
   ELIF next inputsymbol IN $D (nont, alt$_2)$
   THEN alt$_2$
    ...
   ELIF next inputsymbol IN $D (nont, alt$_n)$
   THEN alt$_n$
   ELSE fail
   FI
ENDPROC;
```

If the number of different terminal symbols is low, this can be implemented even more efficiently than RD, using a switch. This is the original form of LL(1) parsers.

4.3.3 Partial backup

A possible solution to the problem of prefix sharing can be found by using backtracking. Assuming the primitives

`remember inputpointer (INT>)`	save current value of inputpointer
`restore inputpointer (>INT)`	restore old value of inputpointer

This idea can be expressed in Lcdl3 as follows:

```
PREDICATE is a program :
    remember inputpointer (P),
      (is compound statement;
       restore inputpointer (P), is block;
       restore inputpointer (P), -).

PREDICATE is compound statement :
    is ("begin"), is statements, is ("end").

PREDICATE is block :
    is ("begin"), is declarations, is statements,
```

```
    is ("end").
```

which can obviously cope with shared prefixes. This form of parser, reinvented by many people, we shall call a **partial backup** parser (partial in the sense that the backtrack is always confined to one alternative).
Upon closer consideration, it turns out that the partial backtrack parsing method is practically useless:

- the class of grammars is only marginally wider than the grammars without prefix sharing, since the method can not cope with local ambiguity
- its applicability is not decideable
- the efficiency may be terrible, due to exponential behaviour.

4.3.4 LR(1) parsing

The principal algorithm for bottom-up parsing of deterministic grammars is the LR(1) parsing method, which is much more widespread than LL(1) parsing, especially in the US. The reason for this popularity is that LR(1) grammars are a wider subset of deterministic grammars than LL(k) (for any k). They can deal with prefix sharing for prefixes up to k symbols long, as well as local ambiguity.

The term **LR(k)** [KNU65] is shorthand for a parser which performs left-to-right parsing building the right-most derivation in reverse (i.e. bottom-up) using at most k terminal symbols lookahead.

LR parsers are members of the class of **shift-reduce parsers** which are a very general type of bottom-up parser. All shift-reduce parsers incorporate a stack for holding constituents as they are built during the parse and have a **shift-reduce table** of states and actions for guiding the parser. This table contains two types of actions: the **shift operation**, which transfers the next word from the input buffer onto the stack, and the **reduce operation**, which replaces several elements on the top of the stack with a new element. See sections WG5.3.3 (Bottom-Up Analysis and LR(k) Grammars) and WG7.3 (LR parsers) of Waite and Goos.

Two variants of LR(1) parsing are frequently used in practice: SLR(1) and LALR(1) parsing, described in WG7.3. Since the construction of such parsers is by no means as intuitively straightforward as the construction of recursive descent parsers, a suitable parser generator has to be used (e.g. Lyacc in conjunction with Llex).

Parsing methods in general (both deterministic and nondeterministic) are treated in-depth in the Parseermethoden course, including the LR methods.

4.4 Exercises

1. Design al algorithm to decide which nonterminals in a given CFG produce empty. Prove its termination.
2. Design al algorithm to decide which nonterminals in a given CFG are left-recursive. Prove its termination.
3. Design an algorithm to compute the follower set of a nonterminal in a given CFG. Prove that it terminates.

Chapter 5

GRAMMAR TRANSFORMATIONS

As we have seen, RD parsers are easy to construct by hand and intuitively very appealing, but they work only for LL(1) grammars. LL (1) Grammars are very restrictive:

- no ambiguity allowed
- no local ambiguity allowed
- no prefix sharing allowed
- no left-recursion allowed
- empty alternative must come last.

Their restrictiveness is impractical when experimenting with a grammar, and the first three restrictions are hard to judge intuitively and impossible to decide. This has led many people to abandoning RD parsing in favour of a more powerful parsing method, e.g. from the LR family.

We will go a different way: In this section we investigate how to extend the class of grammars suitable for deterministic parsing, by attempting to transform non-LL(1) grammars into LL(1) form.

Assume that the language we want to describe possesses an LL(1) grammar, can we find it starting from a non-LL(1) grammar? The question is undecideable, but as long as the grammar is not ambiguous we can of course try to "improve" the grammar, by removing LL(1) violations until we hopefully arrive at an LL(1) grammar.

5.1 The transformations

As a preparation, make sure that the grammar is well-formed. Determine the nonfalse nonterminals. Compute $S(n)$ and $F(n)$ (the latter is only necessary for nonfalse n).

5.1.1 Substitution

The simplest operation on a grammar is the **substitution** of a nonterminal `x`, occurring as a member of some rule, by the right-hand-side of its definition (**unfolding**) or vice versa (**folding**).

```
x: y, z; d.
y: a; b; .
```

Unfolding y:

```
x: a, z; b, z; z; d.
```

Unfolding may somewhat improve the efficiency of a RD recognizer, but it is used most often as a preparation for left-factorizations, or in order to remove nonfalse rules.

5.1.2 Left-recursion removal

Consider the following rule which shows direct left-recursion.

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

Consider also the language produced:

$$\begin{aligned} L(a) &= \{c, cb, cbb, \ldots\} \\ &= cb^* \\ &= ca' \end{aligned}$$

where $L(a') = b^*$. This observation suggests rewriting the rule to

```
a : c, a'.
a': b, a';  .
```

This is the basic transformation for **left-recursion removal**. By the argument given above, the two rules are weakly equivalent.

Observe that the example shows a remarkable similarity to the recursive algebraic equation

$$a = a * b + c$$

which can be algebraically rewritten into

$$\begin{aligned} a &= c + c * b + c * b * b + \ldots \\ &= c * z \end{aligned}$$

where

$$\begin{aligned} z &= 1 + b + b * b + \ldots \\ &= 1 + b * z \end{aligned}$$

The transformation is in fact language preserving because

- concatenation is an associative operation (like the multiplication in algebra)
- choice is associative and commutative (like the addition) with $\oslash$ as identity element.

5.1.2.1 Multiple leftrecursion

In [ACC74], pg. 70ff Mike Griffiths has given an interesting generalization of this transformation, suitable for dealing with multiple left-recursion in a uniform fashion. The basic idea is to consider vectors and matrices of symbols rather than single (scalar) symbols. Consider the pair of mutually recursive rules

```
a1 : a1, b11; a2, b21; c1.
a2 : a1, b12; a2, b22; c2.
```


In matrix/vector notation, let

$$\begin{aligned} a &= \begin{bmatrix} a1 & a2 \end{bmatrix} \\ B &= \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix} \\ c &= \begin{bmatrix} c1 & 2 \end{bmatrix} \end{aligned}$$

The original set of rules corresponds to the matrix equation

$$a = aB + c$$

We now apply the left-recursion removal transformation, turning this into

$$a = cX$$

$$X = I + bX$$

where I is unit matrix of (in this case) size 2.

$$I = \begin{pmatrix} \epsilon & & \ldots & \\ & \epsilon & \ldots & \\ \ldots & & & \\ & & \ldots & \epsilon \end{pmatrix}$$

Introducing new nonterminals

$$X = \begin{bmatrix} X11 & X12 \\ X21 & X22 \end{bmatrix}$$

we can write

```
a1 : c1, X11 ; c2, X21 .
a2 : c1, X12 ; c2, X22 .
```

with

```
X11 : b11, X11 ; b12, X21 ; .
X21 : b21, X11 ; b12, X21   .
X12 : b11, X12 ; b12, X22   .
X22 : b21, X12 ; b22, X22 ; .
```

Two mutually recursive rules have been expanded to 6 non-leftrecursive rules.

Any CF grammar with left-recursion can be transformed by this technique, the **Rosencrantz transformation**, into a weakly equivalent CF grammar without left-recursion, but in removing the left-recursion the grammar may grow quadratically. That is why language designers prefer to avoid left-recursion.

Another problem is that, even though the languages are weakly equivalent, the transformed grammar will produce different parse trees than the original grammar (right-recursive instead of left-recursive). Since the static semantics is usually realized by the insertion of static semantic actions at appropriate places in the grammar, it takes some care to ensure that the transformation leaves the semantics of the program unchanged.

5.1.2.2 An example

Consider the following two-level parser for expressions.

```
PREDICATE is term (VAL>):
  is factor (VAL), term tail (VAL).

ACTION term tail (>VAL>):
  is times,
      (is factor (OPD), mul by (VAL,OPD), term tail (VAL);
       error + "Incorrect expr");
  is div,
      (is factor (OPD), div by (VAL, OPD), term tail (VAL);
       error + "Incorrect expr");
  +.
```

Notice that this form can be obtained by transformation from the leftrecursive form (not defect free)

```
PREDICATE is term (VAL>):
  is term (VAL),
      (is times, is factor (OPD),
             mult by (VAL, OPD);
       is div, is factor (OPD),
             div by (VAL, OPD);
       );
  is factor (VAL).
```

according to the schema

```
a: a, (b; c); d.
```

Introducing a new rule for `b; c` and folding:

```
a: a, bc; d
bc: b; c.
```

Leftfactorization:

```
a: d, at.
at: bc, at; .
```

Unfolding `bc`:

```
a: d, at.
at: b, at; c, at; .
```

The interesting observation is that these transformations have been performed without inverting the flow of information (left to right). This is not a trick but a method, which can be performed automatically by an LL(1) parser generator.

5.1.3 Left-Factorization

Consider the rule

```
q : a, b; a, c; d.
```

The first two alternatives both start with `a`, so that they are not disjoint. We may however factor out this member, introducing a new rule:

```
q : a, q'; d.
q' : b; c.
```

or, using brackets to indicate grouping


```
q : a, (b; c); d.
```

Factoring a common prefix from two or more alternatives in a rule (**direct left-factorization**) as describeld in AHUL is simple. The more general case of **deep left-factorization**, factoring a common nonterminal starter from two or more alternatives, is harder as the following examples show.

5.1.3.1 Some examples

1. From Lalgol60

```
program:
    compound statement; block.
  compound statement:
    "begin", statements, "end".
  block:
    "begin", declarations, statements, "end".
```

   Direct leftfactorization

```
  program:
    "begin", programtail.
  programtail:
    statements, "end";
    declarations, statements, "end".
```

2. Take the Fibonacci grammar

```
S : "a" ;
    "a", "a" ;
    "a", S ;
    "a", "a", S.
```

   Taking out the left-factor `"a"` (using brackets) gives

```
S : "a", ("a" ; S ; "a", S; ).
```

   Leftfactorizing once more leads to

```
S : "a", ("a", (S ; ); S; ).
```

   Of course the example is not deterministic, but the reader will understand that in a straightforward nondeterministic parser the transformation improves efficiency and removes backtracking.

3. A complicated example:

```
a: b, c; d, e.
b: B, x; Z.
d: B, y.
```

   Substitution of `b` and `d`

```
a: B, x, c; Z, c; B, y, e.
```

Leftfactoring out B

```
a: B, n; Z, c.
n: x, c; y, e.
```

4. The following example, taken from Alblas [?], shows that the order of transformations is important.

```
a: b, c; d, e.
b: B, x; Y.
d: b, Z.
```

Unfolding `b` and `d`

```
a: B, x, c; Y, c; b, Z, e.
```

We must again unfold `b`

```
a: B, x, c; Y, c; B, x, Z, e; Y, Z, e.
```

Leftfactoring gives

```
a: B, n1; Y, n2.
n1: x, c; x, Z e.
n2: c; Z, e.
```

Leftfactoring n1

```
n1: x, n3.
n3: c; Z, e.
```

But now we notice that `n3` is the same as `n2`!

```
a: B, n1; Y, n2.
n1: x, n2.
n2: c; Z, e.
```

It would have been better to substitute originally for `d` only

```
a: b, c; b, Z, e.
```

Then leftfactoring gives

```
a : b, n.
n : c; Z, e.
```

Simpler, and more efficient.

5.1.4 Ordering the rules

In the example

```
a : b, c; d, e.
b : B, x; Y.
d : b, Z.
```

we should first substitute for d, because b is a starter of d.

We define a partial ordering among the nonterminals by giving to each nonterminal a **rank**

$$Rank : V_n \to INT$$

such that

$$B \in S(D) \to Rank(B) < Rank(D)$$

This order always exists if there is no left-recursion, since in this case

$$B \in S(D) \to D \notin S(B)$$

$$A \in S(B), B \in S(C) \to A \in S(C)$$

For a grammar without leftrecursion, the rank of the nonterminals can be computed as follows:

1. For all $x \in Vn$ set $r(x)$ to 0
2. $n := 1$
3. repeat
   (a) find some nonterminal y which has no nonterminal starter x with r(x) = 0;
   (b) r(y) := n; n +:= 1

   until all nonterminals have a rank r > 0

This is an optimal order for substitution, closure computation and proofs.

5.1.5 Problems

Given a CF grammar, the given grammar may not be LL(1) even though the language is, i.e. even though the language possesses some LL(1) grammar. Is it possible to obtain an LL(1) grammar for it by systematic application of substitution, left-factoring and left-recursion removal?

NO. The process may loop.

Example (Alblas)

```
a : b; c.
b : f, b; d.
c : f, c; e.
```

Substitution of `b` and `c` in `a`

```
a : f, b; f, c; d; e.
b : f, b; d.
c : f, c; e.
```

Factorizing `f` from `a`

```
a : f, n; d; e.
n : b; c.
b : f, b; d.
c : f, c; e.
```

We are faced with the same problem! On the other hand, there exists an equivalent LL(1) grammar, which can be found by first factoring the common prefix f^* from both `b` and `c`.

```
a : x, y.
x : f, x;.
y : d; e.
```

5.1.6 Non LL(1) grammars

A grammar may be deterministic (i.e. non-ambiguous) and yet not of type LL(1). Of course it is possible to consider a larger lookahead than one symbol, e.g. k symbols with $k > 1$ (**LL(k) grammars**).

The following grammar has a (hidden) prefix problem.

```
expr:
   bool expr; arith expr.
bool expr:
   bool term, rest of bool.
rest of bool:
   or symbol, false symbol;
bool term:
   true symbol; false symbol;
   open symbol, bool expr, close symbol.

arith expr:
   arith term, rest of arith.
rest of arith:
   plus symbol, arith term;.
arith term:
   zero symbol; one symbol;
   open symbol, arith expr, close symbol.
```

Examples of expressions generated:

```
true     true OR false     (true)      ((((true))))
  0         0 + 1            (0)          ((((0))))
```

The grammar is neither ambiguous nor left-recursive. Even so it is not LL(k) for any value of k. It is however LR(1).

Languages whose syntax has not expressly been designed to facilitate top-down parsing may need look-ahead.

- LL(2)?

`begin l m`	lexical problem
`begin l :`	label-identifier
`begin l of t`	field-selector
`begin l +`	identifier

- According to A68 unrevised syntax, each of the following constructs can begin with an arbitrary number of `LONG`s:

`LONG LONG LONG REAL`	declarer
`LONG LONG LONG 1`	denotation
`LONG LONG LONG PERSON`	mode-indication
`LONG LONG LONG +`	operator-indication

  Use a buffer? A counter? A circular buffer?

- Unbounded look-ahead over array bounds:

Common prefix	followed by	construct
`BEGIN [1 :  3] REAL`	`x:= (1, 2, 3)`	object declaration
`BEGIN [1 :  3] REAL`	`+ (1, 2, 3)`	formula

  Of course there can be any number of bounds, which can also be arbitrarily complex expressions.

For ambiguous grammars, completely different recognition techniques have to be used. More about this subject in the course on Parseermethoden.

5.2 Dealing with grammars that are not LL(1)

The syntax of any programming language will tend to be nearly everywhere deterministic, for the simple reason that human readers are not good at dealing with (local) ambiguity and tend to read from left to right. Even in syntactically very rich languages, only in a few places look-ahead is necessary, and even there the look-ahead is usually bounded (and much shorter than the whole program). Therefore the extra effort (for backtracking) is acceptable.

5.2.1 Look ahead of one symbol

In order to deal with LL(2) grammars, we have to introduce an extra **look-ahead** of one symbol. Idea: have two symbols in the buffer

```
---------    ---------
| curr  |    | next   |
| symb  |    | symb   |
---------    ---------
```

```
MODULE lexico = CURR followed by NEXT.

CURR, NEXT = SYMB.

PRED symbol is (>SYMB):
    [SYMB = CURR], next symbol.

TEST followed by (>SYMB):
    [SYMB = NEXT].

ACTION next symbol:
    [NEXT -> CURR], read symb (NEXT).

PRELUDE read symb (CURR), read symb (NEXT).
```

- sufficient, but not very nice due to the fact that following symbol has to be skipped explicitly

- initialization with a “dummy symbol” instead of always reading in two symbols as prelude.

5.2.2 One symbol back-up

Idea: we read ahead until we have gone (one symbol) too far, then back up. Therefore we may have one symbol in stock.

```
--------   ---------  ---------
| curr  |  | next  |  | in     |
| symb  |  |symb   |  | stock  |
--------   ---------  ---------
```

```
MODULE lexico = CURR followed by NEXT if INSTOCK.

CURR, NEXT = SYMB.

INSTOCK::none; one.

PRED symbol is ( > SYMB):
    [SYMB = CURR], next symbol.

ACTION next symbol:
    [INSTOCK -> one], [none -> INSTOCK], [NEXT -> CURR];
    read symb (CURR) .

ACTION back up ( > SYMB):
    has stock, error ("Backup over more than one symbol"),?;
    [one -> INSTOCK],[CURR -> NEXT], [SYMB -> CURR].

PRELUDE  [none -> INSTOCK], read symb (CURR).
```

5.2.3 For larger k

For dealing with look-ahead of length $k >> 1$ two approaches present themselves:

- to store the whole program in random access memory
- to store a window of at most k symbols (see fig. 5.2.3).

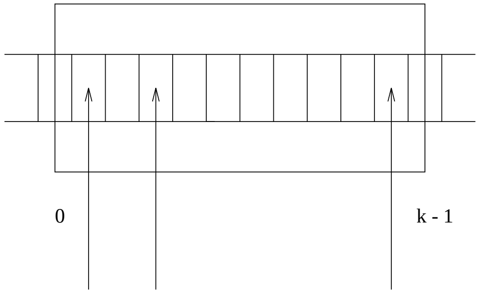

Figure 5.1: Window of length k

The overhead of looking ahead and then backtracking may be acceptable as long as it happens very infrequently and will usually not be over a large distance.

In the figures we use the following identifiers:

`pmax`	index of last symbol read
`pcurr`	index of current symbol
`prec`	index of last symbol recognized.

Rather than moving elements in the buffer to implement the sliding window it is customary to organize it as a circular buffer (fig 5.2.3).

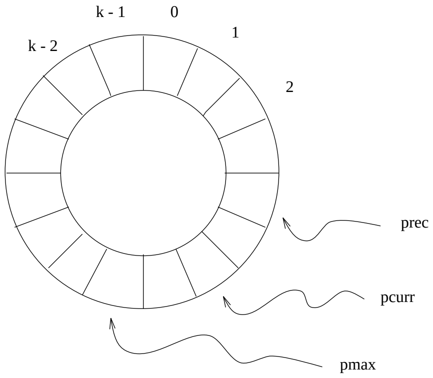

Figure 5.2: A circular buffer

For many students, unfortunately, this is a major programming exercise.

5.2.4 Conclusion: The special case LL(1)

There exist algorithms to check in $O(n^k + 1)$ whether a given grammar is LL(k) for a given k, where n is the sum of the lengths of the right-hand-sides of all productions (see [WAG84], table 7.3).

The special case $k = 1$ is the only case of practical importance, for a number of reasons:

- Usually, languages are designed so that they are LL(1) with the exception of at most a few productions.
- A parser handling more than one look-ahead symbol is inefficient to use, and only mildly more powerful than LL(1).
- Error reporting and recovery becomes more difficult with more than one symbol of look-ahead.

We have described a number of transformation rules to make grammars “more” LL(1). In practice these are quite effective.

5.3 Family relations

In this section we shall briefly discuss the inclusion relations that hold between grammar classes and between language classes for deterministic languages. It provides a refinement of the Chomsky hierarchy, which was introduced earlier.

5.3.1 Grammar classes

In the literature on formal languages various subclasses of the Phrase Structure Grammars are introduced, which are usually characterised by restrictions imposed on the form of production rules of the grammar. The guiding example is the Chomsky hierarchy:

Level	also known as	form of rules
Chomsky 0	Semi Thue system	$V^+ \Rightarrow V^*$
Chomsky 1	Context Sensitive (CS)	$V^+ \Rightarrow V^+$
Chomsky 2	Context Free (CF)	$V_n \Rightarrow V*$
Chomsky 3	Right-linear	$V \Rightarrow V_t$ or $V \Rightarrow V_tV_n$
idem	Left-linear	$V \Rightarrow V_t$ or $V \Rightarrow V_nV_t$
"Chomsky 4"	Finite Choice (FC)	$V_n \Rightarrow V_t$

The last line is an obvious downwards extension to Chomsky's hierarchy, which will be used in some of the following chapters. Right-linear and Left-linear grammars are both equivalent to regular expressions (but a mixture of the two forms of rules is not).

Not every restriction on the form of the rules introduces a separate class, as can be seen from two important special forms of Context Free grammar:

Greibach Normal Form (GNF) $V_n \Rightarrow V_tV^*$
Chomsky Normal Form (CNF) $V_n \Rightarrow V_t$ or $V_n \Rightarrow V_nV_n$ or $V_n \Rightarrow \epsilon$.

Similarly, restrictions like the LL(1) restriction define a certain class of CF grammars. We want to study the relationships between various grammar classes.

We say a grammar is of class T if all of its production rules are of the form required for that class. Under this definition, a given grammar may be of more than one class; all Phrase Structure Grammars are of class Chomsky 0.

We introduce a partial ordering $\subset$ on grammar classes, as follows:

- $G_1 \subseteq G_2 \iff \forall_g g \in G_1 \Rightarrow g \in G_2$
- $G_1 \equiv G_2 \iff G_1 \subseteq G_2 \wedge G_2 \subseteq G_1$
- $G_1 \subset G_2 \iff G_1 \subseteq G_2 \wedge \exists_{g \in G_2} \neg g \in G_1$.

As an example, the Chomsky hierarchy is strict, so
$FC \subset RE \subset CF \subset CS \subset Chomsky0$.

5.3.1.1 Deterministic grammars

The inclusion relations amongst deterministic grammar classes are as follows:

- $LL(1) \subset LL(2) \subset ... \subset LL$ (where LL stands for $Lim_{k\to\infty}LL(k)$
- $LR(0) \subset LR(1) \subset LR(2) \subset ... \subset LR$ (with a similar limit construction for LR)
- $LL(k) \subset LR(k)$ but not $LL(k) \subset LR(k-1)$ (there are some $LL(k)$ grammars which are not $LR(k-1)$)

There are some further deterministic classes described in literature (like $SLL(k)$, $SLR(k)$ and $LALR(k)$), see e.g. [Waite & Goos] or [Fischer & Leblanc]. They show similar inclusion relations.

5.3.2 Language classes

By a *language* we mean a set of strings over some alfabet. A language may be described different formalisms: by enumeration (in the case of a finite language), by set-construction involving regular expressions, by a predicate defining what subset of all strings over the alfabet is a sentence (e.g. $\{w|w = a^nb^{2n}, n \geq 0)$, or by a grammar.

We say a language is of class $\mathcal{T}$ if it possesses a grammar of type $\mathcal{T}$. By definition therefore, each grammar of class $\mathcal{T}$ describes a language of the same class. But any given language possesses arbitrarily many (weakly equivalent) grammars. There is decidedly more than one way to describe a given language. We say a language "is inherently of class $\mathcal{T}$" if it possesses a grammar of class $\mathcal{T}$ but not of a weaker grammar class.

5.3.3 Deterministic languages

Because of e.g. the existence of grammar transformations, the classification of languages is somewhat coarser than that for grammars:

- $LL(1) \subset LL(2) \subset ... \subset LL$
- $LR(0) \subset LR(1) = LR(2) = ... = LR$
- $LL \subset LR(1)$

The language class LR coincides with the class of all deterministic CF languages, stronger language classes are nondeterministic CF, of which the inherently ambiguous form a subset.