Grammar 2

語言的階層

(一) 型態0 文法（type 0 grammar）

1.特點：對於產生規則不作任何的限制；因此刪除產 生規則中的符號是容許的，而且中間過程產生的字串能增長或縮短；例如：產生規則aBBAA→aB中將符號AA刪除。

2.語言的類型：可縮短而且文意相關（contracting context-sensitive）。

3.產生規則的格式及限制：$\alpha \rightarrow \beta$ 其中 $\alpha, \beta \in (N \cup T)^*$ ；$\alpha \neq \lambda$

4.自動化機械（automation）的辨認：turning machine。

(二) 型態1文法（type 1 grammar）

1.特點：限制每一個產生規則中右邊的符號個數不得少於左邊的符號個數；
舉例說明型態1的文法如下：

$N = \{ A, B, \Sigma \}$　　$T = \{ a , b , c \}$

$P = \{ \Sigma \rightarrow Abc$

$Ab \rightarrow aAbB$

$Bb \rightarrow bB$

$Bc \rightarrow bcc$

$A \rightarrow a \}$

上述文法產生的字串形式為$a^n b^n c^n$，$n \geq 1$。

2.語言的類型：不可縮短而且文意相關（noncontracting context-sensitive）

3.產生規則的格式及限制：$\sigma\alpha\tau \rightarrow \sigma\beta\tau$

其中σ，$\tau \in$（N∪T）*；$\alpha,\beta \in$（N∪T）* —λ 而且 length(α)≤ length(β)。

4.自動化機械的辨認：非決定性線性限制的自動化機械（non-deterministic linear-bounded automata）。

(三) 型態2文法（type 2 grammar）

1.特點：限制產生規則的左邊只能是單一個非終端符號；因此產生規則的應用就與符號出現在本文的地方無關了；例如 BACKUS NORMAL FORM (BNF)即為型態2文法。

2.語言的類型：文意無關（context-free）。

3.產生規則的格式及限制：$A \rightarrow \beta$　其中$A \in N$； $\beta \in (N \cup T)^* - \lambda$

4.自動化機械的辨認：非決定性壓入式儲存的自動化機械（non-deterministic push-down storage automata）。

(四) 型態3文法（type 3 grammar）

1.特點：限制每一個步驟的產生規則中終端符號及非終端符號之個數；詳言之，他限制產生規則的左邊只能是單一個非終端符號，而產生規則的右邊之內容有下列兩種情形：

(1)非終端符號出現在非終端符號以外的所有符號的右邊，稱為右線性產生規則（right-linear production）。

(2)非終端符號出現在非終端以為的所有符號的左邊，稱為左線性產生規則（left-linear production）。

2.語言的類型：規則的或有限狀態（regular or finite – state）。

3.產生規則的格式及限制：

(1)右線性：$A \rightarrow aB$

$A \rightarrow a$

(2)左線性：$A \rightarrow Ba$

$A \rightarrow a$

其中 $a \in T$；A，$B \in N$

4.自動化機械的辨認：有限狀態的自動化機械（finite-state automata）。

圖　語言的四種型態

Type	Type of language and recognizing automation	Production form and restrictions
0	Contracting context-sensitive (post systems): Turing machines	$\alpha \rightarrow \beta$ $\alpha, \beta \in (N \cup T)^*$ ； $\alpha \neq \lambda$
1	Noncontracting context-sensitive: Non-deterministic linear-bounded automata	$\sigma\alpha\tau \rightarrow \sigma\beta\tau$ $\sigma, \tau \in (N \cup T)^*$；$\alpha, \beta \in (N \cup T)^* - \lambda$ length(α) ≤ length(β)
2	Context-free: non-deterministic push-down storage automata	$A \rightarrow \beta$ $\beta \in (N \cup T)^* - \lambda$ ； $A \in N$
3	Regular or finite-state: finite-state automata	Right-linear　　Left-linear $A \rightarrow aB$　　$A \rightarrow Ba$ $A \rightarrow a$　　$A \rightarrow a$ $a \in T$；A，$B \in N$

註：X*表示由集合X中有限個符號依各種可能方式相連接所形成的串列之集合。

語言的型態與產生該語言的文法型態是相關的，從上述的討論得：

(一)型態3是型態2的部分集合，型態2是型態1的部分集合，型態1是型 態0的部分集合。

(二)型態 i 的語言能由型態 i 的文法產生，但不能由型態 i+1的文法產生，其中 i等於0或1或2。

Classes of Grammars

Chomsky [1965] distinguished four classes of grammars. The most general class, the unrestricted grammars, is not phrase-structured, and may follow any conceivable set of rules. The other three classes are phrases-structured:

the context-sensitive, context-free, and right-linear grammars.

The most general phrase-structured class is the context-sensitive grammar. In this class, each production has the form:

$$x \rightarrow y$$

where x and y are members of $(N \cup \sum)$, x contains at least one member of N, and $|x| \leq |y|$. Note that the last requirement implies that y cannot be empty.

An example of a context-sensitive grammar is:

$$G1 = (\{S, B, C\}, \{a, b, c\}, P, S)$$

where the production P are:

1. $S \rightarrow aSBC$
2. $S \rightarrow abC$
3. $CB \rightarrow BC$
4. $bB \rightarrow bb$
5. $bC \rightarrow bc$
6. $cC \rightarrow cc$

Let us develop a set of replacements in this grammar. Because S is the designated starting, we look for a production with S as its left member. Either of the first two will do:

$$S \rightarrow aSBC$$

so that aSBC is a new string. In string aSBC, we can use only another S rule; let us choose the second one:

aSBC→ aabCBC

Here, the third or fifth rule may be chosen; let us choose the third:

aabCBC→aabBCC

Continuing, we find the following sequence of replacements:

aabbCC

aabbcC

aabbcc

We end up with all terminals, so this is the end of the possible replacements.

We could reach a string for which no production can apply. For example, in the string aabCBC, if we choose the fifth rule instead of the third, we obtain aabcBC, and we find that no rule can be applied to this string. The consequence of such a failure to obtain a terminal string is simply that we must try other possibilities until we find those that yield terminal strings.

Context-free Grammars

The next most general class of grammars is the one that we shall be studying in most of this text-the context-free grammars. In a context-free grammar, or CFG, each production has the form $x \rightarrow y$, where x is a member of N, and y is any string in $(N \cup \sum)^*$. Note that y may be the empty string. Hence, any CFG with a rule $A \rightarrow \varepsilon$ cannot be context-sensitive: the latter class does not permit such a rule.

An example of CFG that we shall be using repeatedly is an arithmetic expression grammar G0:

$N = \{E, T, F\}$

$\sum = \{+ , * , (,), a\}$

$S = E$

P = the set

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow a$

Here, the nonterminal set is clearly {E, T, F}, the terminal set is {+, *, (,), a}, and the start symbol is E. We may obtain a typical expression by applying the replacement rules, as before:

E derives E + T, using the first rule
E + T derives T + T, using the second rule
T + T derives F + T, using the fourth rule
F + T derives a + T, using the last rule
a + T derives a + F, using the fourth rule
a + F derives a + a, using the last rule

Hence, the string a + a is in G0. Many other examples of derived terminal strings in this grammar may be obtained.

When a grammar has several productions with the same left member, we sometimes will use the symbol | , which stands for alternation, as an abbreviation for two rules. Thus the two rules $E \rightarrow E + T$ and $E \rightarrow T$ may be written:

$$E \rightarrow E + T \mid T$$

Significance of the Grammar Classification

These grammar classifications are to some extent arbitrary. One may define many variations on the basic patterns given. However, these definitions lead to particularly simple classes of sentence recognizing machines or *automata*.

An *automaton*, for our purposes, is some machine with a finite description (but not necessarily containing a finite number of parts) that, given a grammar, can accept some string of terminal symbols and can determine whether the string can be derived in the grammar.

The process of finding a derivation, given a grammar and a terminal string supposedly derivable in the grammar, is called *parsing*, and an automation capable of parsing I called a *parser*. A parsing automaton is of value in a compiler. A grammar is a concise yet accurate description of some language; it expresses the class of structures permissible in the language. However, so far we see only how to construct legal strings in the language. We need to solve the opposite problem: given some string, how do we determine if it is legal? We also need to go further than that; we must determine the sequence of productions needed to obtain the string. For that we need a parser.

Each of the three phrase-structured grammar classes has a fairly simple yet powerful automaton associated with it:

1. The right-linear grammars can be recognized by a finite-state automaton. which consists merely of a finite set of states and a set of transitions between pairs of states. Each transition is associated with some terminal symbol. We shall define finite-state automata more completely in the next chapter.
2. The CFGs are accepted by a finite-state automaton controlling a push-down stack, with certain simple rules governing the operations. The push-down stack is the only element that can be indefinitely large. However, only a finite group of top stack members are ever referenced in the description of this automaton.
3. The CSGs are accepted by a two-way, linear bounded automaton, which is essentially a Turing machine the tape of which is not permitted to grow longer than the input string.

Right-Linear Grammars

If each production in P has the form $A \rightarrow xB$ or $A \rightarrow x$, where A and B are in N and x is in $\sum^*$, the grammar is said to be *right-linear*.

The right-linear grammars clearly are a subset of the CFGs. The following grammar G2 is an example of a right-linear grammar; it defines a set of ternary fixed point numbers, with an optional plus or minus sign:

$V \rightarrow N \mid +N \mid -N$
$N \rightarrow 0 \mid 1 \mid 2$
$N \rightarrow 0N \mid 1N \mid 2N$

Two other related grammars are the *left-linear* and *regular grammar*. A left-linear grammar has productions in P of the form $A \rightarrow Bx$ or $A \rightarrow x$, where A, B, and x have previously defined meanings. A regular grammar is such that every production in P, with the exception of $S \rightarrow \varepsilon$ (S is the start symbol) is of the form $A \rightarrow aB$ or $A \rightarrow a$, where a is in $\sum$. Further, if $S \rightarrow \varepsilon$ is in the grammar, then S does not appear on the right of any production.

The following example of a regular grammar defines the fixed point decimal numbers with a decimal point; the d stands for a decimal digit:

$S \rightarrow dB \mid +A \mid -A \mid .G$
$A \rightarrow dB \mid .G$
$B \rightarrow dB \mid .H \mid d$
$G \rightarrow dH$
$H \rightarrow dH \mid d$

Comparison to Regular Expression Notation

Consider how the above sample context-free grammar compares to the regular expression rules given for **number** in the previous chapter:

number = digit digit*
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]

In basic regular expression rules we have three operations: choice (given by the vertical bar metasymbol), concatenation (with no metasymbol), and repetition (given by the asterisk metasymbol). We also use the equal sign to represent the definition of a name for a regular expression, and we write the name in italics to distinguish it from a sequence of actual characters.

Grammar rules use similar notations. Names are written in italic (but now in a different font, so we can tell them from names for regular expressions). The vertical bar still appears as the metasymbol for choice. Concatenation is also used a standard operation. There is, however, no metasymbol for repetition (like the * of regular expressions), a point to which we shall return shortly. A further difference in notation is that we now use the arrow symbol $\rightarrow$ instead of equality to express the definitions of names. This is because names cannot now simply be replaced by their definitions, but a more complex defining process is implied, as a result of the recursive, in that the name *exp* appears to the right of the arrow.

Note, also, that the grammar rules use regular expressions as components. In the rules for *exp* and *op* there are actually six regular expressions representing tokens in the language. Five of these are single-character tokens: +, -, *, (, and). One is the name ***number***, the name of a token representing sequences of digits.

Grammar rules in a similar form to this example were first used in the description of the Algo160 language. The notation was developed by John Backus and adapted by Peter Naur for the Algo160 report. Thus, grammar rules in this form are usually said to be in **Backus-Naur form**, or **BNF**.

The Chomsky Hierarchy and the Limits of Syntax as Context-Free Rules

When representing the syntactic structure of a useful and powerful tool. But it is also important to know what can or should be represented by the BNF. We have already seen a situation in which the grammar may be left ambiguous intentionally (the dangling else problem), and thus not express the complete syntax directly. Other situations can arise where we may try to express too much in the grammar, or where it may be essentially impossible to express a requirement in the grammar. In this session, we discuss a few of the common cases.

A frequent question that arises when writing the BNF for a language is the extent to which the lexical structure should be expressed in the BNF rather than in a separate description (possibly using regular expression). The previous discussion has shown that context-free grammars can express concatenation, repetition, and choice, just as regular expressions can. We could, therefore, write out grammar rules for the construction of all the tokens from characters and dispense with regular expressions altogether.

For example, consider the definition of a number as a sequence of digits using regular expressions:

***digit* = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
number = digit digit*

we can write this definition using BNF, instead, as

digit $\rightarrow$ **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
number $\rightarrow$ *number digit* | *digit*

Note that the recursion in the second rule is used to express repetition only. A grammar with this property is said to be a ***regular grammar****, and regular grammars can express everything that regular expressions can. A consequence of this is that we could design a parser that would accept characters directly from the input source file and dispense with the scanner altogether.*

Why isn't this a good idea? Efficiency would be compromised. A parser is a more powerful machine than a scanner but is correspondingly less efficient. Nevertheless, it may be reasonable and useful to include a definition of the tokens in the BNF itself-structure. Of course, the language implementor would be expressed to extract these definitions from the grammar and turn them into a scanner.

A different situation occurs with respect to **context rules**, which occur frequently in programming languages. We have been using the term context-free without explaining why such rules are in fact "free of context. "The simple reason is that nonterminals appear by themselves to the left of the arrow in context-free rules. Thus, a rule say that A may be replaced by *α anywhere*, regardless of where the A occurs. On the other hand, we could informally define a **context** as a pair of strings (of terminals and nonterminals) β, γ such that a rule would apply only if β occurs before and γ occurs after the nonterminal. We would write this as

$$\beta A \gamma \rightarrow \beta \alpha \gamma$$

such a rule in which α≠εis called a **context-sensitive grammar rule**. Context-sensitive grammars are more powerful than context-free grammars but are also much more difficult to use as the basis for parser.

What kind of requirements in programming languages require context-sensitive rules? Typical examples involve the use of names. The C rule requiring declaration before use is a typical example. Here, a name must appear in a declaration before its use in a statement or expression is allowed:

```
{ int x;
  …
  …x…
  …
}
```

if we were to try to deal with this requirement using BNF rules, we would, first, have to include the name strings themselves in the grammar rules rather than include all names as identifier tokens that are indistinguishable. Second, for each name we could have to write a rule establishing its declaration prior to a potential use. But in many languages, the length of an identifier is unrestricted, and so the number of possible identifiers is (at least potentially) infinite. Even if names are allowed to be only two characters long, we have the potential for hundreds of new grammar rules. Clearly, this is an impossible situation. The solution is similar to that of a disambiguating rule: we simply state a rule (declaration before use) that is not explicit in the grammar. There is a difference, however: such a rule cannot be enforced by the parser itself, since it is beyond the power of (reasonable) context-free rules to express. Instead, this rule becomes part of semantic analysis, because it depends on the use of the symbol table (which records which identifiers have been declared).

The body of language rules that are beyond the scope of the parser to check, but that are still capable of being checked by the compiler, are referenced to as the **static semantics** of the language. These include type checking (in a statically typed language) and such rules as declaration before use. Henceforth, we will regard as *syntax* only those rules that can be expressed by BNF rules. Everything else we regard as semantics.

There is one more kind of grammar that is even more general than the context-sensitive grammars. These grammars are called **unrestricted grammars** and have grammar rules of the form $\alpha \rightarrow \beta$, where there are no restrictions on the form of the strings α and β (except that α cannot be ε). The four kinds of grammars—unrestricted, context sensitive, context free, and regular—are also called type 0, type 1, type 2, and type 3 grammars, respectively. The language classes they construct are also referred to as the **Chomsky hierarchy**, after Noam Chomsky, who pioneered their use to describe natural languages. These grammars represent distinct levels of computational power.

Indeed, the unstricted (or type 0) grammars are equivalent to Turing machines in the same way regular grammars are equivalent to finite automata, and thus represent the most general kind of computation known. Context-free grammars also have a corresponding equivalent machine, called a pushdown automaton, but we will not need the full power of such a machine for our parsing algorithms and do not discuss it further.

We should also be aware that certain computationally intractable problems are associated with context-free languages and grammars. For example, in dealing with ambiguous grammars, it would nice if we could state an algorithm that would convert an ambiguous grammar into an unambiguous one without changing the underlying language. Unfortunately, this is known to be an undecidable problem, so that such an algorithm cannot possibly exist. In fact, there even exist context-free languages for which *no* unambiguous grammar exists (these are called **inherently ambiguous languages**), and determining even whether a language is inherently ambiguous is undecidable.

Fortunately, complications such as inherent ambiguity do not as a rule arise in programming languages, and the ad hoc techniques for removing ambiguity that we have described usually prove to be adequate in practical cases.