Chapter 3

Chomsky Grammars

3.1 Formal Grammars

Formal grammars are a special form of a more general concept called a rewriting system. There are many general rewriting systems. Some are Markov algorithms, Post productions, and Thue and semi-Thue systems. These are not covered here, but the reader should be aware that the formal grammars presented here are those studied by Noam Chomsky and can be viewed as restricted semi-Thue system. Before showing the classes of grammars and the corresponding languages generated by these Chomsky grammars, several definitions along with notation is presented.

In the context of studying programming languages, we are interested in using these grammars for not only describing what is and is not in a given programming language, but also use these for writing translators. The grammar-related concepts of parsing, parse trees, ambiguity, and the relationship to translation and compiler building are presented. It will be shown that two restricted forms of Chomsky grammars, namely Context-Free and Regular, are the most useful for writing translators.

3.1.1 Generating Languages with Grammars

The definition given next is one for a general Unrestricted Grammar, also known as a Phrase-Structured grammar.

Definition 3.1.1 Grammar: A *grammar* defined by four items represented as a quadruple, $< V_N, V_T, P, S >$, where:

- V_N is a nonempty, finite set of symbols called *nonterminals.*
- V_T is a nonempty, finite set of symbols called *terminals* such that $V_N \cap V_T = \emptyset$.
- $P \subset V^+ \times V^*$ is a non-empty, finite set of production rules, where $V = V_N \cup V_T$.
- $S \in V_N$ is a one and only one designated symbol called the *start symbol.* ■

Definition 3.1.2 Immediate (Direct) Derivation: If $\alpha \to \beta$ is a production and $\omega\alpha\phi$ is a string, where $\omega, \phi \in V^*$ and $\alpha \in V^+$, then $\omega\alpha\phi \Rightarrow \omega\beta\phi$ is an *immediate (or direct) derivation.* ■

Definition 3.1.3 Derives and Reduces Relations: A sequence of immediate derivations $\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{n-1} \Rightarrow \alpha_n$, usually written, $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$ is a *derivation of length n.* We say that α_1 *derives* α_n and α_n *reduces* to α_1; similarly, α_n is *derivable* from α_1 and α_1 is *reducible* from α_n. We represent this as $\alpha_1 \Rightarrow^+ \alpha_n (n > 1)$. We write $\alpha_1 \Rightarrow^* \alpha_n$ $(n \geq 1)$. ■

Definition 3.1.4 Sentential Forms and Sentences: A string α derivable from the start symbol, S, of a grammar, $G =< V_N, V_T, P, S >$, is called a *sentential form*, i.e. $S \Rightarrow^* \alpha$, where $\alpha \in V^* = (V_N \cup V_T)^*$. A string α, is a *sentence* iff α is a sentential form and $\alpha \in V_T^*$. ■

Definition 3.1.5 Grammar-Generated L(G): Language, $L(G)$, *generated by grammar G*, is defined as the set of sentences:

$$L(G) = \{\alpha \in V_T^* \mid S \Rightarrow^+ \alpha\}$$

Generally, for definitions and abstract examples, we will continue to use the metasymbols '$\to$' and '|'. In addition, when the non-terminal and terminal sets are not explicitly listed, the following conventions will be used:

- Nonterminal symbols:strings starting with upper-case Latin letters (e.g. A,B,C,Expr,...)
- Terminal symbols: strings starting with lower-case Latin letters and strings of printable non-alpha characters, excluding '|', and '$\to$') (e.g. a,b,c, id, *, (,) ...)
- Lower-case Greek letters to denote strings in V^* (e.g. $\alpha, \beta, \gamma, \ldots$).

3.1.2 Examples of Grammars

Example 3.1 ***Binary Number Language:***

Let $G =< \{N, A, B\}, \{., 0, 1\}, P, N >$ *where P:*

$$\begin{aligned} N &\to A \mid A.A \\ A &\to B \mid AB \\ B &\to 0 \mid 1 \end{aligned}$$

Here, we have:

$$V = V_N \cup V_T = \{N, A, B, ., 0, 1\}$$

$$N \Rightarrow A.A \Rightarrow AB.A \Rightarrow A1.A$$

$A1.A$ is a sentential form, but it is not a sentence.

$$\begin{array}{ccccccc} N & \Rightarrow & A.A & \Rightarrow & AB.A & \Rightarrow \\ A1.A & \Rightarrow & B1.A & \Rightarrow & 11.A & \Rightarrow \\ 11.B & \Rightarrow & 11.0 \end{array}$$

$N \Rightarrow^* 11.0$ so 11.0 is a sentential form and is also a sentence. After trying some derivations of sentences you should see (intuitively) that $L(G)$ is the language of binary numbers. ■

Example 3.2 ***Expression Grammar: E, T, F***

Let $G =< \{E, T, F\}, \{id, (,), *, +\}, P, E >$ where P:

$$\begin{array}{ccc} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid id \end{array}$$

This grammar is used to describe arithmetic expressions in many programming languages. *E, T,* and *F* stand for *Expression, Term,* and *Factor*, respectively. Some derivations of sentences are shown below:

1.

$$E \Rightarrow T \Rightarrow F \Rightarrow id$$

2.

$$\begin{array}{ccccccc} E & \Rightarrow & E + T & \Rightarrow & E + T * F & \Rightarrow \\ E + F * F & \Rightarrow & E + id * F & \Rightarrow & E + id * id & \Rightarrow \\ T + id * id & \Rightarrow & F + id * id & \Rightarrow & id + id * id \end{array}$$

3.

$$\begin{array}{ccccccc} E & \Rightarrow & E + T & \Rightarrow & E + F & \Rightarrow \\ E + (E) & \Rightarrow & E + (T) & \Rightarrow & E + (T * F) & \Rightarrow \\ E + (T * id) & \Rightarrow & E + (F * id) & \Rightarrow & E + (id * id) & \Rightarrow \\ T + (id * id) & \Rightarrow & F + (id * id) & \Rightarrow & id + (id * id) \end{array}$$

4.

$$\begin{array}{ccccccc} E & \Rightarrow & E + T & \Rightarrow & T + T & \Rightarrow \\ F + T & \Rightarrow & id + T & \Rightarrow & id + F & \Rightarrow \\ id + (E) & \Rightarrow & id + (T) & \Rightarrow & id + (T * F) & \Rightarrow \\ id + (F * F) & \Rightarrow & id + (id * F) & \Rightarrow & id + (id * id) \end{array}$$

■

Note that the last two derivations produced the same sentence. Derivations 3 and 4 are called *rightmost* and *leftmost*, respectively. In the right(left)most derivations the right(left)most nonterminal is replaced. The formal definition for a rightmost derivation is as follows:

Let $G =< V_N, V_T, P, S >$ be a grammar. The derivation

$$\alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_{n-1}$$

$(n > 1)$ is a *rightmost derivation* if and only if each direct derivation $\alpha_{i-1} \Rightarrow \alpha_i$ $(1 \leq i \leq n-1)$ is of the form $\alpha_{i-1} = \phi A \omega$ and $\alpha_i = \phi\beta\omega$ where $\omega \in V_T^*, A \in V_N$, and $\phi \in V^*$ Remember $V = (V_N \cup V_T)$. A *leftmost derivation* is defined similarly — just swap the ϕ with the ω.

3.2 The Chomsky Hierarchy

Up to this point the reader may have observed that the productions of all the examples have been restricted forms of that given in the definition of a grammar: each left-hand side of the productions have consisted only of a single nonterminal. Grammars with these types of productions are called *context-free* grammars (CFGs). Computer scientists have found that CFGs are very useful in describing many artificial languages used in computer science, especially programming languages. The other Chomsky grammars are also very important to computer scientists and other researchers who find their research intersecting the general areas of theory of computation and linguistics — there are too many specific areas to begin listing them.

In this section we will examine four types of grammars that make up the Chomsky hierarchy and mention a few of the relationships between them.

3.2.1 Definitions of the Grammar Classes

We begin by considering the grammar given in the definition of grammars as the most general type and proceed by imposing more restrictions on the production forms. Note that the type-0 grammar is identical to the original definition of a grammar given in Definition 3.1.1. For the more restrictive grammar types, 1,2,and 3, we add the condition that only the productions with ϵ on the right-hand side, is the ones with only the start symbol on the left-hand side; no other production can have ϵ on the right-hand side.

There are four classes of Chomsky grammars, each adding more constraints on the production rule form.

Definition 3.2.1 Phrase-Structured Grammar (type 0): A *phrase-structured* grammar (PSG) has productions of the form:

$$\alpha \rightarrow \beta$$

where $\alpha \in V^+$ and $\beta \in V^*$. This type of grammar is like a *semi-Thue* with the designated string restricted to the start symbol; this type 0 grammar form is also called *unrestricted* in many computer science and mathematics journals and textbooks.

If G is a PSG, the L(G) is a phrase-structured language (PSL).

Definition 3.2.2 Context-Sensitive Grammar (type 1): A *context-sensitive* grammar (CSG) has productions of the form:

$$\alpha \rightarrow \beta \qquad \alpha \in V^+,\ \alpha \neq \beta, \text{ and } |\beta| \geq |\alpha|.$$

Note that since β must be at least as long as α, so obviously $\beta \in V^+$. This length restriction classifies this type of grammar as *non-contracting.*

The reason CSGs are called "context-sensitive" is because each CSG production set can be transformed into productions that generate the same but have the following form:

$$\phi_1 A \phi_2 \rightarrow \phi_1 \omega \phi_2$$

where $\phi_1, \phi_2 \in V^*$ and $\omega \in V^+$. In other words nonterminal A can only be replaced in the "context" of ϕ_1 and ϕ_2. We will not prove that all CSGs can be written in this form.

If G is a CSG, then L(G) is a context-sensitive language (CSL).

Definition 3.2.3 Context-Free Grammar (type 2: A *context-free* grammar (CFG) has productions of the form:

$$A \rightarrow \beta \qquad A \in V_N \text{ and } \beta \in V^+.$$

Note that the productions of CFGs are special cases of CSG productions: simply let $\phi_1 = \epsilon$ and $\phi_2 = \epsilon$. Also note that all of the examples presented in the preceding sections are CFGs.

If G is a CFG, then L(G) is a context-free language (CFL).

Definition 3.2.4 Regular (or linear) Grammar (type 3): A *regular* (RG) grammar is either a left or right linear grammar. A *left linear* grammar (RG) is a restricted CFG that has productions only of the form:

$$A \rightarrow Ba \quad \mathit{or} \quad A \rightarrow a \quad A, B \in V_N \text{ and } a \in V_T.$$

A *right linear* grammar has productions only of the form:

$$A \rightarrow aB \quad \mathit{or} \quad A \rightarrow a \quad A, B \in V_N \text{ and } a \in V_T.$$

If G is an RG, L(G) is a regular language (RL).

We need to discuss languages that contain the empty string, ϵ. For these languages we may derive it as simply: $S \Rightarrow \epsilon$, where $S \in V_N$ is the start symbol, no matter what type of grammar. For describing some languages, it may be a matter of convenience to allow some productions of the form $A \rightarrow \epsilon$, where $A \in V_n$ and A not the start symbol. However, it can be shown that if grammar G_1 has $n > 0$ ϵ-productions, where ϵ may or may not be in $L(G_1)$, then there exists a grammar G_2 where G_2 has no ϵ-productions and where $L(G_2) = L(G_1)$. If $\epsilon \in L(G_1)$, then one must include in G_2 the production $S \rightarrow \epsilon$, where S is the start symbol.

For the remainder of this textbook, we will assume that ϵ-productions are allowed for grammars. Just remember that the remaining CSG productions may not contract.

3.2.2 Examples of Classifying Grammars

When classifying a grammar within the context of the Chomsky hierarchy, always begin by trying to see if every one of the grammar's production rules has the form of a regular grammar. If it does not, check whether it is a CFG. If not, then check whether it is a CSG. Remember that every Chomsky grammar is a PSG and that RGs are the smallest, most restrictive class; always start with the smallest class, RGs, and "work your way up" to the most general class.

Note that if there is more than one symbol, or if there is a terminal on the left-hand side of any production, then it cannot be an RG or CFG. To help distinguish between CSGs and PSGs remember that PSGs can have productions $\alpha \to \beta$ with $|\alpha| \geq |\beta|$; i.e. PSGs are contracting grammars and CSGs are non-contracting. This can help reduce the amount of work classifying a Chomsky grammar.

Example 3.3 ***Context-Sensitive Grammar***

$$\begin{aligned} S &\to cE \mid cS \mid dS \mid b \\ cE &\to cdS \mid cd \end{aligned}$$

The last two productions do not fit the form of CFG production rules. Thus, we must check to see if they fit the CSG form. Recalling the definition of CSGs all CSG productions are non-contracting or have the following form $\phi_1 A \phi_2 \to \phi_1 \omega \phi_2$ where $\phi_1, \phi_2 \in V^*$ and $\omega \in V^+$. Grammar G2 is non-contracting so it is a CSG. Moreover, it fits the second form also. Consider the production $cE \to cdS$. We see that if $\phi_1 = c$, $A = E$, $\phi_2 = \epsilon$, and $\omega = dS$, the production fits the form. Similarly, the last production also fits the form if the same assignments are made with the exception of $\omega = d$. ■

Example 3.4 ***Context-Free Grammar***

A context-free grammar, G3:

$$\begin{aligned} S &\to bDc \mid Db \mid b \\ D &\to bbDc \mid b \mid \epsilon \end{aligned}$$

Grammar G3 is not an RG because the first and fourth productions do not fit the form of an RG. Since we are allowing ϵ-productions in our CFG production forms all productions in G3 do fit the form of a CFG. Try describing the language $L(G3)$. ■

Example 3.5 ***Regular Grammar***

A regular grammar, G4:

$$\begin{aligned} S &\to aU \mid bV \\ U &\to bS \mid b \\ V &\to aS \mid a \end{aligned}$$

The grammar G4 is a right linear grammar. An RG is either right linear or left linear and does not mix productions of the two types. ■

3.2.3 Relationships of the Grammar and Language Classes

As discussed and illustrated earlier, the four types of Chomsky grammars are identified by the form of their productions. Regular grammar productions are special cases of context free productions and context free productions are special cases of context sensitive productions. All productions are phrase structured. Each class of grammars has a corresponding class of languages. Given a language, L, one is not only interested in constructing a grammar G such that $L = L(G)$, but also constructing a grammar of the most restrictive type possible. This begs the questions: are there languages that no regular grammar can generate? Are there languages that no context free grammar can generate? The answers to these questions are all yes. To discuss this we introduce some definitions and notation and then look at some examples.

Let $\mathcal{L}_i$ be the class of languages that can be generated by a grammar of type i, $i \in \{0, 1, 2, 3\}$.

It is beyond the scope of this textbook, but it can be shown that:

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$$

Example 3.6 ***Three Languages***

Shown below are three languages and their respective types and sample grammars.

$L(G)$	Language Type	G
$\{a^n b^m \mid n > 0, m > 0\}$	Regular	$S \rightarrow aA$ $A \rightarrow aA \mid bB \mid b$ $B \rightarrow bB \mid b$
$\{a^n b^n \mid n > 0\}$	Context Free	$S \rightarrow aSb \mid ab$
$\{a^n b^n c^n \mid n > 0\}$	Context Sensitive	$S \rightarrow aSBC \mid aBC$ $CB \rightarrow BC$ $aB \rightarrow ab$ $bB \rightarrow bb$ $bC \rightarrow bc$ $cC \rightarrow cc$

■

Suppose that a software engineer is given a language L and must find a grammar G, such that $L = L(G)$. How does the software engineer know that they have found the most restrictive grammar type that can generate language L? Computer scientists have a collection of theorems and lemmas that can identify the smallest class of languages to which a language belongs. If the smallest language type is known, grammars of a particular form can be pursued and properties can be exploited. Once the language type is known the appropriate known algorithms and models of computations can be utilized in solving the decision language membership problem: "$\alpha \in L(G)$?" Some of these algorithms try to construct a derivation of the string being tested for membership in the usual sense by starting with the start symbol and reading one symbol at a time in one direction (usually left-to-right) with possibly looking $k \geq 0$ symbols ahead. Alternatively, one can re-construct the derivation "backward" through a series of reductions while reading the string in one direction with possibly looking ahead.

Another approach to testing membership is to utilize a model of computation, such as a finite state automaton or push-down automaton. Later we will see that finite state machines can be used for testing string membership in regular languages. Push-down automata can be used for testing string membership in context-free languages. It should be pointed out that linear-bounded Turing machines can be used for testing string membership in context-sensitive languages and Turing machines for testing membership of any language. We will see that these machines can be represented as formal rewrite rules and also have direct relationships with the corresponding grammars in that there are algorithms for translating between the two types of formalisms.

These membership algorithms are referred to as *parsers.* An important underlying abstraction used implicitly and, sometimes explicitly, by these parsers is a parse tree.

3.3 Parse Trees

As mentioned earlier, to solve problem of the testing $\alpha \in L(G)$, one tries to construct a derivation of the string α using productions of G. The construction of a derivation is called *parsing.* A *parse* of some sentence (or more generally a sentential form) shows how the sentence (or sentential form) was derived. A *parser* is simply the name for the procedure that performs the construction of the derivation. Each parse has a corresponding parse tree. We define a parse tree.

3.3.1 Terminology

A *directed acyclic graph* is a directed graph with no cycles. A *tree* is a directed acyclic graph with a unique path from a designated node called the *root* to each of the other nodes. A node n_1 is a *descendant* of another node n_2 if there is a directed path from that n_2 to n_1. *Leaf* nodes have no descendants and the root node is not a descendant of any node. A node that is neither a root node nor a leaf node is an *interior* node. A node n_1 is a *child* of node n_2 if (n_2, n_1) is an edge (branch) in the tree. An *ordered tree* has a linear order on the children of every node.

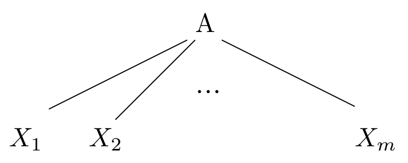


Figure 3.1: $A \to X_1 X_2 \dots X_m$

Definition 3.3.1 Parse Tree: Given a CFG, $G =< V_N, V_T, P, S >$ an ordered tree is a *parse tree* if the following conditions are met:

1. The root is labeled by the start symbol.
2. A leaf node is labeled by a terminal or ϵ.
3. Each interior node is labeled by a nonterminal.
4. If A is an interior node and $X_1,\ X_2,\ \dots,\ X_m$ are children of the node, then $A \to X_1 X_2 \dots X_m$ is a production of the G, where $X_i \in V \cup \{\epsilon\}$. See Figure 3.1.

Note that we have defined parse trees for CFGs. Since the usage of one production rule in a particular context might modify the context of applying another production rule, trees are not usually used as a means of studying context-sensitive grammars. The orders of nonterminal substitution in a CFG derivation does not matter, whereas for CSGs it does matter in some cases.

3.3.2 A Parse Tree Example

Example 3.7 ***Parse Tree for*** $id + id * id$ ***using EFT-Grammar***

Consider the expression grammar:

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid id \end{aligned}$$

The parse tree for the derivation of the sentence $id + id * id$ *is shown in Figure 3.2. The derivation of* $id + id * id$ *is:*

$$\begin{array}{ccccc} E & \Rightarrow & E + T & \Rightarrow & E + T * F \ \Rightarrow \\ E + F * F & \Rightarrow & E + id * F & \Rightarrow & E + id * id \ \Rightarrow \\ T + id * id & \Rightarrow & F + id * id & \Rightarrow & id + id * id \end{array}$$

■

Try doing a rightmost derivation then a leftmost derivation of the same sentence: $id + id * id$. Draw and compare the parse trees for each of the derivations.

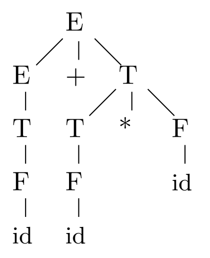


Figure 3.2

3.3.3 Ambiguity

In the previous Example (3.7) you should have discovered that each derivation corresponded to the same parse tree. Also, you should note that there is only one left- and rightmost derivation for that sentence. Some grammars, however, may allow more than one parse tree for a sentence or more than one left- or rightmost derivation. This is referred to as an *ambiguous* grammar. We define ambiguity formally as:

Definition 3.3.2 Ambiguous Sentences, Grammars, and Languages: An *ambiguous sentence* in $L(G)$ is a sentence that has two or more distinct leftmost derivations with respect to grammar G. An *ambiguous grammar* is a grammar that generates at least one ambiguous sentence. A *language* is *(inherently) ambiguous* if and only if there are no unambiguous grammars that generate the language.

NOTE: We could have used "*rightmost derivations*" or "*parse trees*" in place of "*leftmost derivations.*" Each definition would be equivalent. Note that a sentence may still be considered ambiguous even if the two corresponding **unlabeled** parse trees have the same structure, i.e. a different labeling of the interior nodes indicates ambiguity.

Example 3.8 ***Structural Ambiguity***

Consider the following grammar:
$G =< \{E\}, \{*, +, id\}, P, E >$ *where P is:*

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

This grammar generates the same language as the grammar of example 3.7. One of the differences between the two grammars is that this grammar is ambiguous and the one in example 3.7 is not. For example, there are two leftmost derivations for the following sentence $id + id * id$. *Using grammar G in this example we have at least two leftmost derivations:*

1.

$$\begin{array}{ccccccc} E & \Rightarrow & E + E & \Rightarrow & id + E & \Rightarrow \\ id + E * E & \Rightarrow & id + id * E & \Rightarrow & id + id * id \end{array}$$

2.

$$\begin{array}{ccccccc} E & \Rightarrow & E * E & \Rightarrow & E + E * E & \Rightarrow \\ id + E * E & \Rightarrow & id + id * E & \Rightarrow & id + id * id \end{array}$$

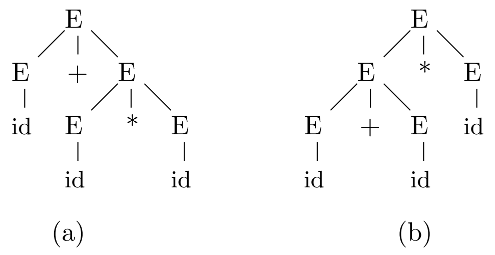


Figure 3.3: Structural ambiguity

The parse trees in Figures 3.3(a) and (b) correspond to derivations 1 and 2, respectively. This example illustrates a sentence with two structurally distinct parse trees. Compare these trees to the one in Figure 3.2. The parse tree in Figure 3.3(a) is the structure that corresponds to the semantics or evaluation that is used in mathematics. When defining the semantics or meaning of programs, one must keep in mind the underlying intended semantics when designing a grammar for the syntax. Many translators build parse trees when performing the syntax analysis and attach semantic attributes to the nodes for use by other components of the compiler, such as the code generator. ■

Example 3.9 ***Labeling Ambiguity***

Consider the following grammar:

$$G = < \{S, A\}, \{a, b\}, P, S >$$

where P is:

$$\begin{aligned} S &\rightarrow Sa \mid Aa \mid b \\ A &\rightarrow Aa \mid b \end{aligned}$$

The sentence baa can be shown to be ambiguous by looking at the two parse trees for it shown in figure 3.4. This illustrates labeling ambiguity. ■

3.4 Parsers

As the construction of a derivation is attempted, the parser needs to read the input string either left-to-right or right-to-left. We will assume left-to-right.

There are essentially two categories of parsers: top-down and bottom-up. Top-down builds parse trees from the root to the leaves and bottom-up builds from the leaves to the root. Many top-down parsers can be implemented by writing a program to parse a particular language, although there are some software tools for some subclasses of CFGs that automatically build top-down parsers directly from the grammar.

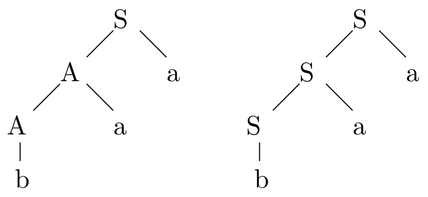


Figure 3.4: Labeling ambiguity

Bottom-up parsers can handle more CFGs and are usually implemented by a collection of software tools that automatically build the parsers directly from a grammar specification. The Unix tool, yacc[1], is an example of one such tool.

Suppose α is a string. Top-down parsers operate much the way derivations are presented in this paper: beginning with the start symbol a sequence of immediate derivations are executed, each with a nonterminal replaced with the corresponding right side of a production rule that has a matching left side. This continues until α (the goal) or no production rule can be applied. In the latter case the string is considered *ill-formed*.

Bottom-up parsers operate "backwards" relative to top-down parsers. A sequence of *reductions* are performed. A reduction is simply the replacement of a substring of a sentential form that matches the right side of a production with the corresponding left side. A bottom-up parser begins with a sentence α and performs a sequence of reductions until the start symbol S is reached, or no right-hand side matches any substring of the current sentential form — the latter being ill-formed.

We shall concentrate on top-down parsing. Recursive-descent parsers are one such class of parsers. This will be discussed in more detail later.

3.5 MetaLanguages for Presenting Grammars

The prefix, "meta-", in this context is used for symbols or syntactic structures that are used to describe, express and define grammars, that, in turn, express the syntactic structure of languages. In this section, several metanotations that allow designers, implementors, and programmers to communicate the syntactic structure of programming languages are shown. The material covered in this section provides the background needed for designing and implementing parsers for programming languages.

3.5.1 BNF Notation

Backus-Naur Form (BNF) is a metasyntactic notion for expressing grammars. It was originally introduced by two computer language and compiler designers, John Backus and Peter Naur. Emil Post's notation for

[1]yacc stands for *y*et *a*nother *c*ompiler-*c*ompiler.

his production system computational provided a starting point for John Backus to utilize an early version of BNF for describing a language, called IAL, that would later evolve into Algol 58. Later Peter Naur utilized it for describing Algol 60. Although John Backus is most known for his leadership on the design team for the first designs of Fortran in the mid 1950's, he and Peter Naur were on the design team for Algol 60. For details and the history of the development of BNF, the interested reader should consult [1] and [9].

A variation of BNF has been utilized earlier in the text for presenting grammar-related and language related concepts. Thus far, for the purposes definitions, theorems, and examples illustrating the definitions, the meta-symbols,of $\rightarrow$ and $|$ have been utilized. Uppercase and lowercase letters have been used to represent nonterminals and terminals, respectively. For "real" programming languages, more meaningful names, such as *parameter-list* as apposed to A, need to be used by language designers, implementors, when writing and reading grammars. In order to distinguish between terminals and nonterminal symbols, the strings representing nonterminals are simply surrounded with a pair of angle-brackets, <>. Several metanotations are used for terminals. With the ability to distinguish between nonterminals and terminals, one need not explicitly list the elements of the nonterminal and terminal sets, V_N and V_T, respectively. One can express the four sets that comprise a grammar, $< V_N, V_T, P, S >$, by simply listing the production rule set in BNF and putting the production rule(s) with the start symbol as the left-hand side symbol as the first one(s) listed. A summary of the BNF metasymbols and their meanings is shown in Table 3.1.

MetaSymbol	Meaning
::= or $\rightarrow$	delimits left-hand side from right-hand side
$\vert$	"or"
< >	Surrounds *nonterminal* names
Unaltered strings not between < > Underlined string bold-faced **string** Single quotes 'string' Double quotes "string"	terminals

Table 3.1: BNF Metasymbols

When BNF was first introduced in the late 1950's, there were not many fonts and typefaces available on most printers used within general purpose computer systems; even if one could interchange the font and typeface, the ability to mix fonts and typeface during the printing of one file was almost non-existent at that time. Word processors with bold and underline functions were not widely available. So, the adoption of quoting terminals or leaving strings without surrounding angle brackets as terminals were utilized also. Usually one font was used using a single character set, such as ASCII. So, many of these notational conventions were adopted because of this. For example, the ::=[2] was adopted instead of $\rightarrow$, since ASCII does not include $\rightarrow$, but consists of a subset of ASCII characters.

[2]A personal speculation: because = and := were terminals in Algol-like languages at the time, ::= seemed to be the most natural metasymbol.

Note that when metasymbols are part of the language generated by a grammar expressed in BNF, single or double quotes must be used for those symbols. If single or double quotes are part of the language, then two consecutive single or double quotes, respectively, are often used when expressing a grammar in BNF for that language. For example, two consecutive double quotes, `""`, used presenting a grammar using EBNF specifies a single double quote terminal symbol, `"`. Note this is how C and C++ programmers can write a single-quote character literal and express a string literal with a double quote embedded.

For some purposes it is also convenient to mix notations for terminals in a set of productions. For example, one might put single quotes around single character terminals, underline reserved words and/or leave operator symbols unquoted in the same EBNF-specified production rule set.

As one can see there are many metanotations for even simple BNF. One must keep in mind that notational consistency provides the best readability and understandability of the grammar and, in turn, of the language generated from that grammar. If you are designing a language and/or a grammar, for a language, it is always best to supply a legend for your readers, the programmers and implementors.

Example 3.10 ***Expression Grammar in BNF***

In this example a grammar for representing the syntactic structure of simple expressions used in many imperative languages is shown. What the type of the expressions are or the operator symbols actually represent is not known; this only expresses the syntax.

$$\begin{array}{rcllll}
<\mathit{Expr}> & ::= & <\mathit{Expr}> + <\mathit{Term}> & \mid <\mathit{Expr}> - <\mathit{Term}> & \mid <\mathit{Term}> & \\
<\mathit{Term}> & ::= & <\mathit{Term}> * <\mathit{Factor}> & \mid <\mathit{Term}> / <\mathit{Factor}> & \mid <\mathit{Factor}> & \\
<\mathit{Factor}> & ::= & <\mathit{Primary}> & \mid + <\mathit{Primary}> & \mid - <\mathit{Primary}> & \mid (<\mathit{Expr}>) \\
<\mathit{Primary}> & ::= & \underline{\mathit{identifier}} & \mid \underline{\mathit{number}} & &
\end{array}$$

From the BNF notation the set of terminals and nonterminals can be inferred:

$$\begin{aligned}
V_T &= \{+, -, *, /, (,), \mathit{identifier},\ \mathit{number}\} \\
V_N &= \{\mathit{Expr},\ \mathit{Term},\ \mathit{Factor},\ \mathit{Primary}\}
\end{aligned}$$

■

3.5.2 EBNF

BNF can be extended by introducing additional metasymbols so that production rules can be combined and reduced. There are many different extensions and a common set of symbols along with their meanings is summarized in Table 3.2. The variable α represents any legal right-hand side of a BNF or EBNF rule. One

BNF + Additional MetaSymbols = EBNF	
MetaSymbols Added to BNF	Meaning (α is a EBNF expression)
$[\ \alpha\ ]$	α is optional
$\{\alpha\}$	α can be repeated 0 or more times
$\{\alpha\}^+$	α can be repeated 1 or more times
$(\ \alpha\ )$	Groups α into a EBNF subexpression

Table 3.2: EBNF's Additional Metasymbols

can view the right-hand side of a production rule as an expression with these additional metasymbols as operators. The precedence rules for these additional metasymbol operators are described with the following metagrammar expressed in EBNF in Figure 3.5: From this grammar one can see that the metasymbol, |, has lower precedence than concatenation, which is implicit by the juxtaposition of EBNF expressions. The metasymbol pairs, (), [], and {}, are used to form subexpressions, with the parentheses having no operational meaning; the parentheses, (), simply can alter the precedence rules.

$$\begin{aligned}
<EBNF> &\rightarrow <A> \{ \text{'|'} <A> \} \\
<A> &\rightarrow <B> \{<B>\} \\
<B> &\rightarrow \text{'['}<C>\text{']'} \quad | \quad \text{'\{'}<C>\text{'\}'} \quad | \quad <C> \\
<C> &\rightarrow \text{'('}<EBNF>\text{')'} \mid <terminal> \mid <nonterminal> \\
<terminal> &\rightarrow \underline{\text{string}} \mid \text{""}\underline{\text{string}}\text{""} \mid \text{' '}\underline{\text{string}}\text{' '} \mid \underline{\text{string}} \\
<nonterminal> &\rightarrow \text{'<'}\underline{\text{string}}\text{'>'}
\end{aligned}$$

Figure 3.5: Metagrammar for EBNF written in EBNF

Example 3.11 ***Expression Grammar in EBNF***

In this example we utilize EBNF notation and rewrite Example 3.10.

$$\begin{aligned}
<Expr> &::= <Term> \{(+ \mid -) <Term>\} \\
<Term> &::= <Factor> \{(* \mid /) <Factor>\} \\
<Factor> &::= [+ \mid -] <Primary> \mid \text{'('} <Expr> \text{')'} \\
<Primary> &::= \underline{identifier} \mid \underline{number}
\end{aligned}$$

Note that in the parentheses are quoted since they are metasymbols in EBNF. As with BNF used in Example 3.10, the EBNF allows one to distinguish nonterminal symbols from terminals.

$$\begin{aligned}
V_T &= \{+, -, *, /, (,), identifier,\ number\} \\
V_N &= \{Expr,\ Term,\ Factor,\ Primary\}
\end{aligned}$$

■

3.5.3 Syntax Diagrams

Just as finite state diagrams give a two-dimensional representation of a finite state machine, context free grammars can be depicted visually by utilizing a set of diagrams. Both types of diagrams are intended to aid programmers with understanding the syntactic structure of a given language and to aid compiler implementors with designing parsers, especially top-down parsers, of a given language. Although all context-free grammars can be represented with syntax diagrams, traditionally syntax diagrams have been utilized for designing a particular subset of top-down parsers, called recursive-descent parsers. Before discussing this, it is shown how to represent a grammar G a set of syntax diagrams.

Each syntax diagram represents a group of related production rules, namely ones with the same left-hand side nonterminal. Each group is called a syntactic category. Using the BNF (or EBNF) metasymbol, |, we combine rules that are not already combined as follows. Production rules

$$\begin{aligned}
<A> &\rightarrow \alpha_1 \\
<A> &\rightarrow \alpha_2 \\
&\cdots \\
<A> &\rightarrow \alpha_n
\end{aligned}$$

are grouped into:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

This is done for each nonterminal. Each syntactic category, for example A, is represented by a syntax diagram. The diagram is named by the name of the syntactic category it represents.

Toward a more formal characterization we note that each syntax diagram is a node-labeled, directed graph. We will see that a finite state diagram is a node-labeled, edge-labeled, directed graph. Whereas finite state machine has one corresponding diagram (graph), a grammar is represented by possibly many syntax diagrams (graphs). In a finite state diagram the nodes are labeled with state names, represented as circles; transitions are directed edges that are labeled by a symbol by one symbol from the terminal set. In a syntax diagram there are three types of nodes, junction nodes, terminal nodes, and nonterminal nodes. Conventionally junction nodes are implicit in the drawing, a terminal node is represented by an ellipse and labeled with a terminal (token) symbol, and a nonterminal node is represented by a rectangle and labeled with a nonterminal symbol. All edges in a syntax graph are unlabeled, directed edges.

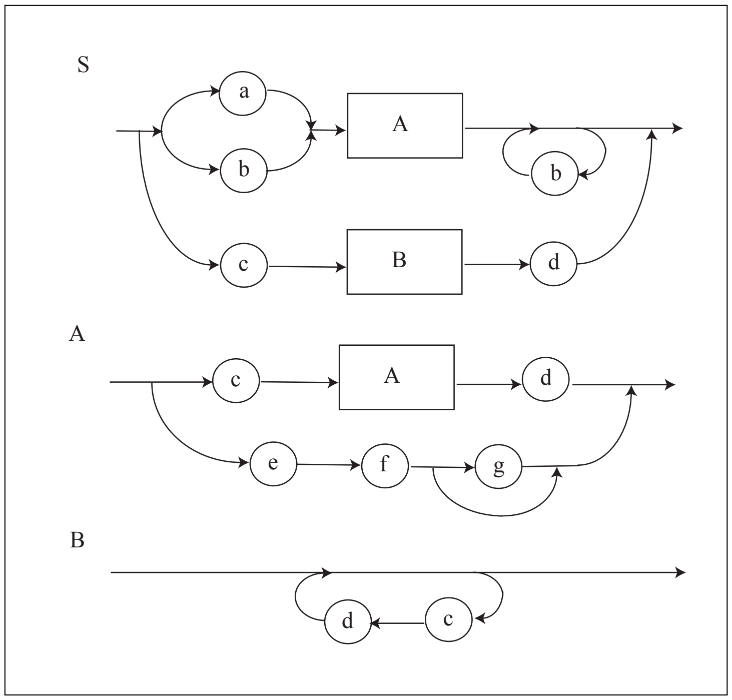
When using a finite state diagram, one can trace a path through the state diagram by reading the input string and following the labels of the transition edges. Parsing the input string requires following a directed path through a syntax diagram and may require jumping between several syntax graphs, each of which correspond to a group of production rules. Informally, when traversing the syntax graph, if one encounters a terminal node, it requires matching the next input symbol. In such a case, this requires moving the input pointer to the next input symbol and continuing down a path in the graph. upon encountering a nonterminal rectangle, one jumps to the syntax graph named by that nonterminal (maybe even the same graph in the case of a direct recursive rule); after traversing that graph and, possibly several others, one returns and continues on the edge emanating from the rectangle. One can convert each syntactic category into a syntax graph by putting each right-hand side as a branch at a junction node. Each right-side is a sequence of EBNF elements: terminals, nonterminals, optional elements, iterative elements, and nested alternative elements. The junction nodes represents forks in the road where decisions a have to be made or they represent the merging of two or more paths.

Example 3.12 ***Syntax Graph of a Simple Graph***

Consider the grammar G with start symbol E, written in EBNF:

$$\begin{aligned}
<S> &\rightarrow (a|b) <A> \{b\}|c <B> d \\
<A> &\rightarrow c <A> d|ef[g] \\
<B> &\rightarrow \{cd\}
\end{aligned}$$

The corresponding syntax graphs is:



To trace a path of edges in syntax graph is equivalent to parsing an input string. For example, the string ccdcdd may be parsed by tracing a path through the syntax diagrams. One starts with graph S. The first symbol c matches the terminal node labeled c and moves past that node to the rectangle node labeled with nonterminal B and the input pointer advances to the next symbol in the string, which is the second c. Because a rectangle B is encountered, one jumps to syntax diagram B. There is a path that can be taken that goes through a fork junction node and leads to a match with another terminal node c. Continuing further down the input string d is read and is matched against the next terminal node labeled d. One continues around passing through the merge junction node and then passing through the fork junction node to match the next symbol c, then d. Finally, when reading the last symbol of the input string d, there is no match at the fork junction point the B diagram is exited back to the diagram from which the jump originated. In this case, that diagram of S and one jumps back to the backside of the B rectangle and a path is traced from there. A terminal node d is matched and the string is accepted since the starting graph can be exited. ■

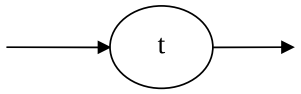
To convert a set of production rules written in EBNF notation, intermediate graphs are created. Before beginning the conversion, the EBNF grammar is put into a "canonical form" by grouping all context-free productions with same left-hand symbol. For each nonterminal A, group all rules with A as the left-hand side in following form:

$$A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

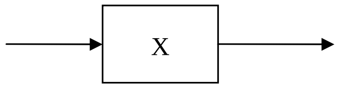
Let the group be called a *syntactic* category A. The right-hand side of each production consists of an EBNF expression, which in turn, consists of subexpressions. In the following algorithm, a syntax graph is recursively constructed from each production's right-hand side EBNF expression. The base case is represented by converting terminals and nonterminals to their respective graphs.

1. For each syntactic category, A, with production rule, $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \alpha_n$, let γ represent the right-hand-side EBNF expression and recursively construct syntax diagram, named A, according to rules 2-8 by converting the underlying subexpressions of γ as structured by the metagrammar for EBNF.

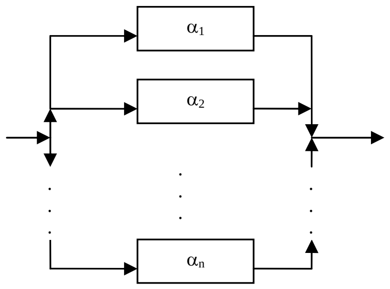
2. if $\gamma = t$, where t is a terminal, let the diagram be represented as:



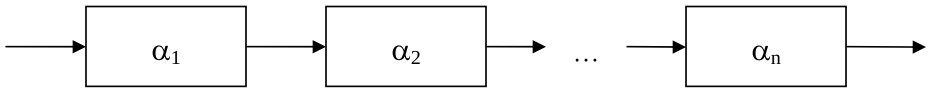
3. If $\gamma = X$, where X is a nonterminal let the graph be represented as:



4. If $\gamma = \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, a syntax graph for γ is constructed as follows.

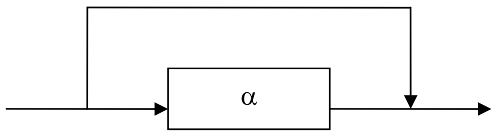


5. If $\gamma = \alpha_1\alpha_2 \ldots \alpha_n$, a syntax graph for γ is constructed as follows.



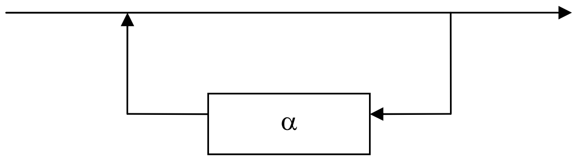
Each EBNF subexpression, α_i, are converted to syntax graphs, which are represented as $\boxed{\alpha_i}$, by recursively applying rules 2-8.

6. If $\gamma = [\alpha]$, a syntax graph is constructed as follows.



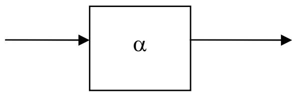
EBNF subexpression, α, is converted to a syntax graph by recursively applying rules 2-8. The syntax graphs is represented here as: $\boxed{\alpha}$.

7. If $\gamma = \{\alpha\}$, a syntax graph is constructed as follows.



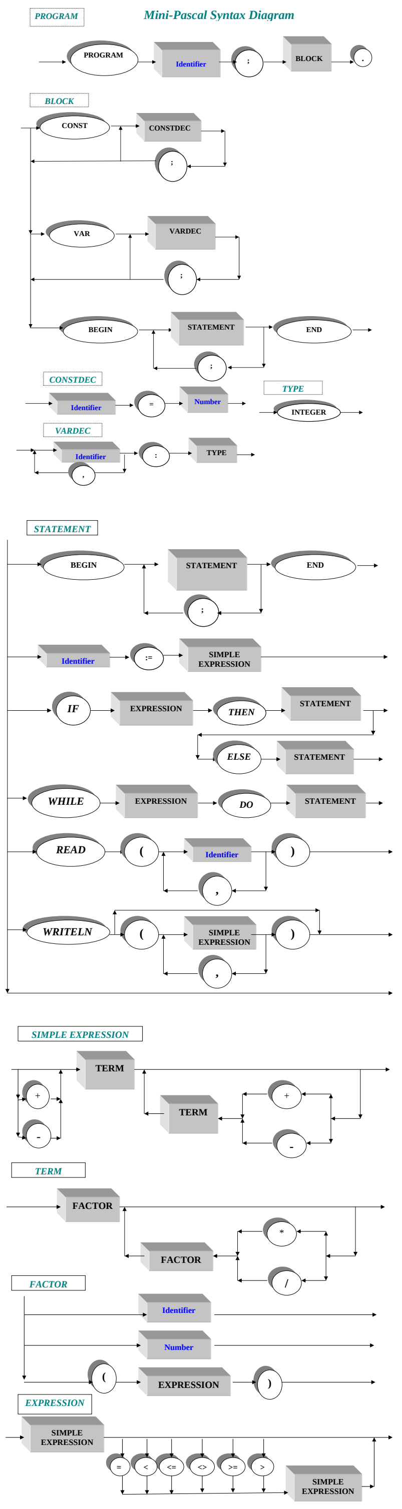
EBNF subexpression, α, is converted to a syntax graph by recursively applying rules 2-8. The syntax graphs is represented here as: $\boxed{\alpha}$.

8. If $\gamma = (\alpha)$, a syntax graph is constructed as follows.



EBNF subexpression, α, is converted to a syntax graph by recursively applying rules 2-8. The syntax graphs is represented here as: $\boxed{\alpha}$.

An example of a grammar for a subset of Pascal expressed as a syntax diagram is shown below:


Mini-Pascal Syntax Diagram
PROGRAM
PROGRAM
Identifier
;
BLOCK
.
BLOCK
CONST
CONSTDEC
;
VAR
VARDEC
;
BEGIN
STATEMENT
END
;
CONSTDEC
Identifier
=
Number
TYPE
INTEGER
VARDEC
Identifier
,
:
TYPE
STATEMENT
BEGIN
STATEMENT
END
;
Identifier
:=
SIMPLE EXPRESSION
IF
EXPRESSION
THEN
STATEMENT
ELSE
STATEMENT
WHILE
EXPRESSION
DO
STATEMENT
READ
(
Identifier
,
)
WRITELN
(
SIMPLE EXPRESSION
,
)
SIMPLE EXPRESSION
+
-
TERM
TERM
+
-
TERM
FACTOR
FACTOR
*
/
FACTOR
Identifier
Number
(
EXPRESSION
)
EXPRESSION
SIMPLE EXPRESSION
=
<
<=
<>
>=
>
SIMPLE EXPRESSION