

Chapter 2

Syntax

Grammar:
that part of the study of language which deals with
form and structure of words (*morphology*)
and with their customary arrangement in
phrases and sentences (*syntax*),
usually distinguished from
the study of word meanings (*semantics*).
– Webster

When something is written down there must be an agreement between writers and readers so that they understand each other. The agreement includes the form and the meaning of what is written. The form can be dealt with directly; the meaning is defined in terms of the form. This chapter is concerned with form. The next chapter is concerned with meaning. The presentation covers the following topics:

- the description of grammars
- the parse, which results from applying a grammar to a text
- the relation of a parse to parenthesis-free notation
- applications of grammars, including
 - probabilistic grammars
 - grammar grammars
 - executable grammars
- regular expression grammars
- lexical structure and scanners
- recursive descent parsing
- grammars for X

The viewpoint will alternate between that of the reader of syntax and the writer of syntax.

2.1 Grammars and Languages

All things are lawful unto me; but all things are not expedient.

—*The Apostle Paul*

A grammar is a means for defining correct usage of a language. In English, for example, one would expect to find the fact that an adjective must precede the noun it modifies, as in “red shirt”, expressed in the English grammar. Beginning users of a language often find grammatical rules pedantic and constraining. A child may say “shirt red” and expect the understanding parent to figure out what is meant. One can build “understanding” programs too, but the payoff is short lived; mature users of computer languages prefer to be precise and expect the computer to reject imprecise input.

Grammars present obstacles to language designers as well as language learners. Getting a grammar “just right” is a challenge akin to poetics. The cost of using grammars is somewhat more care in writing things down; the intellectual benefit is a sense of elegance and an organizing principal for dealing with languages; the practical benefit is fewer misunderstandings and availability of computer programs to deal with grammars.

Given a grammar and a text, there are several important questions that can be answered:

- Is the text in the language defined by the grammar? This is the same as asking if the text is grammatically correct (with respect to the grammar, of course). It takes some work to answer this question. There are algorithms for this work. The implementation of such an algorithm is called a *recognizer*. Given a text, a recognizer will answer ‘yes’ or ‘no’, depending on whether the text is in the language or not. If the answer is ‘yes’ the text is *accepted*, otherwise it is *rejected*.
- If the text is rejected, where does it first fail to be grammatical? Being able to answer this question is essential for producing diagnostic information for a user who has attempted, and failed, to write correct text.
- If the text is accepted, what is its structure? The process of answering the recognition question ‘yes’ can be expected to reveal the grammatical structure of the text. Analysis of the text to discover its meaning is greatly aided by knowing the structure. A recognizer that also reveals the structure is called a *parser*. Understanding, designing and constructing parsers is the main goal of this chapter.

A grammar contains of a set of rewriting rules. A simple recognizer might repeatedly try to apply every rule in every possible way to some text. If some sequence of rewritings succeeds, by criteria to be explained below, the text is accepted. The record of the rule application sequence leading to success is called a *parse*. The parse allows anyone to repeat the rewritings and therefore check the acceptance.

Practical parsing has two essential properties: it is efficient and the parse leads to a straightforward definition of meaning. The grammar writer needs to develop skills to achieve both properties simultaneously.

One should not infer that grammars are the only way to define language — rather grammatical description is a constraint on definitional expressiveness accepted by programming language designers because the advantages of grammars outweigh their limitations. Grammars are typically “loose” definitions in that only some of the defined texts are meaningful.

Grammars are an interesting topic in themselves. There are many kinds. This chapter and Chapter 5 present grammars that are used in the construction of compilers. More complete presentations of grammars can be found in texts on Automata Theory.[?]

Formal Definitions

For any grammar there are two finite sets of symbols called its *terminal* and *non-terminal* vocabularies. A *text* is a sequence of terminal symbols. Nonterminal symbols are used to describe structure.

It takes four things to fully define a grammar \mathcal{G} :

$$\mathcal{G} \stackrel{\text{def}}{=} \langle V_T, V_N, G, \Pi \rangle \quad (2.1)$$

V_T is the set of terminals. V_N is the set of nonterminals. G is a nonterminal symbol called the *goal*.¹ Π is a set of rewriting rules. The *vocabulary* is the union of terminal and nonterminal vocabularies.

$$V \stackrel{\text{def}}{=} V_T \cup V_N$$

Formally, the 4-tuple $\langle V_T, V_N, G, \Pi \rangle$ is a *context-free grammar*² if

$$\begin{aligned} V_T \cap V_N &= \{\} \\ G &\in V_N \\ \Pi &\subseteq V_N \times V^* \end{aligned}$$

The members of Π are pairs: each consisting of a nonterminal on the left and a sequence of terminals and nonterminals on the right. We might write a pair

$$\langle \text{sentence, subject predicate} \rangle$$

to display a rule defining a sentence in English. The right member of the pair is an acceptable structure for the nonterminal on the left. In this case, a sentence may be a subject followed by a predicate. Each nonterminal typically has several rules, each defining one possible structure for texts described by that nonterminal.

¹In natural language grammars the goal is typically *sentence* or S; in programming languages the goal is typically *program* or P; here we often use G.

²Context-free grammar is abbreviated CFG.

The rules may be repeatedly applied to a text, rewriting the text with each application. If, somewhere in a text, one finds a sequence of symbols exactly matching the right-hand side of a rule in the grammar, that rule may be applied. The sequence is removed from the text and replaced by the nonterminal on the left of the rule. The removed sequence is a *phrase* and the nonterminal replacing it is its *phrase name*.

As one continues to apply rules to a text, the text gets shorter and contains phrase names and ever fewer of the original terminal symbols. The process terminates when no more rules can be applied. The process succeeds (a different matter) when all that is left in the text is G , the goal symbol of the grammar. The parse is the record of rule applications that is needed for semantic analysis.

Everyday Notation for Grammars

Proposition	
Disjunction	$r0$
Disjunction	
Disjunction \vee Conjunction	$r1$
Conjunction	$r2$
Conjunction	
Conjunction \wedge Complement	$r3$
Complement	$r4$
Complement	
\neg Boolean	$r5$
Boolean	$r6$
Boolean	
t	$r7$
f	$r8$
(Disjunction)	$r9$

Table 2.1: A CFG for Propositions

In day-to-day use of grammars, one writes down Π and infers V_T , V_N and G . The ordered pairs appear cluttered, so we pick a visually more attractive notation.

By convention, the first rule of Π defines G . It is typically the case that G is also used only on the left of rules and there is no other symbol used only on the left. Thus G is also called the *leftmost symbol* of the grammar. The nonterminal vocabulary is the set of symbols used on the left of the rules of Π . Since Π is a set of pairs, we can write down the relation $V_N = \mathcal{D}(\Pi)$. The right sides of the rules use all the terminal symbols, thus terminal vocabulary V_T is related to Π by $V_T = \mathcal{R}(\Pi)^{1/*} - V_N$.

Since there are often several rewriting rules defining one nonterminal, it is convenient to write those rules together when displaying Π , to avoid visually repeating the left side of the rule. And it is often helpful to give each rule a

short unique name for easy reference later. In fact the numerical position in the list representing Π is often used as a short name for each rule in the internal workings of compilers.

In all of the grammar notations to be introduced in this chapter, there are *metasymbols* which must be distinguished from symbols in V_T and V_N .

Table 2.1 presents a practical notation. The nonterminals show up on the left margin. Subsequent lines give the right sides of the rules for that nonterminal. An empty right-hand-side is represented by a blank line.³ The short rule names at the far right are not a part of the grammar, but rather more like comments in a programming language. In this case there are ten rules defining well-formed formulas⁴ in the propositional calculus.

There are other notations for grammars. The definition of Algol 60[?] introduced the popular notation called Backus-Naur Form⁵ in which the nonterminal symbols are bracketed (e.g. (arithmetic expression)) so that phrase names can consist of more than one word; the metasymbol ‘::=’ stands for “is defined as” and ‘|’ for “or is defined as.” BNF is not much used because it is cluttered and also poses some technical difficulties as an input language. The ISO standard for C uses ‘:’ for “is defined as”, end-of-line for “or is defined as”, and different type fonts for V_T and V_N . It looks much like Table 2.1.

Exercises

1. [1,1] Define terms recognizer, accepted, rejected, parser, terminal and non-terminal vocabularies, phrase name, goal, context-free grammar parse, CFG, metasymbol.
2. In terms of the CFG in Table 2.1:
 - (a) [1,50] Does it make sense to write
 $\Pi = \{r0, r1, r2, r3, r4, r5, r6, r7, r8, r9\}$?
 - (b) [1,50] Does it make sense to write $(\text{Complement}, \neg\text{Boolean}) \in \Pi$?
 - (c) [1,50] What are V_T , V_N , G and Π ?
 - (d) [1,1] The text $f \vee t$ is a WFF. One can show this fact by applying rules $r8, r6, r4, r2, r7, r6, r4, r1$ and $r0$ in that order. Do it. Give a “rule of thumb” for finding such a sequence of rules for this grammar. Apply it to text $t \wedge f$. Try numerous other examples.
 - (e) [1,1] What is the effect of replacing rule $r5$
 Complement
 \neg Boolean
 with rule

³The empty right-hand-side is usually not the last alternative; it would be too easy to confuse the blank line with mere whitespace used for readability. There are no empty rules in the CFG for propositions, however look ahead to Exercise 2.3.

⁴Well-formed formula is abbreviated WFF.

⁵Backus-Naur Form is abbreviated BNF

Complement

\neg Complement

- (f) The text “tt” is not a WFF. What happens when you try to rewrite it? What is the best diagnostic you can imagine to give to the author of “tt”? What is the best you might expect from a recognizer?

3. [1,50] Write down an everyday grammar for

$$\mathcal{G} = \langle \{b, c\}, \{G, A, B\}, G, \{ \langle G, A \rangle, \langle A, Bb \rangle, \langle A, Bc \rangle, \langle B, \rangle \} \rangle$$

4. [1,1] Using the grammar in Table 2.1 as a guide, write a grammar describing instead arithmetic expressions (digits, +, −, ×, /). Use it to parse

$$1 + 2 \times (3/4) - 5$$

What you should have learned

Formal and informal definition of CFG. The concept of rewriting and phrase structure. A specific grammar for propositions. A practical notation for grammars.

2.2 The Parse

For the body does not consist of one member, but many.
—*The Apostle Paul*

While it is easy enough to show small examples of rummaging about in a text, finding phrases, applying rules, and completing a rewriting sequence, one needs to be systematic for practical parsing. The goal is to produce a parse automatically, with consistency and efficiency for a text of any size.⁶

Production and Reduction

If $\langle B, \beta \rangle \in \Pi$, one can write

$$\alpha B \gamma \xrightarrow{\langle B, \beta \rangle} \alpha \beta \gamma$$

for any strings α and γ . One says that the text $\alpha B \gamma$ *immediately produces* text $\alpha \beta \gamma$ by rule $\langle B, \beta \rangle$. If one writes

$$\sigma \xrightarrow{r} \mu$$

⁶Notation: Recall from Chapter 1 that Greek lower case letters $\alpha, \beta \dots$ are used to designate sequences. The letter λ is used to denote the empty sequence. λ is therefore used to denote an empty right-hand-side of a rule. Most sequences in this chapter are in fact strings of symbols.

one means that σ immediately produces μ by an application of rule r . That is:

$$\sigma \xrightarrow{r} \mu \stackrel{def}{=} \exists \alpha \beta \gamma B. \sigma = \alpha B \gamma \wedge \mu = \alpha \beta \gamma \wedge r = \langle B, \beta \rangle \quad (2.2)$$

The symbol \rightarrow is called a *production arrow*. Used by itself \rightarrow means that there exists some rule with which to do the rewrite:

$$\sigma \rightarrow \mu \stackrel{def}{=} \exists r. \sigma \xrightarrow{r} \mu \quad (2.3)$$

The production arrow defines a relation on $V^* \times V^*$. This relation is the basis for parsing methods.

Exercise

5. [1,50] List a dozen or so strings related by \rightarrow for the grammar for propositions.

Sentential Forms and Language

<i>predicate</i>	<i>axiom</i>
$SF(\text{Proposition})$	
$SF(\text{Disjunction})$	r0
$SF(\text{Disjunction} \vee \text{Conjunction})$	r1
$SF(\text{Disjunction} \vee \text{Conjunction} \wedge \text{Complement})$	r3
$SF(\text{Disjunction} \vee \text{Conjunction} \wedge \neg \text{Boolean})$	r5
$SF(\text{Disjunction} \vee \text{Conjunction} \wedge \neg f)$	r8
$SF(\text{Disjunction} \vee \text{Complement} \wedge \neg f)$	r4
$SF(\text{Disjunction} \vee \text{Boolean} \wedge \neg f)$	r6
$SF(\text{Disjunction} \vee t \wedge \neg f)$	r7
$SF(\text{Conjunction} \vee t \wedge \neg f)$	r2
$SF(\text{Complement} \vee t \wedge \neg f)$	r4
$SF(\text{Boolean} \vee t \wedge \neg f)$	r6
$SF(f \vee t \wedge \neg f)$	r8

Table 2.2: A Sentential Form Proof

One can regard a grammar as an abbreviation of an axiom system in which to prove things about strings of symbols. There is a predicate SF which defines the so-called sentential forms of the grammar. SF is defined by two axioms. Suppose the grammar is $\mathcal{G} = \langle V_T, V_N, G, \Pi \rangle$. The first axiom is for *rewrites*

$$\sigma \rightarrow \mu \Rightarrow (SF(\mu) \Leftrightarrow SF(\sigma))$$

The second axiom is the *basis*

$$SF(G)$$

The basis says that the goal symbol, by itself, is a sentential form. The rewrite axiom allows one to show texts other than G are also sentential forms, only if

they are reached from G by applying grammar rules. The rewrite axiom can be applied to replace a phrase name by a phrase or — opposite the direction of the production arrow — to replace the phrase by its name.

Alternatively one may define SF directly from \rightarrow . The reflexive transitive completion of relation \rightarrow is written \rightarrow^* . If $\alpha \rightarrow^* \beta$ one says α *produces* β .⁷

$$SF(\beta) \Leftrightarrow G \rightarrow^* \beta$$

Table 2.2 can be read in two directions. From top to bottom it shows phrase names (from Table 2.1) being replaced by phrases. From bottom to top, the phrases are replaced their names. The downward direction is called *production*; the upward direction is called *reduction*. One could get into a loop of applying and disapplying rules. In compilers, only the direction for reductions is used.

Since the grammar for Proposition defines WFFs, Table 2.2 gives a proof that $f \vee t \wedge \neg f$ is a WFF. Each time a rewrite axiom is used, its existential quantifier is instantiated with a rule (see Equation 2.3); that rule is noted between the lines of the proof it establishes.

The string β is in the *language*, $\mathcal{L}(\mathcal{G})$, defined by grammar \mathcal{G} if $SF(\beta)$ is provable and β is a string of terminal symbols.

$$\mathcal{L}(\mathcal{G}) \stackrel{\text{def}}{=} \{\beta \mid SF(\beta)\} \cap V_T^*$$

Exercises

6. [1,1] Define terms sentential form, rewrite and basis axioms, production, reduction, language.
7. [1,50] Is $G \in \mathcal{L}(\mathcal{G})$ for any \mathcal{G} ?
8. [1,50] Write a grammar \mathcal{G} for which $\lambda \in \mathcal{L}(\mathcal{G})$.
9. [1,50] For several values, $n = 0, 1, 2, \dots$, write a grammar for which $\text{size}(\mathcal{L}(\mathcal{G})) = n$.
10. [1,50] Solve the previous two problems simultaneously.
11. [1,50] Write a grammar for which $\text{size}(\mathcal{L}(\mathcal{G})) = \infty$.
12. [1,50] Say how you would write a grammar for which the only valid string is Lincoln's Gettysburg address.

The Canonical Parse

Equation 2.2

$$\sigma \xrightarrow{r} \mu \stackrel{\text{def}}{=} \exists \alpha \beta \gamma B. \sigma = \alpha B \gamma \wedge \mu = \alpha \beta \gamma \wedge r = \langle B, \beta \rangle$$

⁷See Table 1.6.

describes the parse step. If, in the production step from $\alpha B \gamma$ to $\alpha \beta \gamma$, the string γ consists only of terminal symbols (i.e., $\gamma \in V_T^*$), then B , the target of the substitution, is the *rightmost nonterminal*. Since it does not restrict future choices in any way to choose an arbitrary nonterminal in $\alpha B \gamma$ for expansion, one loses nothing by systematically choosing the rightmost nonterminal at every step. Such an order of rule application is called *canonical*. The reverse of a canonical sequence of production steps is a *canonical parse*.

It is harder to see that the reduction step reversing a canonical production step is the *leftmost* substitution that will lead to a canonical parse. One is always reaching ahead into the text for the next phrase to be reduced; a canonical step minimizes how far ahead one reaches. It is possible to reach too far and still satisfy $\gamma \in V_T^*$. Eventually, however, such a misstep will catch up with you, forcing a substitution to the left of some nonterminal. The principal parsing problem is finding each leftmost string β leading to a canonical sequence of parse steps. Formally the sequence

$$\rho = \langle r_0, r_1, \dots, r_k \rangle$$

is a canonical parse of β if

$$G \xrightarrow{r_k} \alpha_k \xrightarrow{r_{k-1}} \dots \xrightarrow{r_0} \beta$$

and for each step $\gamma \in V_T^*$. Much like an earlier convention one can write

$$G \xrightarrow{\rho}^* \beta$$

and mean that ρ is a canonical parse of β .

There are several reasons to single out the canonical parse. It is the order that naturally occurs when parsing a text (from left to right, of course). It allows a compact linear description of a parse (without having to identify the place each rule was applied). It leads to efficient parsing. It is the basis for the definition of syntactic ambiguity. It is the linchpin between syntax and semantics. Each of these facts will be further discussed below.

Exercises

13. [1,50] Using Table 2.1, give two parses for $t \vee t$, one canonical and one not. Make up more strings to parse and find the canonical parse for each.
14. [1,50] Using Table 2.1 and Table 2.2
 - (a) On each line of the proof, identify the rule and all of the quantified variables in Equation 2.2. E.g., on the r_0 line, $\alpha = \gamma = \lambda$, $B =$ Proposition, $\beta =$ Disjunction, $r = r_0$.
 - (b) Verify that the derivation in Table 2.2 is canonical.
 - (c) Prove that each of t , f , (t) and $t \vee (f)$ are Propositions.

- (d) Show that $f(\vee)t$ cannot be a Proposition.
- (e) Prove $SF(\text{Disjunction } \vee t)$
15. [1,50] Write down $\mathcal{L}(\mathcal{G})$ where $\Pi = \{\langle G, A \rangle, \langle A, Bb \rangle, \langle A, Bc \rangle, \langle B, \lambda \rangle\}$
16. [1,50] Find two different grammars \mathcal{G} , each describing $\mathcal{L}(\mathcal{G}) = \{ab, ba\}$
17. [10,50] Find a grammar \mathcal{G} describing all sequences of a's and b's with an odd number of a's.

Parse Trees and Operator Precedence

Order is a lovely thing;
 On disarry it lays its wing,
 Teaching simplicity to sing.
 — *Anna Hempstead Branch*

The grammar in Table 2.1 has another interesting property: it determines the associativity and precedence of the boolean connectives. As one can see by referring to rule $r1$, the operator ‘ \vee ’ acts on the Disjunction to its left and the Conjunction to its right.

Disjunction		
Disjunction \vee Conjunction		$r1$

Intuition demands that whatever needs be done to compute the operands of ‘ \vee ’ must be completed before this ‘ \vee ’ itself can operate. The parse has just this property: before rule $r1$ can be applied, all the rules for the left and right nonterminals must have already been applied. The implication is that all other occurrences of ‘ \vee ’ in the phrase Disjunction have already been evaluated (left associativity) and all occurrences of ‘ \wedge ’ and ‘ \neg ’ to the left and right of ‘ \vee ’ have already been evaluated (‘ \neg ’ and ‘ \wedge ’ have higher precedence than ‘ \vee ’).

This relation is seen graphically in the *parse tree* for $f \vee t \wedge \neg f$ in Figure 2.1. Each node is labeled with a phrase name; the arrows point to the symbols constituting the phrase. The root node is always labeled with the goal symbol of the grammar; the leaf nodes are always terminal symbols. Each set of arrows can be labeled, in addition, with the rule that is applied. Whenever a string can be parsed, it has a parse tree. In some compilers the parse tree, or some abstraction of it, is used as an internal representation of the program during compilation.

The canonical parse is the order of rule applications from a postorder walk of the parse tree and also the order that occurs naturally when processing an input text from start to finish (left to right, if you prefer).

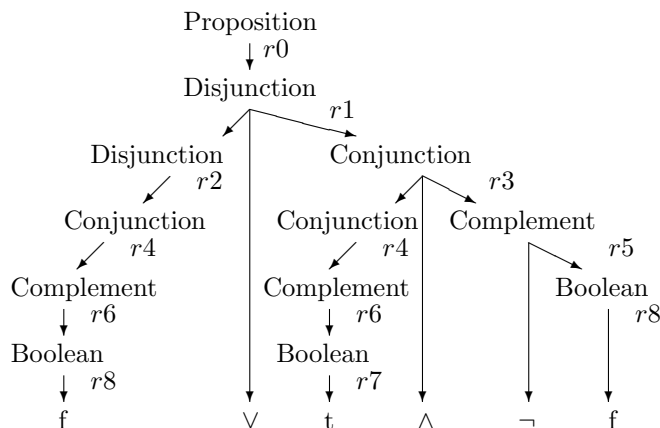


Figure 2.1
Parse Tree for $f \vee t \wedge \neg f$

Using grammar to define precedence and associativity is a well-established convention, both for readers of grammars and implementors of compilers. A left-recursive rule, such as $r1$, implies left associativity. A right-recursive rule, such as the following one for exponentiation

$$\begin{array}{l} \text{Factor} \\ \text{Primary} \uparrow \text{Factor} \end{array}$$

implies right associativity; the expression $2 \uparrow 3 \uparrow 4$ means $2 \uparrow (3 \uparrow 4)$. This is a significant choice since the alternative interpretation, $(2 \uparrow 3) \uparrow 4$, would have the same effect as the simpler formula $2 \uparrow (3 \times 4)$. Which rule of associativity the language designer *should* choose for exponentiation is controversial. One choice gives consistency of form while the other makes the default (no parentheses) more interesting.

Exercises

18. [1,1] Define terms associativity, precedence, parse tree.
19. [1,1] Redraw the parse tree in Figure 2.1 replacing the phrase names with the name of the rule that was applied. Since the phrase name is the left side of the rule, no information has been lost. This labeling is less mnemonic, but more useful and less verbose, than the one using phrase names.
20. [1,1] What associativity rule is used for exponentiation in PL/1? Why?
21. [1,50] Using Table 2.1, draw parse trees for $((t))$ and $f \vee t \wedge (t \vee f)$.

22. [20,50] Devise a grammar for $+$, $-$, \times , $/$, and \uparrow (meaning the four arithmetic functions plus exponentiation) so that the four arithmetic functions are left associative, exponentiation is right associative and there are three levels of precedence (\uparrow over \times and $/$, and these over $+$ and $-$). Restrict the operands to single digits. Draw parse trees for several expressions. How does the result compare with your answer to Exercise 4.
23. [10,10] Repeat the previous exercise adding parentheses to the language.
24. [10,10] Repeat the previous exercise adding unary ‘ $-$ ’ to the language. Decide what precedence unary ‘ $-$ ’ should have. Can you justify your choice? Is it consistent with PASCAL or C?
25. [10,10] Verify that a postorder walk of the tree in Figure 2.1 gives the canonical parse for $f \vee t \wedge \neg f$.
26. [50,10] Prove that a postorder walk of a parse tree always gives the canonical parse.
27. [1,1] Describe an algorithm to build the parse tree from the canonical parse and the grammar. Could you implement it?
28. [20,10] Show that for any CFG, each subtree of a parse tree corresponds to a contiguous subsequence of the canonical parse.
29. [20,10] Extend the previous exercise to show that any rule r appearing in a canonical parse is immediately preceded by m contiguous subsequences, one for each nonterminal on the right-hand of the rule.
30. [10,10] Using the results of the previous exercise, design a space-efficient computer representation of parse trees. Is the canonical parse itself a solution?

What you should have learned

The relation between CFG, canonical parse, parse tree and the structure of arithmetic expressions.

Parenthesis-free Notation

Exp	Pfn
Pfn	Pfn Pfn Op2
	Pfn Op1
	t
	f
Op2	

	∨
	∧
Op1	¬

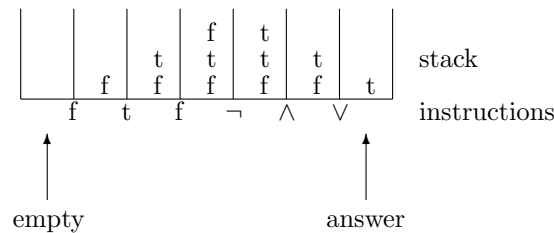
Table 2.3: A CFG for PFN

The grammar in Table 2.3 describes a parenthesis-free notation (PFN) for logical expressions.⁸ For every expression there is a parenthesis free form. PFN has been used as a basis for instruction codes for stack computers, and for interpreters that use a pushdown store for the evaluation of arithmetic expressions. As executable instructions, PFN has two rules: operands get stacked; operators get executed on the values on the top of the stack. The single value remaining on the top of the stack after the PFN is executed is the value of the expression. The origin of the grammar in Table 2.3 should not trouble the reader. Like that for Proposition in Table 2.1, it is a notation out of mathematics.

In the grammar in Table 2.3 there are two kinds of operators: one Op1 that pops a single value off the stack and replaces it with a new value and two Op2 operators that pop off two values and replace them with a single new value. In this case Op1 is used for what is a prefix operator in conventional logical expressions and Op2 is used for what are infix operators in conventional logical expressions. PFN is not limited to prefix and postfix operators. It is also reasonable to add postfix operators to Op1 (the * in V^* is a postfix operator). One can add a class Op3 for if-then-else, and so on. PFN is a flexible concept.

The PFN is used in this book, and in HYPER, as an internal representation of the program being compiled.⁹

For example the WFF $f \vee t \wedge \neg f$ has the same meaning as PFN $ftf\neg\wedge\vee$. Used for execution, the PFN leads to the sequence of stack states depicted in Figure 2.2. PFN instructions are displayed between and below the stack transitions they cause. The original stack state is empty. The final stack state contains a single value.



If the canonical parse for WFF $f \vee t \wedge \neg \vee$ is displayed together with the operator and operand symbols that are consumed by each rule, PFN appears. From Table 2.2 we get

r8	r6	r4	r2	r7	r6	r4	r8	r5	r3	r1	r0
f			t			f	\neg	\wedge	\vee		

This is not as magical as it might seem. Operands appear in the same order in conventional text and PFN. Operators are delayed (forced right of their operands) because the operands are consumed in phrase classes that must be completely parsed before the phrase class containing the operator. Precedence and associativity are decided by the grammar writer.¹⁰ Magical or not, having the PFN imbedded in the canonical parse allows an elegant simplification to the problem of compiling expressions.

Parentheses cloud this elegant picture a bit. One can either ignore them or make them into an Op1 operator that pops the stack and pushes the value unchanged right back onto it. That is: Op1 ‘()’ is a no-op.

Exercises

31. [1,50] Compute PFN for $((t))$ and $t \wedge (t \vee f)$ from the canonical parse (use Table 2.1). Draw the parse tree.
32. [1,50] Compute PFN for $Pfn\ tftt \vee \wedge \neg f \wedge \wedge$ from the canonical parse (use Table 2.3). Draw the parse tree. What is an equivalent WFF? Is it unique?
33. [1,1] Use a stack to evaluate PFN $tftt \vee \wedge \neg f \wedge \wedge$. Try some other examples.
34. [1,1] Show, by induction on the CFG for PFN (Table 2.3), that the number of infix operators is always one less than the number of t/f operands.
35. [1,1] Use the result of the previous exercise to prove that exactly one value remains on the stack after evaluation, thereby insuring the integrity of the stack and the uniqueness of the value of a PFN expression.
36. [1,1] Add a nonterminal ‘Conditional’ for if-then-else to propositions. It should describe proposition
 if if t then f else t then f else t
 among others. Add an Op3 operator ‘?’ for if-then-else to the PFN CFG. Make up some examples to turn into PFN and execute.
37. [1,1] Remove operands t and f from the CFG for PFN and add a phrase name Op0 to the CFG for PFN with two zero-ary operators t and f. As the phrase name indicates, these new PFN operators do not pop anything from the stack. Like all PFN operators they push a single result on the stack. Now repeat Exercise 2.31 and the one following it. Does it change anything to have only operators in PFN? Extend the stack-integrity proof to PFN containing operators OpN from 0 up to some arbitrary value of N.

¹⁰See page 44.

38. [1,1] Substitute 1 for t and 0 for f in both the PFN and Proposition CFGs. This changes nothing. Now generalize 0 and 1 to binary integers and the operators to act bitwise. What is the relation between the two languages and stack machines?
39. [1,1] Continuing the previous exercise, add a postfix operator ‘.’ to propositions and a corresponding Op1 operator to the PFN. Add a conventional memory to the stack machine and interpret the ‘.’ operator as ‘fetch from memory’. The effect is to add variables to propositions, using memory addresses as operands.

What you should have learned

The relation between PFN and the canonical parse. The relation between PFN and stack machines. A way to evaluate expressions. A foreshadowed version of variables in expressions.

2.3 Applying Grammars

Nonstandard Grammars

Not all context-free grammars are useful and not all useful grammars fit within the constraints of context-free. The purpose of this section is to describe some variants of the CFG, both generalizations and restrictions. Each of these variants has some value. The commonest situation is a restriction that does not hurt much and makes some other part of the task easier.

It is sometimes the case that there are two (or more) grammars sharing most of their rules. One convenient way to express this situation is to allow more than one goal symbol G . The formalism would replace G with V_G , each element in it defining a separate language.¹¹

Another way to share a single grammar representation is to have more than one terminal vocabulary V_T, V'_T, \dots . If all but one set of terminals is ignored, the grammar is just a CFG. If each set of terminals comes from a different input stream, the grammar describes concurrent languages. If one set of terminals is regarded as an output stream and another as an input stream, the recognition of one language will produce a string in a second language. This latter interpretation allows one to build entirely syntax-directed translation over some small range of languages; it is an organizing concept of the remainder of this section.

¹¹See Equation 2.1 and a similar concept denoted V_F in the treatment of regular expressions in Chapter 5.

$\Pi \triangleright \{\lambda\} = \{\}$	no erasure
$A \not\rightarrow^+ A\alpha$	no left recursion
$A \not\rightarrow^+ A$	no circularity
$\{\alpha \mid A \rightarrow^* \alpha\} \cap V_T^* \neq \{\}$	no nonterminating symbols
$\{\alpha \mid G \rightarrow^* \alpha\}^{1/*} = V$	no unreachable symbols
$G \xrightarrow{\rho} \alpha \wedge G \xrightarrow{\sigma} \alpha \Rightarrow \rho = \sigma$	unambiguous

Table 2.4: Possible Restrictions on CFGs

Staying within the formal definition of CFG, there are some restrictions that are commonly accepted.

A rule with an empty right-hand side is said to *erase*. (recall Exercise 2.3) Avoiding erasure simplifies the task for some kinds of parsers. It is, therefore, a common restriction of convenience to parser implementors. Any grammar with erasure can be rewritten without erasing rules,¹² so it would seem to not be of much consequence to forbid erasure to the grammar writer. In fact erasure is of use merely for conciseness, and also when matching the grammar to the semantics, as will be seen in Chapter 3.

To remove erasing from Π it is helpful to define the erasing set

$$V_\lambda \stackrel{\text{def}}{=} \{A \mid A \rightarrow^+ \lambda\} \quad (2.4)$$

which contains the nonterminals that can produce the empty string. The erasing set and the erasure-free set of rules Π' can be computed together. The following iteration converges rapidly.

$$\begin{aligned} P_0 &= \Pi \\ P_{i+1} &= P_i \cup \{(B, \alpha\gamma) \mid A \rightarrow \lambda \wedge B \rightarrow \alpha A \gamma\} \\ P_\infty &= \lim_{i \rightarrow \infty} P_i \\ V_\lambda &= \mathcal{D}(P_\infty \triangleright \{\lambda\}) \\ \Pi' &= P_\infty \triangleright \overline{\{\lambda\}} \cup G \triangleleft P_\infty \end{aligned}$$

Table 2.5: The Erasing Set and λ Removal

Left recursion is essential to the description of arithmetic precedence but it is fatal to some methods of parsing. Thus there is some work to do here. *Right recursion* is of interest for the detection of ambiguity. One can detect left recursion efficiently by defining a relation on V_N

$$r = \{(A, B) \mid B \in V_N \wedge A \rightarrow B\alpha\}$$

and noting that any symbol such that $A \rightarrow^+ A$ is left recursive. Right recursion is similarly dealt with.

¹²... unless the empty string itself is in the language.

Circularity and *nonterminating symbols* are purely evil. Both are easy to detect and can always be removed. The first step in detecting circularity is to remove erasing rules (see Table 2.5). Then simplify the rule set Π by discarding all rules with anything but a single nonterminal symbol on the right, giving a relation on V_N . Compute both the identity relation $(\Pi \triangleright V_N)^0$ and the transitive completion $(\Pi \triangleright V_N)^+$. If they are disjoint, the grammar is not circular. That is, no nonterminal produces itself.

Unreachable symbols are also of no use (except for the multiple goal symbols mentioned above, in which case V_G is used in the definition of grammar instead of G ; recall Equation 2.1). The set of reachable symbols can be computed with a rapidly converging iteration.

$$\begin{aligned} R_0 &= \{G\} \\ R_{i+1} &= R_i \cup \mathcal{R}(R_i \triangleleft \Pi)^{1/*} \\ R_\infty &= \lim_{i \rightarrow \infty} R_i \end{aligned}$$

$$R_\infty = V \Rightarrow \text{all symbols reachable}$$

Table 2.6: Reachable Symbols

Ambiguity occurs when some string has two different canonical parses. One can often devise a more concise grammar by allowing it to be ambiguous. The advantage is offset by the increased chance of causing misunderstandings and also in defeating the tools designed to deal only with unambiguous grammars.¹³

Exercises

40. [1,1] Define terms erasure, left recursive, right recursive, circular, unreachable, ambiguous. Give an example of each.
41. [10,10] Note that the grammar for Propositions is left recursive. Implement an algorithm to detect left recursion in a CFG.
42. [10,10] Write a circular CFG.
43. [10,10] Implement an algorithm to detect circularity in a CFG. What is the best diagnostic are you able to provide?
44. [20,10] Implement a program to remove erasure from any CFG \mathcal{G} such that $\lambda \notin \mathcal{L}(\mathcal{G})$. Show that applying your algorithm leaves $\text{size}(\Pi \triangleright \{\lambda\}) = 1$ if $\lambda \in \mathcal{L}(\mathcal{G})$
45. [10,10] Write a CFG with an unreachable nonterminal.
46. [10,10] Implement an algorithm to detect unreachable nonterminals. What is the best diagnostic you are able to provide?

¹³The original grammar supplied by the authors of `c` is ambiguous. The tool (`yacc`) they used to build `c` contains a fairly complex extension to its basic algorithm in order to deal with this kind of ambiguity. The grammar for ANSI `c` is just a little ambiguous.

47. [10,10] Write a CFG with a nonterminating nonterminal.
48. [10,10] Implement an algorithm to detect nonterminating nonterminals. What is the best diagnostic you are able to provide?
49. [10,50] Show that the following CFG is ambiguous.¹⁴

$$\begin{array}{l}
 G \\
 S \\
 S \\
 \quad i S t S e S \\
 \quad i S t S \\
 \quad x
 \end{array}$$

50. [10,50] Write an ambiguous CFG shorter than the one in the previous exercise.
51. [1,1] Write an ambiguous CFG with $\text{size}(\Pi) = \text{size}(\mathcal{R}(\Pi))$.
52. [1,1] Show that you cannot write an ambiguous CFG with $\text{size}(\Pi) = \text{size}(V_N)$.
53. [1,1] Show that any CFG with a nonterminal A for which both $A \rightarrow^+ A\alpha$ and $A \rightarrow^+ \beta A$ (left and right recursive in A) is ambiguous. Note: circularity is a special case.
54. [10,20] Implement an algorithm that, given a CFG, either correctly reports whether it is ambiguous or unambiguous, or reports it failed to make the distinction.
55. [1,1] Extend the previous exercise with the additional constraint that the algorithm never reports failure.
56. [∞ ,1] Extend the previous exercise with the additional constraint that the algorithm always terminates. This is the same as devising an algorithm to detect all ambiguous CFGs.

What you should have learned

Some ways to break out of the CFG straightjacket and some things that can go wrong with CFGs.

Probabilistic Grammars

Suppose that each rule r of a CFG has an associated probability $p(r)$ such that they add up to 1.0 for each $B \in V_N$.

$$\sum_{r \in \{B\} \triangleleft \Pi} p(r) = 1.0$$

¹⁴This is the classic “dangling else” ambiguity from ALGOL 60. Recall Exercise 2.36. Why was ambiguity not a problem there?

Story		
Story Sentence		0.9
Sentence		0.1
Sentence		
Subject Predicate .		1.0
Subject		
ProperNoun		0.2
Article NounPhrase		0.8
NounPhrase		
Noun		0.7
Adjective NounPhrase		0.3
Predicate		
Verb		0.8
Predicate Adverb		0.2
Article		
The		0.5
A		0.5
Adjective		
red		0.5
angry		0.5
Averb		
swiftly		0.5
badly		0.5
Verb		
ran		0.5
bled		0.5
ProperNoun		
Sally		0.5
George		0.5
Noun		
bear		0.5
salad		0.5

Table 2.7: A Trivial Probabilistic CFG for English

At each step of a sentential form proof, one could choose which rule to use to expand the rightmost nonterminal randomly in accordance with the given probabilities. Assuming that no rule has zero probability, it would seem that such an algorithm would eventually generate any particular syntactically correct string.

The probabilistic CFG in Table 2.7 describes a small part of English. The terminal symbols are English words and punctuation. The structure the grammar defines gives subject-predicate sentences like “Sally ran swiftly” and “A red angry bear bled badly.”

Some such sentences are credible; most are clearly mechanical — things only a computer would say. One can elaborate on this grammar to produce a richer set of sentences.¹⁵ What turns out not to be very helpful is manipulating the grammar to produce a higher proportion of meaningful sentences. Syntax at best exposes meaning (or the lack thereof) but is rather clumsy for excluding nonsense.

This same problem affects one serious use of probabilistic grammars — the generation of test programs for compilers. Without some nonsyntactic way to exclude semantic violations, most random programs merely test the diagnostic capability of the compiler, and then not very deeply. One can use the compiler itself to screen meaningless programs (by ignoring those that fail to compile cleanly) but that is a rather expensive filter technique because the proportion that succeed is very small.

Exercises

57. [1,1] Show that for any probabilistic grammar,

$$\sum_{r \in \Pi} p(r) = \text{size}(V_N)$$

58. [10,1] Add, change or remove some rules and reassign probabilities in the grammar in Table 2.7. What are your V_N and V_T ? Generate some “English” sentences using a random number generator (10-sided dice are convenient). To what extent do the sentences satisfy your instincts for English style and content? What is the expected length in sentences of a “Story”?
59. [10,10] Devise a probabilistic grammar that could generate “Jack and Jill”. What is the probability that it would?
60. [20,5] Pick a child’s first reader and devise a probabilistic CFG that can generate all of the sentences in the reader.
61. [40,1] Devise an interesting probabilistic CFG that can generate Lincoln’s Gettysburg address. The CFG that generates *only* the LGA is not interesting; neither is the CFG that generates *all* addresses. What is the probability of generating the LGA with your CFG?

¹⁵There have been English grammars with as many as 30,000 rules.

62. [10,1] Devise a probabilistic CFG to generate interesting programs in C (or some other programming language).
63. [0,0] One measure of success of the CFG in the previous exercise is the proportion of generated programs that compile and run. Can you estimate (or experimentally discover) this measure for your CFG?
64. [0,0] Another measure of success is the proportion of all legal programs that would be generated. Can you estimate (or experimentally discover) this measure for your CFG?
65. [0,0] A perfect generator is capable of generating every runnable program and never generates programs that violate the language standard. Could one invent a (useful) programming language for which implementing a perfect generator is possible?
66. [30,10] Pick a specific probabilistic CFG and write a program to randomly generate the strings it describes. Hint: Write one procedure per nonterminal and call the print statement for each terminal.
67. [10,30] Write a program that accepts any probabilistic CFG as input and generates syntactically correct strings according to the assigned probabilities.
68. [99,1] Write a program to accept a sequence of strings and compute an interesting probabilistic grammar that generates these same strings with approximately the observed input probabilities. The length of the computed grammar should grow more slowly than the length of the input sequence.[??]
69. [50,50] Devise a test program generator with three parameters:
- (a) LENGTH: approximate length of generated program in tokens (0 to 10000).
 - (b) CONSTRUCT: principal construct to be generated (`Expression`, `IfThenElse`, `Declaration...`).
 - (c) LANGUAGE: programming language (`X`, `C`, `Pascal`, `Ada...`).

Test it and save it for the exercises on compiler performance in Chapter 4.

70. [1,1] One could build a parser for any CFG by assigning probabilities to its rules and keeping track of the generated strings. To parse some string α , the generator would be started. When α finally showed up, the record of how it was generated would be the parse. How long would you expect such a parser to run as a function of the length of its input?

What you should have learned

The concept of probabilistic CFG and some ideas for its application.

Grammar grammars

The syntax will be described
with the aid of metalinguistic formulae.
Their interpretation is best explained by an example.
— *Report on Algol 60*

Proposition	=	Disjunction;
Disjunction	=	Disjunction '∨' Conjunction;
Disjunction	=	Conjunction;
Conjunction	=	Conjunction '∧' Complement;
Conjunction	=	Complement;
Complement	=	'¬' Boolean;
Complement	=	Boolean;
Boolean	=	't';
Boolean	=	'f';
Boolean	=	'(' Disjunction ')';

Table 2.8: Proposition CFG (second form)

A second class of practical grammar notations arises from the need to describe grammars themselves. The requirement that $V_T \cap V_N = \{\}$ prevents the notation of Section 2.1 from describing elements of V_N . One solution is to quote terminal symbols, use freestanding identifiers for nonterminal symbols, and leave everything else for meta-use.

The CFG for propositions is rewritten in such a notation in Table 2.8. The symbol indicating “is defined as” becomes ‘=’; left-hand sides are repeated as often as necessary; “end of rule” is indicated by ‘;’. The notation in Table 2.1 allowed V_N and V_T to be inferred from a concrete representation of Π . The inference required global information: one could not know if a symbol was terminal or nonterminal until the entire set of rules had been examined (t and f, for example, are terminal only because they fail to appear on the left of a defining rule). In the new notation each rule stands by itself and terminals can be immediately distinguished from nonterminals. These benefits are offset by a busier appearance for grammars.

Ignoring whitespace, the new notation can be used to describe itself as in Table 2.9.

Grammar	=	Grammar Rule;
Grammar	=	;
Rule	=	Nonterminal '=' Phrase ';';
Phrase	=	Phrase Symbol;
Phrase	=	;
Symbol	=	Terminal;
Symbol	=	Nonterminal;
Terminal	=	' ' '=' ' ' ' ';

Terminal	=	' ' ; ' ' ;
Terminal	=	' ' ' ' ' ' ;
Terminal	=	' ' Letter ' ' ;
Nonterminal	=	Letter;
Nonterminal	=	Nonterminal Letter;
Letter	=	' A ' ;
Letter	=	' B ' ;
		...
Letter	=	' z ' ;

Table 2.9: A Grammar-grammar

Exercises

71. [1,50] What is V_T in the grammar-grammar in Table 2.9?
72. [20,10] Use the CFG in Table 2.9 to parse itself.
73. [20,10] Write a self-describing grammar-grammar that neither uses nor allows erasure (second and fifth rules in Table 2.9).
74. [∞ ,1] Use BNF to unambiguously describe itself (recall the discussion on page 39).

What you should have learned

A grammatical notation capable of self-description.

Executable Grammars

A CFG is intuitively an executable code since the rules can be mechanically applied. The problem is choosing the sequence of rules. One solution is clairvoyance. Another solution is exhaustive trials.¹⁶ In between are practical combinations of restrictions on the defining CFGs and parsing machines. Two such combinations are presented in the next few sections.

g=rg;	p=' ' c ' ' ;	c=l;
g=;	p=' ' ' c ' ' ' ;	l=' g ' ;
	p=l;	l=' r ' ;
r=l ' = ' f ' ; ' ;	c=' ' ' ;	l=' f ' ;
	c=' ' ' ' ;	l=' p ' ;
f=pf;	c=' = ' ;	l=' c ' ;
f=;	c=' ; ' ;	l=' l ' ;

Table 2.10

Self.g — An Executable Self-describing Grammar

¹⁶Recall the discussion on page 36 and Exercise 70.

The three-column grammar in Table 2.10 is another example of a self-describing grammar. This one is executable. Strictly speaking, the texts described by this grammar are not all CFGs because the grammar permits two terminal vocabularies.¹⁷ One set of terminals is delimited by apostrophes (') and the other by quote marks ("). The second set of terminals is not actually used in Table 2.10.

A file naming convention is used for **Self.g** to emphasize that it is a computational object. It can be read as follows. All symbols are restricted to single characters. Each rule is terminated by a semicolon. The first rule says *g* may consist of a rule *r* followed by a list of rules *g*. The second rule says the list may be empty. The third rule says a rule *r* consists of a letter *l*, followed by a literal '=', followed by a formula *f*, and finally a semicolon. This rule is self-describing. Every character is significant (except in printed form as above, whitespace is insignificant).

Two rules (one for *g* and one for *f*) have empty right-hand sides. As mentioned above, there are two quoting conventions, one using '' and one using ". The characters delimited by '' are the input vocabulary; the characters delimited by " are the output vocabulary. The self-describing grammar in Table 2.10 itself does no output. No symbol is left recursive.

Suppose that the rules for each nonterminal are tried in the order of the grammar: first rules first. If a rule fails (i.e., needs some input that is not there), the next rule is tried. The algorithm GEM in Table 2.13, written in an extension to X, works this way.¹⁸

One may look at GEM as a function from a pair of strings to a string:

$$Gem(grammar, input) = output$$

with the additional characteristic that it may fail altogether, signalling that the *input* was not grammatically correct according to the *grammar* or that there was something wrong with the *grammar*. For the self describing grammar **Self.g**,¹⁹

$$Gem(\mathbf{Self.g}, \mathbf{Self.g}) = \lambda$$

Before describing how GEM works, some theory is necessary.

Exercises

75. [1,1] What are the structural differences between the grammars in Tables 2.9 and 2.10?
76. [1,50] What is the shortest string generated by **Self.g**? What is its canonical parse?
77. [1,1] Use **Self.g** to parse its own first rule.

¹⁷See page 49

¹⁸An extension to GEM, written in C, is in the HYPER distribution package. It is called Gem.c for *Grammar Executing Machine*

¹⁹Recall that λ is the empty string.

78. [1,1] Show $Gem(\mathbf{Self.g}, \mathbf{Self.g}) = \lambda$.
79. [1,1] Write a grammar $\mathbf{Trans.g}$ such that $Gem(\mathbf{Trans.g}, \alpha) \neq \lambda$ for some string α .
80. [1,1] Write a grammar $\mathbf{Zip.g}$ such that $Gem(\mathbf{Zip.g}, \mathbf{Zip.g}) = \mathbf{Zip.g}$. Is $\mathbf{Self.g}$ a good starting point? What is the shortest $\mathbf{Zip.g}$?

Top Down Parsing

From the viewpoint of the compiler writer, the predicate SF is a little frustrating. The proof starts from the answer and proceeds to the question; that is: from the goal symbol to the text. Supposing one starts at the text, and making correct choices of which rule to apply, gets to the goal. Then one also knows there is a proof. The problem arises when one gets stuck. It is hard to know whether a bad choice of rule has been made somewhere along the line or if the text should just be rejected.

For large classes of grammars there are methods which always give the correct parsing choice, if there is one, and indicate failure if there is none. In broad terms each such method requires examining the grammar and tabulating every parsing decision that will ever have to be made. The parser consults the table to make decisions.

There are two kinds of parsers: *bottom-up* and *top-down*. Bottom-up parsers pick the rule in each step of the predicate SF , and also locate the phrase to be reduced. While the task of building the tables for a bottom-up parser could be done by hand, this is almost never done in practice because the process is voluminous, tedious and error-prone. Parser generator tools are used instead. Bottom-up methods are discussed in Chapter 5.

There is a simpler method, called *top-down* parsing. This method, like the bottom-up methods, can be automated with table generating tools. In contrast to bottom-up methods, it is also feasible for a programmer to implement a top-down parser for most grammars by systematically writing procedures (perhaps in C) to correspond to grammar rules. In this case no special tools are required. The hand-written top-down technique is described in the next few sections. The first task is to provide a formal description.

Consider an alternative to predicate SF : predicate $TD(\Sigma, \tau)$.²⁰ The parameter Σ is a stack of things to do — in the form of partial grammar rules. The parameter τ is the as-yet-unprocessed text. At each step the stack of things to do grows or something has been recognized and can be dropped from both parameters. This new predicate better reflects the operation of executable grammars than does SF . The representation of a stack of strings requires a notation for stack element separator; the symbol \circ is used. In fact we could use a string to represent the stack, merely concatenating the partial rules, except that we want to call out the ascent step separately. It is the sequence of ascents that yield

²⁰ TD stands for top-down.

the canonical parse. Finally, since we intend to use TD on grammar grammars, terminal strings and symbols will be quoted (for example ‘f’ is the terminal letter f as distinct from the unquoted nonterminal f).

basis	$TD(\lambda, \lambda)$	
descent	$B \rightarrow \beta \wedge TD(\beta \circ \alpha \circ \Sigma, \tau)$	$\Rightarrow TD(B\alpha \circ \Sigma, \tau)$
ascent	$TD(\Sigma, \tau)$	$\Rightarrow TD(\lambda \circ \Sigma, \tau)$
shift	$TD(\alpha \circ \Sigma, \tau)$	$\Rightarrow TD(a\alpha \circ \Sigma, a\tau)$

Table 2.11: Description of Top-down Parsing

There are four axioms: a basis axiom, a descent axiom, an ascent axiom and a shift axiom. There is also a query which proposes an input text to be parsed. The parse proof is constructed from query to basis, then verified in the reverse direction. Thus during proof construction the rules are applied against the \Rightarrow so that during verification the implications can be carried back. The starting condition to parse text τ is the query $TD(G, \tau)$. The four axioms reflect four basic activities of a top down parser. The first axiom is interpreted to say that if there is nothing to be done (empty stack) and no remaining input (empty text) then the parse must have succeeded; it is time to quit. The second axiom is interpreted to mean that when there is a nonterminal at the left of the topmost to-do string on the stack, then it can be removed from its string and the whole right-hand-side defining that rule pushed on the to-do stack. The choice of which rule to apply is not arbitrary; a bad choice will force backup and retry. Practical top-down parsers will always make the right decision, usually by peeking ahead into the input text to see what is coming and choose a rule that can be applied without getting into trouble later. The third rule is an intermediate success: a rule has been fully processed on the to-do stack so that the stack can be popped and work continued on the next thing down. It is at this moment that the application of this particular rule can be reported for the canonical parse. Finally, if both the input text and the top of the to-do stack have the same symbol on the left (demand matches supply), then the symbol can be discarded from both. This action mirrors the scanning of a new symbol from the input.

Table 2.12 shows the application to text ‘f=pf;’ of the grammar-grammar in Table 2.10. The lack of left recursion in this grammar makes it easier to use for the example. The right choice is, in fact, always apparent from examining the leading input character.

The query has only g on Σ , its things-to-do stack. The text to be processed, τ , is ‘f=pf;’. The shift axiom cannot be applied because the leading symbol on Σ does not match the leading symbol in τ . The ascent axiom cannot be applied because λ is not on the top of Σ . The basis axiom for TD cannot be applied because neither parameter is empty. That leaves only descent taking g for B.

Trying the first of the two alternatives in Π , g is shortened to λ and grammar rule right-hand-side rg is stacked on top of it. This makes a new situation, and the process is repeated. It may be that at some point TD will get stuck with no applicable axiom — no matter, TD backs out, undoing things until reaching the

most recent arbitrary choice (trying the rules in grammar-order was arbitrary) and then tries again with the a different choice.

In Table thechapter.theTopDownProof, the arguments to function *TD* are shown, the axiom applied is named, and the grammar rule started or finished is noted. Once a particular grammar rule is started, all other rules started before this particular rule finishes must finish. That is, the applications nest, each descent is followed eventually by a matching ascent. The descent starts a rule and the matching ascent finishes it. In the example, the first rule started ($g=rg$) is therefore the last rule finished. The list of finished rules in the order that they appear is the canonical parse. Vertical alignment is used to maintain visual track of contents of Σ and τ from step to step in Table 2.12.

	<i>to do</i>	<i>input</i>	<i>axiom</i>	<i>started</i>	<i>finished</i>
<i>TD</i> ($g,$	$'f=pf;'$	query		
<i>TD</i> ($rg \circ \lambda,$	$'f=pf;'$	descent	$g=rg$	
<i>TD</i> ($l='f;'\circ g \circ \lambda,$	$'f=pf;'$	descent	$r=l='f;'$	
<i>TD</i> ($'f'\circ '='f;'\circ g \circ \lambda,$	$'f=pf;'$	descent	$l='f'$	
<i>TD</i> ($\lambda \circ '='f;'\circ g \circ \lambda,$	$'=pf;'$	shift		
<i>TD</i> ($'='f;'\circ g \circ \lambda,$	$'=pf;'$	ascent		$l='f'$
<i>TD</i> ($f;'\circ g \circ \lambda,$	$'pf;'$	shift		
<i>TD</i> ($pf \circ ;'\circ g \circ \lambda,$	$'pf;'$	descent	$f=pf$	
<i>TD</i> ($l \circ f \circ ;'\circ g \circ \lambda,$	$'pf;'$	descent	$p=l$	
<i>TD</i> ($'p'\circ \lambda \circ f \circ ;'\circ g \circ \lambda,$	$'pf;'$	descent	$l='p'$	
<i>TD</i> ($\lambda \circ \lambda \circ f \circ ;'\circ g \circ \lambda,$	$'f;'$	shift		
<i>TD</i> ($\lambda \circ f \circ ;'\circ g \circ \lambda,$	$'f;'$	ascent		$l='p'$
<i>TD</i> ($f \circ ;'\circ g \circ \lambda,$	$'f;'$	ascent		$p=l$
<i>TD</i> ($pf \circ \lambda \circ ;'\circ g \circ \lambda,$	$'f;'$	descent	$f=pf$	
<i>TD</i> ($l \circ f \circ \lambda \circ ;'\circ g \circ \lambda,$	$'f;'$	descent	$p=l$	
<i>TD</i> ($'f'\circ \lambda \circ f \circ \lambda \circ ;'\circ g \circ \lambda,$	$'f;'$	descent	$l='f'$	
<i>TD</i> ($\lambda \circ \lambda \circ f \circ \lambda \circ ;'\circ g \circ \lambda,$	$';'$	shift		
<i>TD</i> ($\lambda \circ f \circ \lambda \circ ;'\circ g \circ \lambda,$	$';'$	ascent		$l='f'$
<i>TD</i> ($f \circ \lambda \circ ;'\circ g \circ \lambda,$	$';'$	ascent		$p=l$
<i>TD</i> ($\lambda \circ \lambda \circ \lambda \circ ;'\circ g \circ \lambda,$	$';'$	descent	$f=$	
<i>TD</i> ($\lambda \circ \lambda \circ ;'\circ g \circ \lambda,$	$';'$	ascent		$f=$
<i>TD</i> ($\lambda \circ ;'\circ g \circ \lambda,$	$';'$	ascent		$f=pf$
<i>TD</i> ($';' \circ g \circ \lambda,$	$';'$	ascent		$f=pf$
<i>TD</i> ($\lambda \circ g \circ \lambda,$	λ	shift		
<i>TD</i> ($g \circ \lambda,$	λ	ascent		$r=l='f;'$
<i>TD</i> ($\lambda \circ \lambda \circ \lambda,$	λ	descent	$g=$	
<i>TD</i> ($\lambda \circ \lambda,$	λ	ascent		$g=$
<i>TD</i> ($\lambda,$	λ	ascent		$g=rg$

Table 2.12: A Top-down Proof

Exercises

81. [1,1] Verify that each rule started (*descent*) in Table 2.12 is matched with

a later finish (*ascent*). Verify that starts and finishes nest. Verify that the list of finishes is the canonical parse.

82. [1,1] Noting that the grammar-grammar used in the example is not left recursive, change the two right recursions into left recursions and repeat the derivation. You will have to become clairvoyant on some decisions.
83. [10,10] Duplicate the proof in Table 2.2 by deriving
 $TD(\text{Proposition}, f \vee t \wedge \neg f)$ from $TD(\lambda, \lambda)$.
84. [50,50] Show for any CFG $\mathcal{G} = \langle V_T, V_N, G, \Pi \rangle$ and $\alpha \in V_T^*$

$$\alpha \in \mathcal{L}(\mathcal{G}) \Leftrightarrow TD(G, \alpha)$$

A Grammar Executing Machine

The extension to x in which GEM (see Table 2.13) is written uses only a few programming primitives: tests for equality and set membership, assignment, incrementing and indirection. The idea is to keep close to something that could be conveniently mechanized.

In a conventional computer instructions are taken from a program counter, input is read and output is produced. In GEM the analog to machine language is the grammar itself. The program counter, designated p , is named after the pc register. In this case p points to characters in the grammar. The input is also characters. One task of GEM is to determine if the input is correct according the the grammar. As a side effect, GEM produces output.

The algorithm of GEM is an application of TD , with the addition of a concrete mechanism to find and select rules, to recover from bad choices, and to produce output.

In summary, the grammar, input and output are strings. The corresponding characters are accessed via pointers p, i and o . There is a stack for pointer p (ps) and also one to record the completed rules (es).

The notation describing GEM mirrors some low-level concepts in C: $+p$ means “increment p by 1” and $*p$ means the value pointed to by p . Analogous interpretations are give to $++p$, $--p$, $**p$, $+*p$, and so on. EoS is a symbol that terminates strings.²¹

GEM works as follows. It has three modes: parsing, backtracking and searching. The mode is determined by the value of variable *mode*. The machine is started in the searching mode, looking for a grammar rule to apply.

The grammar rule separator is semicolon. Each semicolon is followed by the name of the next rule (a letter) and an ‘=’. The semicolon is used both to terminate the rule and signal that another rule is coming. So that the very first rule is consistently delineated, an extra semicolon precedes the input grammar.

Things are started with the top of the parse stack ps pointing at a special starting string containing the goal symbol in a context of an artificial “rule”

²¹0 is used in c.

```

es := newstack(); mode := SEARCH           `Initialization
ps := newstack(); p0 := ps; *ps := " = G' E' ";
+* ps; p := grammar; +p; ** ps := *p; p := *ps; ++ p; *p := EoS;
p := grammar; i := input; o := "";       `',' must begin grammar
it
  if mode = PARSE →                       `Executing a Rule.
    if *p ∈ Letter →                       `recursive call
      mode := SEARCH;
      +ps; *ps := p; p := grammar         `descent
    [] *p = ''' →                          `input
      +p;                                  `skip ''
      if *p = *i → +i; ++ p               `shift
      [] else mode := BACK; -- p          `no match
      fi
    [] *p = ''' →                          `output
      +p; +o; *o := *p; ++ p
    [] *p = ';' →                          `return
      -p; +es; *es := p;
      if ps < p0 → exit                    `stack empty: done
      [] else p := *ps; -ps; +p           `ascent
      fi
    fi
  [] mode = BACK →                         `Backtracking.
    if *p ∈ Letter → +ps; *ps := p; p := *es; -es
    [] *p = ''' → -i; --- p                `unshift input terminal
    [] *p = ''' → -o; --- p                `unshift output terminal
    [] *p = '=' → mode := SEARCH; +p
    fi
  [] mode = SEARCH →                       `Searching for a Rule.
    if *p ∈ Letter → +p                    `skip nonterminal
    [] *p = ''' → +++ p                    `skip input terminal
    [] *p = ''' → +++ p                    `skip output terminal
    [] *p = ';' →                          `end of rule
      +p;                                  `skip ';'
      if *p = ** ps → mode := PARSE        `match, descent
      [] else ++ p                          `no match
      fi
    [] *p = EoS →                          `search failed
      if ps ≠ p0 →                          `continue or abort
      fi;
      p := *ps; -ps; mode := BACK; -p     `backtrack
    fi
  fi
ti

```

Table 2.13: GEM — A Grammar Executing Machine

which demands the goal symbol followed by EoS. This is the query setup for *TD*. The special starting string pointed to by *ps* has two dummies, *G* and a quoted *E*. The first is replaced during initialization by the goal symbol from *grammar* and the second is replaced by EoS.

Parsing begins. The grammar pointer *p* points at the extra leading semicolon. As mentioned before, the mode set to **SEARCH**, and searching therefore starts right after the semicolon with the first rule of the grammar.

Each time a semicolon is passed during search, the next symbol (the left-hand side of some rule) is compared to match the search objective on top of *ps*. If it matches, the mode is changed to **PARSE** and the rule is applied. This is an application of the descent axiom. Otherwise searching continues. If the search fails by running off the end of the grammar, there is no applicable rule and the previously applied rule must be disappplied by backtracking.

Backtracking is accomplished by running the grammar backwards. Note that grammar rule bodies are symmetrical, i.e. can be read right-to-left as easily as from left-to-right. The main difference is that increments to *p* become decrements and previous input/output is undone character-by-character. Backtracking continues until the left end of the failing rule is reached (marked by an unquoted '='), then searching is resumed, looking for the next applicable rule. The failed rule is skipped in the new search because a search comparison is tried only after reaching the rule-ending semicolon of the failed rule.

Each time an input terminal is encountered in the grammar during parsing and it agrees with the current input symbol pointed to by *i*, the input symbol is *shifted* to reveal the following input symbol and parsing continues. If, on the other hand, an input terminal in the grammar fails to agree with the current input symbol, the current rule has failed and backtracking is commenced; in this case the increments to *p* turn into decrements to *p* right in the middle of the case in **GEM** for comparing input and grammar terminals.

There are three connections between data structures in **GEM** and function *TD*. The parse stack is pointed to by *ps*. The entries on the parse stack point at nonterminals in the grammar, each in the middle of some rule. The remainder of each rule, beyond the nonterminal pointed to from the parse stack, and on to the end of the rule, is what remains to be done for that rule. The list of these remains is the stack Σ , which is the first argument to function *TD* on page 59. The unconsumed input is the second argument to *TD*. The backtracking stack pointed to by *es* contains the canonical parse, coded as the index of the last symbol of each rule that has been applied. The use of *es* is twofold: it records which rule needs to be backtracked, and also the place where backtracking on that rule must begin. It could, in addition, be used to display the canonical parse when **GEM** terminates successfully.

There are three kinds of failure. An unexpected character can be encountered in the grammar. This situation is analogous to an illegal instruction in machine language. **GEM** (as printed) runs out of alternatives and aborts an if-fi. The second kind of failure occurs when the input is not correct according to the grammar; **GEM** backtracks all the way to the start of the input and reports

the failure. This situation is the analog of “syntax error” in a conventional compiler. The third kind of failure is when an otherwise perfectly acceptable grammar turns out to be left-recursive in some rule; GEM loops.

There are some subgrammars that occur frequently enough to justify one-time definitions. The subgrammars for letter, digit and character are given in Table 2.14.²²

L='a' "a";	D='0' "0";	C=L;
L='b' "b";	D='1' "1";	C=D;
...	...	C='+' "+";
L='z' "z";	...	C='-'"-";
L='A' "A";	...	C= <i>all of ASCII</i>
...
L='Z' "Z";	D='9' "9";	...

Table 2.14
Subgrammars for L, D and C

Using these subgrammars (as though they were implicitly appended to any grammars that use L, D or C), some translators can be written. The following grammar will remove whitespace (blank and new-line) from any other grammar. A blank is made visible (to you) by giving it the glyph ‘␣’. The new-line is obvious as a rule that does not have its terminating ‘;’. The fifth and sixth rules protect quoted characters; otherwise the third and fourth rules capture and discard whitespace before the last rule can reflect it into the output. Order is important. The first rule causes the whole input to be examined, terminating on a single invocation of the second rule when the input is exhausted.

```

g=pg;
g=;
p='␣';
p='
';
p='''''C'''''';
p='''''C'''''';
p=C;

```

Table 2.15
Scan.g — An Executable Deblanker

A function can be defined in terms of the deblanking grammar. For any string s ,

$$\text{Scan}(s) \stackrel{\text{def}}{=} \text{Gem}(\mathbf{Scan.g}, s)$$

²²ASCII stands for the American Standard Code for Information Interchange.

The deblanker must already be deblanked to use as the first parameter to GEM. (It is presented in Table 2.15 with the line breaks because it is not very readable otherwise.²³)

A more interesting grammar accepts other grammars and inverts them. In programming jargon, it builds a decompiler out of a compiler.

```
g = pg;
g = ;
p = ' ' ' ' ' ' C ' ' ' ' ' ' ;
p = ' ' ' ' ' ' C ' ' ' ' ' ' ;
p = C;
```

Table 2.16

Invert.g — An Executable Grammar Inverter

As there was for the deblanker, there is a function for the inverter:

$$\mathit{Invert}(s) \stackrel{\text{def}}{=} \mathit{Gem}(\mathit{Scan}(\mathbf{Invert.g}), s)$$

In this case whitespace can be used freely in **Invert.g** to improve readability because *Scan* removes it before GEM has to deal with it.

Like a rattlesnake, the inverting grammar is immune to itself. That is, for all strings s

$$\mathit{Gem}(\mathit{Scan}(\mathit{Invert}(\mathbf{Invert.g})), s) = \mathit{Invert}(s)$$

An interesting application of the inverter is to regenerate texts from the canonical parse. Suppose we define a grammar for right-associative sums and differences in Table 2.17.

```
g = e      "0";
e = t '+' e "1";
e = t '-' e "2";
e = t      "3";
t = 'x'    "4";
```

Table 2.17

Sum.g — Canonical Parse for a Right-associative Sum

The input is an expression such as $x + x - x$; the corresponding output is the canonical parse 4443210. Suppose that the grammar above is inverted:

$$\mathbf{UnSum.g} \stackrel{\text{def}}{=} \mathit{Invert}(\mathbf{Sum.g})$$

²³The name of **Scan.g** indicates a close relation to scanning, a topic appearing later in this section.

```

g = e      '0';
e = t "+" e '1';
e = t "-" e '2';
e = t      '3';
t = "x"    '4';

```

Table 2.18
UnSum.g — Inverted Sums Grammar

Given the parse 4443210, the inverted grammar will produce the original input string $x + x - x$. That is, for sum s ,

$$Gem(Scan(\mathbf{UnSum.g}), Gem(Scan(\mathbf{Sum.g}), Scan(s))) = Scan(s)$$

As a final step, a filter grammar can be defined to extract the PFN from the canonical parse.²⁴

```

g = t g;
g = ;
t = '0';
t = '1' "+";
t = '2' "-";
t = '3';
t = '4' "x";

```

Table 2.19
Filter.g — Filter for Canonical Parse

$$Pfn(s) \stackrel{\text{def}}{=} Gem(Scan(\mathbf{Filter.g}), Gem(Scan(\mathbf{Sum.g}), Scan(s)))$$

A more difficult problem is making the executable grammars produce PFN for left associative operators. The “natural” grammar for left associativity is left recursive in each level of arithmetic hierarchy. Yet it happens that left recursion defeats GEM; it will go into a recursion loop, eventually overrunning the storage allocated for the parse stack. There is a pretty solution, which requires extending GEM with a meta symbol for repetition (regular expression ‘*’), and lead directly to the method of recursive descent. This solution is discussed in the following sections.

There is, in fact, a solution using the unmodified GEM which leads to an ugly grammar. The ugly trick is to insert some new rules in the sum grammar, simulating the and, in each such repetition, move each infix operator “over” its right operand. The grammar in Table 2.20 does the job.

²⁴See page 48

$g = e;$	$s = '*' f "*" s;$
$e = t r;$	$s = '/' f "/" s;$
$r = '+' t "+" r;$	$s = ;$
$r = '-' t "-" r;$	$f = L;$
$r = ;$	$f = D;$
$t = f s;$	$f = '(' e ')';$

Table 2.20

Expr.g — Left-associative Expression Grammar

If the rule for parentheses is removed, this grammar is invertible. The inverse takes PFN back into its original expression form. Parentheses are fatal for the reverse transformation because the PFN does not contain the information on redundant pairs in the original input. At any place where there is an x , there might have been (x) or $((x))$ and so on.²⁵ The problem shows up as left recursion in the inverted grammar. What is perhaps surprising is that the simple the grammar in Table 2.3 cannot be used to accept PFN as input. Note that the grammar for PFN is twice left recursive.

In contrast to left recursion, ambiguity poses no problem for GEM; when a choice of rule application need be made, GEM takes the first rule that works. When it is in trouble it backtracks. The first parse that uses the entire input is accepted.

Exercises

85. [10,50] Show that **Scan.g** is ambiguous (ignore the output vocabulary for this exercise).
86. [1,50] Parse x and $x + x$ using **Sum.g**, keeping track of the output.
87. [20,50] Translate the grammar for Proposition (Table 2.8) into the executable notation. Add emitters for the parse sequence. Parse $f \vee t \wedge \neg f$ by the new grammar, duplicating the information in Table 2.2. (Note, the clumsier executable notation will cause you to add some rules for which you will not produce output. Also, since the grammar is left recursive, it is not useful with GEM.)
88. [20,10] Implement GEM. Invert the parser **Sum.g**. Call the result **UnSum.g**. Parse an expression and use **UnSum.g** to recover the original input. (Note: recall that grammars must start with a gratuitous ';' for GEM to work.)
89. [1,1] How does GEM respond to a missing right quote (either ' or ") in the grammar? Can you suggest an improvement?
90. [10,20] How does GEM respond to errors in the input? (See page 65.) Can anything be done to make GEM easier to use?

²⁵GEM can generate a lot of left parentheses trying.

91. [10,10] Write a grammar to take deblanked grammars and restore them to readable form. Call it **Pretty.g**; it is a kind of inverse to **Scan.g**. Write a formula describing the relation between **Scan.g** and **Pretty.g**.
92. [5,10] Extend **Scan.g** so that it also removes comments in the style of those in x.
93. [20,50] Change the expression-to-PFN translator **Expr.g** to process Propositions instead. This is a useful restatement of Exercise 2.87.
94. [20,20] Try the grammar for PFN in Table 2.3 on GEM and explain the behavior. Then devise a grammar that *does* allow GEM to accept PFN as input.
95. [20,50] Extend the expression-to-PFN translator **Expr.g**. Add right associative exponentiation. Without using **Scan.g**, allow blanks in the expression input. Allow multicharacter identifiers and integers. Separate all PFN output by single blanks.
96. [10,1] Speed up GEM by making L, D and C reserved names, implementing them directly in the “hardware” of GEM.²⁶
97. [10,1] Speed up GEM SEARCH mode by building a table of starting points for the search, and direct links to next-rule rather than rummaging past rule after rule.

What you should have learned

An interpretation of grammar as a programming language. A reversible instruction stream for backtracking. Some interesting little grammars. The concepts of scanning and inversion.

2.4 Regular Expressions

Regular expressions serve the same purpose as grammars: they syntactically define strings of symbols. Their advantage is conciseness and ease of implementation. Their disadvantage is that they can describe fewer things than CFGs. Two side issues favor a discussion of regular expressions here. The first is that their expressive power can be mixed in with grammars, making the grammars more concise while not changing much else. The second is a close relation of a subset of the regular expressions (as defined here) to finite automata. Finite automata, in turn, form the basis for automatic parsers. The use of regular expressions to extend the expressiveness of grammars is presented in this section. The new style of grammars will be designated REG in contrast to CFG. The necessary restrictions to establish a relation to finite automata are presented in the Chapter 5.

²⁶This sped up GEM is in the HYPER distribution package.

$\mathcal{L}(\lambda)$	$\stackrel{\text{def}}{=} \{\lambda\}$
$\mathcal{L}(a)$	$\stackrel{\text{def}}{=} \{a\}$
$\mathcal{L}(A)$	$\stackrel{\text{def}}{=} \mathcal{L}(A)$
$\mathcal{L}(AB)$	$\stackrel{\text{def}}{=} \{\alpha\beta \mid \alpha \in \mathcal{L}(A) \wedge \beta \in \mathcal{L}(B)\}$
$\mathcal{L}(A B)$	$\stackrel{\text{def}}{=} \mathcal{L}(A) \cup \mathcal{L}(B)$
$\mathcal{L}(A\&B)$	$\stackrel{\text{def}}{=} \mathcal{L}(A) \cap \mathcal{L}(B)$
$\mathcal{L}(A - B)$	$\stackrel{\text{def}}{=} \mathcal{L}(A) - \mathcal{L}(B)$
$\mathcal{L}(\sim A)$	$\stackrel{\text{def}}{=} V_T^* - \mathcal{L}(A)$
$\mathcal{L}(A^0)$	$\stackrel{\text{def}}{=} \{\lambda\}$
$\mathcal{L}(A^i)$	$\stackrel{\text{def}}{=} \mathcal{L}(A^{i-1}A)$
$\mathcal{L}(A_j^k)$	$\stackrel{\text{def}}{=} \bigcup_{i=j}^k \mathcal{L}(A^i)$
$\mathcal{L}(A^*)$	$\stackrel{\text{def}}{=} \mathcal{L}(A_0^\infty)$
$\mathcal{L}(A^+)$	$\stackrel{\text{def}}{=} \mathcal{L}(A_1^\infty)$
$\mathcal{L}(A^?)$	$\stackrel{\text{def}}{=} \mathcal{L}(A_0^1)$

Table 2.21: Regular Expression Languages

A regular expression is a combination of terminal symbols (V_T) and meta symbols. The meta symbols are integer constant subscripts and superscripts, parentheses, and the following operators:

	regular expression ‘or’
&	regular expression ‘and’
-	regular expression difference
*	zero or more repetitions
+	one or more repetitions
?	zero or one (optional item)
~	regular expression complement

Let $a \in V_T$, $\alpha \in V_T^*$, and A, B be regular expressions. Given A , the set of strings, $\mathcal{L}(A)$, defined by it can be built up by repetitive application of the rules in Table 2.21. To be careful, for the time being, we use parentheses wherever the precedence or associativity is ambiguous.

Exercises

98. [1,1] What are $\mathcal{L}(\sim())$, $\mathcal{L}(a|b)$, $\mathcal{L}(a|a)$, $\mathcal{L}(a\&b)$, $\mathcal{L}(a\&a)$, $\mathcal{L}(a^*)$, $\mathcal{L}(a^+)$, $\mathcal{L}(a^9)$, $\mathcal{L}(a_3^9)$, and $\mathcal{L}(cr^?a(zy|t))$?
99. [1,1] Where is $((H|n)o)^2(lu)^2 - ((\sim())H|n)\sim()$?
100. [1,1] Letting L stand for any letter and D for any digit, describe the following programming language constructs as regular expressions:
- (a) identifier
 - (b) real constant
 - (c) C reserved word
101. [1,1] Using D for dime, N for nickel and P for penny give regular expressions for change for a dime and change for a quarter.
102. [1,1] Give a regular expression for regular expressions.

Regular Expression Grammars

The right-hand sides of grammar rules can be regular expressions. The canonical parse and parse tree are not defined, but the grammars are straightforward to write and understand. The grammar in Table 2.8, for instance, is rewritten using the meta operators in Table 2.22. The rewrite is shorter and has less recursion. More importantly, writing a REG is an essential step in building a recursive descent parser for its language.

```

Proposition = Disjunction;
Disjunction = Conjunction ('∨' Conjunction)*;
Conjunction = Complement ('∧' Complement)*;
Complement = '¬'? Boolean;
Boolean     = 't' | 'f' | '(' Disjunction ')';

```

Table 2.22
REG Proposition Grammar

Exercises

103. [1,1] Elaborate on the following “regular schedule” by adding rules to describe more symbols. How far does it pay to carry such a description? Would $V_G = \{His, Hers\}$ be a good idea?

```

DailyGrind = (Weekday5 Saturday Sunday)*;
Weekday    = Alarm AMchores Drive Work Drive PMchores Sleep;
Saturday   = Alarm Sleep AMchores (Shopping | Lawns) PMchores Sleep;
Sunday     = AMchores Church (TV | Outing) PMchores Sleep;

```

104. [1,1] The essence of grammar is phrase structure; things defined in terms of sequences of things, to any depth. Describe something else (other than DailyGrind) with a REG. Extra credit for creativity.
105. [10,50] The grammar-grammar in Table 2.9 can also be rewritten as a REG, and then rewritten once again to be self describing. Do both, including only meta operators ‘*’, ‘?’, parentheses, and ‘|’. You may assume an ASCII form of input.
106. [20,20] Repeat the previous exercise but use all of the regular expression meta operators. You must do some language design to decide how to represent the subscripts and superscripts within the limits of ASCII.
107. [0,0] What is the shortest self describing grammar of this kind you can devise? You may want to include or exclude some of the meta symbols.
108. [10,20] Consider a program with the following properties:
- The input consists of letters and blanks. A word is any sequence or 12 or fewer letters. A telegram is any sequence of words (but not XXXX), separated by one or more blanks, terminated by word XXXX. A batch is a sequence of telegrams, also terminated by word XXXX. Use a REG to describe a batch. (Hint: Use operator ‘—’ and integer superscripts.)
 - The output consists of a sequence of 40-character lines. Each line starts with a sequence of words separated by single blanks, and is padded on the right with zero or more blanks. The output words are the same as the input words, and in the same order, except that word XXXX does not appear and telegrams are separated by a blank line. Use a REG to describe output resulting from an acceptable batch (hint: use operator ‘&’). Is your grammar strict enough to make the output well defined?
 - [1,20] Combine your grammars by letting $V_G = \{\text{input}, \text{output}\}$. This is an example of a nonstandard grammar.
 - [1,1] How would you send *this* exercise as a telegram?

Michels’ Grammar-grammar

There is an interesting feedback phenomenon in self-compiling compilers. The simpler the implementation language, the simpler the compiler. Extrapolating this concept raises a cute question: “What is the minimal extendable self-compiling compiler?” The GEM grammars above are surely much closer to this endpoint than C-in-C. Here is another, even more concise, grammar-grammar²⁷[?]:

²⁷This solution presented here was taken from Doug Michels’ undergraduate thesis at the University of California.

|'S|.|'.|'.|'.SS.''|'.|'|''S

The symbols are '|' meaning 'or', the quotation mark "'", meaning take the next symbol literally (terminal vocabulary), an explicit catenation operator '.' and finally S which is the only nonterminal and therefore also the goal symbol. The expression, which could be loosely designated as a REG in prefix PFN, is the recursive definition of S. Put back into infix notation and using white space for readability, the grammar is

S = 'S
 | ('.|'|)SS
 | '' (''|'.|'| | 'S)

Everything to the right of the = is a regular expression, which translates to the prefix PFN constituting Michels' grammar-grammar.

Some of the terminal strings described are

S
 'S
 .SS
 ..SSS
 .'.'.
 |'S|.|'.|'.|'.SS.''|'.|'|''S

The association of operators and operands in the prefix PFN self-description is given in Figure 2.3.

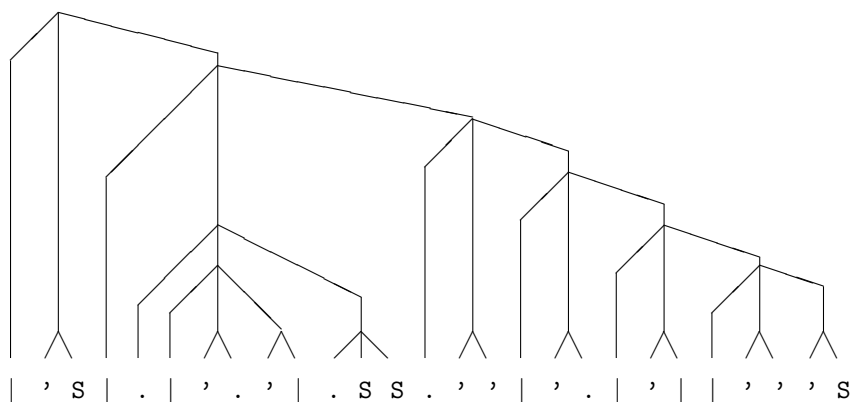


Figure 2.3
 Operator and operand association for '| 'S|. |'. |'. |'. SS.' '|'. |'|''S

What you should have learned

The concept of a regular expression for defining language. The equivalences between regular expressions and grammars and how to combine them.

2.5 Parsing Expressions

Extending Gem

Executable grammars²⁸ can do quite a bit and it does not take much machinery to get them going. They are, however, too inefficient for most practical applications. The principal problem is that right recursion causes the parse stack to grow with the length of the input text. One can surmount this problem by implementing part of the regular expression syntax. Iteration for any nonterminal B can be represented by a metasymbol asterisk following the nonterminal, e.g. B*. Using the transformation from left recursion to repetition discussed below, one can make a REG equivalent to the more natural CFG. The grammar in Table 2.23 is the self-describing grammar expressed in REG form.

g = r*;	c = ' ';	c = l;
r = l '=' p* ';' ;	c = ' ' ' ';	l = 'c' ;
p = ' ' ' ' c ' ' ' ' ;	c = ' ' ' ' ' ' ;	l = 'g' ;
p = ' ' ' ' ' c ' ' ' ' ;	c = '=' ;	l = 'l' ;
p = l ;	c = '* ' ;	l = 'p' ;
p = l '* ' ;	c = ';' ;	l = 'r' ;

Table 2.23:
Executable Self-describing REG

The first rule says a grammar g consists of zero or more rules r. The second rule says a rule r consists of a letter followed by '=' followed by zero or more primaries p, and terminated by a semicolon. The letters l defined include only those also used, to keep the grammar as short as possible for this presentation.

Given the new repetition operator, and faster rule search, GEM can be used for practical applications. Such a program, called **gemstar.c**, is included in the HYPER distribution package. There is, in fact, no loss in generality in restricting the application of '*' to nonterminals. Any terminal or parenthesized regular expression, which might have been repeated, can be made the right side of a new rule for a new nonterminal and the nonterminal repeated instead. The '|' operator can likewise be broken out into two or more new rules. The '+' can be built out of '*', as in BB*. The metasymbols '&', '~' and '-' are beyond the range of GEM. The canonical parse is a meaningful concept for this limited kind of REG. The effect is to have zero or more rule applications for the star-ed nonterminal.

Most CFGs are mostly left recursive, a fact ultimately deriving from our choice of reading and writing from left to right. The principal remaining problem in applying the new grammars is removing left recursion from the original CFG without recourse to the nonintuitive and space-inefficient approach of in Table 2.20.

²⁸See page 57.

Supposing that there is a CFG for the language to be compiled. It can usually be turned into a REG with no left recursion. The transformation takes rules of the form

$$\begin{aligned} N &= N \alpha \\ N &= \beta \end{aligned}$$

to a rule of the form

$$N = \beta(\alpha)^*$$

Nonterminal N is no longer left recursive. There may also be other recursions in the CFG — they are usually best left alone. If $*$ and parentheses are implemented, the task is done.

Exercises

109. [1,1] The exercises 84 to 97 should be reread and perhaps reworked using the regular expression operator $*$.
110. [1,1] Express the grammar in Table 2.22 as an executable REG and use it to test the extended version of GEM.

Here is one final comment on executable grammars. The output symbols can be replaced with, instead, calls to routines. Instead of writing

$$\begin{aligned} D &= C B^* \\ B &= '\vee' C \text{"orOP"} \end{aligned}$$

we might write something like:

$$\begin{aligned} D &= C B^* \\ B &= '\vee' C \{\text{emit(orOP)};\} \end{aligned}$$

where everything inside the curly braces is to be executed each time it is reached during the parse. This is, in fact, the method of the UNIX **yacc** program. Of course, there is no possibility of automatically backtracking across whatever arbitrary C code the user might write inside the curly braces. If one, however, saves a *journal of calls* to C code instead of actually calling them, and later processes the journal on a second pass, one can backtrack by erasing parts of the journal. This issue does not arise in **yacc** because the method of constructing a bottom-up **yacc** processor precludes the need to backtrack.

2.6 Parser Construction

Since the canonical parse is useful in determining the sequence of expression operations, and because it is also useful in providing a structure on which other compile-time actions may be taken, building a parser is an essential step in building a compiler.

Construction Steps

A recursive descent parser is designed from the CFG and REG for its language. There is one recursive function per nonterminal in the REG; the output is the sequence of rule numbers from the CFG. The problem is to get the functions written. There are four steps:

1. Describe the source language with a CFG.
2. Build an equivalent REG without left recursion.
3. Write one parameterless procedure for each nonterminal of the REG.
 - (a) There will be one REG input action for each terminal in the rule.
 - (b) There will be one recursive call for each nonterminal in the REG rule.
4. Provide one output action for each rule in the CFG.

The proposition grammars from Tables 2.8 and 2.22 are such a CFG-REG pair. Here they are again.

Proposition	=	Disjunction;	r0
Disjunction	=	Disjunction '∨' Conjunction;	r1
Disjunction	=	Conjunction;	r2
Conjunction	=	Conjunction '∧' Complement;	r3
Conjunction	=	Complement;	r4
Complement	=	'¬' Boolean;	r5
Complement	=	Boolean;	r6
Boolean	=	't';	r7
Boolean	=	'f';	r8
Boolean	=	'(' Disjunction ')';	r9

Proposition	=	Disjunction;
Disjunction	=	Conjunction ('∨' Conjunction)*;
Conjunction	=	Complement ('∧' Complement)*;
Complement	=	'¬'? Boolean;
Boolean	=	't' 'f' '(' Disjunction ')';

As a first approximation to the parser code to be written, each rule in the REG is a template for the procedure named by the left side of the rule. Each nonterminal symbol on the right side of a rule is a procedure call and each terminal symbol is a command to examine the input for the terminal symbol, to discard it if found, or otherwise to issue a diagnostic and quit.

The implementation starts (in C programs) with function `main`. It must get the initial state established. Then, since the entire language consists of a Proposition, `main` calls Proposition. Assuming that Proposition does its job,

`main` only need clean up upon regaining control. `Proposition`, likewise, may rely on `Disjunction` to do its job.

The generation of output for grammar rules to form the canonical parse is independent of the process of recognition. The insertion of the generators can be done after the recognizer is built. Where to insert them is relatively obvious from the observation of the two grammars side-by-side. Here is the code in C.

Given rule

Proposition = Disjunction

write

```
int ch;                                /* global next character */

main() {
    ch = getchar();                    /* first character */
    Proposition();                     /* goal symbol of grammar */
    if (ch == EoF) return 0;          /* all text used? */
    else Error();                     /* abnormal exit */
}

Proposition() {
    Disjunction();                    /* descent */
    Rule(r0);                          /* record parse step */
    return;                            /* ascent */
}
```

The implementation of the rule for `Disjunction` has done its job when it has found a `Conjunction` followed by the longest possible series of: \vee followed by `Conjunction`. The code follows. Note that the regular expression repetition operator `*` turns into a loop which is terminated when it does not find \vee . It isn't always quite this easy.

Given rule

Disjunction = Conjunction (' \vee ' Conjunction)*

write

```
Disjunction() {
    Conjunction();                    /* descent */
    Rule(r2);                          /* report parse step */
    while (ch == '\vee') {            /* look for vee */
        ch = getchar();                /* discard \vee */
        Conjunction();                /* descent */
        Rule(r1);                      /* report parse step */
    }
}
```

```

    }
    return;                                /* ascent */
}

```

Skipping ahead to the last nonterminal, there are three alternatives for Boolean:

```

Boolean = 't' | 'f' | '('Disjunction ')'
write

```

```

Boolean() {
  switch (ch) {                            /* 3 cases */
  case 't':                                /* constant true */
    ch = getchar();                        /* discard t */
    Rule(r7); break;
  case 'f':                                /* constant false */
    ch = getchar();                        /* discard f */
    Rule(r8); break;
  case '(':                                /* parentheses */
    ch = getchar();                        /* discard ( */
    Disjunction();                         /* descent */
    if (ch == ')') ch = getchar();        /* discard ) */
    else Error();                          /* abnormal exit */
    Rule(r9); break;                       /* found parenthesized expr */
  default: Error("something bad");
  }
  return;                                  /* ascent */
}

```

Exercises

111. [10,99] Given the rules

Conjunction = Complement ('^' Complement)*

Complement = '¬'? Boolean

write procedures Conjunction and Complement to generate rules $r3$, $r4$, $r5$, $r6$ so that

- All matches are positive (use '==').
- After every match on ch , call `getchar`. There should be exactly one “discard” for every terminal in the REG.
- There is one call to generator `Rule` for each rule of the CFG.

112. [10,50] Put the procedures all together, compile them, and parse Propositions. The implementation of `Rule()` can just contain a print statement.

113. [1,1] The calls to the error routine could pass a string parameter giving a diagnostic. Write down the best diagnostic you can think of for each of the three calls to `Error()`. What other information would you like the error routine to have to enhance the quality of its diagnostics?
114. [1,1] It would seem that since there are so many things that are not syntactically a Proposition that there would be more calls to `Error()`. Why aren't there?
115. [1,1] Suppose that instead of aborting the run, `Error()` returns to continue translation after issuing a diagnostic. What must you do to insure that the parser does not get into a loop?
116. [1,1] Implement a parser for Michels' language (see Section 2.4).

What you should have learned

How to transform a CFG for expressions into a REG. How to use the REG as a design template to build a recursive descent parser. What it takes to formulate and report diagnostics for syntax errors.

2.7 Scanning

Up to this point text has been treated as a sequence of abstract symbols. When a concrete form of symbols was needed, ASCII characters were used. Modern programming languages need one more layer of descriptive power — called the lexicon. A lexicon for natural language describes words, numbers, punctuation and whitespace. The lexicon for a programming language is used to distinguish reserved words like `if` from identifiers, discard whitespace and comments, and to gather into handy chunks the various constants of the language such as strings, characters and natural numbers.

The implementation of the lexicon is done in a module called the *scanner*; every compiler has one. Recall the grammar **Scan.g** for GEM. Its purpose was to discard non-significant characters so that GEM could accept the text as executable code. This was the first hint that what our eye demands in terms of spatial layout will require some preprocessing to be done.

Scanning itself has two phases: first the raw input text is broken into pieces called *lexemes*, and the lexemes are associated with the CFG of the source language. The result is to: (1) have discarded the parts of the input not needed for later processing, (2) to identify the words and operators of the language (called *tokens*), and (3) save other information just in case it is needed later.

The name of the file from which the token came, the file line and column location of a lexeme, and textual representation of the lexeme are examples of “other” information. One could stretch this concept of “other” to include the last date/time the token was modified and the telephone number of the person who modified it. Often the motivation for saving information about tokens is to

Program	= Lexeme* EoF
Lexeme	= Word Integer Operator Separator String WhiteSpace Comment ;
Word	= Letter (Letter Digit)*;
Integer	= Digit ⁺ ;
Operator	= Op ⁺ ;
Op	= '+' '-' '*' '/' '\' '~' ':' '=' '<' '>'
Separator	= '.' '(' ')' ',' ';' '#';
String	= '"' (Ascii - EoL - EoF - '"')* '"';
WhiteSpace	= ' '+ EoL;
Comment	= '`' (Ascii - EoL - EoF)*;

Table 2.24: An REG for the Lexical Structure of X

supply it to diagnostic routines, debuggers and other programming environment tools that are used in conjunction with a compiler.

Lexical Structure

A conventional programming language will have a dozen or fewer kinds of lexemes, and a hundred or more kinds of tokens. Which token is which is a matter of table lookup. The language X has 7 kinds of lexemes and 38 kinds of tokens.

A REG for the lexical structure of X is given in Table 2.24. The REG is, in fact, just a readable way to present a largish regular expression, as one can see by back-substituting the definition for each nonterminal. One should note that such a REG contains a hidden kind of ambiguity: one does not know whether to treat AB as one word AB or two — A and B. The conventional solution is to resolve the ambiguity in favor of the longest match.

The definitions of Letter, Digit, Ascii, EoL and EoF are not included in the grammar.²⁹ From the viewpoint of the human reader, the definitions are clutter. From the viewpoint of implementation, the definitions are treated as suggested for L, D and C in GEM: predefined and to be implemented as efficiently as the hardware will allow³⁰.

²⁹EoL stands for end-of-line; EoF stands for end-of-file

³⁰See Table 2.theLCD

An important property of the REG in Table 2.24 is that the kind of lexeme is determined by its first character. For example, if a lexeme starts with a Letter, it is a Word. It is also true that the right end of each lexeme can be determined by looking for the first character that is not allowed inside it. For example, a Comment is terminated by an end-of-line. Conventional programming languages have neither property. (Consider comments in C, for instance.) These two conditions are accepted limitations on language design for X because it makes scanning simpler.

Isolating Lexemes

Supposing that the kind of lexeme can be determined by examining its first character, the lexeme class can be looked up using the ASCII code of the first character as key. Given that one knows what the class of the lexeme is, the first character can be processed. This is relevant only for String since no other lexeme in X has a beginning delimiter that is not allowed inside the lexeme. The remainder of the lexeme can now be processed, down to the final delimiter.

The function in Table 2.25 gives the general outline of the code to isolate a lexeme. The variable 'ch' holds the next input; the variable α accumulates the lexeme. Functions in X are called by placing their name (in quotes) in the middle of an assignment. The function "Inside", derived from the lexical REG, is defined as a set or order pairs in Table 2.26. If "Lex.x" calls itself recursively, it is the characters isolated by the last call that form α .

File Lex.x

```

lc := "LexemeClass" := ch;
bs := "Inside" := lc;
if lc = String →
    lexeme := ch;                `capture "
    ch := "getchar" :=          `discard "
[] else
    lexeme := λ;                `start empty
fi;
it                               `collect chrs
    if ch ∈ bs →
        lexeme := lexeme ch;    `catenate
        ch := "getchar" :=
    [] else exit
    fi
ti;
if lc = String →
    if ch = EoL ∨ ch = EoF →    `not allowed
        := "Error" :=;        `complain
    fi;
    lexeme := lexeme ch;        `capture "
    ch := "getchar" :=;        `discard "

```

```

[] lc = Separator →
    lexeme := ch;
    ch := "getchar" :=
[] lc = WhiteSpace ∨ lc = Comment →
    lexeme := "Lex.x" :=          ` recur
[] else
fi
α := lexeme                       ` output of Lex.x

```

Table 2.25: Model Lexer

$$\left\{ \begin{array}{ll} \langle \text{Word, Letter} \cup \text{Digit} \rangle, & \langle \text{Integer, Digit} \rangle, \\ \langle \text{Operator, Op} \rangle, & \langle \text{Separator, \{ \}} \rangle, \\ \langle \text{String, Ascii} - \{ \text{EoL, EoF, "} \} \rangle, & \\ \langle \text{WhiteSpace, \{ \}, \text{EoL}} \rangle, & \langle \text{Comment, Ascii} - \{ \text{EoL, EoF} \} \rangle \end{array} \right\}$$

Table 2.26: Function Inside: The Internal Structure of Lexemes

The treatment of operators in *x* is unconventional. Some set of characters is chosen from which multicharacter operators are to be built; these characters are like letters in identifiers in that any number of them can be clumped to describe an operator. The assignment operator ‘:=’ is a typical Operator. The effect is to require blanks in some places where conventional languages do not. For instance, the C expression “*x-- +x*” means “*x-- +x*”. The space is optional in C but would be required by a scanner of the kind that is being described. One advantage of the Operator concept is to make it easy to put new operators into the language, or to take them out, as the language is developed.

Exercises

117. [1,50] Extend the lexical structure of *x* to allow simple character constants.
118. [1,50] Extend the lexical structure of *x* to allow underbar in identifiers.
119. [1,50] Extend the lexical structure of *x* to allow underbar inside identifiers (but not starting or ending them).
120. [20,10] Write an unambiguous REG for the lexical structure of *x* (that is, one that does not depend on longest-match). (Hint: have several kinds of lexeme lists, one ending in Identifier, one ending in Integer, one ending in WhiteSpace, and so on.)
121. [10,50] The real number 12.31 can be processed by turning ‘.’ into a digit, as the cost of 1.2.3.1 also being accepted by the lexer. This usually does not cause any trouble because the mistake will be caught during numeric conversions. One can also look at 12.31 as three lexemes, ‘12’, ‘.’ and ‘31’. The disadvantage is that now 12 . 31, with imbedded blanks, is also acceptable. What modifications would have to be made to the lexical REG to allow the second alternative?

122. [10,10] In C the assignment `*x++<= -+++y;` is allowed. What are the tradeoffs between this kind of lexical structure and the one proposed for x? Why is the blank after the = necessary? Or is it? Or do you care to know?

Leaving a terminating quote off a string turns a program inside out and can cause troubles for translators and programmers alike. The exclusion of end-of-line from strings in the lexicon for x is not necessary, but rather intended to provide robustness in programs. The need for long strings may be handled by interpreting successive strings in the text with nothing but whitespace separating them as one concatenated string. This is the solution in ANSI C; it pushes the problem of handling line breaks into the parser, rather than leaving the complicated interaction of long strings and short lines to scanning.

One can implement a lexer patterned after the one shown in Table 2.25. What this program actually does is examine and discard each lexeme.

Exercises

123. [1,1] Explain the function inside in Table 2.26 line by line.
124. [10,10] Extend the solution in Table 2.25 to handle the backslash escape in C strings. You must be able to accept `"\n"` as well as `"\007"`. It is convenient to allow `"\x"` to stand for x wherever x does not have some other interpretation.
125. [1,1] In C division by a dereferenced value caused a translation diagnostic.

```
*ThisIsAPointer = ThisIsANumerator/*ThisIsAPointer;
```

What is the problem? What is the cure? (Turning the clock back is not an acceptable answer.)

126. [50,25] In C all of the following: `"\n"`, `1.2e-3`, `/* This is a Comment */`, `a**b`, as well as a convention for hiding linebreaks, defeat the relatively simple lexer algorithm just presented. Other conventional languages have similarly complex lexical constructs. It is not hard to process such things but it does complicate the lexer. There is a single loop in the one presented here; there may be a loop per kind of lexeme in a more complex lexer. The multiple loop lexer will look like the program in Table 2.27. Even though it is not consistent with other simplifications in the presentation you are enjoying here, implement a lexer for C.³¹

File BigLex.x

```
lexeme := λ;
if ch ∈ Letter →
```

³¹This can be done in about 500 lines of C code.

```

    it
      if ch ∈ (Letter ∪ Digit) →
        lexeme := lexeme ch;
        ch := "getchar" :=
      [] else
        exit
      fi
    ti;
    α := lexeme
  [] ch ∈ Digit →
    it
      if ch ∈ Digit →
        lexeme := lexeme ch;
        ch := "getchar" :=;
      [] else
        exit
      fi
    ti;
    α := lexeme
  [] etc.
    etc. etc.
  fi

```

Table 2.27: A Bigger, More General Lexer

What you should have learned

How to describe the lexical structure of a programming language and how to implement descriptions.

Lexemes to Tokens

Once a lexeme is extracted from the text, it must be compared with the symbols of the language. The reserved symbols of X are presented in Table 2.28. The table represents a discrete function, L2T, mapping strings to token names. Token names typically stand for unique but otherwise arbitrary small integers. There is really no one best naming scheme for tokens, and it is tedious to invent, tabulate and maintain large tables of arbitrary names. One simple scheme is to give short name to every ASCII character and name sequences with a catenation of the short names. The letters can stand for themselves. The reader can deduce the rest of the character names from the table below. It can be generated automatically.

{	$\langle \text{if}, \text{if} \text{TOKEN} \rangle,$	$\langle \text{fi}, \text{fi} \text{TOKEN} \rangle,$	$\langle \text{else}, \text{else} \text{TOKEN} \rangle,$
	$\langle \text{it}, \text{it} \text{TOKEN} \rangle,$	$\langle \text{ti}, \text{ti} \text{TOKEN} \rangle,$	$\langle \text{exit}, \text{exit} \text{TOKEN} \rangle,$
	$\langle \text{be}, \text{be} \text{TOKEN} \rangle,$	$\langle \text{eb}, \text{eb} \text{TOKEN} \rangle,$	
	$\langle \text{true}, \text{true} \text{TOKEN} \rangle,$	$\langle \text{false}, \text{false} \text{TOKEN} \rangle,$	$\langle \text{b2i}, \text{b2i} \text{TOKEN} \rangle,$
	$\langle \rightarrow, \text{subgt} \text{TOKEN} \rangle,$	$\langle ::, \text{clncln} \text{TOKEN} \rangle,$	$\langle :=, \text{cIneq} \text{TOKEN} \rangle,$
	$\langle \setminus, \text{lsrs} \text{TOKEN} \rangle,$	$\langle \wedge, \text{rlls} \text{TOKEN} \rangle,$	$\langle \sim, \text{twidle} \text{TOKEN} \rangle,$
	$\langle <, \text{lt} \text{TOKEN} \rangle,$	$\langle \leq, \text{lteq} \text{TOKEN} \rangle,$	$\langle =, \text{eq} \text{TOKEN} \rangle,$
	$\langle <>, \text{ltgt} \text{TOKEN} \rangle,$	$\langle \geq, \text{gteq} \text{TOKEN} \rangle,$	$\langle >, \text{gt} \text{TOKEN} \rangle,$
	$\langle +, \text{add} \text{TOKEN} \rangle,$	$\langle -, \text{sub} \text{TOKEN} \rangle,$	
	$\langle *, \text{mul} \text{TOKEN} \rangle,$	$\langle \setminus, \text{div} \text{TOKEN} \rangle,$	$\langle //, \text{divdiv} \text{TOKEN} \rangle,$
	$\langle \cdot, \text{dot} \text{TOKEN} \rangle,$	$\langle (, \text{lpar} \text{TOKEN} \rangle,$	$\langle \rangle, \text{rpar} \text{TOKEN} \rangle,$
	$\langle ;, \text{semi} \text{TOKEN} \rangle,$	$\langle ,, \text{comma} \text{TOKEN} \rangle,$	$\langle \#, \text{sharp} \text{TOKEN} \rangle$

Table 2.28: Function L2T Mapping Reserved Lexemes to Token Codes

Scanning turns a sequence of characters into a sequence of tokens. A function Scanner is defined in Table 2.29. When Scanner fails (because it is undefined for its input), it reports an error. In the case of x, the only errors are illegal symbol, string missing a trailing quote, and unknown Operator. Function MakeToken constructs a token out of the information provided.

For any string of characters γ , there is a function Scanner(γ) which gives the sequence of tokens. Scanner is defined as follows: $\gamma = \alpha\beta$ and at each recursive application of Scanner, α is the longest string to satisfy one of the conditions in Table 2.29.

<i>condition</i>	<i>result</i>
$\alpha\beta = \lambda$	\Rightarrow MakeToken(λ, EoF)
$\alpha \in \mathcal{D}(\text{L2T})$	\Rightarrow MakeToken($\alpha, \text{L2T}(\alpha)$)Scanner(β)
$\alpha \notin \mathcal{D}(\text{L2T}) \wedge \text{Word} \rightarrow^* \alpha$	\Rightarrow MakeToken($\alpha, \text{Identifier}$)Scanner(β)
Integer $\rightarrow^* \alpha$	\Rightarrow MakeToken($\alpha, \text{Integer}$)Scanner(β)
String $\rightarrow^* \alpha$	\Rightarrow MakeToken(α, String)Scanner(β)
WhiteSpace $\rightarrow^* \alpha$	\Rightarrow Scanner(β)
Comment $\rightarrow^* \alpha$	\Rightarrow Scanner(β)

Table 2.29: Definition of Scanning

It is not usually the case that the whole input is scanned before the tokens are used. Instead a function Scan provides one token per call. It is defined in terms of the function Scanner by

$$\text{Scanner}(\alpha) = \langle t_1, t_2, \dots \rangle \Rightarrow \text{Scan}(\alpha) = t_1$$

Exercise

127. [40,1] A particularly straightforward way to implement Scan() is to switch on every character, without regard for classes such as Letter and Digit,

until the token is isolated. Such a scanner is big but easy to understand and also fast. The isolation of lexemes and the identification of tokens is all mixed together. A fragment of such a scanner written in a C-like notation follows:

```

Scan() {
  switch (ch) {
  case 'i':
    ch = getchar();
    switch (ch) {
    case 'f':
      ch = getchar();
      lexeme = "if";
      if (ch ∉ Letter ∪ Digit) {
        return MakeToken(lexeme, ifTOKEN);
      } else {
        while (ch ∈ Letter ∪ Digit) {
          lexeme = lexeme, ch;
          ch = getchar();
        }
        return MakeToken(lexeme, Identifier);
      }
      }
    }
  }
}

```

Implement a scanner for X this way. Compare the result with that of the previous exercise. How deeply would switches nest for C? For ADA?

What you should have learned

The means for describing and implementing lexical analysis. The concept of a token and scanner.

Lexicon

The lexical description of a language can be summarized by a lexicon, formed by adding the information on reserved words and operators the lexical REG. The lexicon is principally of value in user-level description since the actions of the scanner are defined in terms of the lexical grammar and reserved lexeme table L2T rather than the lexicon itself.

The lexicon for X is given in Table 2.30. It adds to the description of the lexemes the additional detail required for tokens.

Program	= (Token WhiteSpace Comment)* EoF;
Token	= Identifier Constant OperatorId ReservedWord ReservedOperator Separator String ;
Identifier	= Letter (Letter Digit)* – ReservedWord;
ReservedWord	= 'if' 'fi' 'else' 'it' 'ti' 'exit' 'be' 'eb' 'true' 'false' 'b2i' ;
Constant	= Digit ⁺ ;
OperatorId	= Operator ⁺ – ReservedOperator;
Operator	= '+' '-' '*' '/' '\' ':' '=' '<' '>' ;
ReservedOperator	= '::' '->' ':= '<' '<=' '=' '>' '>=' '>' '+' '-' '*' '/' '//' ;
Separator	= '.' '(' ')' ',' ';' '#';
String	= '"' (Ascii – EoL – '"')* '"';
WhiteSpace	= ' ' + EoL;
Comment	= '`' (Ascii – EoL)*;

Table 2.30: The Lexicon for x

Exercise

128. [10,10] Revisit the exercises (Exercises 124 ff) to extend the lexical REG. Extend the lexicon instead.

What you should have learned

A grammatical technique for describing the token structure of a language.

Providing Lookahead

Up until now, it has been assumed that the caller of the scanner is ready to use the tokens, one at a time, as they come. This is often inconvenient: a decision may be required that depends on the *following* token, rather than just the current one. The obvious programming solution is for the caller to tuck away the current token in a temporary, call the scanner once again, make the decision and then go ahead with two tokens in hand.

This works, but it is not the best idea for two reasons. The first is that lookahead happens a lot, so there are potentially a lot of temporaries, and a lot of places where the user must be prepared to take care of two tokens rather than the usual one. There is a lot of potential for simple minded blunders. The second reason is that the most elegant code might require that the saved token actually show up in the usual global variable, rather than a special temporary.

One solution is to implement the lookahead function once and for all in the scanner, leaving the mechanism hidden to the users. The interface is a function `Lookahead` which has the same value as `Scan` except that it can be called as often as one likes while leaving the input undisturbed. The implementation of lookahead is not an issue in Syntax and is not discussed further here.

Efficiency

While `x` is a relatively simple language, its lexical structure is of the same order of complexity as that of conventional languages. When one knows how to do it, writing a scanner is a few hours of work. When one modifies a scanner already built and understood, it takes even less time.

The most expensive operations are finding α , the longest-match string described by the lexical grammar, and looking in 'L2T' to see if α is reserved. The scanner must be especially efficient because it is dealing with the program a character at a time. In a compiler where everything else has been done to speed things up, the scanner will be the bottleneck. The scanner for `HYPER` uses a hash access method to speed up access to the reserved lexeme table 'L2T'. The scanner in `HYPER` is 2 to 3 times slower than it would be with every speedup applied.

To avoid moving strings around after they are scanned, `HYPER` enters every new token into 'L2T' with appropriate values, and returns only handles for elements in 'L2T'. This means that the definition of 'L2T' changes as scanning

proceeds. This can be represented by a definition of `Scanner` which is parameterized by a changing ‘L2T’ as shown in Table 2.31.

<i>condition</i>	<i>result</i>
$\alpha\beta = \lambda$	\Rightarrow <code>MakeToken</code> (λ , <i>EoF</i>)
$\alpha \in \mathcal{D}(\text{L2T})$	\Rightarrow <code>MakeToken</code> (α , <code>L2T</code> (α)) <code>Scanner</code> _{L2T} (β)
$\alpha \notin \mathcal{D}(\text{L2T}) \wedge \text{Word} \rightarrow^* \alpha$	\Rightarrow <code>Scanner</code> _{L2T'} ($\alpha\beta$) where $\text{L2T}' = \text{L2T} \cup \{\langle \alpha, \text{Identifier} \rangle\}$
$\alpha \notin \mathcal{D}(\text{L2T}) \wedge \text{Integer} \rightarrow^* \alpha$	\Rightarrow <code>Scanner</code> _{L2T'} ($\alpha\beta$) where $\text{L2T}' = \text{L2T} \cup \{\langle \alpha, \text{Integer} \rangle\}$
$\alpha \notin \mathcal{D}(\text{L2T}) \wedge \text{String} \rightarrow^* \alpha$	\Rightarrow <code>Scanner</code> _{L2T'} ($\alpha\beta$) where $\text{L2T}' = \text{L2T} \cup \{\langle \alpha, \text{String} \rangle\}$
<code>WhiteSpace</code> $\rightarrow^* \alpha$	\Rightarrow <code>Scanner</code> _{L2T} (β)
<code>Comment</code> $\rightarrow^* \alpha$	\Rightarrow <code>Scanner</code> _{L2T} (β)

Table 2.31: Definition of `Scanner`_{L2T}

In fact `HYPER` uses an incremental algorithm, avoiding scanning almost altogether when compilation is repeated after a small program change. The implementation of incremental scanning is not an issue in `Syntax`, thus discussion is deferred until later.

What you should have learned

The concept of lookahead. A technique of defining scanning in terms of a changing lexeme-to-token table. Some efficiency considerations.

2.8 Recursive Descent Parsing

The final topic in syntax is parsing when things get tough. The grammar in Table 2.32 extends the proposition grammar to assignment statements; a Disjunction by itself is also allowed. This CFG is a prototype of a problem situation for recursive descent parsers. The problem is that the `Identifier` starting an `Assignment` may or may not start a `Disjunction`. If it does, `Assignment` should call `Disjunction`. Otherwise it should call `Variable`, discard `:=`, and call itself recursively.³²

The grammar in Table 2.32 introduces one additional concept; the symbol `Identifier` is a terminal (not in $\mathcal{D}(\Pi)$) but also not quoted. This is the convention to indicate that a terminal is defined in the lexicon, at a lower level of abstraction. The use of a scanner also allows constants `true` and `false` to be spelled out as multicharacter words. In the recursive descent code, the input is changed from

```
ch = getchar();
to
```

³²This grammar poses no problem for the bottom-up LALR(1) parsers described in Chapter 5.

```
token = Scan();
```

Statement	=	Assignment;	r0
Assignment	=	Variable ':=' Assignment;	r1
Assignment	=	Disjunction;	r2
Disjunction	=	Disjunction '∨' Conjunction;	r3
Disjunction	=	Conjunction;	r4
Conjunction	=	Conjunction '^' Complement;	r5
Conjunction	=	Complement;	r6
Complement	=	'¬' Boolean;	r7
Complement	=	Boolean;	r8
Boolean	=	'true';	r9
Boolean	=	'false';	r10
Boolean	=	Variable;	r11
Boolean	=	(' Disjunction ')';	r12
Variable	=	Identifier;	r13

Table 2.32: CFG for Assignments

Exercises

129. [10,99] Try writing Statement right now, before reading about the standard solutions. If you cannot do it, then read the first solution below and return to do this exercise.
130. [20,30] Suppose that Variable were arbitrarily complex, containing subscripts, structure qualification, indirection or anything else that makes trying to look ahead futile. Now repeat the previous exercise. If you cannot do it, then read the second solution below and return to this exercise.

The first step in either case is, of course, to write a REG as in Table 2.33.

Statement	=	Assignment ';';
Assignment	=	Variable ':=' Assignment Disjunction;
Disjunction	=	Conjunction ('∨' Conjunction)*;
Conjunction	=	Complement ('^' Complement)*;
Complement	=	'¬'? Boolean;
Boolean	=	'true'
		'false'
		Variable
		(' Disjunction ')';
Variable	=	Identifier;

Table 2.33: REG for Assignments

As hinted in the first exercise, the scanner's lookahead feature often will suffice to resolve the difficulty posed by the local ambiguity introduced by Identifier starting Assignments. One writes:

```

Statement() {
    Assignment();
    if (token == semiTOKEN) token = Scan();
    else Error();
        Rule(r0);
}

Assignment() {
    if (token == Identifier ^ LookAhead() == asgTOKEN) {
        Variable();
        token = Scan();           /* discard := */
        Assignment();
        Rule(r1);
    } else {
        Disjunction();
        Rule(r2);
    }
}

```

and carries on as before.

Suppose that, as in the second exercise above, one cannot use lookahead. The second solution is to first parse the common part of the input as described in the REG and then finish it off only after the correct path is known. Since the common part of the input will have already been processed, provision must be made to skip it in the lower level recursive routines. For the grammar above we get the following implementation.

```

Assignment() {
    if (token == Identifier) {
        Variable();
        if (token == asgTOKEN) {
            token = Scan();           /* discard := */
            Assignment();
            Rule(r1);
        } else {
            Disjunction(YES);
            Rule(r2);
        }
    } else {
        Disjunction(NO);
        Rule(r2);
    }
}

```

```

}

Disjunction(int VariableGone) {
    Conjunction(VariableGone);
    Rule(r3);
    while (token == orTOKEN) {
        token = Scan();           /* discard  $\vee$  */
        Conjunction(NO);
        Rule(r4);
    }
}

Conjunction(int VariableGone) {
    Complement(VariableGone);
    Rule(r5);
    while (token == andTOKEN) {
        token = Scan();         /* discard  $\wedge$  */
        Complement(NO);
        Rule(r6);
    }
}

Complement(int VariableGone) {
    if (token == negTOKEN) {
        token = Scan();         /* discard  $\neg$  */
        Boolean(NO);
        Rule(r7);
    } else {
        Boolean(VariableGone);
        Rule(r8);
    }
}

Boolean(int VariableGone) {
    if (VariableGone == YES) {
        Rule(r11)               /* no Identifier to discard */
    } else if (token == trueTOKEN) {
        token = Scan();         /* discard true */
        Rule(r9);
    } else if (token == falseTOKEN) {
        token = Scan();         /* discard false */
        Rule(r10);
    } else if (token == Identifier) {
        Variable();
        Rule(r11);
    }
}

```

```

    } else if (token == lparTOKEN) {
        token = Scan();                /* discard ( */
        Disjunction(NO);
        if (token == rparTOKEN) token = Scan();
        else Error("Missing ");
        Rule(r12);
    } else Error("Unknown operand");
}

Variable() {
    token = Scan();                    /* discard Identifier */
    Rule(r13);
}

```

Exercises

131. [1,1] Rework the parser for Assignment, this time allowing Assignment instead of Disjunction inside the parentheses of Boolean.
132. [1,1] Devise a CFG for arithmetic assignments analogous to the CFG for boolean assignments presented in this section. Implement a recursive descent parser for it.
133. [1,1] Implement a recursive descent parser for English, using the CFG in Table 2.7 or your own extension of it.
134. [1,1] Conventional programming languages present a fair challenge to the writer of a recursive descent parser. The following grammar is abstracted from the ANSII C REG. Write function AbstractDeclarator. You see that 9 functions are required according to the REG. Since a Pointer *always* starts with a '*' and neither AbstractDeclarator nor AbstractQualifier do, the problem of the optional Pointers is resolvable. The next problem arises in ParameterDeclaration; when does one call Declarator and when does one call AbstractDeclarator? It turns out they are distinguished, in general, only when the Identifier of the Declarator shows (or fails to show) up. That is:

Declarator \rightarrow "*"**x[]"

AbstractDeclarator \rightarrow "*"**[]"

Try your hand at a solution and then read the following material.

Lookahead won't help with the C declarator problem. One solution is to implement a parametric function Declarator(which) and call it with parameters Abstract, Plain, DontKnow. In ParameterDeclaration(), call Declarator(DontKnow). As soon as the situation resolves itself, then continue the processing with the now-known value. As in the case of the assignments, the

parameter must be carried through the recursion to be available to lower level routines. The outline of a solution follows:

AbstractDeclarator:

```
Pointer
Pointer? '(' AbstractDeclarator ')' AbstractQualifier*
Pointer? AbstractQualifier+
```

AbstractQualifier:

```
'[' Constant? ']'
 '(' ParameterTypeList? ')'
```

ParameterTypeList:

```
ParameterDeclaration (',' ParameterDeclaration)* (',' '...'?)?
```

ParameterDeclaration:

```
DeclarationSpecifiers Declarator
DeclarationSpecifiers AbstractDeclarator?
```

DeclarationSpecifiers:

```
'int' | 'static'
```

Declarator:

```
Pointer? (Identifier | '(' Declarator ')') DeclaratorQualifier*
```

DeclaratorQualifier:

```
'[' Constant? ']'
 '(' IdentifierList | ParameterTypeList? ')'
```

Pointer:

```
('*' 'volatile'?)*
```

IdentifierList:

```
Identifier (',' Identifier)*
```

It is worth a moment or two of reflection after one completes this exercise. Is the baroque C declaration syntax worth it? Did it extend gracefully to function prototypes (the reason for AbstractDeclarator)? The issue here is whether a difficult-to-implement construct is also difficult-to-use.

Exercises

135. Implement a parser for arithmetic assignments in C.
136. Implement a parser for AbstractDeclarator.

What you should have learned

Two techniques to extend the top-down parsing technique to more complex grammars

Structural Grammars for X

The form of X is defined by a REG and a CFG with the lexical level items left as terminal symbols as shown in Tables 2.36, 2.34 and 2.35.

Exercises

137. [1,1] Confirm that the CFG and REG for X in Tables 2.36 and 2.34 and 2.35 describe the same language.
138. [1,1] Write a parser for X.

What you should have learned

The structure of X and, in principal, how to write a parser for it.

Program:	Statements
Statements:	Statement Statements ';' Statement
Statement:	'exit' Block Selection Iteration Assignment Subprogram
Block:	'be' Identifiers '.' Statements 'eb'
Identifiers:	Identifier Identifiers Identifier
Selection:	'if' Hint1 Alternatives 'fi' 'if' Hint1 Alternatives 'else' Statements 'fi'
Hint1:	λ
Alternatives:	Alternative Alternatives '[' Alternative
Alternative:	Guard '→' Statements
Guard:	Expression
Iteration:	'it' Hint2 Statements 'ti'
Hint2:	λ
Assignment:	Variables ':=' Expressions Outputs ':=' String ':=' Inputs
Outputs:	Variables
Inputs:	Expressions
Subprogram:	String
Variables:	Identifier Variables ',' Identifier
Expressions:	Expression Expressions ',' Expression

Table 2.34: CFG for X (control)

Expression:	Disjunction
Disjunction:	Conjunction Disjunction '∨' Conjunction
Conjunction:	Complement Conjunction '^' Complement
Complement:	Relation '¬' Relation
Relation:	Sum Sum '<' Sum Sum '≤' Sum Sum '=' Sum Sum '≠' Sum Sum '≥' Sum Sum '>' Sum
Sum:	Term '−' Term Sum '+' Term Sum '−' Term
Term:	Factor Term '*' Factor Term '/' Factor Term '//' Factor
Factor:	'true' 'false' Integer Identifier '('Expression ')' 'b2i' '('Expression ')'

Table 2.35: CFG for X (expressions)

Program:	Statements
Statements:	Statement (';' Statement)*
Statement:	λ 'exit' 'be' Identifier ⁺ '.' Statements 'eb' 'if' Alternative ('[' Alternative)* ('[' 'else' Statements)? 'fi' 'it' Statements 'ti' Variables ':=' Expressions Variables [?] ':=' String ':=' Expressions [?] String
Alternative:	Expression '→' Statements
Expression:	Disjunction
Disjunction:	Conjunction ('∨' Conjunction)*
Conjunction:	Complement ('^' Complement)*
Complement:	'¬' [?] Relation
Relation:	Sum (('<' '≤' '=' '≠' '≥' '>') Sum) [?]
Sum:	'-' [?] Term (('+' '-') Term)*
Term:	Factor (('*' '/' '//') Factor)*
Factor:	'true' 'false' Integer Identifier '(' Expression ')' 'b2i' '(' Expression ')'
Variables:	Identifier (',' Identifier)*
Expressions:	Expression (',' Expression)*

Table 2.36: REG for X

Semantics in the CFG

We rely on the CFG to sort out the precedence and associativity rules of expressions by simply collecting the canonical parse and acting appropriately for each rule. It is clear, both from the output of executable grammars, and from the ability of the programmer to put *anything* into the recursive recognizer procedures, that we need not be limited to just the canonical parse. But it is well to do so for as long as possible for two reasons. The canonical parse is usually enough; if one habitually make other *ad hoc* provisions, the code will be unnecessarily complex. Moreover, other programmers, looking at the code sometime later, are aided by assuming that the standard canonical parse information was used. It is a discipline that is not often onerous.

But the exceptions occur. The prototype of this situation is the loop. The CFG is

```
Iteration      = 'it' Statements 'ti'
```

Just before leaving `Iteration`, the compiler must generate a branch back to the beginning of the loop. Unfortunately, there is no convenient rule of the canonical parse upon which to hang the action of saving the location of the start of the loop. One can violate the discipline of using only the canonical parse or one can hack the grammar.

The grammar hack involves adding a “fake” rule to enrichen the canonical parse. For example, we can write

```
Iteration      = LoopHead Statements 'ti'
LoopHead       = 'it'
```

or

```
Iteration      = 'it' Hint2 Statements 'ti'
Hint2          =
```

and hang the “save pc” action on the rule for `Hint2`. However practical, this is ugly from the viewpoint of the grammatical poet. The inserted empty rule is the least obtrusive and therefore the one used here. In fact, for large languages, the number of unpretty grammar rules may outnumber the pretty ones. For this reason it is often advisable to keep a pretty reference grammar and a useful one to define the parse. For `Iteration`, the C code is:

```
Iteration() {
    token = Scan();                /* discard it */
    Rule(Hint2);
    Statements();
    if (token == tiTOKEN) token = Scan(); /* discard ti */
    else Error();
    Rule(r);
}
```

The semantic additions to the grammar that reduce its value for documenting the form of the language are kept as few and as simple as possible.

Exercises

139. needed

What you should have learned.

A regular way of augmenting the canonical parse to reflect semantics.

Chapter Summary

Purely grammatical means can be used to describe a restricted class of translators. The executable grammars illustrate recognition, parsing and translation. The implementation of an executor for grammars can be accomplished in about 100 lines of C.

The form of traditional computer languages demands a two-level grammar to separate off the lexical detail from the structure of the language. This separation of concerns carries through to the implementation of a scanner and parser, one for each the two levels of grammar.

Top-down mechanisms of translation requires that grammatical left-recursion be turned into an iteration, which is expressed with the formalism of regular expressions.

The translation process automatically provides much of the sequencing information required for translating to a sequential machine code. There is some art to handling difficult grammars and difficult sequencing. One applies the art either by bending the grammars to one's purpose, or simplifying the computer language so as not to need bent grammars.

Bibliography