THESE PAGES COME FROM MY BOOK
COMPUTATION ENGINEERING - APPLIED AUTOMATA THEORY AND LOGIC
SPRINGER 2006
PROVIDED AS A COURTESY FOR YOU TO DO ASSIGNMENT 7 and STUDY ABOUT CFGs

13

Context-free Languages

A *context-free language* (CFL) is a language accepted by a *push-down automaton* (PDA). Alternatively, a context-free language is one that has a *context-free grammar* (CFG) describing it. This chapter is mainly about context-free grammars, although a brief introduction to push-down automata is also provided. The next chapter will treat push-down automata in greater detail, and also describe algorithms to convert PDAs to CFGs and vice versa. The theory behind CFGs and PDAs has been directly responsible for the design of *parsers* for computer languages, where parsers are tools to analyze the syntactic structure of computer programs and assign them meanings (e.g., by generating equivalent machine language instructions).

A CFG is a structure (N, Σ, P, S) where N is a set of symbols known as *non-terminals*, Σ is a set of symbols known as *terminals*, $S \in N$ is called the *start symbol*, and P is a finite set of *production rules*. Each production rule is of the form

$$L \to R_1 \; R_2 \; \ldots R_n,$$

where $L \in N$ is a non-terminal and each R_i belongs to $N \cup \Sigma_\varepsilon$. We will now present several context-free grammars through examples, and then proceed to examine their properties.

Consider the CFG $G_1 = (\{S\}, \{0\}, P, S)$ where P is the set of rules shown below:

Grammar G_1:

```
S -> 0.
```

A CFG is machinery to produce strings according to production rules. We start with the start symbol, find a rule whose left-hand side matches the start symbol, and derive the right-hand side. We then repeat the

process if the right-hand side contains a non-terminal. Using grammar G_1, we can produce only one string starting from S, namely 0, and so the derivation stops. Now consider a grammar G_2 obtained from G_1 by adding one extra production rule:

Grammar G_2:

```
S -> 0
S -> 1 S.
```

Using G_2, an infinite number of strings can be derived as follows:

1. Start with `S`, calling it a *sentential form*.
2. Take the current sentential form and for one of the non-terminals N present in it, find a production rule of the form $N \rightarrow R_1 \ldots R_m$, and replace N with $R_1 \ldots R_m$. In our example, `S -> 0` matches `S`, resulting in sentential form `0`. Since there are no non-terminals left, this sentential form is called a *sentence*. Each such sentence is a member of the *language* of the CFG - in symbols, $0 \in \mathcal{L}(G_2)$. The step of going from one sentential form to the next is called a *derivation step*. A sequence of such steps is a *derivation sequence*.
3. Another derivation sequence using G_2 is

$$S \Rightarrow 1S \Rightarrow 11S \Rightarrow 110.$$

To sum up, given a CFG G with start symbol S, S is a sentential form. If $S_1 S_2 \ldots S_i \ldots S_m$ is a sentential form and there is a rule in the production set of the form $S_i \rightarrow R_1\ R_2\ \ldots R_n$, then $S_1 S_2 \ldots\ R_1\ R_2\ \ldots R_n\ \ldots S_m$ is a sentential form. We write

$$S \Rightarrow \ldots \Rightarrow S_1 S_2 \ldots S_i \ldots S_m \Rightarrow S_1 S_2 \ldots\ R_1\ R_2\ \ldots R_n\ \ldots S_m \Rightarrow \ldots.$$

As usual, we use $\Rightarrow^*$ to denote a multi-step derivation.

Given a CFG G and one of the sentences in its language, w, a *parse tree* for w with respect to G is a tree with frontier w, and each interior node corresponding to one derivation step. The parse tree for string 110 with respect to CFG G_2 appears in Figure 13.1(a).

13.1 The Language of a CFG

The language of a CFG G, $\mathcal{L}(G)$, is *the set of all sentences* that can be derived starting from S. In symbols, for a CFG G,

$$\mathcal{L}(G) = \{w \mid S \Rightarrow^* w \wedge w \in \Sigma^*\}.$$

According to this definition, for a CFG G_3, with the only production

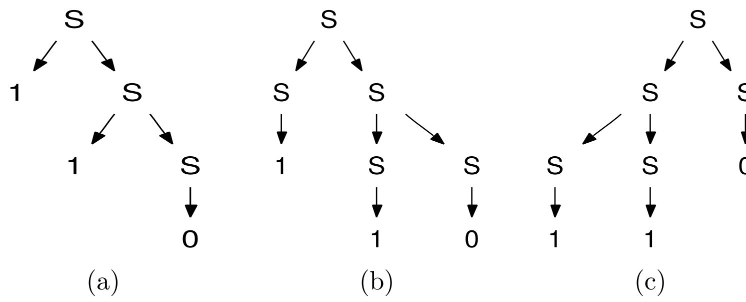

Fig. 13.1. (a) The parse tree for string 110 with respect to CFG G_2; (b) and (c) are parse trees for 110 with respect to G_4.

Grammar G_3**:**
$S \to S.$

we have $\mathcal{L}(G_3) = \emptyset$. The same is true of a CFG all of whose productions contain a non-terminal on the RHS, since, then, we can never get rid of all non-terminals from any sentential form.

A derivation sequence, in which the leftmost non-terminal is selected for replacement in each derivation step, is known as a *leftmost derivation.* A rightmost derivation can be similarly defined. Specific derivation sequences such as the leftmost and rightmost derivation sequences are important in compiler construction. We will employ leftmost and rightmost derivation sequences for pinning down the exact derivation sequence of interest in a specific discussion. This, in turn, decides the shape of the *parse tree.* To make this clear, consider a CFG G_4 with three productions

Grammar G_4**:**
$S \to SS \mid 1 \mid 0.$

The above notation is a compact way of writing three distinct *elementary productions* $S \to SS$, $S \to 1$, and $S \to 0$. A string 110 can now be derived in two ways:

- Through the leftmost derivation $S \Rightarrow SS \Rightarrow 1S \Rightarrow 1SS \Rightarrow 11S \Rightarrow 110$ (Figure 13.1(b)), or
- Through the rightmost derivation $S \Rightarrow SS \Rightarrow S0 \Rightarrow SS0 \Rightarrow S10 \Rightarrow 110$ (Figure 13.1(c)).

Notice the connection between these derivation sequences and the parse trees.

Now consider grammar G_5 with production rules

Grammar G_5**:**
$S \rightarrow aSbS \mid bSaS \mid \varepsilon.$

The terminals are $\{a, b\}$. What CFL does this CFG describe? It is easy to see that in each replacement step, an S is replaced with either ε or a string containing an a and a b; and hence, all strings that can be generated from G_5 have the same number of a's and b's. Can *all* strings that contain equal a's and b's be generated using G_5? We visit this (much deeper) question in the next section. If you try to experimentally check this conjecture out, you will find that no matter what string of a's and b's you try, you can find a derivation for it using G_5 so long as the string has an equal number of a's and b's.

Note: We employ ε, `e`, and `epsilon` interchangeably, often for the ease of type-setting. □

13.2 Consistency, Completeness, and Redundancy

Consider the following CFG G_6 which has one extra production rule compared to G_5:

Grammar G_6**:**
$S \rightarrow aSbS \mid bSaS \mid SS \mid \varepsilon.$

As with grammar G_5, all strings generated by G_6 also have an equal number of `a`'s and `b`'s. If we identify this property as *consistency*, then we find that grammars G_5 and G_6 satisfy consistency. What about completeness? In other words, will *all* such strings be derived? Does it appear that the production $S \rightarrow SS$ is essential to achieve completeness? It turns out that it is not - we can prove that G_5 is complete, thus showing that the production $S \rightarrow SS$ of G_6 is *redundant*.

How do we, in general, prove grammars to be complete? The general problem is undecidable,[1] as we shall show in Chapter 17. However, for *particular* grammars and *particular* completeness criteria, we can establish completeness, as we demonstrate below.

Proof of completeness:

The proof of completeness typically proceeds by induction. We have to decide between arithmetic or complete induction; in this case, it turns out that complete induction works better. Using complete induction, we write the *inductive hypothesis* as follows:

[1] The undecidability theorem that we shall later show is that for an *arbitrary* grammar G, it is not possible to establish whether $L(G)$ is equal to Σ^*.

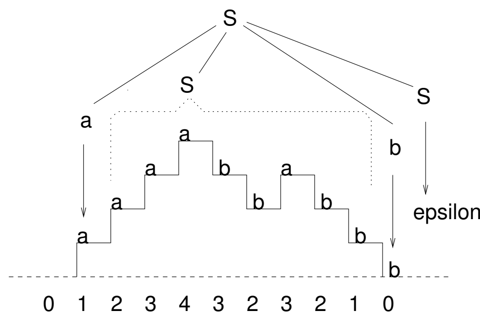

Fig. 13.2. A string that does not cause zero-crossings. The numbers below the string indicate the running difference between the number of a's and the number of b's at any point along the string

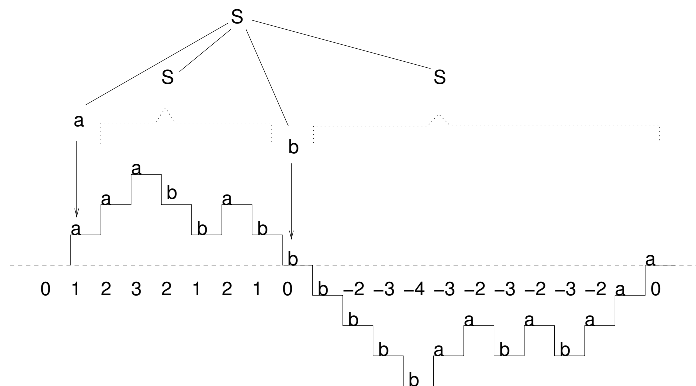

Fig. 13.3. A string that causes zero-crossings

> Suppose G_5 generates all strings less than n in length having an equal number of `a`'s and `b`'s.

Consider now a string of length $n+2$ – the next longer string that has an equal number of `a`'s and `b`'s. We can now draw a graph showing the running difference between the number of `a`'s and the number of `b`'s, as in Figure 13.2 and Figure 13.3. This plot of the running difference between `#a` and `#b` is either fully above the x-axis, fully below, or has zero-crossings. In other words, it can have many "hills" and "valleys." Let us perform a case analysis:

1. The graph has no zero-crossings. There are further cases:
   a) it begins with an `a` and ends with a `b`, as in Figure 13.2.

   b) it begins with a `b` and ends with an `a` (this case is symmetric and hence will not be explicitly argued).
2. It has zero-crossings, as in Figure 13.3. Again, we consider only one case, namely the one where the first zero-crossing from the left occurs after the curve has grown in the positive direction (i.e., after more `a`'s occur initially than `b`'s).

Let us consider case 1a. By induction hypothesis, the shorter string in the middle can be generated via `S`. Now, the entire string can be generated as shown in Figure 13.2 using production `S -> aSbS`, with `a` matching the first `a`, the first `S` matching 'the shorter string in the middle,' the `b` matching the last `b` in the string, and the second `S` going to ε. Case 1b may be similarly argued. If there is a zero-crossing, then we attack the induction as illustrated in Figure 13.3, where we split the string into the portion before its last zero-crossing and the portion after its last zero-crossing. These two portions can, by induction, be generated from G_5, with the first portion generated as `aSb` and the second portion generated as an `S`, as in Figure 13.3. $\square$.

Illustration 13.2.1 *Consider*

$$L_{a^m b^n c^k} = \{a^m b^n c^k \ \mid \ m, n, k \geq 0 \text{ and } ((m = n) \text{ or } (n = k))\}$$

Develop a context-free grammar for this language. Prove the grammar for consistency and completeness.
Solution: The grammar is given below. We achieve "equal number of `a`'s and `b`'s" by growing "inside out," as captured by the rule `M -> a M b`. We achieve zero or more `c`'s by the rule `C -> c C` or `e`. Most CFGs get designed through the use of such "idioms."

```
S -> M C | A N

M -> a M b | e

N -> b N c | e

C -> c C | e

A -> a A | e
```

Consistency: No string generated by `S` must violate the rules of being in language $L_{a^m b^n c^k}$. Therefore, if `M` generates matched a's and b's, and `C` generates only c's, consistency is guaranteed. The other case of `A` and `N` is very similar.

Notice that from the production of `M`, we can see that it generates matched a's and b's in the `e` case. Assume by induction hypothesis

that in the `M` occurring on the right-hand side of the rule, `M -> a M b`, respects consistency. Then the `M` of the left-hand side of this rule has an extra `a` in front and an extra `b` in the back. Hence, it too respects consistency.

Completeness: We need to show that any string of the form $a^n b^n c^k$ or $a^k b^n c^n$ can be generated by this grammar. We will consider all strings of the kind $a^n b^n c^k$ and develop a proof for them. The proof for the case of $a^k b^n c^n$ is quite similar and hence is not presented.

We resort to arithmetic induction for this problem. Assume, by induction hypothesis that the particular $2n + k$-long string $a^n b^n c^k$ was derived as follows:

- S $\Rightarrow$ M C.
- M $\Rightarrow^*$ $a^n b^n$ through a derivation sequence that we call S1, and
- C $\Rightarrow^*$ c^k through derivation sequence S2.
- S $\Rightarrow$ M C $\Rightarrow^*$ $a^n b^n$ C $\Rightarrow^*$ $a^n b^n$ c^k. Notice that in this derivation sequence, the first $\Rightarrow^*$ derivation sequence is what we call S1 and the second $\Rightarrow^*$ derivation sequence is what we call S2.

Now, consider the next legal longer string. It can be either $a^{n+1} b^{n+1} c^k$ or $a^n b^n c^{k+1}$. Consider the goal of deriving $a^{n+1} b^{n+1} c^k$. This can be achieved as follows:

- S $\Rightarrow$ M C $\Rightarrow$ a M b C.
- Now, invoking the S1 sequence, we get $\Rightarrow^*$ a $a^n b^n$ b C.
- Now, invoking the S2 sequence, we get $\Rightarrow^*$ a $a^n b^n$ b c^k; and hence, we can derive $a^{n+1} b^{n+1} c^k$.

Now, $a^n b^n c^{k+1}$ can be derived as follows:

- S $\Rightarrow$ M C $\Rightarrow$ M c C.
- Now, invoking the S1 derivation sequence, we get $\Rightarrow^*$ $a^n b^n$ c C.
- Finally, invoking the S2 derivation sequence, we get $\Rightarrow^*$ $a^n b^n$ c^{k+1}.

Hence, we can derive any string that is longer than $a^n b^n c^k$, and so by induction we can derive all legal strings.

13.2.1 More consistency proofs

In case of grammars G_5 and G_6, writing a consistency proof was rather easy: we simply observed that all productions introduce equal a's and b's in each derivation step. Sometimes, such "obvious" proofs are not possible. Consider the following grammar G_9:

```
S -> ( W S | e
W -> ( W W | ).
```

It turns out that grammar G_9 generates the language of the set of *all* well-parenthesized strings, even though three of the four productions appear to introduce *unbalanced* parentheses. Let us first formally define what it means to be well-parenthesized (see also Exercise 12.10), and then show that G_9 satisfies this criterion.

Well-parenthesized strings

A string x is well-parenthesized if:

1. The number of (and) in x is the same.
2. In any prefix of x, the number of (is greater than or equal to the number of).

With this definition in place, we now show that G_9 generates only consistent strings. We provide a proof outline:

1. Conjecture about S: *same number of* `(` *and* `)`. Let us establish this via induction.
   a) Epsilon (`e`) satisfies this.
   b) How about `( W S` ?
   c) OK, we need to "conquer" `W`.
      i. Conjecture about `W`: *it generates strings that have one more* `)` *than* `(`.
      ii. This is true for both arms of `W`.
      iii. Hence, the conjecture about `W` is true.
   d) Hence, the conjecture about S is true.
   e) Need to verify one more step: *In any prefix, is the number of* `(` *more than the number of* `)`*?*
   f) Need conjecture about `W`: *In any prefix of a string generated by* `W`*, number of* `)` *at most one more than the number of* `(`. Induct on the `W` production and prove it. Then `S` indeed satisfies consistency. □.

13.2.2 Fixed-points again!

The language generated by a CFG was explained through the notion of a derivation sequence. Can this language also be obtained through fixed-points? The answer is 'yes' as we now show.

Consider recasting the grammar G_5 as a language equation (call the language $L(S_5)$):

$$L(S_5) = \{a\}\ L(S_5)\ \{b\}\ L(S_5) \ \cup\ \{b\}\ L(S_5)\ \{a\}\ L(S_5) \ \cup\ \{\varepsilon\}.$$

Here, juxtaposition denotes language concatenation. What solutions to this language equation exist? In other words, find languages to plug in place of $L(S_5)$ such that the right-hand side language becomes equal to the left-hand side language. We can solve such language equations using the fixed-point theory introduced in Chapter 7. In particular, one can obtain the least fixed-point by iterating from "bottom" which, in our context, is the empty language $\emptyset$. *The least fixed-point is also the language computed by taking the derivation sequence perspective.*

To illustrate this connection between fixed-points and derivation sequences more vividly, consider a language equation obtained from CFG G_6:

$$L(S_6) = \{a\}L(S_6)\{b\}L(S_6) \cup \{b\}L(S_6)\{a\}L(S_6) \cup \{b\}L(S_6)L(S_6) \cup \{\varepsilon\}$$

It is easy to verify that this equation admits two solutions, one of which is the desired language of equal a's and b's, and the other language is a completely uninteresting one (see Exercise 13.5). The language equation for $L(S_5)$ does not have multiple solutions. It is interesting to note that the reason why $L(S_6)$ admits multiple solutions is because of the redundant rule in it. To sum up:

- Language equations can have multiple fixed-points.
- The least fixed-point of a language equation obtained from a CFG also corresponds to the language generated by the derivation method.

These observations help sharpen our intuitions about least fixed-points. The notion of least fixed-points is central to programming because programs also execute through "derivation steps" similar to those employed by CFGs. Least fixed-points start with "nothing" and help attain the least (most irredundant) solution. As discussed in [20] and also in Chapter 22, in Computational Tree Logic (CTL), both the least fixed-point and the greatest fixed-point play significant and meaningful roles.

13.3 Ambiguous *Grammars*

As we saw in Figures 13.1(b) and 13.1(c), even for one string such as 1110, it is possible to have two distinct parse trees. In a compiler, the existence of two distinct parse trees can lead to the compiler producing two *different* codes - or can ascribe two different meanings - to the same sentence. This is highly undesirable in most settings. For example, if we

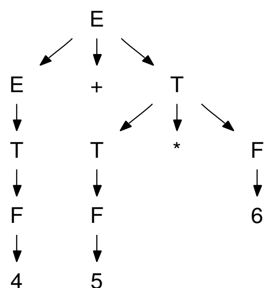

Fig. 13.4. Parse tree for $4 + 5 * 6$ with respect to G_8.

have an arithmetic expression $4+5*6$, we want it parsed as $4+(5*6)$ and not as $(4+5)*6$. If we write the expression grammar as G_7,

Grammar G_7:
$E \rightarrow E + E \ | \ E * E \ | \ number$,

then *both* these parses (and their corresponding parse trees) would be possible, in effect providing an *ambiguous* interpretation to expressions such as $4 + 5 * 6$. It is necessary to *disambiguate* the grammar, for example by rewriting the simple expression grammar above to grammar G_8. One such disambiguated grammar is the following:

Grammar G_8:
$E \rightarrow E + T \ | \ T$
$T \rightarrow T * F \ | \ F$
$F \rightarrow number \ | \ (E)$.

In the rewritten grammar, for any expression containing $+$ and $*$, the parse trees will situate $*$ deeper in the tree (closer to the frontier) than $+$, thus, in effect, forcing the evaluation of $*$ first, as illustrated in Figure 13.4.

13.3.1 If-then-else ambiguity

An important practical example of ambiguity arises in the context of grammars pertaining to `if` statements, as illustrated below:

```
STMT ->   if EXPR then STMT
        | if EXPR then STMT else STMT
        | OTHER

OTHER -> p
```

```
EXPR  -> q
```

The reason for ambiguity is that the `else` clause can match either of the *then* clauses. Compiler writers avoid the above if-then-else ambiguity by modifying the above grammar in such a way that the `else` matches with the closest unmatched `then`. One example of such a rewritten grammar is the following:

```
STMT ->  MATCHED | UNMATCHED

MATCHED -> if EXPR then MATCHED else MATCHED | OTHER

UNMATCHED ->  if EXPR then STMT
            | if EXPR then MATCHED else UNMATCHED

OTHER -> p

EXPR  -> q
```

This forces the `else` to go with the closest previous unmatched `then`.

13.3.2 Ambiguity, inherent ambiguity

In general, it is impossible to algorithmically decide whether a given CFG is ambiguous (see Chapter 17 and Exercise 17.2 that comes with hints). In practice, this means that there cannot exist an algorithm that can determine whether a given CFG is ambiguous. To make things worse, there are *inherently ambiguous languages* – languages for which every CFG is ambiguous. If the language that one is dealing with is inherently ambiguous, it is *not* possible to eliminate ambiguity in all cases, such as we did by rewriting grammar G_7 to grammar G_8.

> Notice that the terminology is not *inherently ambiguous grammar* but *inherently ambiguous language* - what we are saying is *every grammar* is ambiguous for *certain CFLs.*

An example of an inherently ambiguous language is

$$\{0^i 1^j 2^k \mid i, j, k \geq 0 \wedge i = j \vee j = k\}.$$

The intuition is that every grammar for this language must have productions geared towards matching 0s against 1s and 1s against 2s. In this case, given a string of the form $0^k 1^k 2^k$, *either* of these options can be exercised. A formal proof may be found in advanced papers in this area, such as [82].

13.4 A Taxonomy of Formal Languages and Machines

Machines	Languages	Nature of grammar
DFA/NFA	Regular	Left-linear or Right-linear productions
DPDA	Deterministic CFL	Each LHS has one non-terminal The productions are deterministic
NPDA (or "PDA")	CFL	Each LHS has only one non-terminal
LBA	Context Sensitive Languages	LHS may have length > 1, but $\mid$ LHS$\mid \leq \mid$RHS$\mid$, ignoring ε productions
DTM/NDTM	Recursively Enumerable	General grammars ($\mid$LHS$\mid \geq \mid$RHS$\mid$ allowed)

Fig. 13.5. The Chomsky hierarchy and allied notions

We now summarize the formal machines, as well as languages, studied in this book in the table given in Figure 13.5. This is known as the *Chomsky hierarchy* of formal languages. For each machine, we describe the nature of its languages, and indicate the nature of the grammar used to describe the languages. It is interesting that simply by varying the nature of production rules, we can obtain *all* members of the Chomsky hierarchy. This single table, in a sense, summarizes some of the main achievements of over 50 years of research in computability, machines, automata, and grammars. Here is a summary of the salient points made in this table, listed against each of the language classes:[2]

Regular languages:

DFAs and NFAs serve as machines that recognize regular languages. Context-free grammars written with only left-linear or only right-linear productions can generate or recognize regular languages. The linearity

[2] We prefer to highlight the language classes as they constitute the more abstract concept, while machines and grammars are two different syntactic devices that denote languages.

of the production rules means that there is only *one* non-terminal allowed on the RHS of each production, which appears leftmost or rightmost. Hence, these non-terminals can be regarded as states of an NFA, as discussed in Section 13.6.

Deterministic context-free languages (DCFL):

Push-down automata are machines that recognize DCFLs, as illustrated in Illustration 13.4.2. In effect, they can parse sentences in the language without backtracking (deterministically). As for grammars, the fact that each context-free production specifies the expansion of *one* and only one non-terminal on the left-hand side means that this expansion is good *wherever the non-terminal appears—i.e.*, regardless of the context (hence "context-free"). The grammars are *deterministic*, as illustrated in Illustration 13.4.2.

Context-free languages (CFL):

These are more general than DCFLs, as the constraint of determinism is removed in the underlying machines and grammars.

Context-sensitive languages (CSL):

CSLs can be recognized by *linear bounded automata* which are described in Section 15.2.3. Basically, they are restricted Turing machines which can write only on that portion of the input tape on which the input was originally laid out. In particular, given any LBA M and a string w, it can be conclusively answered as to whether M accepts w or not. This is impossible with a Turing machine.

As for grammars, CSLs are recognized by productions in which the length of a left-hand side is allowed to be more than 1. Such a *context-sensitive* production specifies a *pattern* on the LHS, and a sentential form on the RHS. In a sense, we can have a rule of the form `a A d -> a a c d` and another of the form `a A e -> a c a d`. Notice that `A`'s expansion when surrounded by `a` and `d` can be different from when surrounded by `a` and `e`, thus building in context sensitivity to the interpretation of `A`. The length of the RHS is required to be no less than that of the LHS (except in the ε case) to ensure decidability in some cases.

Recursively enumerable or Turing recognizable (RE or TR) languages:

These form the most general language class in the Chomsky hierarchy. Notice that Turing machines as well as unrestricted productions, form the machines and grammars for this language class.

CFGs and CFLs are fundamental to computer science because they help describe the structure underlying programming languages. The basic "signature" of a CFL is "nested brackets:" for example, nesting occurs in expressions and in very many statically scoped structures in computer programs. In contrast, the basic signature of regular languages is "iteration (looping) according to some ultimate periodicity."

Illustration 13.4.1 Let us argue that the programming language C is not regular. Let there be a DFA for C with n states. Now consider the C program

$$C_{NOP} = \{main()\{^n\}^n \mid n \geq 0\}.$$

Clearly, the DFA for C will loop in the part described by $main()\{^n$, and by pumping this region wherever the loop might fall, we will obtain a malformed C program. Some of the pumps could, for instance, result in the C program $maiaiain()\{\ldots$, while some others result in strings of the form $main\{\{\{\}\}$, etc.
Using a CFG, we can describe C_{NOP} using production rules, as follows:

```
L_C_nop -> main Paren Braces
Paren -> ()
Braces -> epsilon | { Braces }.
```

13.4.1 Non-closure of CFLs under complementation

It may come as a surprise that most programming languages are *not* context-free! For instance, in `C`, we can declare function *prototypes* that can introduce an arbitrary number of arguments. Later, when the function is defined, the same arguments must appear in the same order. The structure in such "define/use" structures can be captured by the language

$$L_{ww} = \{ww \mid w \in \{0,1\}^*\}.$$

As we shall sketch in Section 13.8 (Illustration 13.8.1), this language is *not* context-free. It is a context-sensitive language which can be accepted by a *linear bounded automaton* (LBA). Basically, an LBA has a tape, and can sweep across the tape as many times as it wants, writing "marking bits" to compare across arbitrary reaches of the region of the tape where the original input was presented. This mechanism can easily spot a w and a later w appearing on the tape. The use of *symbol tables* in compilers essentially gives it the power of LBAs, making compilers able to handle C prototype definitions.

While L_{ww} is not context-free, its complement, $\overline{L_{ww}}$, is indeed a CFL. This means that CFLs are *not* closed under complementation!

$\overline{L_{ww}}$ is generated by the following grammar $G_{\overline{ww}}$:

Grammar $G_{\overline{ww}}$:

```
S -> AB | BA | A | B
A -> CAC | 0
B -> CBC | 1
C -> 0 | 1.
```

Illustration 13.4.2 For each language below, write

- R if the language is regular,
- $DCFL$ if the language is deterministic context-free (can be recognized by a DPDA),
- CFL if it can be recognized by a PDA **but not** a DPDA,
- IA if the language is CFL but is inherently ambiguous, and
- N if not a CFL.

Also provide a one-sentence justification for your answer. *Note:* In some cases, the language is described using the set construction, while in other cases, the language is described via a grammar ("L(G)").

1. $\{x \mid x \text{ is a prefix of } w \text{ for } w \in \{0,1\}^*\}$.
   Solution: This is R, because the language is nothing but $\{0,1\}^*$.
2. L(G) where G is the CFG $S \rightarrow 0\,S\,0 \mid 1\,S\,1 \mid \varepsilon$.
   Solution: CFL, because nondeterminism *is required* in order to guess the midpoint.
3. $\{a^n b^m c^n d^m \mid m, n \geq 0\}$.
   Solution: The classification is N, because comparison using a single stack is not possible. If we push a^n followed by b^m, it is no longer possible to compare c^n against a^n, as the top of the stack contains b^m. Removing b^m "temporarily" and restoring it later isn't possible, as it is impossible to store away b^m in the *finite-state* control.
4. $\{a^n b^n \mid n \geq 0\}$.
   Solution: $DCFL$, since we can deterministically switch to matching b's.
5. $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } j = k\}$.
   Solution: IA, because for $i = j = k$, we can have two distinct parses, one comparing a's and b's, and the other comparing b's and c's (the capability for these two comparisons must exist in any grammar, because of the "or" condition).

Illustration 13.4.3 Indicate which of the following statements pertaining to closure properties is true and which is false. For every true assertion below, provide a one-sentence supportive answer. For every false assertion below, provide a counterexample.

1. The union of a CFL and a regular language is a CFL.
   Solution: True, since regular languages are also CFLs. Write the top-level production of the new CFL as `S -> A | B` where `A` generates the given CFL and `B` generates the given regular language.
2. The intersection of any two CFLs is always a CFL.
   Solution: False. Consider $\{a^m b^m c^n \mid m, n \geq 0\} \cap \{a^m b^n c^n \mid m, n \geq 0\}$. This is $\{a^n b^n c^n \mid n \geq 0\}$, which is not a CFL.
3. The complement of a CFL is always a CFL.
   Solution: False. Consider $L_{ww} = \{ww \mid w \in \{0,1\}^*\}$ which was discussed on page 230. Try to come up with another example yourself.

Illustration 13.4.4 Describe the CFL generated by the following grammar using a regular expression. Show how you handled each of the non-terminals.
$S \rightarrow TT$
$T \rightarrow UT \mid U$
$U \rightarrow 0U \mid 1U \mid \varepsilon.$

It is easy to see that U generates a language represented by regular expression $(0+1)^*$, while T generates U^+. Note that for any regular expression R, it is the case that $(R^*)^+$ is $R^* \cup R^*R^* \cup R^*R^*R^* \ldots$ which is R^*. Therefore, T generates $(0+1)^*$. Now, S generates TT, or $(0+1)^*(0+1)^*$, which is the same as $(0+1)^*$. Therefore, $L(S) = \{0,1\}^*$.

13.4.2 Simplifying CFGs

We illustrate a technique to simplify grammars through an example.

Illustration 13.4.5 Simplify the following grammar, explaining why each production or non-terminal was eliminated:

$S \rightarrow A\,B \mid D$
$A \rightarrow 0\,A \mid 1\,B \mid C$
$B \rightarrow 2 \mid 3 \mid A$
$D \rightarrow A\,C \mid B\,D$

$E \rightarrow 0.$

Solution: Grammars are simplified as follows. First, we determine which non-terminals are *generating* - have a derivation sequence to a terminal string (if a non-terminal is non-generating, the language denoted by it is $\emptyset$, and we can safely eliminate all such non-terminals, as well as, recursively, all other non-terminals that use them). We can observe that in our example, B is generating. Therefore, A is generating. C is an undefined non-terminal, and so we can eliminate it. Now, we observe that S is generating, since AB is generating; so we had better retain S (!). D is reachable ('reachable' means that it appears in at least one derivation sequence starting at `S`) but non-generating, so we can eliminate D. Finally, E is *not* reachable from S through any derivation path, and hence we can eliminate it, all productions using it (none in our example), and all productions expanding E (exactly one in our example). Therefore, we obtain the following simplified CFG:

$S \rightarrow A\,B$
$A \rightarrow 0\,A \ \mid 1\,B$
$B \rightarrow 2 \ \mid 3 \ \mid A.$

Here, then, are sound steps one may employ to simplify a given CFG (it is assumed that the productions are represented in terms of *elementary productions* in which the disjunctions separated by | on the RHS of a production rule are expressed in terms of separate productions for the same LHS):

- A non-generating non-terminal is useless, and it can be eliminated.
- A non-terminal for which there is no rule defined (does not appear on the left-hand side of any rule) is non-generating in a trivial sense.
- The property of being non-generating is 'infectious' in the following sense: if non-terminal N_1 is non-generating, and if N_1 appears in every derivation sequence of another non-terminal N_2, then N_2 is also non-generating.
- A non-terminal that does not appear in any derivation sequence starting from S is unreachable.
- Any CFG production rule that contains either a non-generating or an unreachable non-terminal can be eliminated.

Illustration 13.4.6 Simplify the following grammar, clearly showing how each simplification was achieved (name criteria such as 'generating' and 'reaching'):

```
S -> A B | C D
A -> 0 A | 1 B
B -> 2   | 3
D -> A C | B D E
E -> 4 E | D | 5.
```

B is generating. Hence, A is generating. S is generating. B, A, and S are reachable. Hence, S, A, and B are essential to preserve, and therefore C and D are reachable; however, C is not generating. Hence, production CD is useless. Hence, we are left with:

```
S -> A B
A -> 0 A | 1 B
B -> 2 | 3.
```

□

We now examine *push-down automata* which are machines that recognize CFLs, and bring out some connections between PDAs and CFLs.

13.5 Push-down Automata

A push-down automaton (PDA) is a structure $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ where Q is a finite set of states, Σ is the input alphabet, Γ is the stack alphabet (that usually includes the input alphabet Σ), q_0 is the initial state, $F \subseteq Q$ is the set of accepting states, z_0 the initial stack symbol, and

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}.$$

In each move, a PDA can *optionally* read an input symbol. However, in each move, it *must* read the top of the stack (later, we will see that this assumption comes in handy when we have to convert a PDA to a CFG). Since we will always talk about an NPDA by default, the δ function returns a set of nondeterministic options. Each option is a next-state to go to, and a stack string to push on the stack, with the first symbol of the string appearing on top of the stack after the push is over. For example, if $\langle q_1, ba\rangle \in \delta(q_0, x, a)$, the move can occur when x can be read from the input and the stack top is a. In this case, the PDA moves over x (it cannot read x again). Also, an a is removed from the stack. However, as a result of the move, an a is promptly pushed back on the stack, and is *followed by* a push of b, with the machine going to state q_1. The transition function δ of a PDA may be either *deterministic* or nondeterministic.

13.5.1 DPDA versus NPDA

A push-down automaton can be deterministic or nondeterministic. DPDA and NPDA are *not equivalent* in power; the latter are strictly more powerful than the former. Also, notice that unlike with a DFA, a deterministic PDA can move on ε. Therefore, the exact specification of what *deterministic* means becomes complicated. We summarize the definition from [60]. A PDA is deterministic if and only if the following conditions are met:

1. $\delta(q, a, X)$ has at most one member for any q in Q, a in Σ_ε, and X in Γ.
2. If $\delta(q, a, X)$ is non-empty, for some a in Σ, then $\delta(q, \varepsilon, X)$ must be empty.

In this book, I will refrain from giving a technically precise definition of DPDAs. It really becomes far more involved than we wish to emphasize in this chapter, at least. For instance, with a DPDA, it becomes necessary to know when the string ends, thus requiring a *right end-marker* $\dashv$. For details, please see [71, page 176].

13.5.2 Deterministic context-free languages (DCFL)

Current State	Input	Stack top	String pushed	New State	Comments
q0	0	z0	0 z0	q1	0. Have to push on this one
q0	1	z0	1 z0	q1	...or this one
q1	0	0	0 0	q1	1a.Assume not at midpoint
q1	0	1	0 1	q1	Have to push on this one
q1	0	0	ε	q1	1b. Assume at midpoint
q1	1	1	1 1	q1	2a. Assume not at midpoint
q1	1	0	1 0	q1	Have to push on this one
q1	1	1	ε	q1	2b. Assume at midpoint
q1	ε	z0	z0	q2	3. Matched around midpoint

Fig. 13.6. A PDA for the language $L_0 = \{ww^R \mid w \in \{0,1\}^*\}$

A deterministic context-free language (DCFL) is one for which there is a DPDA that accepts the same language. Consider the language

Language $L_0 = \{ww^R \mid w \in \{0,1\}^*\}$.

L_0 is not a DCFL because in any PDA, the use of nondeterminism is essential to "guess" the midpoint. Figure 13.6 presents the δ function of a PDA designed to recognize L_0. This PDA is described by the structure

$$P_{L_0} = (\{q_0, \}, \{0,1\}, \{0,1,z_0\}, \delta, q_0, z_0, \{q_0, q_2\}).$$

This PDA begins by stacking 0 or 1, depending on what comes first. The comments `1a` and `1b` describe the nondeterministic selection of assuming *not* being at a midpoint, and being a midpoint, respectively. A similar logic is followed in `2a` and `2b` as well. Chapter 14 describes PDA construction in greater detail.
Let us further our intuitions about PDAs by considering a few languages:

L_1: $\{a^i b^j c^k \mid \textit{if } i = 1 \textit{ then } j = k\}$.

L_2: $\{a^i b^j c^k d^m \mid i,j,k,m \geq 0 \wedge \textit{if } i = 1 \textit{ then } j = k \textit{ else } k = m\}$

L_3: $\{ww \mid w \in \{0,1\}^*\}$.

L_4: $\{0,1\}^* \setminus \{ww \mid w \in \{0,1\}^*\}$.

L_5: $\{a^i b^j c^k \mid i = j \textit{ or } i = k\}$.

L_6: $\{a^n b^n c^n \mid n \geq 0\}$.

L_1 is a DCFL, because after seeing whether $i = 1$ or not, a deterministic algorithm can be employed to process the rest of the input. A DPDA can be designed for $reverse(L_1)$ also. Likewise, a DPDA can be designed for L_2. However, as discussed in Section 8.1.4, $reverse(L_2)$ is not a DCFL, as it is impossible to keep *both* decisions – whether $j = k$ or $k = m$ – ready by the time i is encountered. L_3 is not a CFL at all. However, L_4, the complement of L_3, is a CFL. L_5 is a CFL (but not a DCFL) – the guesses of $i = j$ or $i = k$ can be made nondeterministically. Finally, L_6 is not a CFL, as we cannot keep track of the length of three distinct strings using one stack. □

13.5.3 Some Factoids

Here are a few more factoids that tie together ambiguity (of grammars) and determinism (of PDA):

- If one can obtain a DPDA for a language, then that language is not inherently ambiguous. This is because for an inherently ambiguous language, *every* CFG admits two parses, thus meaning that there cannot be a *D*PDA for it.
- There are CFLs that are not DCFLs (have no DPDA), and yet they have non-ambiguous grammars. The grammar

  ```
  S -> 0 S 0 | 1 S 1 | e
  ```

  is non-ambiguous, and yet denotes a language that is not a DCFL. In other words, this CFG generates all the strings of the form $ww,^R$ and these strings have only one parse tree. However, since the midpoint of such strings isn't obvious during a left-to-right scan, a nondeterministic PDA is necessary to parse such strings.

13.6 Right- and Left-Linear CFGs

A right-linear CFG is one where every production rule has exactly one non-terminal and that it also appears rightmost. For example, the following grammar is right-linear:

```
S -> 0 A | 1 B | e
A -> 1 C | 0
B -> 0 C | 1
C -> 1 | 0 C.
```

Recall that `S -> 0 A | 1 B | e` is actually *three* different production rules `S -> 0 A`, `S -> 1 B`, and `S -> e`, where each rule is right-linear. This grammar can easily be represented by the following NFA obtained almost directly from the grammar:

```
IS - 0 -> A
IS - 1 -> B
IS - e -> F1
A  - 1 -> C
A  - 0 -> F2
B  - 0 -> C
B  - 1 -> F3
C  - 0 -> C
C  - 1 -> F4.
```

A left-linear grammar is defined similar to a right-linear one. An example is as follows:

```
S -> A 0 | B 1 | e
A -> C 1 | 0
B -> C 1 | 1
C -> 1 | C 0.
```

A purely left-linear or a purely right-linear CFG denotes a regular language. However, the converse is not true; that is, if a language is regular, it does not mean that it has to be generated by a purely left-linear or purely right-linear CFG. Even non-linear CFGs are perfectly capable of sometimes generating regular sets, as in

```
S -> T T | e
T -> 0 T | 0.
```

It also must be borne in mind that we cannot "mix up" left- and right-linear productions and expect to obtain a regular language. Consider the productions

```
S -> 0 T  | e
T -> S 1.
```

In this grammar, the productions are linear - left or right. However, since we use *left- and right-linear rules*, the net effect is as if we defined the grammar

```
S -> 0 S 1  | e
```

which generates the non-regular context-free language

$$\{0^n 1^n \mid n \geq 0\}.$$

Conversion of purely left-linear grammars to NFA

Converting a left-linear grammar to an NFA is less straightforward. We first *reverse* the language it represents by reversing the grammar. Grammar reversal is approached as follows: given a production rule

$$S \to R_1 R_2 \ldots R_n,$$

we obtain a production rule for the reverse of the language represented by S by reversing the production rule to:

$$S^r \to R_n^r R_{n-1}^r \ldots R_1^r.$$

Applying this to the grammar at hand, we obtain

```
Sr -> 0 Ar | 1 Br | e
Ar -> 1 Cr | 0
Br -> 1 Cr | 1
Cr -> 1 | 0 Cr.
```

Once an NFA for this right-linear grammar is built, it can be reversed to obtain the desired NFA.

13.7 Developing CFGs

Developing CFGs is much like programming; there are no hard-and-fast rules. Here are reasonably general rules of the thumb for arriving at CFGs:

1. *(Use common idioms):* Study and remember many common patterns of CFGs and use what seems to fit in a given context. Example: To get the effect of matched brackets, the common idiom is

```
S -> ( S ) | e.
```

2. *Break the problem into simpler problems:*
   Example: $\{a^m b^n \mid m \neq n,\ m, n \geq 0\}$.

   a) So, a's and b's must still come in order.
   b) Their numbers shouldn't match up.
      i. Formulate matched up a's and b's

```
M -> e | a M b
```

      ii. Break the match by adding either more A's or more B's

```
S -> A M | M B
A -> a | a A
B -> b | b B
```

13.8 A Pumping Lemma for CFLs

Consider any CFG $G = (N, \Sigma, P, S)$. A Pumping Lemma for the language of this grammar, $L(G)$, can be derived by noting that a "very long string" $w \in L(G)$ requires a very long derivation sequence to derive it from S. Since we only have a finite number of non-terminals, some non-terminal must repeat in this derivation sequence, and furthermore, the *second* occurrence of the non-terminal must be a result of expanding the first occurrence (it must lie within the parse tree generated by the first occurrence).

For example, consider the CFG

```
S -> ( S ) | T | e

T -> [ T ] | T T | e.
```

Here is an example derivation:

```
S => ( S ) => (( T )) => (( [ T ] )) => (( [  ] ))
                 ^            ^

              Occurrence-1   Occurrence-2

Occurrence-1 involves Derivation-1: T => [ T ] => [ ]
Occurrence-2 involves Derivation-2: T => e
```

Here, the second T arises because we took T and expanded it into [T] and then to [ ]. Now, the basic idea is that we can use Derivation-1 used in the first occurrence in place of Derivation-2, to obtain a longer string:

```
S => (S) => ((T))  => (( [ T ] )) => (( [[ T ]] )) => (( [[ ]] ))
               ^            ^

             Occurrence-1 Use Derivation-1 here
```

In the same fashion, we can use Derivation-2 in place of Derivation-1 to obtain a shorter string, as well:

```
S => ( S ) => ( ( T ) ) => ( ( ) )
                  ^

                Use Derivation-2 here
```

When all this happens, we can find a repeating non-terminal that can be pumped up or down. In our present example, it is clear that we can manifest $(([^i\ ]^i))$ for $i \geq 0$ by either applying Derivation-2 directly, or by applying some number of Derivation-1s followed by Derivation-2. In order to conveniently capture the conditions mentioned so far, it is good to resort to parse trees. Consider a CFG with $|V|$ non-terminals, and with the right-hand side of each rule containing at most b syntactic elements (terminals or non-terminals). Consider a b-ary tree built up to height $|V|+1$, as shown in Figure 13.7. The string yielded on the frontier of the tree $w = uvxyz$. If there are two such parse trees for w, pick the one that has the fewest number of nodes. Now, if we avoid having the same non-terminal used in any path from the root to a leaf, basically each path will "enjoy" a growth up to height at most $|V|$ (recall that the leaves are terminals). The string $w = uvxyz$ is, in this case, of length at most $b^{|V|}$. This implies that *if we force the string to be of length* $b^{|V|+1}$ (called p hereafter), a parse tree for this string will have some path that repeats a non-terminal. Call the higher occurrence V_1 and the lower occurrence (contained within V_1) V_2. Pick the lowest two such repeating pair of non-terminals. Now, we have these facts:

- $|vxy| \leq p$; if not, we would find two other non-terminals that exist lower in the parse tree than V_1 and V_2, thus violating the fact that V_1 and V_2 are the lowest two such.

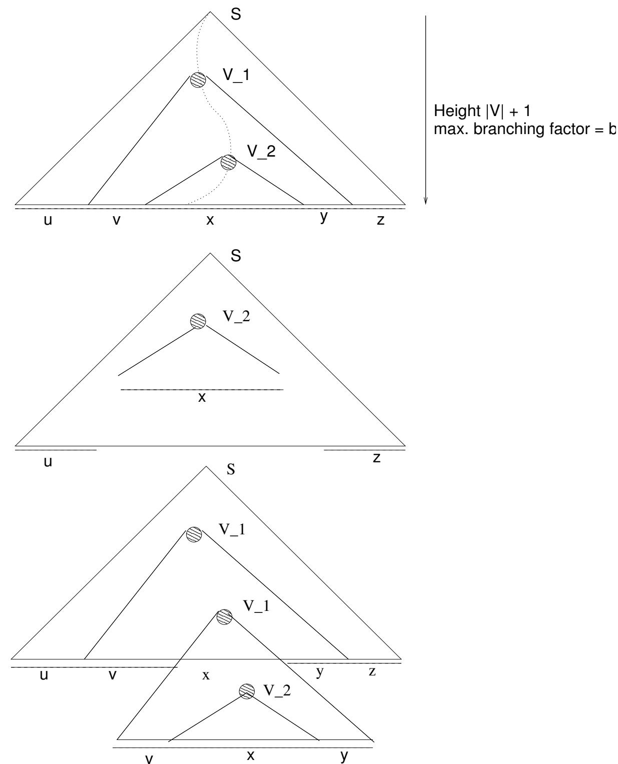

Fig. 13.7. Depiction of a parse tree for the CFL Pumping Lemma. The upper drawing shows a very long path that repeats a non-terminal, with the lowest two repetitions occurring at V_2 and V_1 (similar to `Occurrence-1` and `Occurrence-2` as in the text). With respect to this drawing: (i) the middle drawing indicates what happens if the derivation for V_2 is applied in lieu of that of V_1, and (ii) the bottom drawing depicts what happens if the derivation for V_2 is replaced by that for V_1, which, in turn, contains a derivation for V_2

- $|vx| \geq 1$; if not, we will in fact have $w = uxz$, for which a shorter parse tree exists (namely, the one where we directly employ V_2).
- Now, by pumping, we can obtain the desired repetitions of v and y, as described in Theorem 13.1.

Theorem 13.1. *Given any CFG $G = (N, \Sigma, P, S)$, there exists a number p such that given a string w in $L(G)$ such that $|w| \geq p$, we can split w into $w = uvxyz$ such that $|vy| > 0$, $|vxy| \leq p$, and for every $i \geq 0$, $uv^i xy^i z \in L(G)$.*

We can apply this Pumping Lemma for CFGs in the same manner as we did for regular sets. For example, let us sketch that L_{ww} of page 230 is not context-free.

Illustration 13.8.1 Suppose L_{ww} were a CFL. Then the CFL Pumping Lemma would apply. Let p be the pumping length associated with a CFG of this language. Consider the string $0^p 1^p 0^p 1^p$ which is in L_{ww}. The segments v and y of the Pumping Lemma are contained within the first $0^p 1^p$ block, in the middle $1^p 0^p$ block or in the last $0^p 1^p$ block, and in each of these cases, it could also have fallen entirely within a 0^p block or a 1^p block. By pumping up or down, we will then obtain a string that is not within L_{ww}. □

Exercise 13.13 demonstrates another "unusual" application of the CFG Pumping Lemma.

Chapter Summary

This chapter discussed the notion of context-free grammars and context-free languages. We emphasized 'getting a grammar right' by showing that it has two facets—namely *consistency* and *completeness*. Fixed-point theory helps appreciate context-free grammars in terms of recursive equations whose least fixed-point is the "desired" context-free language. We discussed ambiguity and disambiguation—two topics that compiler writers deeply care about. After discussing the Chomsky hierarchy, we discuss the topics of closure properties (or lack thereof under intersection and complementation). We present how CFGs may be simplified. We then move on to push-down automata, which are machines with a finite control and *one* stack. We discuss the fact that NPDAs and DPDAs are not equivalent. We close off with a discussion of an *incomplete* Pumping Lemma for CFLs. Curiously, there is also a complete Pumping Lemma for CFLs ("strong Pumping Lemma" [124]). We do not discuss this lemma (it occupies nearly one page even when stated in a formal mathematical notation).

Exercises

13.1. Draw the parse tree for string

$$a\ a\ a\ b\ b\ a\ b\ b\ b\ b\ b\ a\ a\ b\ a\ b\ a\ a\ a$$

with respect to grammar G_5, thus showing that this string can be derived according to the grammar.

13.2.

1. Parenthesize the following expression according to the rules of standard precedence for arithmetic operators, given that $\sim$ stands for unary minus:
$$\sim 1 * 2 - 3 - 4 / \sim 5.$$
2. Convert the above expression to Reverse Polish Notation (post-fix form).

13.3. Prove by induction that the following grammar generates only strings with an odd number of 1s. Clearly argue the basis case(s), the inductive case(s), and what you prove regarding T and regarding S.

$$S \rightarrow S\,T\,0 \;\mid\; 0\,1$$
$$T \rightarrow 1\,1\,T \;\mid\; \epsilon$$

13.4. Write the consistency proof pertaining to G_9 in full detail. Then write a proof for the completeness of the above grammar (that it generates *all* well-parenthesized strings).

13.5. Which other solution to the language equation of $L(S_6)$ of page 225 exists?

13.6. *Prove* that $G_{\overline{ww}}$ of Page 231 is a CFG for the language $\overline{L_{ww}}$. *Hint:* The productions `S -> A` and `S -> B` generate odd-length strings. Also, `S -> AB` and `S -> BA` generate all strings that are *not* of the form ww. This is achieved by generating an even-length string pq where $|p| = |q|$ and if p is put "on top of" q, there will be at least one spot where they both differ.

13.7. Argue that a DPDA satisfying the definition in Section 13.5.1 cannot be designed for the language $\{ww^R \mid w \in \Sigma^*\}$.

13.8. (Adapted from Sipser [111]) Determine whether the context-free language described by the following grammar is regular, showing all the reasoning steps:

```
S -> T T | U
T -> 0 T | T 0 | #
U -> 0 U 0 0 | #.
```

13.9. Answer whether true or false:

1. *There are more regular languages (RLs) than CFLs.*
2. *Every RL is also a CFL.*

3. *Every CFL is also a RL.*
4. *Every CFL has a regular sublanguage* ("sublanguage" means the same as "subset").
5. *Every RL has a CF sublanguage.*

13.10.

1. Obtain one CFG G_1 that employs left-linear and right-linear productions (that cannot be eliminated from G_1) such that $L(G_1)$ is regular.
2. Now obtain another grammar G_2 where $L(G_2)$ is non-regular but is a DCFL.
3. Finally, obtain a G_3 where $L(G_3)$ is not a DCFL.

13.11. Using the Pumping Lemma, show that the language $\{0^n 1^n 2^n \mid n \geq 0\}$ is not context-free.

13.12. Show using the Pumping lemma that the language $\{ww \mid w \in \{0,1\}^*\}$ is not context-free.

13.13. Prove that any CFG with $|\Sigma| = 1$ generates a regular set. *Hint:* use the Pumping Lemma for CFLs together with the ultimate periodicity result for regular sets. Carefully argue and conclude using the PL for CFLs that we are able to generate only periodic sets.

13.14. Argue that the *syntax* of regular expressions is *context-free* while the syntax of *context-free grammars* is *regular*!

Index

$<_{lex}$, 109
$=_{lex}$, 109
$>_{lex}$, 109
A_{TM}, 298
CP, the set of all legal C programs, 44
E_{TM}, 304
Eclosure function, 157
$Halt_{TM}$, 299
Ibs, infinite bit-sequences, 44
R^*, given R, 60
R^+, given R, 60
R^0, given R, 60
$Regular_{TM}$, 305
$S/\equiv$, 60
T_{ID_DFA}, 124
$\Leftarrow$, 76
$\Leftrightarrow$, 76
$\Rightarrow$, 75
Σ, 105
$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, 141
δ, for a DFA, 121
δ, for an NFA, 141
$\emptyset$, 107
$\hat{\delta}$, 124, 147
$\leq_{lex}$, 109
$|$, Unix pipe, 153
$\neq$-CNF, 340
$\overline{Clique}$, 349
$\perp$, 98
$\preceq$, 64
$\rightarrow$, 146
$\doteq$, 76
$\triangleq$, 76
ε, 106
ε-closure, 145
$\vdash$, 124
$\vdash^*$, 124
as notation, 77, 113
$q \rightarrow$, 146
q_0, 121
', quoting, 153
$\stackrel{\varepsilon}{\rightarrow}$, 146
$\stackrel{\varepsilon}{\rightarrow}^*$, 146
'*,' Kleene star, 112
'*,' Star, 112
'-', Read standard input, 153
`graphviz`, 129
2-CNF, 341
2-CNF satisfiability, 341
2-partition, 364
3-CNF, 340
3-SAT, 354
  NPC, 354
3-partition, 364
3x+1 problem, 28

Acceptance of strings, 124
Acceptance problem, 299
Accepting state, 119
Agrawal, Kayal, Saxena, 363

Aleph, 37
Algorithm, 3, 27
Algorithmically computable, 27
Algorithmically decidable, 126
Alphabet, 105
Ambiguity, 253
  if-then-else , 226
  of CFGs , 225
  undecidability, 227
Ambiguity , 225
Ambiguity and nondeterminism, 253
Ambiguous grammar, 225
Approximant, 194
  for fixed-points, 194
Ariane, 8
AU, 425
  GFP, 427
  LFP, 426
  recursion, 425
Automata theory, 6
Automaton/logic connection, 186
Avoid wasted work, 11
Axiomatization, 324

Büchi Automata, 389
Büchi automata, 428
  emptiness, 433
  expressiveness, 428
Bag, 16
Barter, 38
Basketball, 56
BDD, 186, 199
  as minimized DFA, 187
  backward reachability, 192
  BED tool, 190
  Boolean functions, 191
  forward reachability, 192
  reachability analysis, 192
  redundant decoding, 189
  variable ordering, 190
BDD vs ROBDD, 187
BED
  reachability example, 197
BED tool, 190
Binary decision diagrams, 187
Black hole state, 123
Boolean functions, 191
Boolean satisfiability, 331
Bottom, 98
  function, 100
  value, 98
Bryant, 186

Cantor, 39
Cardinal numbers, 49
Cardinality, 37
  $\aleph_0$, 37
  $\aleph_1$, 37
  $\aleph_{0.5}$??, 43
  trap, 39
Cartesian product, 21
  examples of, 21
Certificate, 348
  Diophantine, 361
CFG, 217
  fixed-points , 224
  left-linear , 237
  right-linear , 237
  simplification, 232
CFG to NFA, 238
CFG to PDA, 254
CFL
  deterministic, 235
  pumping lemma, 239
CFLs and decidability, 264
Characteristic sequence, 22
  infinite, 108
Chomsky hierarchy , 228
Chomsky normal form, 262
Church-turing thesis, 272
CKY algorithm, 262
Clay institute, 345
Clique, 349
Closure, 60, 124
  prefix, 115
  reflexive, 60
  reflexive and transitive, 60, 124
  transitive, 60
Closure of CFLs, 264
Closure properties of CFLs , 230

CNF, 332
CNF conversion, 336
Cocke-Kasami-Younger Algorithm, 262
Code, 292
  angle bracket, 292
Coiled, 179
Collatz's problem, 28
Complete, 323
Completeness, 220
  of a CFG, 220
  undecidable , 220
Complexity theory, 5
Composite, 350, 363
  selling, 350
Computation, 105
Computation engineering, 1
Computation history, 312
Computation tree, 401
Computational procedure, 3
Computational process, 27
Computational tree logic, 403
Computer engineering, 1
Congruence
  radio receiver, 65
Congruence relation, 64
Conjunctive normal form, 332
coNP, 362
coNPC, 362
Consistency
  of a CFG, 220
Consistency , 220
Constructor, 80
Context-free, 229
Context-free grammar, 217
  fixed-points, 224
Continuum Hypothesis, 43
Contradiction
  proof by, 207
Contrapositive, 207
Contrapositive rule, 77
Conversion
  NFA to DFA, 159
Cryptography, 74
  RSA, 74
CTL, 403
  enumerative model checking, 419
  expressiveness, 428
  fixed-point iteration, 421
  fixed-point semantics, 410
  illustration using BED, 424
  LFP and GFP, 425
  semantics, 408
  symbolic model checking, 421
  syntax, 407
Curried form, 30

DBA, 430
DCFL, 235
De-fred, 24, 93, 94, 96
Deadlock, 392
Decidable, 126
Decider, 351
Decision problem, 273
Definitional equality, 76
Demorgan's law, 80
  generalized, 80
Derivation, 219
  leftmost , 219
  parse tree shape, 219
  rightmost , 219
Derivation sequence , 218
Derivation step , 218
Determinism vs ambiguity , 236
Deterministic BA, 430
Deterministic CFLs, 235
Deterministic finite automaton, 4
DFA, 4
  complementation, 165
  Hamming distance, 176
  intersection, 165
  intersection with PDAs, 293
  Kleene star, 166
  minimization, 174
  Myhill-Nerode Theorem, 174
  relationship with BDDs, 174
  string classifier, 124
  ultimate periodicity, 179
  union, 162
DFA Transition function, 121

Diagonalization, 39
  proof, 41
DIMACS format, 337
Diophantine, 360
Disambiguation , 226
Disjunctive normal form, 332
DNF, 332
DNF to CNF, 333
Dog, 37
  Fifi, 37
  Howard, 37
Dominance relation, 62
DPDA and NPDA , 235
Duality
  between ∀ and ∃, 80

Eclosure, 144
Effective computability, 272
Effective procedure, 27
Effectively computable, 27
EG, 421
  recursion, 421
Elementary productions, 233
Engineering mathematics, 2
Enumerative model checking, 419, 432
Equivalence, 58
  class, 60
Equivalence class, 60
Error correcting DFA, 176
Exists
  as infinite disjunction, 79
Exponential DFA, 139
Expressiveness, 428
  Büchi automata, 428
  CTL, 428
  LTL, 428

FBI, 8
Final state, 121
Finite regular language, 187
First-order logic, 79, 323
First-order validity, 326
Fixed-point, 95, 192
  and automata, 101
  equation, 95
  immovable point, 95
  least, 100
  multiple, 100
  of a Photocopying machine, 95
  on a calculator, 95
Fixed-point iteration, 194
Fixed-points
  greatest, 101
FOL
  enumerable, 331
  semi decidable, 331
Forall
  as infinite conjunction, 16, 78
Formal methods, 8
  disappearing, 13
Formal model, 2
Formal verification, 136, 155
Forward reachability, 193
Fred, 23, 93
Frontier , 218
Full contradiction, 209
Function, 22
  1-1, 25
  bijection, 26
  computable, 27
  correspondences, 26
  domain, 22
  gappiest, 100
  injection, 25
  into, 26
  many to one, 25
  non computable, 27
  non-termination, 25
  one to one, 25
  onto, 26
  partial, 25
  range, 22
  signature, 22
  surjection, 26
  total, 25
Function constant, 324

Gödel, 185
Game

mastermind, 78
General CNF, 340
Generalized NFA, 170
GNFA method, 170
Grail, 151
|, 153
dot, 153
fa2grail.perl, 153
fmcment, 153
fmcross, 153
fmdeterm, 153
fmmin, 153
ghostview, 153
gv, 153
retofm, 153
Grail tool, 151
Greibach's Theorem, 312

Halting Problem
diagonalization, 50
Halting problem, 301
Hamming distance, 176
Hampath, 358
Higher-order logic, 323
Hilbert, 185
23 problems, 185
first problem, 42
problems, 42
Hilbert style, 324
Hilbert's problems, 42
History, 12
computation theory, 12
Homomorphism, 113
inverse, 114

ID, 124
Identity relation, 59
Iff, 76
Image, 23, 192, 193
Implication, 17
antecedent, 75
consequent, 75
Independent, 323
Individual, 324
Individuals, 79
Induction, 81
arithmetic, 82
complete, 82
noetherian, 84
structural, 81, 85
Inductive, 80
basis elements, 80
closed under, 81
constructor, 80
definition, 80
free, 81
least set, 81
set, 80
Inductive assertions, 247
Infinite loop, 25
Infinitely often, 394
Inherently ambiguous, 227
Input encoding, 353
strong NPC, 364
unary, 353
Instantaneous description, 124
Interior node , 218
Interleaving, 383
Interpretation, 323
Intractable, 345
Invariant, 192
checking, 192
Inverse homomorphism, 114, 168
Irredundant name, 24, 94
Isomorphism, 174
Ivory soap, 40

Kripke structure, 399

Lambda calculus, 3, 23
alpha conversion, 25
alpha rule, 24
associativity, 24
beta reduction, 24
beta rule, 24
bindings, 24
function application, 24
irredundant names, 24
syntactic conventions, 30
Y, 94
Language, 5, 105, 107, 125

cardinality, 108
concatenation, 110, 111
context-free , 217
exponentiation, 111
homomorphism, 113, 114
intersection, 110
of a CFG, 218
prefix closure, 115
recursive definition, 101
regular, 125
reversal, 113
setminus, 110
star, 112
symmetric difference, 110
uncountably many, 108
union, 110
universality, 294
language
empty, 107
Language , 217
Lasso, 180
Lasso shape, 179
Lattice, 64
$\preceq$, 64
all equivalences, 64
glb, 64
greatest lower bound, 64
illustration on 2^S, 64
least upper-bound, 64
lub, 64
LBA, 128, 277
acceptance decidable, 313
undecidable emptiness, 313
Least and greatest FP, 425
Least fixed-point, 192
Left-end marker, 276
Left-to-right scan, 119
Lexical analyzer, 149
Lexicographic, 109
strictly before, 109
Lexicographic order, 109
LHS, 77
Linear bounded automata, 277
Linear bounded automaton, 128
Linear CFGs, 237
Linear-time temporal logic, 405
Livelock, 393
Liveness, 389
Logic
$\exists$, 78
$\forall$, 78
if-then, 75
axiom, 75
axiomatization, 324
complete, 323
first-order, 323
FOL validity, 326
higher-order, 323
Hilbert style, 324
if, 75
implication, 75
independent, 323
individual, 324
interpretation, 323
modus ponens, 325
predicate, 324
proof, 75, 323
propositional, 323
quantification, 78
rule of inference, 75
sound, 323
substitution, 325
theorem, 323
vacuous truth, 75
validity, 323
well-formed formula, 323
wff, 323
zeroth-order, 323
logic
$\Rightarrow$, 75
Logic/automaton connection, 186
Loop invariant, 253
LTL, 405
enumerative model checking, 432
expressiveness, 428
semantics, 406
syntax, 406
LTL vs. CTL, 405, 428

Machines

with one stack, 4
with two stacks, 4
with zero stacks, 4
Map of USA, 39
Mapping reduction, 301
$\leq_m$, 301
Matrix, 372
Mechanical process, 27
Message sequence chart, 394
Millennium problem, 345
Minimalist approach, 3
Mixed left/right linear, 238
Model checking, 381
vs. testing, 387
BDD, 383
disappearing, 387
history, 381
Modular, 113
homomorphism, 113
substitution, 113
Modus ponens, 325
MSC, 394
Multiple fixed-points, 197
Myhill-Nerode Theorem, 174

Natural number
as set, 20
NBA, 428, 431
versus DBA, 431
Nested DFS, 434
NFA, 141
δ, 142
$\rightarrow$, 146
ε moves, 143
ε-closure, 145
$\vdash$, 142
$\vdash^*$, 142
concatenation, 165
generalized, 170
homomorphism, 168
ID, 142
instantaneous description, 142
inverse homomorphism, 168
Kleene-star, 166
language, 147
prefix-closure, 169
reversal, 167
to DFA, 159
to Regular Expressions, 170
token game, 148
union, 162
NFA transition function, 141
NLBA, 277
Non-terminals , 217
Non-trivial property, 312
Nondeterminism, 135, 136, 387
abstraction, 387
over-approximation, 387
power of machines, 137
Nondeterministic Büchi automata, 431
Nondeterministic BA, 428
Nondeterministic machines, 137
NP, 345, 348, 350
NP decider, 350
NP verifier, 348
NP-completeness, 345
NP-hard, 350
Diophantine, 360
NPC, 345
2-partition, 364
3-SAT, 354
3-partition, 364
decider, 351
funnel diagram, 354
strong, 364
tetris, 364
verifier, 348
Number
$\aleph_{0.5}$??, 43
cardinal, 37, 38
integer, 15
natural, 15
real, 15
Numeric
strictly before, 110
Numeric order, 110

Omega-regular language, 430
syntax, 430

One-stop shopping, 224
Order
  partial, 57
  pre, 57
Over-approximation, 135

P, 345
P versus NP, 288
P vs. NP, 345
Parse tree , 218
Parser, 217
PCP, 315
  computation history, 318
  dominoes, 315
  solution, 315
  tiles, 315
  undecidable, 316
PDA, 4, 245, 253
  $\vdash$, 246
  ID, 246
  instantaneous description, 246
  intersection with DFAs, 293
  proving their correctness, 247
  undecidable universality, 313
PDA acceptance, 247
  by empty stack, 249
  by final state, 247
PDA to CFG, 257
Periodic
  ultimate, 76
Philosophers, 390
Photocopying machine
  fixed-point, 95
  image transformation, 95
Pigeon, 85
Pigeon-hole principle, 85
POS, 332
Post's correspondence, 315
Power of computers, 5
Power of machines, 3, 62, 137
Pre-image, 192
Predicate constant, 324
Predicate logic, 324
Prefix, 372
Prefix closure, 115
Prenexing, 371
Presburger, 371
  atomic, 371
  conversion to automata, 376
  encoding, 373
  interpretation, 373
  pitfall to avoid, 378
  quantification, 376
  sentences, 371
  term, 371
Presburger arithmetic, 126, 370
Primed variable, 193
Primes, 363
Procedure, 3
Product of sums, 332
Production, 217
  elementary, 219
Promela, 384
  accept label, 394
  never automaton, 394
  proctype, 394
  progress label, 394
Proof, 75, 323
  reductio ad absurdum, 77
  axiom, 75
  by contradiction, 77
  machine-checked, 441
  mistakes, 439
  model-checker, 441
  of a statement, 75
  reliability, 439
  rule of inference, 75
  theorem prover, 441
Proof by contradiction, 207
Property, 28
Propositional logic, 323
Proving PDAs, 247
Pumping
  case analysis, 207
  full contradiction, 209
  stronger, incomplete, 209
Pumping Lemma, 205
  complete, 205, 212
  incomplete, 205
  Jaffe, 212

one-way, 205
quantifier alternation, 206
Stanat and Weiss, 213
Purely left-linear , 238
Purely right-linear , 238
Push-down automata, 245
Push-down automata , 234
Push-down automaton, 4
Putative queries, 10

Quantifier alternation, 206

RE, 134, 137, 295
closure, 134
Complementation, 138
DeMorgan's Law, 138
Reachability
in graphs, 60
multiple fixed-points, 197
Recognize, 124
Recursion, 93
nonsensical, 94
solution of equations, 97
solving an equation, 94
Recursive definition, 77
Recursively enumerable, 295
Reflexive and transitive closure, 124
Reflexive transitive closure, 60
Regular, 125
Regular Expression
to NFA, 169
Regular expression, 137
Regular expressions, 134
Regular language
closure properties, 211
Regular set
closure properties, 211
Regular sets, 211
Regularity, 211
preserving operations, 211
Rejecting state, 119
Relation, 28
irr, 53
non, 53
antisymmetric, 55
asymmetric, 55
binary, 28, 53
broken journey, 56
co-domain, 29
complement, 29
domain, 29, 53
equivalence, 58
functional, 30
identity, 59
intransitive, 56
inverse, 29
irreflexive, 54
non-reflexive, 55
non-symmetric, 55
non-transitive, 57
partial functions as, 30
partial order, 57
preorder, 57
reflexive, 54
restriction, 29, 60
short cut, 56
single-valued, 30
symmetric, 55
ternary, 29
total, 58
total functions as, 30
total order, 58
transitive, 56
unary, 28
universal, 59
Resistor, 68
Respect, 65, 113
concatenation, 113
operator, 65
Reverse, 238
of a CFG, 238
of a CFL, 238
RHS, 77
Rice's Theorem
corrected proof, 311
failing proof, 310
Rice's theorem, 309
partition, 309
Robustness of TMs, 276
Run, 105

Russell's paradox, 17, 21

Safety, 389
SAT solver, 338
Satisfiability, 331
  2-CNF, 341
Scanner, 149
  telephone number, 149
Schöenfinkeled form, 30
Schröder-Bernstein Theorem, 43
  $Nat \rightarrow Bool$, 44
  all C programs, 44
Second-order logic, 79
Sentence , 218
Sentential form , 218
Sequence, 105
Set, 16
  Real versus *Nat*, 43
  cardinality, 37
  complement, 20
  comprehension, 16
  countable, 38
  empty, 16
  intersection, 19
  numbers as, 20
  powerset, 16, 22
  proper subset, 19
  recursively defined, 18
  subtraction, 19
  symmetric difference, 20
  union, 19
  unique definition, 18
  universal, 18, 19
Skolem constant, 326
Skolem function, 326
Skolemization, 326
Solving one implies all, 11
SOP, 332
Sound, 323
SPIN, 384
  interleaving product, 394
  message sequence chart, 394
  MSC, 394
  property automaton, 394
Stack, 128
  single, 128
  two, 128
Start symbol , 217
State
  accepting, 121
  black hole, 123
  final, 121
State explosion, 383
State transition systems, 192
String, 105
  ε, 106
  length, 107
  substr, 107
  concatenation, 107
  empty, 106
  in living beings, 106
  lexicographic order, 109
  numeric order, 110
  reversal, 113
String classifier, 124
Strong NP-completeness, 364
Strongly NP-complete, 364
Structural induction
  proof by, 81
Substitution, 325
Sum of products, 332
Symbol, 105, 107
  bounded information, 107
Symbolic model checking, 421

Tape, 275
  doubly infinite, 275
  singly infinite, 275
Telephone number NFA, 149
Temporal logic, 382
Term, 68
Terminals , 217
Testing computers, 9
Tetris, 364
Theorem, 323
Theorem prover, 73
  ACL2, 74
  floating point arithmetic, 75
  Flyspec project, 74
Therac-25, 8

Tic-tac-toe, 198
TM, 274
  deterministic, 274
  nondeterministic, 274
  robust, 276
TR, 108, 295
Transition function, 124
  δ, 121
  $\hat{\delta}$, 124
  for strings, 124
  total, 121
Transition systems, 192
Triple, 21
Trivial partition, 309
Tuple, 21
Turing machine, 2, 128
Turing recognizable, 108, 295
Turing-Church thesis, 272
Two stacks + control = TM, 276
Twocolor, 347
Type, 20
  versus sets, 20

Ultimate periodicity, 179, 181
Ultimately periodic, 76
Unambiguous, 10
Unique definition, 18
  function, 77, 94
  functions, 82
  sets, 18
Uniqueness, 77
Universal relation, 59
Unsatisfiability core, 340
Unsatisfiable CNF, 340
Until
  GFP, 427
  LFP, 426
  recursion, 425

Vacuously true, 75
Valid, 323
Validity, 326
  undecidable, 329
Variable ordering, 190
Venus probe, 267
Verifier, 348

Weird but legal C program, 44
Well-formed formula, 323
Well-founded partial order, 84
Well-parenthesized, 224
well-parenthesized, 215
WFF, 323
What if queries, 10
Wild-card, 137

Y function, 96

Zchaff, 338
Zeroth-order logic, 323