Appendix A

Context free languages and push down automata

This document is an appendix to the book What Can Be Computed?: A Practical Guide to the Theory of Computation *by John MacCormick (Princeton University Press, 2018).*

Previously, we have seen two distinct computational models: Turing machines (chapter 5), and finite automata (chapter 9). These two models represent two extremes in the world of computability. On the one hand, Turing machines are as powerful as any physically-realistic computer. On the other hand, the finite automaton seems to be the weakest computational model that deserves serious study: it can decide regular languages but nothing else, making it strictly weaker than the Turing machine. Are there any interesting models between these two extremes? The answer is "yes, but not many." Surprisingly, there are only a few useful models of intermediate power. Of these, the most important are the *push down automata* (pdas), which are the main topic of this chapter.

To understand pdas, we must first understand one of the most fundamental data structures in computer science: the *stack*. As you probably know, a stack is a data structure that allows you to store as many pieces of information as you want, but the only way to access the information is to look at, and optionally remove, the *top* of the stack. It might help to think of having a large supply of blank cards in a desk drawer and a desktop where you can store a single pile (i.e. stack) of cards that have symbols written on them. There are two operations you can perform on this pile:

Push You can take a blank card from the drawer, write a symbol s on the card, and place it on top of the pile. We say you have *pushed* the symbol s onto the stack.

Pop Provided the stack isn't empty, you can pick up the top card from the pile, look at the symbol s, then throw the card in the trash. We say you have *popped* the stack, obtaining the symbol s.

Recall that a finite automaton is a specialized, restricted form of Turing machine: it can't edit the tape, and its read-write head always moves to the right. A pda is also a restricted form of Turing machine, although it's not usually described like that. Usually, a pda is described as a finite automaton augmented with a stack. Perhaps a sensible name for these things would be "stack automata." Why are they instead called "push down automata"? This comes from a commonly-used physical analogy for stacks. Sometimes we think of a stack as being spring-loaded, so that every time we add a new item to the top, everything gets "pushed down" and the top of the stack stays at the same level. Some cafeterias use a system like this with stacks of dining plates.

Pdas come in two flavors: we have the deterministic pda (dpda) and the nondeterministic pda (npda). Surprisingly, and very importantly, dpdas and npdas are not equivalent in terms of computational power. This will be one of the main results of the chapter, demonstrating that (in contrast with dfas/nfas, and dtms/ntms) nondeterminism can affect computability. We won't give a proof of the non-equivalence of dpdas and npdas, but we will see a persuasive example at the end of section A.1.

The other key result of the chapter is that pdas correspond to a fundamental and important class of languages known as *context free languages* (cfls). Cfls are a central concept in the theory of compilers and programming languages. It turns out that an understanding of pdas is very useful for building practical compilers. The proof of equivalence for pdas and cfls is given in sections A.3 and A.5.

A.1 Definition and examples of pdas

We mentioned above that a pda can be thought of as a finite automaton augmented with a stack. Our formal definition, however, is based on Turing machines:

> **Pda, dpda, and npda**. A *pda* is a 2-tape Turing machine with the additional properties (1)–(4) below. If the Turing machine is deterministic, we have a deterministic pda or *dpda*. If the Turing machine is nondeterministic, we can emphasize this by

calling it an *npda*, but the term "pda" incorporates both npdas and dpdas. The two tapes of a pda have special names: the first tape is called the *input*, and the second tape is called the *stack*. The input has the same restrictions as a finite automaton with ϵ-transitions:

(1) Cells on the input tape cannot be altered.

(2) The head for the input tape always stays or moves to the right (it cannot move left).

The stack tape is restricted to push and pop operations, formalized as follows:

(3) The stack is initially empty (i.e. the stack tape contains only blank symbols).

(4) The only permitted sequences of operations on the stack tape are:

- (Push) Write a non-blank symbol s, then move right one cell.
- (Pop) Move left one cell, read symbol s, then overwrite s with a blank and stay at the current cell.[1]

Naturally, the input to a pda is provided at the start of the input tape before the computation begins. By definition, this input consists of a sequence of non-blank symbols followed by infinitely many blanks.

The easiest way to describe a specific pda is via a transition diagram, such as figure A.1. The details of this figure will be explained shortly. First, let's understand the specialized notation on these transition diagrams, which is somewhat different to the finite automaton and Turing machine notation from earlier chapters. Specifically, each transition is labeled

$$X, s; s'$$

where X is the scanned input tape symbol, s is the popped stack symbol, and s' is the symbol (or symbols, as described soon) pushed back onto the stack. For example, the transition label

$\mathtt{C}, \mathtt{g}; \mathtt{a}$ means "if the scanned input symbol is $\mathtt{C}$ and the popped stack symbol is a $\mathtt{g}$, push an $\mathtt{a}$ onto the stack and move right to the next input symbol."

[1]Attempting to pop an empty stack is equivalent to doing nothing. In terms of the underlying Turing machine operations, this corresponds to our convention that commanding a Turing machine to move left from cell 0 leaves the head where it is.

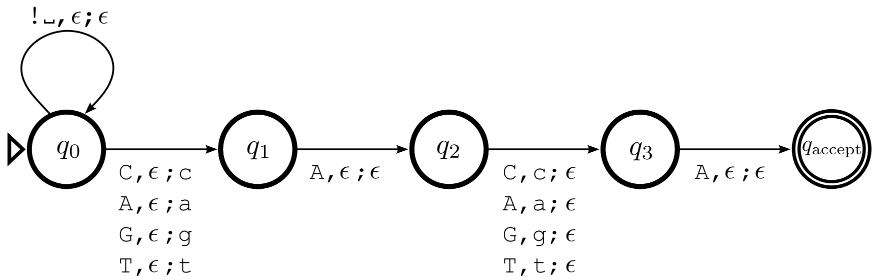


Figure A.1: A pda solving the problem ContainsNANA.

Any or all of the three components can be replaced with ϵ, as the following examples show:

ϵ, `g`; `a`	means	"if popped `g`, then push `a`, and don't move input head"
`C`, ϵ; `a`	means	"if scanned `C`, then push `a` without popping anything, and move input head right"
`C`, `g`; ϵ	means	"if scanned `C` and popped `g`, move input head right"
`C`, ϵ; ϵ	means	"if scanned `C`, move input head right without altering the stack"

Figure A.1 provides a concrete example. This pda solves the problem ContainsNANA, which searches a string for one of the substrings "CACA", "GAGA", "TATA", or "AAAA" (for details, see figure 8.1). As with the transition diagrams in earlier chapters (see section 5.1), we allow abbreviated notation, so "!␣" matches any single symbol other than a blank.

Hence, our `containsNANA` pda is nondeterministic. When reading a C, G, A, or T, the state q_0 produces two clones: one returns to q_0, and the other advances to q_1. Note how the stack is used to remember the first character of the string to be matched. For example, whenever the pda encounters a G, one of the clones pushes "g" onto the stack and transitions to q_1. If the next input symbol is an A, the pda moves to q_2. At this point, the clone rejects unless the top stack symbol matches the scanned input symbol. In our specific example, then, the pda transitions to q_3 only if the next input symbol is a G.

The `containsNANA` pda illustrates a useful convention: it often improves readability to use a separate input alphabet and stack alphabet. In this book, we will typically use uppercase letters for the input and lowercase letters for the stack. This is not formally required by the definition of a pda, but it does make it easier to read transitions like "C, g; ϵ." The

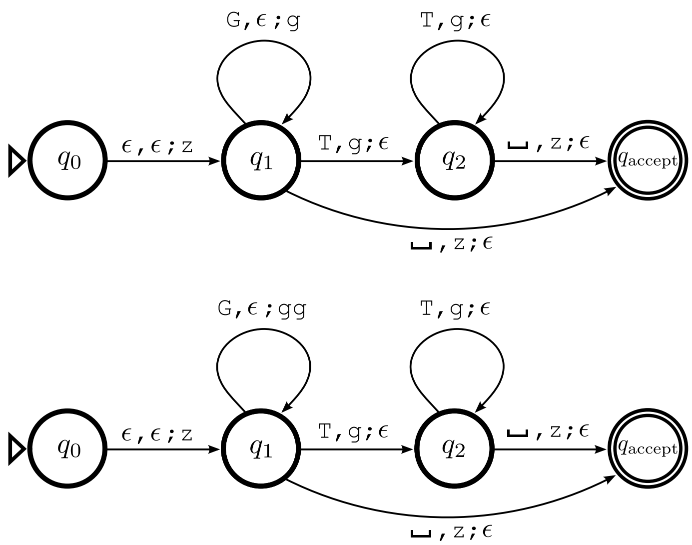


Figure A.2: Top: The pda `GnTn`, which decides the language $\texttt{G}^n\texttt{T}^n$. Bottom: The pda `GnT2n`, which decides the language $\texttt{G}^n\texttt{T}^{2n}$. The only difference between these pdas is in the $q_1 \to q_1$ transition.

alphabet for the underlying Turing machine is just the union of the input and stack alphabets.

The book materials provide the simulators `simulateDpda.py` and `simulateNpda.py` for deterministic and nondeterministic pdas respectively. The format for describing pdas in ASCII is similar to the Turing machine descriptions in chapter 5 and the finite automata descriptions in chapter 9. ASCII descriptions of all pdas in this chapter are provided with the book materials; see `containsGAGA.pda` `containsNANA.pda` for specific examples. The simulators can be invoked using the same style of commands as for Turing machines and finite automata:

```
>>> simulateDpda(rf('containsGAGA.pda'), 'TTGAGATT')
>>> simulateNpda(rf('containsNANA.pda'), 'TTGAGATT')
```

Note that our `containsNANA` pda decides a language that can also be decided by an nfa (or dfa). The nfa would need more states: instead of a single q_1, for example, we would need four separate states to remember which character needs to be matched. So in this case, the presence of the stack yields a more compact automaton but doesn't appear to provide extra computational power.

In contrast, our next two pda examples demonstrate that pdas are strictly more powerful than dfas and nfas. The top panel of figure A.2 shows a pda called GnTn that decides the language $\texttt{G}^n\texttt{T}^n$. In section 9.5, we proved that this language cannot be decided by any dfa or nfa. Hence, this example provides an immediate proof that pdas are more powerful than dfas.

Let's investigate the GnTn pda in more detail. Its first action, in the $q_0 \to q_1$ transition, is to push the symbol "z" onto the stack without reading any input. This is a trick that our pdas will use frequently: there is no built-in way of recognizing the bottom of the stack, but we can achieve this by pushing a unique symbol onto the stack before doing anything else. In this book, we will always use "z" to mark the bottom of the stack, but of course any other symbol could be used for that purpose.

After this, the GnTn pda loops in q_1, pushing one "g" for every "G" encountered in the input. As soon as a "T" is encountered, it pops a "g" and switches to q_2, where it continues to pop one "g" for every "T". The automaton accepts if and only if the blank marking the end of the input is encountered at the same time that the bottom of the stack ("z") is popped. This means the number of g's pushed equals number of g's popped, so the number of G's equals the number of T's. The lower curved transition enables the special case of the empty string being accepted. Thus, the pda accepts precisely the strings of the form $\texttt{G}^n\texttt{T}^n$ for $n \geqslant 0$.

The lower panel of figure A.2 shows a slight variant. The only difference is the $q_1 \to q_1$ transition, which pushes two g's instead of one. The notation for this transition, "$\texttt{G}, \epsilon; \texttt{gg}$", demonstrates another useful convention for our transition diagrams: we allow a single transition to push two or more symbols onto the stack, if desired. This is just a notational convenience, since we could have written out a sequence of states that push one symbol at a time instead. The order of the pushes runs right to left. For example:

$\texttt{C}, \texttt{g}; \texttt{abc}$	means	"if the scanned input symbol is C and the popped stack symbol is a g, push a c, then a b, then an a onto the stack, and move right to the next input symbol."

Returning to the lower panel of figure A.2, let's determine what language it decides. The input must begin with some G's, say n of them. For each of these G's, two g's get pushed. So, immediately before the first T is encountered, there are $2n$ g-symbols on the stack. For the remainder of the input, one g is popped for every T. Thus, the PDA accepts precisely strings of the form $\texttt{G}^n\texttt{T}^{2n}$ for $n \geqslant 0$. Although we didn't explicitly prove it in chapter 9, it's easy to see that this language is also irregular and therefore another example of the extra power of pdas. The next claim summarizes these observations.

Claim A.1 There exist non-regular languages that can be decided by dpas. Hence, dpdas are strictly more powerful than dfas or nfas.

Proof of the claim. To prove the claim, we need only a single example of a non-regular language that is decidable by a dpda. The `GnTn` example discussed above is a suitable example. □

The `GnTn` example shows us how we can use a pda's stack to count things. In the next example, we use it for a more detailed comparison. For this, we need some new notation. Given a string s, write s^R for the reverse of s. For example, if s = "`GAT`" then s^R = "`TAG`". (We used a similar notation to reverse an entire language in section 9.7; now we use it to reverse individual strings.) Recall that a *palindrome* is a string that reads the same backwards as forwards. More formally, s is a palindrome if $s = s^R$. For example, the following are all palindromes: ϵ, "`G`", "`GG`", "`CTC`", and "`GTTATTG`". We use this idea to define two new languages that form palindromes from "`A`" and "`T`" characters:

$$\texttt{EvenPalindromes} = \{s = (\texttt{T}\,|\,\texttt{A})^* \text{ such that } s = s^R \text{ and } |s| \text{ is even}\}$$
$$\texttt{MarkedPalindromes} = \{s = (\texttt{T}\,|\,\texttt{A})^*\,\texttt{C}\,(\texttt{T}\,|\,\texttt{A})^* \text{ such that } s = s^R\}$$

So `EvenPalindromes` contains strings such as "`ATTA`" and "`TTAATT`", whereas `MarkedPalindromes` contains strings such as "`ACA`" and "`TTACATT`". The key difference is that, in `MarkedPalindromes`, the center of each string is *marked* with a "`C`". In `EvenPalindromes`, we can only determine the center of the string by examining the entire string. By the way, we consider palindromes of even length purely to simplify the pda that decides this language. It is a useful exercise to create a pda that decides palindromes of any length.

Figure A.3 shows pdas deciding each of our palindrome languages. These pdas use the same trick of pushing a "`z`" to mark the bottom of the stack. The key observation is that state q_2 pops "`A`" and "`T`" characters in precisely the reverse order that they were pushed on in state q_1—by definition, stacks release their contents in the reverse of the insertion order. This is how we enforce the requirement that the string reads the same backwards or forwards.

The only difference between the two pdas of figure A.3 is in the transition from q_1 to q_2. The top pda can follow this transition at any time without consuming any input or stack symbol, whereas the bottom pda insists on reading the center-marking "`C`" before transitioning to q_2. This apparently-small difference is deceptive: it means that the top panel's q_1 is

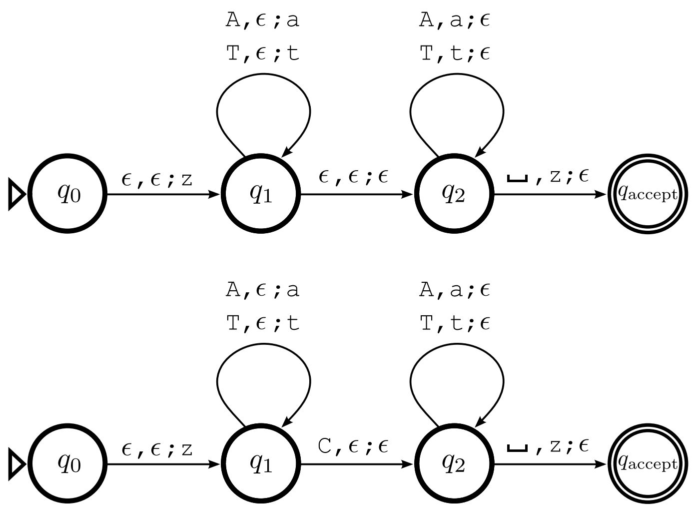


Figure A.3: Top: An npda deciding the language `EvenPalindromes`. Bottom: A dpda deciding the language `MarkedPalindromes`. The only difference between these pdas is in the $q_1 \to q_2$ transition.

nondeterministic, since any "`A`" or "`T`" in the input string offers the choice of remaining in q_1 or transitioning to q_2. In effect, the top pda launches a new clone for every "`A`" or "`T`", just in case the scanned input symbol is the start of the second half (that is, the reversed half) of the string. The bottom pda doesn't need this nondeterminism: because of the center-marking "`C`", it can tell exactly when it reaches the middle of the string. In fact, a careful analysis of each transition in the bottom pda reveals that the entire pda is deterministic.

It turns out that we have encountered a truly fundamental difference between `EvenPalindromes` and `MarkedPalindromes`: we constructed an npda that decides `EvenPalindromes`, but it can be shown that no deterministic pda decides `EvenPalindromes`. This important and surprising difference between npdas and dpdas is highlighted in the following claim.

Claim A.2 There exist languages that can be decided by an npda but not by any dpda.

The proof of this claim lies beyond the scope of this book and is omitted, but the example of figure A.3 is persuasive. Spend a few minutes trying to construct a dpda for `EvenPalindromes`! The set of languages that can be decided by dpdas is known as the *deterministic context free languages*.

We briefly return to this concept in section A.6.

We finish our overview of the computational power of pdas by stating another result without proof, this time demonstrating that Turing machines have strictly greater power than pdas.

Claim A.3 There exist languages that can be decided by a Turing machine but not by any pda.

Again, a proof of this claim is beyond the scope of this book, but we can provide a few hints about why it is true. One example of a language that can't be decided by pdas is $\{\mathtt{G}^n\mathtt{T}^n\mathtt{A}^n \mid n \geqslant 0\}$, or GnTnAn for short. Clearly, GnTnAn can be decided by a Python program and hence by a Turing machine. But it can be shown that, because they have only a single stack, pdas can't keep track of *both* matching pairs of G's and T's *and* matching pairs of T's and A's. The proof employs a more advanced variant of the pumping lemma, known as the "pumping lemma for context free languages."

A.2 Context free grammars

In computer science, a "grammar" is a set of rules for producing strings. Let's start with an informal example:

$$\mathtt{s} \to \mathtt{CGaT}$$
$$\mathtt{a} \to \epsilon \mid \mathtt{Aa}$$

Our grammars will always generate strings by begining with the *start symbol*, usually denoted s. Symbols appearing on the left-hand side of a rule (before the "→") are called *variables*. In this book, variables will usually be lowercase ASCII letters; the variables in the example above are s and a. The remaining symbols are called *terminals*. In this book, terminals will usually be uppercase ASCII letters; the terminals in the example above are C, A, G, and T. In most of our examples, we will restrict the terminals to this genetic alphabet. In general, of course, the variables and terminals could be drawn from any disjoint alphabets.

The rules in the example grammar above tell us we can replace "s" with "CGaT", and we can replace "a" with either ϵ or "Aa". Rules can be applied as many times as we wish. For example, in this grammar we can obtain the sequence

$$\mathtt{s} \to \mathtt{CGaT} \to \mathtt{CGAaT} \to \mathtt{CGAAaT} \to \mathtt{CGAAT}.$$

A sequence like this is called a *derivation*. Any string that can be derived is called a *sentential form*. In our example, sentential forms include "`S`", "`CGaT`", "`CGAAaT`", and "`CGAAT`". Most often, we are interested in deriving strings that contain no variables. We call these *terminal strings*, or just *strings*. So, "`CGAAT`" is a terminal string generated by our example grammar, whereas "`CGAAaT`" is a sentential form but not a terminal string. The set of all terminal strings that can be derived by a grammar G is called the *language generated by* G. It's easy to see that the language generated by our example grammar is represented by the regular expression `CGA*T`.

Grammars are an important and fundamental concept. They can be used to unify all of theoretical computer science. For example, there exists a class of *regular grammars* that generate all regular languages. Every regular grammar G corresponds to some dfa D (and vice versa), such that D decides the language generated by G. Similarly, there exists a class of *unrestricted grammars* that generate all recognizable languages. Every unrestricted grammar G corresponds to some Turing machine M (and vice versa), such that M recognizes the language generated by G. This fact is particularly important for historical reasons, since researchers such as Emil Post developed a complete theory of computation based on unrestricted grammars, independently of Alan Turing's development of the theory in terms of Turing machines.

Despite their fundamental importance, we mostly ignore grammars in this book. Our approach is instead based on Turing machines and special cases of Turing machines, such as dfas and pdas. However, there is one class of grammars that we cannot ignore, since it is ubiquitous in computer science. This is the class of *context free grammars*. Here's a formal definition:

> **Context free grammar, context free language**. A context free grammar (cfg) consists of: an alphabet of *variables* including the *start symbol*, a separate and disjoint alphabet of *terminals*, and a finite set of *rules*. Each rule maps a single variable to a string of variables and terminals.
>
> A language generated by a cfg is called a *context free language* (cfl).

As a notational convenience, we often combine rules that map a given variable using the vertical bar, "|". For example, our initial example at the start of section A.2 combined the two rules `a` $\to \epsilon$ and `a` $\to$ `Aa` into the more compact notation `a` $\to \epsilon \,|\,$ `Aa`.

What makes a grammar "context free"? The key part of the definition is that the left-hand side of any rule contains exactly one variable. More general grammars allow additional constraints on the left-hand side, as in the rule $\texttt{aG} \to \texttt{ATG}$, which allows us to replace the variable "a" with "AT" only when the "a" is followed by a "G". In other words, this rule can only be applied when the "context" of the variable includes a "G" on the right. By insisting that the left-hand side of a rule contains only a single variable v, we are declaring that the rule can be applied whenever v occurs in a sentential form, regardless of v's context.

Derivation trees and ambiguity

Let's now focus our attention on the following cfg, which we'll call G_1:

$$\begin{aligned} \texttt{s} &\to \texttt{sc} \,|\, \texttt{st} \,|\, \epsilon \\ \texttt{c} &\to \texttt{CAT} \\ \texttt{t} &\to \texttt{TAG} \end{aligned} \tag{A.1}$$

As a simple but valuable exercise, experiment with the G_1 grammar until you understand exactly what language it generates. After a minute or two, it should become clear that this language is represented by the regular expression $(\texttt{CAT} \,|\, \texttt{TAG})^*$. But you probably found several different ways of deriving strings in the language. For example, consider the string "TAGCAT". There are in fact four different ways to derive this string, including the following two that will be of special interest to us:

$$\underline{\texttt{s}} \to \underline{\texttt{s}}\texttt{t} \to \underline{\texttt{s}}\texttt{ct} \to \underline{\texttt{c}}\texttt{t} \to \texttt{CAT}\underline{\texttt{t}} \to \texttt{CATTAG} \tag{A.2}$$

$$\underline{\texttt{s}} \to \texttt{s}\underline{\texttt{t}} \to \underline{\texttt{s}}\texttt{TAG} \to \texttt{s}\underline{\texttt{c}}\texttt{TAG} \to \underline{\texttt{s}}\texttt{CATTAG} \to \texttt{CATTAG} \tag{A.3}$$

Here we have introduced some new notation that helps us to understand the derivations: in each sentential form, the variable that is about to be replaced is underlined. Clearly, whenever there are two or more variables in a sentential form, we can choose which one to replace next. If a derivation always chooses to replace the leftmost variable in the current sentential form, we call it a *leftmost derivation*. If a derivation always chooses to replace the rightmost variable in the current sentential form, we call it a *rightmost derivation*. Line (A.2) above is a leftmost derivation, and (A.3) is a rightmost derivation. The real point of leftmost and rightmost derivations is that they capture the underlying structure of a derivation. Although we could continue to analyze both leftmost and rightmost, they turn out to have similar properties. It will be simpler to focus only on leftmost derivations.

Next we must understand *derivation trees*, which are an alternative way of viewing derivations. Given a derivation, the corresponding derivation

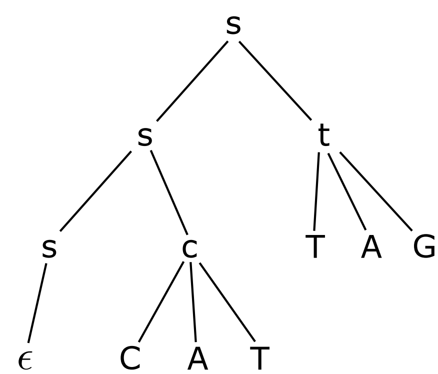


Figure A.4: Derivation tree for "CATTAG" in the G_1 grammar.

tree is defined recursively: the root node is the start symbol s, and the children of any node v are the symbols produced when a rule is applied to v. The children must of course be listed in order from left to right, as specified by the rule that produced them. The *yield* of a derivation tree is the string formed by concatenating the leaves of the tree from left to right. Figure A.4 gives a derivation tree of "CATTAG" in the G_1 grammar; note that the yield is indeed "CATTAG". Before reading on, trace out the leftmost and rightmost derivations on this tree, and convince yourself of the simple correspondence between the tree and these derivations.

It follows immediately from the definition of derivation tree that the internal nodes are all variables and the leaf nodes are all terminals or ϵ. It's also easy to see that a given derivation tree corresponds to a unique leftmost derivation, obtained by generating the tree in the standard left-first, depth-first order. Thus, there is a one-to-one correspondence between derivation trees and leftmost derivations.

Is it possible for a string to have two different leftmost derivations? (Or equivalently, can a string have two different derivation trees?) Unfortunately, the answer is yes. Consider the following cfg, denoted G_2, which adds two new rules to G_1:

$$\begin{aligned} \mathtt{s} &\to \mathtt{sc} \,|\, \mathtt{st} \,|\, \epsilon \\ \mathtt{c} &\to \mathtt{CAT} \,|\, \mathtt{CA} \\ \mathtt{t} &\to \mathtt{TAG} \,|\, \mathtt{AG} \end{aligned}$$

Now consider the string "CATAG". As a simple but important exercise, try to write down two distinct leftmost derivations and two distinct derivation trees for this string, before looking at the solutions below. Hopefully,

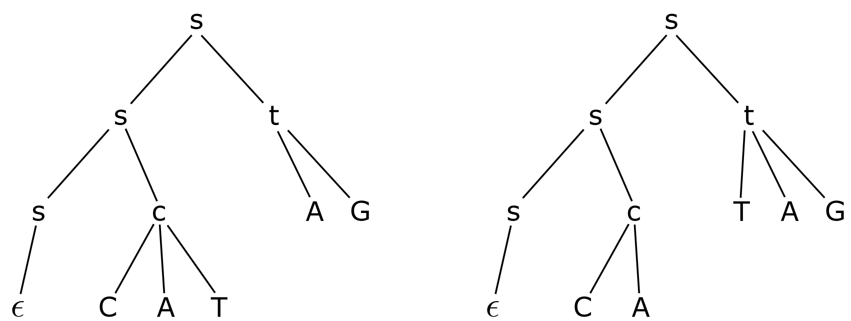


Figure A.5: Distinct derivation trees for “CATAG” in the G_2 grammar..

you were able to obtain the following leftmost derivations, where the only difference is in the fifth sentential form:

$$\underline{\texttt{s}} \to \underline{\texttt{s}}\texttt{t} \to \underline{\texttt{s}}\texttt{ct} \to \underline{\texttt{c}}\texttt{t} \to \texttt{CAT}\underline{\texttt{t}} \to \texttt{CATAG} \tag{A.4}$$

$$\underline{\texttt{s}} \to \underline{\texttt{s}}\texttt{t} \to \underline{\texttt{s}}\texttt{ct} \to \underline{\texttt{c}}\texttt{t} \to \texttt{CA}\underline{\texttt{t}} \to \texttt{CATAG} \tag{A.5}$$

The difference is more obvious in the trees of figure A.5, where we see immediately that the terminal “T” has a different parent in the two derivations. This situation is called *ambiguity*. Formally, a terminal string is *ambiguous* if it has two distinct leftmost derivations (or equivalently, two distinct derivation trees). A cfg is *ambiguous* if it has one or more ambiguous terminal strings in its language.

So, we have already proved that G_2 is an ambiguous cfg, since we exhibited a string with two distinct leftmost derivations. Interestingly, however, the problem of ambiguity in G_2 can be fixed. We will now describe a new cfg, G_3, which generates the same language as G_2, yet is not ambiguous. The rules of G_3 are:

$$\begin{aligned}
&\texttt{s} \to \texttt{sc} \,|\, \texttt{sa} \,|\, \texttt{scT} \,|\, \texttt{sTa} \,|\, \epsilon \\
&\texttt{c} \to \texttt{CA} \\
&\texttt{t} \to \texttt{AG}
\end{aligned}$$

We leave it as an exercise to prove that G_3 is in fact unambiguous and generates the same language as G_2.

Ambiguity in context free grammars is a subtle and fascinating topic, but we don’t dwell on it here. Instead, we will state without proof two important facts about ambiguity:

- There do exist *inherently ambiguous* cfgs: that is, cfgs that cannot be reformulated to remove ambiguity, as we can do with G_3 above.

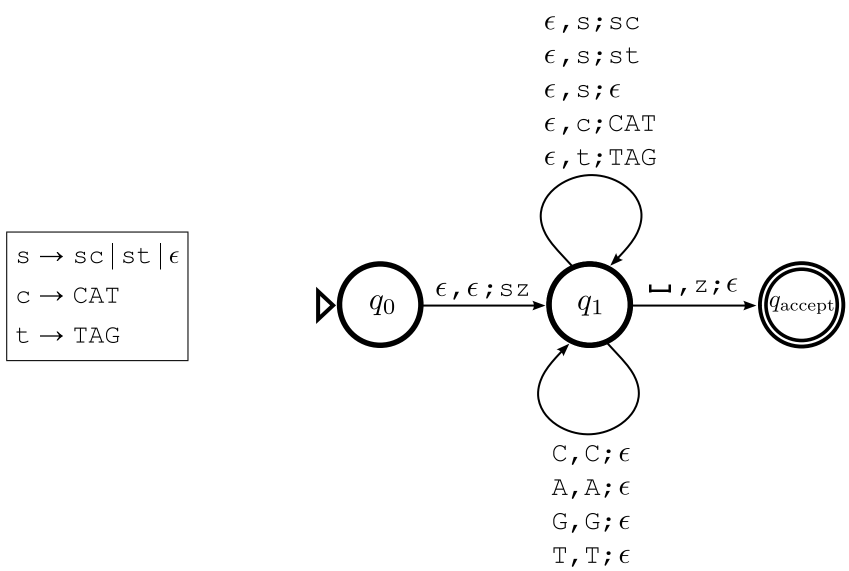


Figure A.6: Left: The grammar G_1, which generates the language L represented by the regex (CAT | TAG)* (see section A.2). Right: A pda equivalent to G_1. This pda recognizes the language L and is provided with the book materials as `cattag.pda`. For additional explanatory details, experiment with `showCATTAGhist.py`.

Consequently, there exist *ambiguous context-free languages*, which are generated only by ambiguous cfgs. One example of an ambiguous cfl is

$$\{\texttt{C}^n\texttt{A}^m\texttt{T}^k \text{ such that } n = m \text{ or } m = k\}.$$

- Let AmbiguousCfg be the decision problem that asks whether a given cfg is ambiguous. Then AmbiguousCfg is undecidable.

A.3 Converting a cfg to a pda

We now turn to the most important result of this chapter, which is that pdas recognize precisely the set of context free languages. Note carefully the use of the word "recognize" rather than "decide" here, and if necessary review section 4.5 to understand the difference between these concepts. For the remainder of this chapter, we focus only on *recognizing* languages and not deciding them. The reason is that there is no easy way for grammars to reject a string. Grammars *generate* strings. So we can imagine using a

step number	state	tape	stack (top to the left)
0	q_0:	$\boxed{\texttt{C}}$ A T	ϵ
1	q_1:	$\boxed{\texttt{C}}$ A T	sz
2	q_1:	$\boxed{\texttt{C}}$ A T	scz
3	q_1:	$\boxed{\texttt{C}}$ A T	cz
4	q_1:	$\boxed{\texttt{C}}$ A T	CATz
5	q_1:	C $\boxed{\texttt{A}}$ T	ATz
6	q_1:	C A $\boxed{\texttt{T}}$	Tz
7	q_1:	C A T $\boxed{␣}$	z
8	q_{accept}:	C A T ␣ $\boxed{␣}$	ϵ

Figure A.7: A complete computation of the pda in figure A.6, accepting the string "CAT". Note that the computation tree of this pda for the input "CAT" is infinite. The computation shown here is the only path leading to a positive leaf in the computation tree. It corresponds to the leftmost derivation s → sc → c → CAT.

grammar to create a list of all the strings in a language: begin with the start symbol, and repeatedly apply all of the possible rules in the grammar. This approach will never terminate, but it will generate every string in the language of the grammar. Thus, given any particular string S, we will eventually recognize S if it is in the language. But there is no obvious way of rejecting S if it is not in the language.[2]

Therefore, we aim to prove that cfgs and pdas are equivalent in the sense that they recognize the same class of languages. To prove this result, we will need to show that (i) given any cfg G, there exists a pda that recognizes the language generated by G; and (ii) Given any pda M, there exists a cfg that generates the language recognized by M. In this section, we prove part (i).

Before giving a general proof of (i), it will be helpful to examine a detailed example. Figure A.6 presents this example, which is based on the grammar G_1. The rules of G_1 are shown again in the left panel of figure A.6. Recall that G_1 was first introduced in section A.2, where we saw that it generates the language L represented by the regex (CAT|TAG)*. So our challenge is to come up with a pda that recognizes any string of the form

[2]The question of membership in a context free language is decidable: a cubic-time algorithm known as CYK achieves this. But CYK itself runs on a Turing machine, not a pda.

(CAT | TAG)*.

One possible solution is shown in the right panel of figure A.6. This is also provided with the book materials as `cattag.pda`. You can of course simulate this pda using `simulateNpda.py`, as mentioned previously. But it is also recommended to experiment with the file `showCATTAGhist.py`, which demonstrates some of the facilities provided for analyzing pda computations in more detail.

The basic idea of the `cattag` pda of figure A.6 is that it will use its stack to mimic any left-most derivation from G_1. The first transition, $q_0 \rightarrow q_1$, pushes the bottom-of-stack marker `z` and the grammar's start symbol `s` onto the stack. The only accepting transition, $q_1 \rightarrow q_{\text{accept}}$, will accept only if we have finished reading the input string (and therefore have reached a blank symbol on the tape) *and* the stack is empty.

At the heart of the `cattag` pda are the nine $q_1 \rightarrow q_1$ transitions. The bottom four are easy to understand: these transitions consume a terminal symbol on the input tape if and only if the same terminal symbol is at the top of the stack. So we need to arrange that, for any string that is in the language, the string's symbols can be pushed onto the stack in the correct order. This is accomplished by the five $q_1 \rightarrow q_1$ transitions shown above q_1. Each of these transitions implements one of the rules of the grammar. There are five transitions here, because there are five rules in the grammar. (Note that the line "s $\rightarrow$ sc | st | ϵ" represents three separate rules.) Each rule is implemented in the pda by a transition that pops the left-hand side of the rule from the stack, and pushes the right-hand side onto the stack. For example, the rule c $\rightarrow$ CAT pops c and pushes CAT. Note that this is an exception to a normal practice of using disjoint input and stack alphabets. In this construction, it is convenient to use symbols from the input alphabet on the stack.

The `cattag` pda incorporates non-determinism, because sometimes there are several rules that could be applied to the top stack symbol. For this particular grammar, the only stack symbol that permits nondeterminism is the start symbol `s`, which generates three different clones corresponding to the three possible rules whose left hand side is `s`.

The overall effect of this pda can be seen by allowing it to process the input string "CAT", as shown in figure A.7. This computation is simulated in detail by the provided program `showCATTAGhist.py`; now would be a good time to try it if you haven't done so already. The sequence of computational steps shown in figure A.7 is only one possible path through the computation tree: step numbers 1 and 2 involve non-determinism, because there are three possible transitions whenever `s` is at the top of the stack. In this figure, only the nondeterministic choices leading to the acceptance of the string "CAT" are shown. Notice how steps 2, 3, and 4 each correspond to the application of a rule in the leftmost derivation of this string, which is $\texttt{S} \to \texttt{SC} \to \texttt{C} \to \texttt{CAT}$. The other steps involve initialization of the stack (step 1), acceptance of the blank symbol with an empty stack (step 8), and consumption of terminal symbols in the input string while simultaneously popping them off the stack (steps 5, 6, 7).

Let's now prove that this construction can be made to work for any cfg.

Claim A.4 Let G be a cfg that generates the language L. Then there exists a pda M that recognizes L.

Proof of the claim. We construct M as in the example of figure A.6. M contains only the three states $q_0, q_1, q_{\text{accept}}$. The transitions $q_0 \to q_1$ and $q_1 \to q_{\text{accept}}$ are exactly as shown in figure A.6. In addition, we have one $q_1 \to q_1$ transition for each terminal symbol, following the same pattern as the rules below q_1 in figure A.6—let's call these the *terminal symbol transitions*. Formally, M has a $q_1 \to q_1$ transition labelled "$X\texttt{,}X\texttt{;}\epsilon$" for each terminal X in G.

Also, we have one $q_1 \to q_1$ transition for each rule in G, following the same pattern as the rules above q_1 in figure A.6—let's call these the *grammar rule transitions*. Formally, M has a $q_1 \to q_1$ transition labelled "$\epsilon\texttt{,}v\texttt{;}W$" for each rule $v \to W$ in G.

Now let T be a string in L. We need to show that M accepts T. Well, we know that T has a leftmost derivation $T_0 \to T_1 \to \ldots \to T_n$, where $T_0 = \texttt{S}$ and $T_n = T$. Each step in this derivation can be mimicked by following the corresponding grammar rule transition in M. If the resulting sentential form T_i has any terminal symbols at its left end, we then consume these symbols using the corresponding terminal symbol transitions, before moving to the next step of the derivation. A completely formal proof of correctness would use induction to show that the following invariant holds: whenever M has finished applying the transitions corresponding to step i in the derivation, M's stack contains precisely T_i, with any prefix of terminal symbols removed, and this same prefix has been read on the input tape. We leave these details as an exercise. Note that the invariant does yield an accepting transition at the end of the derivation, since the stack must be empty and all symbols of T have been read on the input tape.

Finally, we need to show that M does not accept strings outside L. This follows by contradiction, using similar reasoning to the above. Specifically, suppose that M accepts some string $T \notin L$. Then we examine the accepting computation, and note the sequence of grammar rule transitions taken by M in this computation. This sequence of grammar rules corresponds to a leftmost derivation of T, contradicting the fact that $T \notin L$. □

A.4 Subcomputations for pdas

Before investigating the connections between pdas and cfgs any further, we need a more detailed understanding of pdas. This section describes these necessary details, covering the *standard form* of a pda, *matching push-pop transition pairs*, *stack-preserving subcomputations*, and finally the *splitting* and *peeling* operations on these subcomputations.

Standard form of a pda

A pda M is in *standard form* if

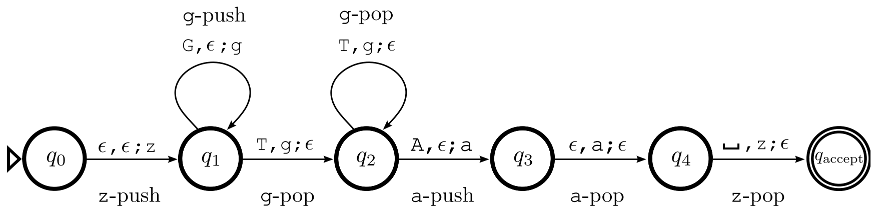
1. M can enter $q_{\texttt{accept}}$ only when the stack is empty;
2. Every transition of M either pushes exactly one symbol onto the stack or pops exactly one symbol off the stack.
3. The input alphabet and stack alphabet of M are disjoint, except for the blank symbol.
4. Before accepting, M always consumes the entire input. By convention, the input is terminated with a blank symbol. Thus, any transition to $q_{\texttt{accept}}$ is guaranteed to read the blank symbol from the input tape while simultaneously, due to condition (2) above, popping the last remaining symbol off the stack.

As the next claim shows, we lose nothing by assuming our pdas are in standard form.

Claim A.5 Given a pda M, there exists an equivalent pda M' in standard form.

Sketch proof of the claim. We sketch the key ideas of the proof, leaving the formal details as an exercise. To achieve condition (1) above, we can use the trick already mentioned in section A.1. First, choose a symbol that is not already in the stack alphabet—in our examples, we always use `z` for this. Insert a new state and transition from the initial state that does nothing but push `z` onto the stack. Insert another state before any transitions to $q_{\texttt{accept}}$. This state uses a self-transition to pop all non-`z`'s, then when it detects a `z` it pops that and transitions to $q_{\texttt{accept}}$. This guarantees the pda's stack is empty when it enters $q_{\texttt{accept}}$, as required.

To achieve condition (2), we simply add new states and transitions wherever necessary, breaking down operations that involve pushing or popping more than one symbol into their constituent parts. For example, a transition that pops `a` then pushes `bc` would be broken down into three transitions: an `a`-push, then a `c`-push, then a `b`-push. We also need a technique



stack symbol	pushes	pops
a	$q_2 \to q_3 : \texttt{A}, \epsilon; \texttt{a}$	$q_3 \to q_4 : \epsilon, \texttt{a}; \epsilon$
g	$q_1 \to q_1 : \texttt{G}, \epsilon; \texttt{g}$	$q_1 \to q_2 : \texttt{T}, \texttt{g}; \epsilon$ $q_2 \to q_2 : \texttt{T}, \texttt{g}; \epsilon$
z	$q_0 \to q_1 : \epsilon, \epsilon; \texttt{z}$	$q_4 \to q_{\mathrm{accept}} : \llcorner\!\lrcorner, \texttt{z}; \epsilon$

Figure A.8: Top: The pda GnTnA in standard form. Bottom: The pda's transitions have been organized into matching pushes and pops, according to which stack symbol they employ. There are four matching push-pop pairs, because the g-push matches two possible g-pops.

for dealing with transitions like "A, ϵ; ϵ", which read a tape symbol without touching the stack. This is converted to standard form by pushing an extra symbol onto the stack and immediately popping it. For example, we could convert "A, ϵ; ϵ" into "A, y; ϵ" followed by "ϵ, ϵ; y".

Observe that condition (3) can be achieved very easily, by substituting new unique symbols for any that are used in both the input and stack alphabets.

Finally, it is also easy to guarantee condition (4), by inserting an extra state before q_{accept} and allowing the new state to consume the remainder of the input before executing the required "␣, z; ϵ" transition to q_{accept}. □

Matching push-pop transition pairs

Once we convert a pda into standard form, its accepting computations have a very nice property: every symbol pushed onto the stack must eventually be popped off at some time later in the computation. (This follows immediately from the fact that a pda in standard form must have an empty stack when it enters q_{accept}.) This motivates us to think about organizing all of the transitions from a given pda into matching pairs that push and pop the same symbol.

The top panel of figure A.8 provides an example of a pda in standard form. This pda recognizes the language $\{\texttt{G}^n\texttt{T}^n\texttt{A} \mid n \geqslant 1\}$, so we will refer to this pda as GnTnA. The figure includes some extra labels on the transitions to help with organizing them into matching push-pop pairs. Because the pda is in standard form, we know every transition pushes or pops exactly one symbol. So it makes sense to describe a transition as, for example, a "g-push" or "z-pop." In the bottom panel figure A.8, the seven transitions of the pda have been placed into a table. The transitions are sorted according to which stack symbol they push or pop; this determines in which row of the table each transition is placed. The transitions are further sorted into pushes and pops, and this determines the column for each transition. From the table, we can quickly read off all possible matching pairs of pushes and pops. For example, for the stack symbol a, we see there is exactly one possible matching pair in the top row: the a-push "$q_2 \to q_3 : \mathtt{A}, \epsilon; \mathtt{a}$" matches the a-pop "$q_3 \to q_4 : \epsilon, \mathtt{a}; \epsilon$". For the stack symbol g, however, there are two possible matching pairs. The g-push "$q_1 \to q_1 : \mathtt{G}, \epsilon; \mathtt{g}$" matches either of the g-pops "$q_1 \to q_2 : \mathtt{T}, \mathtt{g}; \epsilon$" or "$q_2 \to q_2 : \mathtt{T}, \mathtt{g}; \epsilon$".

Clearly, this is a small and simple example. In general, suppose that for a given stack symbol x we have k_1 x-pushes and k_2 x-pops. Then there would be $k_1 k_2$ matching pairs of x-pushes and x-pops.

Stack-preserving subcomputations

Given a pda in standard form, an accepting computation of that pda can be thought of as a sequence of legal configurations and transitions beginning in q_0 and ending in q_{accept} with an empty stack. Figure A.9(a) shows an example of an accepting computation for GnTnA. This example demonstrates our notation for pda computations, which includes the contents of the stack after each transition, written below the current state. The input tape symbol consumed by a transition is written above the arrow between states, and we refer to the string of all of these symbols concatenated together as the string *consumed by* the computation. For example, the accepting computation of figure A.9(a) consumes the string "GGTTA".

We define a *subcomputation* to be any sequence of legal consecutive configurations and intervening transitions. Figure A.9(b) shows an example, consisting of a sequence of four consecutive states and the intervening transitions, drawn from the accepting computation (a). As before, we define the string consumed by the subcomputation in the obvious way, in this case yielding "GTT". Of course, a string consumed by a subcomputation is not necessarily in the language recognized by the pda.

Let us now pay attention not to the input symbols consumed, but the behavior of the stack. Notice how in this particular subcomputation, the stack contains gz at the beginning of the subcomputation and contains z at the end of the subcomputation. Usually, we won't be interested in subcomputations that alter the stack like this. Instead, we concentrate on

$$
\boxed{\begin{array}{ccccccccccccccccc}
q_0 & \xrightarrow{\epsilon} & q_1 & \xrightarrow{G} & q_1 & \xrightarrow{G} & q_1 & \xrightarrow{T} & q_2 & \xrightarrow{T} & q_2 & \xrightarrow{A} & q_3 & \xrightarrow{\epsilon} & q_4 & \xrightarrow{\sqcup} & q_{\text{accept}} \\
\epsilon & & z & & g & & g & & g & & z & & a & & z & & \epsilon \\
 & & & & z & & g & & z & & & & z & & & & \\
 & & & & & & z & & & & & & & & & &
\end{array}}
$$

(a) an accepting computation for `GnTnA`

$$
\boxed{\begin{array}{ccccccc}
q_1 & \xrightarrow{G} & q_1 & \xrightarrow{T} & q_2 & \xrightarrow{T} & q_2 \\
g & & g & & g & & z \\
z & & g & & z & & \\
 & & z & & & &
\end{array}}
$$

(b) a subcomputation for `GnTnA`

$$
\boxed{\begin{array}{ccccccccc}
q_1 & \xrightarrow{G} & q_1 & \xrightarrow{G} & q_1 & \xrightarrow{T} & q_2 & \xrightarrow{T} & q_2 \\
z & & g & & g & & g & & z \\
 & & z & & g & & z & & \\
 & & & & z & & & &
\end{array}}
$$

(c) a stack-preserving subcomputation for `GnTnA`

Figure A.9: Examples of an accepting computation and subcomputations for the `GnTnA` pda (see figure A.8).

$$\boxed{\begin{array}{ccccccccccccc} q_1 & \xrightarrow{G} & q_1 & \xrightarrow{G} & q_1 & \xrightarrow{T} & q_2 & \xrightarrow{T} & q_2 & \xrightarrow{A} & q_3 & \xrightarrow{\epsilon} & q_4 \\ z & & g & & g & & g & & z & & a & & z \\ & & z & & g & & z & & & & z & & \\ & & & & z & & & & & & & & \end{array}}$$

(a) a stack-preserving subcomputation for `GnTnA`, before splitting

$$\boxed{\begin{array}{ccccccccc} q_1 & \xrightarrow{G} & q_1 & \xrightarrow{G} & q_1 & \xrightarrow{T} & q_2 & \xrightarrow{T} & q_2 \\ z & & g & & g & & g & & z \\ & & z & & g & & z & & \\ & & & & z & & & & \end{array}} \quad \boxed{\begin{array}{ccccc} q_2 & \xrightarrow{A} & q_3 & \xrightarrow{\epsilon} & q_4 \\ z & & a & & z \\ & & z & & \end{array}}$$

(b) two stack-preserving subcomputations resulting from splitting (a)

Figure A.10: Example of splitting a stack-preserving subcomputation.

subcomputations that leave the stack undisturbed, and this motivates our next definition.

A subcomputation is *stack-preserving* if the initial and final content of the stack is identical, and none of the initial content is popped during the subcomputation. Note that it is not enough for the final content to be the same as the initial content: we insist that the content remains undisturbed, so it is not permitted to pop any of the initial content during the subcomputation and replace it before the end. Figure A.9(c) shows an example of a stack-preserving subcomputation for `GnTnA`, which consumes the string "`GGTT`". Note that any accepting computation automatically satisfies the conditions for being a stack-preserving subcomputation, so figure A.9(a) provides another example. Any single configuration is defined to be a *trivial* subcomputation. Because it doesn't disturb the stack, a trivial subcomputation is also stack-preserving.

Splitting and peeling pda subcomputations

Stack-preserving subcomputations can be decomposed into simpler parts via two operations that we will call *splitting* and *peeling*.

We tackle splitting first. Suppose that, at some point before the end of the subcomputation, the stack returns to its initial condition. Then we can *split* the subcomputation at that point, creating two shorter subcomputations. The configuration at the point of the split is duplicated, becoming the end of the first component and the start of the second. Figure A.10 gives an example, where we split at the configuration in state q_2, when the stack first returns to its initial content `z`. Note that the components of a

$$\boxed{\begin{array}{ccccccccc} q_1 & \xrightarrow{G} & q_1 & \xrightarrow{G} & q_1 & \xrightarrow{T} & q_2 & \xrightarrow{T} & q_2 \\ z & & g & & g & & g & & z \\ & & z & & g & & z & & \\ & & & & z & & & & \end{array}}$$

(a) a stack-preserving subcomputation for `GnTnA`, before peeling

$$\boxed{\begin{array}{ccccc} q_1 & \xrightarrow{G} & q_1 & \xrightarrow{T} & q_2 \\ g & & g & & g \\ z & & g & & z \\ & & z & & \end{array}}$$

(b) the stack-preserving subcomputation resulting from peeling (a)

Figure A.11: Example of peeling a stack-preserving subcomputation.

split are indeed always stack-preserving, because we split at a point where the stack is in its initial condition.

Next, we move on to peeling. We *peel* a stack-preserving subcomputation by removing its first and last configurations. Figure A.11 gives an example. Note that if we applied the peeling operation a second time in this example, we would be left with a trivial subcomputation. Obviously, trivial subcomputations cannot be peeled.

It's worth noting that peeling a stack-preserving subcomputation doesn't always result in a stack-preserving subcomputation. For example, what would happen if we peeled the subcomputation in figure A.10(a)? The resulting subcomputation would begin with the stack `gz` and end with stack `az`. And even if the stack had ended with the same content, it's possible that the peeled version could disturb the stack while in one of its intermediate configurations. In fact, the importance of the peeling operation results from the following property: *if a stack-preserving subcomputation can't be split, then peeling it will result in another stack-preserving subcomputation.* We prove a more precise formulation of this fact in claim A.6 below. Nevertheless, it would be a valuable exercise to prove it now, before reading on.

A.5 Converting a pda to a cfg

In this section we complete our proof that pdas and cfgs are equivalent, by showing that any pda can be converted to an equivalent cfg. We will first give an explicit recipe for constructing the cfg; later, we will prove that this cfg has the desired properties. Given a pda M in standard form, we will denote the corresponding grammar by G_M. We use as a running example the case of M = GnTnA, where GnTnA is the pda in figure A.8. To describe G_M, we need to describe its variables (including a start variable), terminals, and rules. The terminals are easiest so let's start there: they consist of M's input alphabet, with the blank symbol excluded. For GnTnA, this means the terminals are G, T, and A.

Next we describe the the variables of G_M. If G_M has k states, then there will be k^2 variables: one for each ordered pair of states. Let us to denote these by $v_{i,j}$, where i and j run over all the possible states, including q_{accept}. For GnTnA, there are 36 variables: $v_{0,0}$, $v_{0,1}$, $v_{0,2}$, ..., $v_{0,\mathrm{accept}}$, ..., $v_{\mathrm{accept},3}$, $v_{\mathrm{accept},4}$, $v_{\mathrm{accept},\mathrm{accept}}$. Each variable will have a very useful and important interpretation: the variable $v_{i,j}$ will generate all strings that can be consumed by a stack-preserving subcomputation that begins in state q_i and ends in state q_j. As an example, consider the variable $v_{1,2}$ in the grammar for GnTnA. This variable will turn out to generate all strings of the form $\mathtt{G}^n\mathtt{T}^n$. The variable $v_{2,4}$ will generate only one nonempty string, "A". And the variable $v_{1,1}$ will generate only the empty string. Note that $v_{1,1}$ will *not* generate G, GG, GGG, and so on—these strings are consumed by subcomputations that begin and end at q_1, but none of these subcomputations is stack-preserving. Is also worth emphasizing at this point that this property of the $v_{i,j}$ is something we will have to prove later. It has been mentioned now only to help with understanding and motivation. But note that once we have proved the property, the variable $v_{0,\mathrm{accept}}$ will be particularly important: it will generate all strings that are consumed by stack-preserving subcomputations that begin in q_0 and end in q_{accept}. In other words, $v_{0,\mathrm{accept}}$ will generate precisely the language recognized by M.

Finally we must describe the rules of the grammar G_M. There will be a *start rule* and three other types of rules, which we call *split rules*, *peel rules*, and *vanishing rules*.

The *start rule* is simple: it takes our usual start symbol S and maps it to $v_{0,\mathrm{accept}}$. And as we just noted above, this will go on to generate the language recognized by M. Formally, we have the rule

$$\mathtt{S} \to v_{0,\mathrm{accept}}.$$

The *split rules* are designed to acknowledge the fact that, at least in principle, a subcomputation that starts at q_i and ends at q_j could visit any other state q_k on the way. So we need to allow for the possibility of splitting such a subcomputation into two components: one from q_i to q_k and another from q_k to q_j. Hence, we add all rules of the form

$$v_{i,j} \to v_{i,k}v_{k,j}.$$

stack symbol	pushes	pops
`a`	$q_2 \to q_3 : \mathbf{A}, \epsilon; \texttt{a}$	$q_3 \to q_4 : \epsilon, \texttt{a}; \epsilon$
`g`	$q_1 \to q_1 : \mathbf{G}, \epsilon; \texttt{g}$	$q_1 \to q_2 : \mathbf{T}, \texttt{g}; \epsilon$ $q_2 \to q_2 : \mathbf{T}, \texttt{g}; \epsilon$
`z`	$q_0 \to q_1 : \epsilon, \epsilon; \texttt{z}$	$q_4 \to q_{\text{accept}} : \sqcup, \texttt{z}; \epsilon$

(a) matching push-pop pairs for `GnTnA`

$$
\begin{aligned}
v_{2,4} &\to \mathbf{A}\, v_{3,3} \\
v_{1,2} &\to \mathbf{G}\, v_{1,1}\, \mathbf{T} \\
v_{1,2} &\to \mathbf{G}\, v_{1,2}\, \mathbf{T} \\
v_{0,\text{accept}} &\to v_{1,4}
\end{aligned}
$$

(b) peel rules resulting from the push-pop pairs in (a)

Figure A.12: Example of producing peel rules from matching push-pop pairs.

In the example of `GnTnA`, which has 6 states, this gives us $6^3 = 216$ split rules. For example, the grammar will contain the rule $v_{1,4} \to v_{1,2}v_{2,4}$, which will enable the split shown in figure A.10. Note that our construction creates many more split rules than necessary. In the `GnTnA` grammar, for example, it's clear that a rule such as $v_{3,1} \to v_{3,2}v_{2,1}$ is useless, since there isn't even a path in the transition graph from q_3 to q_1. But it turns out that these useless rules will not affect our proof, and it's easiest to leave them in the grammar rather than trying to calculate exactly which ones will be needed.

The *peel rules* reflect the fact that some stack-preserving subcomputations can be peeled. Here, we will use the structure of M's transitions to ensure that only legal peeling operations are reflected in the grammar. This is done by creating peel rules only for M's matching push-pop transition pairs, which were described earlier. And if any symbols are consumed by the peeled transitions, we allow the rule to generate those symbols as terminals. The details of how this works can difficult to absorb, so we give an example first and then proceed to the general definition.

Recall from figure A.8 that we have four matching push-pop transition pairs for the `GnTnA` pda. These matching push-pop pairs are reproduced in figure A.12(a), except that non-blank symbols consumed from the input tape have been highlighted in bold. The four matching pairs lead to four corresponding peel rules in the grammar; these are shown in figure A.12(b). For example, the first rule in figure A.12(b) originates from the matching pair of an `a`-push and `a`-pop in the first row of figure A.12(a). We imagine a possible stack-preserving subcomputation that begins with the `a`-push, transitioning from q_2 to q_3 while consuming an A, and ends with the a-pop, transitioning from q_3 to q_4. We apply the peel operation to this subcomputation, giving us a new subcomputation that begins in q_3 and ends in q_3. The new subcomputation is represented by $v_{3,3}$, and its overall effect will be the same as the original subcomputation, as long as we record the fact that an A was consumed at the start. That explains the right-hand side of the rule, $\mathtt{A}\, v_{3,3}$.

Similarly, the second rule in figure A.12(b) originates from the matching pair consisting of a g-push ($q_1 \to q_1$) and g-pop ($q_1 \to q_2$). Peeling this subcomputation results in a new subcomputation that begins in q_1 and ends in q_1, but now we recorded the fact that a G was consumed at the start and a T at the end. This yields the right-hand side of the rule, $\mathtt{G}\, v_{1,1}\, \mathtt{T}$.

The other rules originate from similar reasoning. The third rule mimics the peeling operation shown in figure A.11, for example. One technicality is in the final rule, where consuming a blank symbol is not explicitly recorded. This is a minor detail, but it turns out to be the correct behavior because we insist that pdas in standard form consume a blank if and only if they are transitioning to q_{accept}.

The general procedure for creating peel rules follows this same pattern. We create a peel rule for every matching push-pop transition pair in M. Specifically, suppose we have a matching pair in which the push transitions from q_i to q_j while consuming X, and the pop transitions from q_k to q_l while consuming Y. Then we add the following rule to our grammar G_M:

$$v_{i,l} \to X\, v_{j,k}\, Y.$$

In the GnTnA example, this leads to the four rules already discussed above and listed in figure A.12(b).

Finally, we add the *vanishing rules*, which reflect the fact that a trivial subcomputation produces an empty string. So for every state q_i, we add the rule

$$v_{i,i} \to \epsilon.$$

In the GnTnA example, this gives us the six rules $v_{0,0} \to \epsilon$, $v_{1,1} \to \epsilon$, ...

Overview of how the cfg operates

At this point, we have not proved anything about the properties of our grammar G_M. But hopefully it is already intuitively clear how the grammar can mimic the operation of the pda M. The idea is that any accepting computation can be broken down into simpler stack-preserving subcomputations via splitting and peeling. These operations are applied repeatedly until we are left with only trivial subcomputations, which disappear via

$$
\begin{array}{ccccccccccccccccc}
q_0 & \xrightarrow{\epsilon} & q_1 & \xrightarrow{G} & q_1 & \xrightarrow{G} & q_1 & \xrightarrow{T} & q_2 & \xrightarrow{T} & q_2 & \xrightarrow{A} & q_3 & \xrightarrow{\epsilon} & q_4 & \xrightarrow{\sqcup} & q_{\text{accept}} \\
\epsilon & & z & & g & & g & & g & & z & & a & & z & & \epsilon \\
 & & & & z & & g & & z & & & & z & & & & \\
 & & & & & & z & & & & & & & & & &
\end{array}
$$

(a) a computation accepting the input string "`GGTTA`"

`S`	$\rightarrow$	$v_{0,\text{accept}}$	use start rule
	$\rightarrow$	$v_{1,4}$	peel the z-push/z-pop pair
	$\rightarrow$	$v_{1,2}\,v_{2,4}$	split at q_2
	$\rightarrow$	`G` $v_{1,2}$ `T` $v_{2,4}$	peel a g-push/g-pop pair
	$\rightarrow$	`GG` $v_{1,1}$ `TT` $v_{2,4}$	peel another g-push/g-pop pair
	$\rightarrow$	`GGTT` $v_{2,4}$	eliminate trivial subcomputation at q_1
	$\rightarrow$	`GGTTA` $v_{3,3}$	peel the a-push/a-pop pair
	$\rightarrow$	`GGTTA`	eliminate trivial subcomputation at q_3

(b) derivation of the same string using the corresponding grammar

Figure A.13: Example of mimicking a computation on the pda $M =$ `GnTnA`, using the constructed grammar G_M

the vanishing rules. As an example, consider the accepting computation by `GnTnA` in figure A.13(a), which is duplicated from figure A.9(a). This computation accepts the input string "`GGTTA`". The corresponding grammar can mimic this computation, as shown by the derivation in figure A.13(b).

Proof that the cfg operates correctly

We have explained how to construct a cfg G_M from a pda M that is in standard form. It remains to prove that M and G_M are equivalent, i.e. that M accepts a string if and only if G_M generates it. However, it is not easy to prove this directly. Instead, we will prove an even stronger statement which tells us our interpretation of the symbols $v_{i,j}$ is in fact correct. In detail, we would like to prove the following claim:

Claim A.6 Let M be a pda in standard form, and let G_M be the cfg obtained from M via the construction described above, so that G_M possesses the variables $v_{i,j}$. Then $v_{i,j}$ generates the string of terminals S if and only if M has a stack-preserving subcomputation that begins at q_i, ends at q_j, and consumes S.

Proof of the claim. We prove this claim by breaking it into two parts: part 1 for the "if" and part 2 for the "only if." Both parts use the technique of mathematical induction, which has not been employed elsewhere in the book, but is required here.

Part 1 of the proof. We assume that M has a stack-preserving subcomputation that begins at q_i, ends at q_j, and consumes S; we need to show that $v_{i,j}$ generates S. We do this by induction on the length L of the subcomputation, where the "length" is the number of transitions followed. It's worth noting that L is always even, since stack-preserving subcomputations must consist of the same number of pushes and pops. The base case of the induction is a trivial subcomputation, which by definition begins and ends at a single state q_i, has no transitions (i.e., $L = 0$), and consumes only the empty string. Hence, the vanishing rule $v_{i,i} \to \epsilon$ guarantees that the base case holds.

Now we turn to the inductive step. We assume our statement holds for all subcomputations of length at most $L - 2$, and attempt to prove it for L (which we may assume is even). So, suppose we have a stack-preserving subcomputation C of even length $L \geqslant 2$ that begins at q_i, ends at q_j, and consumes S. There are two cases: either (i) C can be split, or (ii) C cannot be split. To assist with visualization and understanding, consult the examples of figure A.10 for case (i) and figure A.11 for case (ii).

Case (i): C can be split, say at q_k, producing two smaller stack-preserving subcomputations: C_1 from q_i to q_k consuming S_1, and C_2 from q_k to q_j consuming S_2, where $S = S_1S_2$. Both C_1 and C_2 are strictly shorter than C, so we can apply the inductive hypothesis to each separately. Hence, we have that $v_{i,k}$ generates S_1 and $v_{k,j}$ generates S_2. Finally, by applying the split rule $v_{i,j} \to v_{i,k}v_{k,j}$, it follows that $v_{i,j}$ generates $S_1S_2 = S$, as desired.

Case (ii): C cannot be split. Let h be the height of the stack when C begins and ends. Because C cannot be split, we know the height of the stack is at least $h + 1$ after every transition except the last. (Otherwise, we could split at the point where the height returned to h.) So the symbol that C initially pushes onto the stack (say, a) remains undisturbed until the very end of the subcomputation, when it is removed by a matching pop. Hence, we can peel this matching pair and the resulting shorter computation will also be stack-preserving (with height at least $h + 1$ throughout the computation). So we will be able to apply the inductive hypothesis to the peeled computation. In detail, suppose C's initial a-push transitions from q_i to q_k consuming X, and suppose C's final a-pop transitions from q_l to q_j consuming Y. (Here, X and Y are either ϵ or symbols from the input alphabet.) Then peeling C results in a shorter stack-preserving subcomputation C' which begins in q_k, ends in q_l, and consumes S', where we must have $S = XS'Y$. Applying the inductive hypothesis to C', we obtain that $v_{k,l}$ generates S'. Finally, by using the peel rule $v_{i,j} \to X\, v_{k,l}\, Y$, we conclude that $v_{i,j}$ generates $XS'Y = S$, as desired. (Note that the peel rule needed for this is actually present in the grammar, because of the matching a-push/a-pop pair described above.)

Part 2 of the proof. We assume that $v_{i,j}$ generates S; we need to show that M has a stack-preserving subcomputation that begins at q_i, ends at q_j, and consumes S. We do this by induction on the length L of the derivation that generates S, where the "length" is the number of rules that are applied. For intuition and visualization in the remainder of the proof, consult figure A.13.

First we deal with the base case of the induction. The shortest possible derivation is a single application of a vanishing rule $v_{i,i} \to \epsilon$, so this is the base case of the induction with $L = 1$. The trivial subcomputation at q_i is stack-preserving and consumes ϵ, so the base case holds.

For the inductive step, we assume the statement holds for all derivations of length less than L, where $L > 1$. We must show that the statement also holds for derivations of length L. The first rule in the derivation is either a split or peel, and we treat these two cases separately. (Why don't we consider the start rule or the vanishing rules? The start rule is irrelevant because it doesn't begin with a variable of the form $v_{i,j}$. The vanishing rules can occur first only when $L = 1$. So we are indeed left with only two cases for the first rule: split or peel.)

Case (i): first rule application is a split. The first step must be of the form $v_{i,j} \to v_{i,k}v_{k,j}$, where $v_{i,k}$ generates some string S_1, $v_{k,j}$ generates some string S_2, and $S = S_1S_2$. The derivations of S_1 and S_2 are shorter than L, so we apply the inductive hypothesis to both, concluding that M has stack-preserving subcomputations C_1, C_2 such that C_1 goes from q_i to q_k consuming S_1, and C_2 goes from q_k to q_j consuming S_2. Concatenating these computations together yields a stack-preserving subcomputation from q_i to q_j that consumes $S_1S_2 = S$, as desired.

Case (ii): first rule application is a peel. Let R denote the peel rule employed as the first step of the derivation. So R must be of the form $v_{i,j} \to X\, v_{k,l}\, Y$, where X and Y are either ϵ or symbols from the input alphabet. We also know that $v_{k,l}$ generates a string S' such that $S = XS'Y$. The derivation of S' is shorter than L, so we can apply the inductive hypothesis and conclude that M has a stack-preserving subcomputation C going from q_k to q_l and consuming S'. The construction of the grammar guarantees that rule R corresponds to a matching push-pop pair for some stack symbol, say a. Therefore, M must possess an a-push that transitions from q_i to q_k consuming X, and an a-pop that transitions from q_l to q_j consuming Y. We can concatenate the a-push with the above subcomputation C and the a-pop. This yields the desired stack-preserving subcomputation, completing the proof. □

Finally, we can tie up the loose ends and use the previous claim to understand the entire language generated by G_M. In essence, the start rule yields exactly the desired behavior. The proof below gives the details.

Claim A.7 Let M be a pda in standard form, and let G_M be the cfg obtained from M via the construction described above. Then the language accepted by M is the same as the language generated by G_M.

Proof of the claim. First we show that a string S accepted by M is in the language generated by G_M. Since S is accepted by M, there is an accepting computation that consumes S. By definition, it is in fact a stack-preserving subcomputation that begins at q_0 and ends at q_{accept}. Applying our previous claim A.6, we conclude that $v_{0,\text{accept}}$ generates S. And the start rule of the grammar, $\mathtt{s} \to v_{0,\text{accept}}$, produces $v_{0,\text{accept}}$. Hence G_M generates S.

Next we complete the proof by showing that if S is generated by G_M, then M accepts S. The derivation of S must begin with the start rule, meaning that $v_{0,\text{accept}}$ generates S. Applying our previous claim A.6, we conclude that M has a stack-preserving subcomputation that begins at q_0, ends at q_{accept}, and consumes S. This is precisely an accepting computation for S, and the proof is complete. □

A.6 Summary of computational power of automata

The previous section completed our proof that pdas (or more specifically, npdas) recognize precisely the set of context free languages. As mentioned in section A.1, the strict subset of the cfls recognized by deterministic pdas is known as the *deterministic context free languages*. Although beyond the scope of this book, the deterministic cfls are of great importance because they can be parsed efficiently by compilers.

Figure A.14 combines the results of this chapter with the earlier ones, summarizing which problems and languages can be decided or recognized by the various computational models we have examined. Models listed in any one row of the table have equivalent computational power, but each row is strictly more powerful than the one above. The examples in the last column demonstrate this by providing examples that cannot be recognized by the model in the row above.

computational model	**languages decided or recognized**	**example that can't be recognized by row above**
dfa, nfa	decide regular languages	any regex, e.g., $\texttt{G}^*\texttt{T}^*$
dpda	decide deterministic context free languages	marked palindromes: $\{\texttt{G}^n\texttt{CT}^n\}$
npda	recognize context free languages	even palindromes: $\{\texttt{G}^n\texttt{T}^n\}$
tm, ntm	decide any decidable language	$\{\texttt{G}^n\texttt{T}^n\texttt{A}^n\}$

Figure A.14: Summary of languages recognized and decided by different computational models.