14

Push-down Automata and Context-free Grammars

This chapter details the design of push-down automata (PDA) for various languages, the conversion of CFGs to PDAs, and vice versa. In particular, after formally introducing push-down automata in Section 14.1, we introduce two notions of acceptance - by final state and by empty stack - in Sections 14.1.2 and 14.1.3, respectively. In Section 14.2, we show how to prove PDAs correct using the *Inductive Assertions* method of Floyd. We then present an algorithm to convert a CFG to a language-equivalent PDA in Section 14.3, and an algorithm to convert a PDA to a language-equivalent CFG in Section 14.4. This latter algorithm is non-trivial - and so we work out an example entirely, and also show how to simplify the resulting CFG *and* prove it correct. In Section 14.5, we briefly discuss a *normal form* for context-free grammars called the *Chomsky normal form*. We do not discuss other normal forms such as the *Greibach normal form*, which may be found in most other textbooks. We then describe the Cocke-Kasami-Younger (CKY) parsing algorithm for a grammar in the Chomsky normal form. Finally, we briefly discuss closure and decidability properties in Section 14.6.

14.1 Push-down Automata

A push-down automaton (PDA) is a structure $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ where Γ is the stack alphabet (that usually includes the input alphabet Σ), z_0 is the initial stack symbol, and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$ is the transition function that takes a state, an input symbol (or ε), and a stack symbol (or ε) to a set of states and stack contents. In particular, the $2^{Q \times \Gamma^*}$ in the range of the signature indicates that the PDA can nondeterministically assume one of many states and stack contents. Also, as the signature of the δ function points out, in each

move, a PDA may or may not read an input symbol (note the ε in the signature), but must read the top of the stack in *every* move (note the absence of a ε associated with Γ).

We must point out that many variations on the above signature are possible. In [111] and in the JFLAP tool [66], for instance, PDAs may also optionally read the top of the stack (in effect, they employ the signature $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow 2^{Q \times \Gamma^*}$). Such variations do not fundamentally change the "power" of PDAs. We adopted our convention—of always reading and popping the stack during every move—because it yields an intuitively clearer algorithm for converting PDAs to CFGs[1] (following [60]).

Notions of Acceptance:

There are two different notions of acceptance of a string by a PDA. According to the first, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. According to the second, a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack. We define these notions in Sections 14.1.2 and 14.1.3. In both these definitions, we employ the notions of *instantaneous descriptions* (ID), and step relations $\vdash$, as well as its reflexive and transitive closure, $\vdash^*$.

Instantaneous Description:

An *instantaneous description* (ID) for a PDA is a triple of the form

(state, unconsumed input, stack contents)

Formally, the type of the instantaneous description of a PDA is $T_{ID} = Q \times \Sigma^* \times \Gamma^*$. The type of $\vdash$ is $\vdash \subseteq T_{ID} \times T_{ID}$. The $\vdash$ relation is as follows:

$$(q, a\sigma, b\gamma) \vdash (p, \sigma, g\gamma)\rangle \textit{ iff } a \in \Sigma_\varepsilon \wedge b \in \Gamma \wedge g \in \Gamma^* \wedge \exists (p, g) \in \delta(q, a, b).$$

In other words, if δ allows a move from state q and stack top b to state p via input $a \in \Sigma \cup \{\varepsilon\}$, then $\vdash$ does allow that. In this process, the stack top b is popped, and the new stack contents described by g is pushed on. The first symbol of g ends up at the top of the stack. Sometimes, the last symbol of g is set to b, thus helping restore b (that was popped). In some cases, g is actually made equal to b, thus modeling the fact that the stack did not suffer any changes.

[1] In fact, a PDA move that optionally reads the top of the stack may be represented by a PDA move that reads whatever is on top of the stack, but pushes that symbol back.

14.1.1 Conventions for describing PDAs

We prefer to draw tables of PDA moves. Please make the tables detailed. *Do write comments* - after all, you are coding in a pretty low-level language that is *highly error-prone*; therefore, the more details you provide, the better it is for readers to follow your work. A diagram will also be highly desirable, as is included in Figure 14.1.

In Section 14.2, we present a method to *formally prove* the correctness of PDAs using the *inductive assertions* method of Floyd [40]. This technique should convince the reader that arguing the correctness of a PDA is akin to verifying a program; both are activities that can be rendered difficult if comments and clear intuitive explanations are not provided.

The diagramming style we employ for PDAs resembles state diagrams used for NFAs and DFAs, the only difference being that we now annotate moves by $insymb, ssymb \rightarrow sstr$ where

- $insymb$ is an input symbol or ε,
- $ssymb$ is a stack symbol, and
- $sstr$ is a string of stack symbols that is pushed onto the stack when the move is executed.

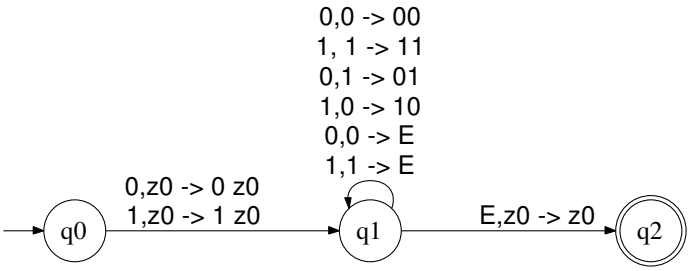
Also, recall that PDAs don't need to specify a behavior for every possible $insymb/ssymb$ combination at every state. If an unspecified combination occurs, the next state of the PDA is undefined. In effect, PDAs are partial functions from inputs and stack contents to new stack contents and new states.
As said earlier, a PDA accepts an input if the input leads it to one of the final states. There is one important difference between DFAs and *D*PDAs: the latter may have undefined input/stack combinations. In other words, one does not have to fully decode inputs and transition to "*black hole*" states upon arrival of illegal inputs, as with a DFA. Finally, recall the difference between NPDAs and DPDAs pointed out in Section 13.5.1. Now we define the different notions of acceptance of PDAs in more detail.

14.1.2 Acceptance by final state

A PDA **accepts a string** w **by final state** if and only if, for some $q_f \in F$, the final set of states of the PDA, $(q_0, w, z0) \vdash^* (q_f, \varepsilon, g)$. For any given PDA, our default assumption will be that of acceptance by final state. The language of the PDA will be defined as follows:

$$\{w \mid \exists g \,.\, (q_0, w, z0) \vdash^* (q_f, \varepsilon, g) \; for \; q_f \in F\}.$$

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




```
      WINNER TOKEN                    LOSER TOKEN

            (q0,001100,    z0)              (q0,001100,  z0)
push    |- (q1, 01100,    0z0)     push   |- (q1, 01100, 0z0)
push    |- (q1,  1100,   00z0)     pop    |- (q1,  1100,  z0)
push    |- (q1,   100,  100z0)     stuck! |- can't accept
pop     |- (q1,    00,   00z0)
pop     |- (q1,     0,    0z0)
pop     |- (q1,       ,    z0)
accept  |- (q2,       ,    z0)

       ACCEPT!                         REJECT!
```

Fig. 14.1. Transition table and transition graph of a PDA for the language $L_0 = \{ww^R \mid w \in \{0,1\}^*\}$, and an illustration of the $\vdash$ relation on input 001100

For a PDA P whose acceptance is defined by final states, we employ the notation "$L(P)$" to denote its language. In contrast, for a PDA P whose acceptance is defined by empty stack, discussed next in Section 14.1.3, we employ the notation "$N(P)$" to denote its language. These are, respectively, subsets of Σ^* that lead the PDA into a final state or cause its stack to be emptied.

14.1.3 Acceptance by empty stack

To further highlight PDAs that accept by empty stack, we leave out the F component from their seven-tuple presentation, thus obtaining the six-tuple $P_2 = (Q, \Sigma, \Gamma, \delta, q_0, z_0)$. For such PDAs, a string w is in its language exactly when the following is true:

$$(q_0, w, z0) \vdash^* (q, \varepsilon, \varepsilon).$$

Here, $q \in Q$, i.e., q is *any state*. All that matters is that the input is entirely consumed *and* an empty stack results in doing so.

Consider the PDA for language L_0 defined in Figure 13.6, reproduced in Figure 14.1 for convenience. This figure also shows how IDs evolve. In particular, nondeterminism is clearly shown by the fact that for the same input string, namely `001100`, one token (called the "winner") can progress towards acceptance, while another token (called "loser") progresses towards demise. Each token also carries with it the PDA stack. As long as one course of forward progress through $\vdash$ exists, and leads to a final state (`q2`, in our present example), the given string is accepted. The other tokens "die out.[2]" Such animations are best observed using tools such as JFLAP [66]. In fact, JFLAP allows users to choose the acceptance criterion—through final states, through empty stack, or *both* (when a final state is reached on an empty stack[3]). JFLAP maintains a view of each token as it journeys through the labyrinth of a PDA transition diagram, therefore watching JFLAP animations is a good way to build intuitions about PDAs.

An arbitrarily given PDA may reach a final state without having emptied its stack. A given PDA may also have an empty stack in a state other than its final state. It is, however, possible to modify a given PDA so that it enters a final state or empties its stack only in a controlled

[2] Nondeterminism in PDAs is akin to the "fork" operation in operating systems such as Unix: an entire clone of the PDA, including its stack, are created at every nondeterministic choice point, and these clones—or tokens as we have been referring to them—either "win" or "lose."

[3] "...on an empty stomach?!"

manner. Specifically, Section 14.1.4 describes how to convert a PDA that accepts by final state into one that empties its stack exactly when in a final state, and Section 14.1.4 describes how to convert a PDA that accepts by empty stack into one that goes into a final state exactly when its stack is empty.

Start state = q00

Current State	Input	Stack top	String pushed	New State	Comments
q00	ε	z00	z0 z00	q0	Start stack with z00; add z0 here.
q0	ε	z0	z0	qS	q0 is a final state; so jump to qS
qS	ε	any	ε	qS	qS drains the stack regardless of what's on top of the stack.
q0	0	z0	0 z0	q1	1a. Decide to stack a 0
q0	1	z0	1 z0	q1	2a. Decide to stack a 0
q1	0	0	0 0	q1	1a'. Decide to stack a 0
q1	0	1	0 1	q1	Forced to stack
q1	0	0	ε	q1	1b. Decide to match
q1	1	1	1 1	q1	2a'. Decide to stack a 1
q1	1	0	1 0	q1	Forced to stack
q1	1	1	ε	q1	2b. Decide to match
q1	ε	z0	z0	q2	Prepare to drain the stack
q2	ε	z0	z0	qS	Jump to stack-draining state qS

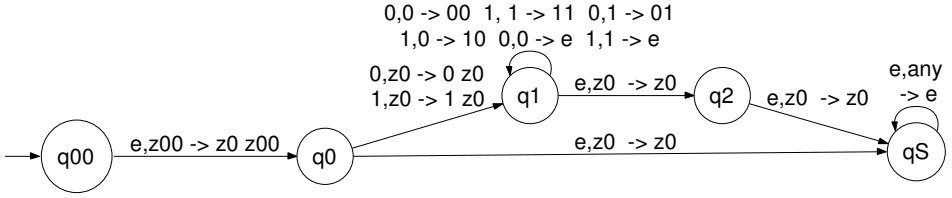


Fig. 14.2. The PDA of Figure 13.6 converted to one that accepts by empty stack. There are some redundancies in this PDA owing to our following a standard construction procedure.

14.1.4 Conversion of P_1 to P_2 ensuring $L(P_1) = N(P_2)$

Given a PDA P_1 that accepts by final state, we can obtain a PDA P_2 that accepts by empty stack such that $N(P_2) = L(P_1)$, simply by ensuring that P_2 has an empty stack exactly when P_1 reaches a final state (for the same input w seen by both these PDAs). The following construction achieves the above condition:

- To avoid the stack of P_2 becoming empty "in between," introduce an extra symbol in P_2's stack alphabet, say z_{00}.
- Start P_2 with its stack containing z_{00}, and then z_0 riding above it.
- The remaining moves of P_2 are similar to that of P_1. However, "final" states are insignificant for P_2. Therefore, whenever P_1 reaches a final state, we introduce in P_2, a move from it to a new *stack-draining state* qS. While in qS, P_2 empties its stack completely.
- No state other than qS tests for the stack-top being z_{00}. Hence, the stack is totally emptied, including z_{00}, only in state qS.

Figure 14.2 illustrates this construction.

14.1.5 Conversion of P_1 to P_2 ensuring $N(P_1) = L(P_2)$

Given a PDA that is defined according to the "accept by empty stack" criterion, how do we convert it to a PDA that accepts by final state? A simple observation tells us that the stack can become empty at any control state. Therefore, the trick is to start the PDA with a new bottom of stack symbol z_{00}. Under normal operation of the PDA, we do not see z_{00} on top of the stack, as it will be occluded by the "real" top of stack z_0. However, in any state, if z_{00} shows up on top of the stack, we add a transition to a newly introduced final state qF. qF is the only final state in the new PDA. Hence, whenever the former PDA drains its stack, the new PDA ends up in state qF.

Illustration 14.1.1 *Develop a push-down automaton for $L_{a^m b^n c^k}$ of Illustration 13.2.1.*

The PDA is shown in Figure 14.3. The PDA will first exercise a nondeterministic option: either I shall decide to match a's and b's, or do b's against c's. Recall that PDAs begin with z0 in the stack, and further we must pop one symbol from the stack in each step. Also, in each move, we can push zero, one, or more (a finite number) symbols back onto the stack.

Here are some facts about this PDA (based on intuitions - no proofs):

Initial state = Q0 Final states = Q0,Qc,Qd

Current State	Input	Stack top	String pushed	New State	Comments
Q0	ε	z0	z0	Qab	Nondeterministically proceed to match a's against b's
Q0	ε	z0	z0	Qbc	..or proceed to match b's against c's
Qab	a	z0	a z0	Qab	Stack the first 'a'
Qab	a	a	a a	Qab	Continue stacking a's
Qab	b	a	ε	Qb	The first b to come; match against an 'a'
Qb	b	a	ε	Qb	One more b came; perhaps more to come; so stay in Qb
Qb	ε	z0	z0	Qc	Go to Qc ("eat c" state), an accept state
Qc	c	z0	z0	Qc	Any number of c's are OK in Qc
Qab	ε	z0	z0	Qc	Enter the "eat c" accept state
Qbc	a	z0	z0	Qbc	Any number of a's can come. Qbc ignores them; it will match b's and c's
Qbc	b	z0	b z0	Qbc1	First b to come; no more a's allowed
Qbc	b	b	b b	Qbc1	Continue stacking b's; no no more a's
Qbc1	c	b	ε	Qm	Continue matching c's; no more b's allowed
Qm	c	b	ε	Qm	Continue matching c's; no more a's or b's
Qm	ε	z0	z0	Qd	A token goes to Qd whenever z0 is on top of the stack

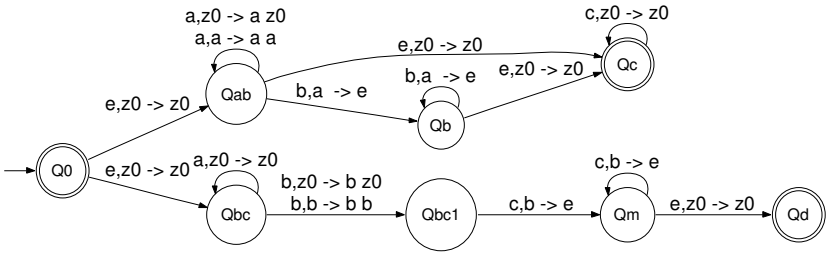


Fig. 14.3. A PDA for $L_{a^m b^n c^k}$ of Illustration 13.2.1

- Nondeterminism is essential. We do not know whether we are going to match a's and b's or b's and c's. In fact, for a string "abc," there must be two different paths that lead to some final state - hence, nondeterminism exists.
- This language is inherently ambiguous. For string "abc" it must be possible to build two distinct parse trees *no matter which grammar is used to parse it.*

14.2 Proving PDAs Correct Using Floyd's Inductive Assertions

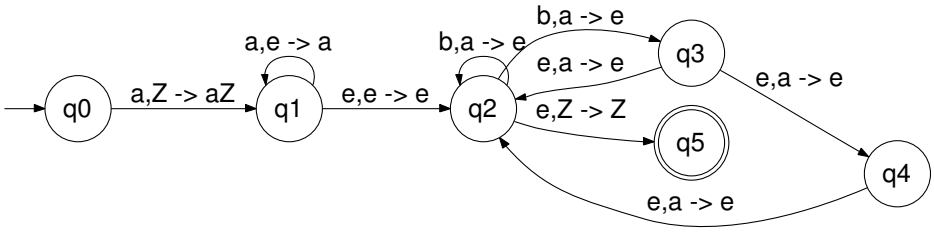


Fig. 14.4. A PDA whose language is being proved correct using Floyd's method

Consider the PDA in Figure 14.4. What is its language? *Think hard before you proceed reading!* □

Guessing the language and proving its correctness

We guess the language of this PDA to be

$$\{a^{i_a}b^{i_b} \mid i_b \leq i_a \leq 3.i_b\}.$$

How do we prove this claim? We will use Floyd's method which rests on finding *loop invariants*. To simplify the discussion of the method a bit, we assume that the PDA has arrived into state q2, having stacked all the a's (this being the only way this PDA can proceed to accept anything). We seek a loop invariant (explained below) for the loop at state q2.

Let i_a be the number of a's initially in the input; likewise for i_b. Let s_a be the number of a's on the stack (note that b's don't get into the

stack, ever). Let n_b be the number of b's yet to be read. Let p_a be the number of a's popped so far.

We explain Floyd's method with respect to a single loop (for more details, see [78]). With this assumption,

- Floyd's method works by pretending that we have arrested the program (PDA in this case) suddenly within its loop, at an arbitrary point during its execution.
- We are then asked to find an accurate description relating all "important" variables used in the loop. This is known as the *loop invariant*.
- The assertion and the variables participating in it must be sufficiently comprehensive so that when we bring the loop to its exit point, the final answer falls out as a special case of the loop invariant.

Considering all this, we come up with these equations:

1. $i_a = s_a + p_a$. This is because all the a's are stacked, and then some are popped, with the rest remaining in the stack.
2. $(i_b - n_b) \leq p_a \leq 3.(i_b - n_b)$. This is because for each 'a' that is popped, we match it against one to three b's. Therefore, the b's read thus far, namely $(i_b - n_b)$, are as per this equation.

Now, these must be *inductive assertions* as far as any q2 to q2 path is concerned. Let us check this:

- In any q2 to q2 traversal, the a's that are popped are the ones that are removed from the stack; hence, the first assertion is inductive.
- Consider the q2 to q3 to q2 traversal (the rest can be similarly argued - see Exercise 14.3). We have n_b going down by 1 while p_a goes up by 2. Thus we have to prove

  $$(i_b - n_b) \leq p_a \leq 3.(i_b - n_b) \Rightarrow (i_b - n_b + 1) \leq p_a + 2 \leq 3.(i_b - n_b + 1),$$

  which follows from simple arithmetic.

Now, specializing the invariant to the exit point, we observe that exiting occurs when $p_a = i_a$ and $n_b = 0$. This immediately gives us $i_b \leq i_a \leq 3.i_b$. □

14.3 Direct Conversion of CFGs to PDAs

When given the option of capturing a context-free language using a PDA or a CFG, what would one choose? In many cases, a CFG would

be easier to first obtain; in that case, there exists a rather *elegant* direct conversion algorithm to convert that CFG into a PDA. This algorithm, in effect, is a *nondeterministic* parsing algorithm. The opposite conversion - a PDA to a CFG - is *much more involved* and is discussed in Section 14.4.

By determinizing the CFG to PDA conversion algorithm, we can obtain an exponential-time parsing algorithm for any CFG. Determinization can be achieved by arranging a *backtracking search*; whenever the NPDA is faced with a nondeterministic choice, we arrange a piece of code that recursively searches for *one* of the paths to accept.[4] In Section 14.5, we discuss the Chomsky Normal Form for a CFG, and in its context, discuss an $O(N^3)$ parsing algorithm attributed to Cocke, Kasami, and Younger (Section 14.5.1).

In the CFG to PDA conversion algorithm, the non-terminals and terminals of the given grammar constitute the stack alphabet of the PDA generated. In addition, the stack alphabet contains z_0. The conversion proceeds as follows:

- Start from state q_0 with z_0 on top of the stack.
- From q_0, jump to state q_M (for "main state") with S, the start symbol of the grammar, on top of the stack, and z_0 below it (restored in the jump).
- In state q_M,
  - If the top of the stack is z_0, jump to state q_F, the only accepting state.
  - If the top of the stack is the terminal x, jump back to state q_M upon input x, without restoring x on top of the stack. Essentially, the parsing goal of x has been fulfilled.
  - If the top of the stack is the non-terminal X, and there is a rule $X \rightarrow R$, where R is a string of terminals and non-terminals, jump to state q_M by popping X and pushing R. Essentially, the parsing goal of X is turned into zero or more parsing subgoals.

Illustration 14.3.1 Let us convert the CFG in Illustration 13.2.1 into a PDA. The resulting PDA is given in Figure 14.5. First set up `S` to be the *parsing goal*. The PDA can then take a nondeterministic jump to two different states. One state sets up the parsing goals `M` and `C`, with `M` on top of the stack. The other path sets up `A` and `N`.

Suppose parsing goal `M` is on top of the stack. We can then set up the parsing goal `a M b`, with `a` on top of the stack. Discharging the

[4] In a technical sense, your computer program would then serve as a deterministic Turing machine that simulates your NPDA.

Initial state = Q0 Final states = QF

Current State	Input	Stack top	String pushed	New State	Comments
Q0	ε	z0	S z0	Qmain	Qmain is the main state of this PDA
Qmain	ε	S	M C	Qmain	Create subgoals, ignoring actual input
Qmain	ε	S	A N	Qmain	Create subgoals, ignoring actual input
Qmain	ε	M	a M b	Qmain	Create subgoals, ignoring actual input
Qmain	ε	M	ε	Qmain	Epsilon production for M
Qmain	ε	N	b N c	Qmain	Create subgoals, ignoring actual input
Qmain	ε	N	ε	Qmain	Epsilon production for N
Qmain	ε	C	c C	Qmain	Create subgoals, ignoring actual input
Qmain	ε	C	ε	Qmain	Epsilon production for C
Qmain	ε	A	a A	Qmain	Create subgoals, ignoring actual input
Qmain	ε	A	ε	Qmain	Epsilon production for A
Qmain	a	a	ε	Qmain	Eat 'a' from input - a parsing goal
Qmain	b	b	ε	Qmain	Eat 'b' from input - a parsing goal
Qmain	c	c	ε	Qmain	Eat 'c' from input - a parsing goal
Qmain	ε	z0	z0	QF	Accept when z0 surfaces (parsing goals met)

Fig. 14.5. CFG to PDA conversion for the CFG of Illustration 13.2.1

parsing goal `a` is easy: just match `a` with the input. On the other hand, with parsing goal `M` on top of the stack, we could also have set up the parsing goal "ε" which means – *we could be done*! Hence, another move can simply empty `M` from the stack. Then, finally, when `z0` shows up on top of stack, we accept, as there are no parsing goals left.

14.4 Direct Conversion of PDAs to CFGs

We first illustrate the PDA to CFG conversion algorithm with an example. As soon as we present an example, we write the corresponding general rule in *slant* fonts. Further details, should you need them, may be found in the textbook of Hopcroft, Motwani, and Ullman [60] whose algorithm we adopt. A slightly different algorithm appears in Sipser's book [111].

```
delta          contains          Productions
-----------------------          -------------------------------
                                 S -> [p,Z0,x] for x in {p,q}

<p,(,Z0>         <p,(Z0>         [p,Z0,r_2] -> ( [p,(,r_1] [r_1,Z0,r_2]

                                     for r_i in {p,q}

<p,(,(>          <p,((>          [p,(,r_2] -> ( [p,(,r_1] [r_1,(,r_2]

                                     for r_i in {p,q}

<p,),(>          <p,e>           [p,(,p]  -> )

<p,e,Z0>         <q,e>           [p,Z0,q] -> e
```

Fig. 14.6. PDA to CFG conversion. Note that `e` means the same as ε.

Consider the PDA that *accepts by empty stack*,

$$(\{p, q\},\ \{(,)\},\ \{(,), Z0\},\ \delta,\ p,\ Z0)$$

with δ given in Figure 14.6. Recall that since this is a PDA that accepts by empty stack, we do not specify the F component in the PDA structure. The above six-tuple corresponds to $(Q, \Sigma, \Gamma, \delta, q_0, z_0)$. This figure also shows the CFG productions generated following the PDA moves. The method used to generate each production is the following. Each step is explained with a suitable section heading.

14.4.1 Name non-terminals to match stack-emptying possibilities

Notice that the non-terminals of this grammar have names of the form `[a,b,c]`. Essentially, such a name carries the following significance:

> It represents the language that can be generated by starting in state `a` of the PDA with `b` on top of the stack, and being able to go to state `c` of the PDA *with the same stack contents* as was present while in state `a`.

This is top-down recursive programming at its best: we set up top-level goals, represented by non-terminals such as `[a,b,c]`, without immediately worrying about how to achieve such complicated goals. As it turns out, these non-terminals achieve what they seek through subsequent recursive invocations to other non-terminals - letting the *magic of recursion* make things work out!

> *General rule: For all states $q_1, q_2 \in Q$ and all stack symbols $g \in \Gamma$, introduce a non-terminal $[q_1, g, q_2]$ (most of these non-terminals will prove to be useless later).*

14.4.2 Let start symbol `S` set up all stack-draining options

All the CFG productions are obtained systematically from the PDA transitions. The only exception is the first production, which, for our PDA, is as follows:

```
S -> [p,Z0,x] for x in {p,q}.
```

In other words, two productions are introduced, they being:

```
S -> [p,Z0,p]
S -> [p,Z0,q].
```

Here is how to understand these productions. `S`, the start symbol of the CFG, generates a certain language. This is the *entire* language of our PDA. The entire language of our PDA is nothing but *the set of all those strings* that take the PDA from its start state `p` to some state, *having gotten rid of everything in the stack*. In our PDA, since it starts with `Z0` on top of stack, that's the only thing to be emptied from the stack. Since the PDA could be either in `p` or `q` after emptying the stack (and since we don't care where it ends up), we introduce both these possibilities in the productions for `S`.

> *General rule: For all states $q \in Q$, introduce one production $S \to [q_0, z_0, q]$.*

14.4.3 Capture how each PDA transition helps drain the stack

A PDA transition may either get rid of the top symbol on the stack *or* may end up *adding* several new symbols onto the stack. Therefore, many PDA transitions do *not* help achieve the goal of draining the stack. However, we can set up recursive invocations to clear the extra symbols placed on top of the stack, thus still achieving the overall goals.

To see all this clearly, consider the fact that δ contains a move, as shown below:

```
delta        contains
----------------------
<p,(,Z0>       <p,(Z0>
```

In this PDA, when in state `p`, upon seeing `(` in the input and `Z0` on top of the stack, the PDA will jump to state `p`, having momentarily gotten rid of `Z0`, but promptly restoring `(` as well as `Z0`. Then the PDA has to "further struggle" and get rid of `(` as well as `Z0`, reaching some states after these acts. It is only *then* that the PDA has successfully drained the `Z0` from its stack. Said differently, to drain `Z0` on the stack while in state `p`, read `(`, invite more symbols onto the stack, and then recursively get rid of them as well. All this is fine, except we don't know rightaway where the PDA will be after getting rid of `(`, and subsequently getting rid of `Z0`. However, this is no problem, as we can *enumerate all possible states*, thus obtaining as many "catch all" rules as possible. This is *precisely* what the set of context-free grammar rules generated for this grammar says:

```
[p,Z0,r_2] -> ( [p,(,r_1] [r_1,Z0,r_2] for r_i in {p,q}
```

The rule says: "if you start from state `p` with a view to *completely* drain `Z0` from the stack, you will end up in some state `r_2`. That, in turn, is a three step process:

- Read `(` and, for sure, we will be in state `p`.
- From state `p`, get rid of `(` recursively, ending up in some state `r_1`.
- From state `r_1`, get rid of `Z0`, thus ending up in the very same state `r_2`!

Fortunately, this is *precisely* what the above production rule says, according to the significance we assigned to all the non-terminals. We will have *sixteen* possible rules even for this single PDA rule!! Many of these rules will prove to be useless.

> *General rule: If* $\delta(p, a, g)$ *contains* $\langle q, g_1, \ldots, g_n$, *introduce one generic rule*
>
> $$[p, g, q_0] \rightarrow a\ [q, a, q_1][q_1, g_1, q_2] \ldots [q_n, g_n, q_0]$$

and create one instance of the rule for each $q_i \in Q$ chosen in all possible ways.

14.4.4 Final result from Figure 14.6

We apply this algorithm to the PDA in Figure 14.6, obtaining an extremely large CFG. We hand simplify, by throwing away rules as well as non-terminals that are never used. We further neaten the rules by assigning shorter names to non-terminals as shown below:

```
Let A=[p,Z0,p], B=[p,Z0,q], C=[q,Z0,p], D=[q,Z0,q],
    W=[p,(,p],  X=[p,(,q],  Y=[q,(,p],  Z=[q,(,q].
```

Then we have the following rather bizzare looking CFG:

```
S -> A | B

A -> ( W A  |  ( X C
B -> ( W B  |  ( X D

W -> ( W W  |  ( X Y
X -> ( W X  |  ( X Z

W -> )
B -> e
```

How are we sure that this CFG is even close to being correct?

We simplify the grammar based on the notions of *generating* and *reachable* from the previous chapter. This process proceeds as follows:

1. Notice that `C,D,Y,Z` are *not* generating symbols (they can never generate any terminal string). Hence we can eliminate production RHS using them.
2. `W` and `B` are generating (`W -> )` and `B -> e`).
3. `X` is not generating. Look at `X -> ( W X`. While `(` is generating and `W` is generating, `X` on the RHS isn't generating – we are doing a "bottom-up marking." The same style of reasoning applies also to `X -> ( X Z`.
4. Even `A` is not generating!

Therefore, in the end, we obtain a short (but still 'bizzare looking') grammar:

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

Fortunately, this grammar is now small enough to apply our verification methods based on *consistency* and *completeness*:

Consistency: Any string s generated by `S` must be such that it has an equal number of `(` and `)`. Further, in any of its proper prefixes, the number of `(` is greater than or equal to the number of `)`.
Completeness: All such strings must be generated by `S`.

Proof outline for consistency:
Let us establish the 'same number of `(` and `)` part. Clearly, `e` (ε) satisfies this part. How about `( W S`? For this, we must state and prove a lemma about `W`:

- Conjecture: W has one more `)` than `(`.
- True for both arms of `W`, by induction.
- Hence, this conjecture about `W` is true.

Therefore, `s` has an equal number of `(` and `)`.

Now, to argue that in any of the proper prefixes of s, the number of `(` is greater than or equal to the number of `)`, we again need a lemma about `W`:

- Conjecture: In any prefix of a string generated by `W`, number of `)` is at most one more than the number of `(`.
- This has to be proved by induction on `W`.

Hence, `S` satisfies consistency.

Completeness

To argue completeness with respect to `S`, we state and prove a completeness property for `W`.

> All the following kinds of strings are generated by `W`: In any prefix of a string generated by `W`, number of `)` is at most one more than the number of `(`.

The proof would proceed as illustrated in Figure 13.3. Now, the completeness of `S` may be similarly argued, as Exercise 14.1 requests.[5]

[5] In fact, the plot will be simpler for this grammar, as there will be no zero-crossings. There could be occasions where the plot touches the x-axis, and if it continues, it promptly takes off in the positive direction once again.

14.5 The Chomsky Normal Form

Given a context-free grammar G, there is a standard algorithm (described in most textbooks) to obtain a context-free grammar $G^{'}$ in the *Chomsky normal form* such that $L(G^{'}) = L(G) - \{\varepsilon\}$. A grammar in the Chomsky normal form has two kinds of productions: $A \rightarrow BC$, as well as $A \rightarrow a$. If ε is required to be in the new grammar, it is explicitly added at the top level via a production of the form $S \rightarrow \varepsilon$. There is another well-known normal form called the *Greibach normal form* (GNF) which may be found discussed in various textbooks. In the GNF, all the production rules are of the form $A \rightarrow aB_1B_2 \ldots B_k$ where a is a terminal and $A, B_1, \ldots, B_k$, for $k \geq 0$, are non-terminals (with $k = 0$, we obtain $A \rightarrow a$). Obtaining grammars in these normal forms facilitates proofs, as well as the description of algorithms. In this chapter, we will skip the actual algorithms to obtain these normal forms, focusing instead on the advantages of obtaining grammars in these normal forms.

A grammar G in the Chomsky normal form has the property that *any* string of length n generated by G must be derived through exactly $2n-1$ derivation steps. This is because all derivations involve a binary production $A \rightarrow BC$ or an unary production $A \rightarrow a$. For example, given the following grammar in the Chomsky normal form,

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

a string `abab` can be derived through a seven step $(2 \times 4 - 1)$ derivation

```
S => SS => ABS => ABAB => aBAB => abAB => abaB => abab.
```

In the next section, we discuss a parsing algorithm for CFGs, assuming that the grammar is given in the Chomsky normal form.

14.5.1 Cocke-Kasami-Younger (CKY) parsing algorithm

The CKY parsing algorithm uses *dynamic programming* in a rather elegant manner. Basically, given any string, such as `0 0 1`, and a Chomsky normal form grammar such as

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

the following steps describe how we "parse the string" (check that the string is a member of the language of the grammar):

- Consider *all possible* substrings of the given string of length 1, and determine all non-terminals which can generate them.

- Now, consider *all possible* substrings of the given string of length 2, and determine all pairs of non-terminals in juxtaposition which can generate them.
- Repeat this for strings of lengths 3, 4, ..., until the full length of the string has been examined.

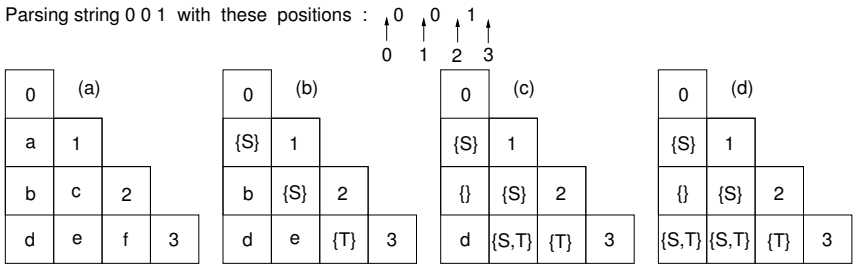


Fig. 14.7. Steps of the CKY parsing algorithm on input 001

To capture all this information, we choose a convenient tabular representation as in Figure 14.7(a). The given string `001` has *four* positions (marked `0` through `3`) in it. Position `a` in the table represents the portion of the string between positions 0 and 1, i.e., the first "`0`" in the string. Likewise, positions `c` and `f` represent `0` and `1`, respectively. Let us fill these positions with the set of all non-terminals that can generate these strings. We know that S can generate a 0, and nothing else. Therefore, the *set* of non-terminals that generates 0 happens to be $\{S\}$. Likewise, $\{T\}$ is the set of non-terminals that generate a 1. Filling the table with these, we obtain Figure 14.7(b).

What can we say about position `b` in this table? It represents the region in the string between positions `0` and `2`. *Which non-terminal can generate the region of the string between positions* `0` *and* `2`? The answer depends on which non-terminals generate the region of the string between positions `0` and `1`, and which non-terminals generate the region of the string between positions `1` and `2`. We know these to be $\{S\}$ and $\{S\}$. The set of non-terminals that generate the substring `02` are then *those non-terminals that yield* SS. Since *no* non-terminals yield SS, we fill position `b` with $\{\}$. By a similar reasoning, we fill position `e` with $\{S, T\}$. The table now becomes as shown in Figure 14.7(c).

Finally, position `d` remains to be filled. Substring `03` can be generated in two distinct ways:

- Concatenating substring `01` and `13`, or

- Concatenating substring `02` and `23`.

Substring `01` is generated by `{S}` and substring `13` by `{S,T}`. Therefore, substring `03` is generated by all the non-terminals that generate `{S}{S,T}`, i.e., those that generate `{SS,ST`, i.e., `{S,T}`. *No* non-terminal generates substring `02`, hence we don't pursue that possibility anymore. Hence, we fill position `d` with `{S,T}` as in Figure 14.7(d).

The parsing succeeds because we managed to write an `S` in position `d`—the *start* symbol can indeed yield the substring `03`.

14.6 Closure and Decidability

In this section, we catalog the main results you should remember, plus some justifications. Details are omitted for now.

1. Given a CFG, it *is* decidable whether its language is empty. Basically, if you find that `S` is not generating, the language of the grammar is empty! It is the bottom-up marking algorithm discussed above.
2. Given a CFG, it is *not decidable* whether its language is Σ^*.
3. The equivalence between two CFGs is not decidable. This follows from the previous result, because one of the CFGs could easily be encoding Σ^*.
4. Given a CFG, whether the CFG is ambiguous is not decidable.
5. Given a CFG, whether the CFG generates a *regular* language is not decidable.
6. CFLs are closed under union, concatenation, and starring because these constructs are readily available in the CFG notation.
7. CFLs are closed under reversal because we know how to "reverse a CFG."
8. CFLs are *not* closed under complementation, and hence also not closed under intersection.
9. CFLs are closed under intersection with a *regular language*. This is because we can perform the product state construction between a PDA and a DFA.
10. CFLs are closed under homomorphism.

14.7 Some Important Points Visited

We know that if L is a regular language, then L is a context-free language, but not vice versa. Therefore, the *space* of regular languages is

properly contained in the space of context-free languages. We note some facts below:

- It **does not** follow from the above that the union of two CFLs is always a *non-regular* CFL; it is *not* so, in general. Think of $\{0^n 1^n \mid n \geq 0\}$ and the complement of this language, both of which are context-free, and yet, their union is Σ^* which is context-free, *but also regular*.
- The union of a context-sensitive language and a context-free language can be a regular language. Consider the languages L_{ww} and $\overline{L_{ww}}$ of Section 13.4.1.

All this is made clear using a real-world analogy:

- In the real world, we classify music (compared to context-free languages) to be "superior" to white noise (compared to the regular language Σ^*) because music exhibits superior patterns than white noise.
- By a stretch of imagination, it is possible to regard white noise as music, but usually not vice versa.
- By the same stretch of imagination, *utter silence* (similar to the regular language $\emptyset$) can also be regarded as music.
- If we mix music and white noise in the air (they are simultaneously played), the result is white noise. This is similar to taking $\{0^n 1^n \mid n \geq 0\} \cup \Sigma^*$ which yields Σ^*.
- However, if we mix music and silence in the air, the result is still music (similar to taking $\{0^n 1^n \mid n \geq 0\} \cup \emptyset$).
- Regular languages other than $\emptyset$ and Σ^* 'sound different.' For instance, $\{(01)^n \mid n \geq 0\}$ 'sounds like' a square wave played through a speaker. Therefore, the result of taking the union of a context-free language and a regular language is either context-free or is regular, depending on whether the strings of the regular language manage to destroy the delicate patterns erected by the strings of the CFL.

It must also be clear that there are $\aleph_0$ regular languages and the same number of context-free languages, even though not all context-free languages are regular. This is similar to saying that not all natural numbers are prime numbers, and yet both have cardinality $\aleph_0$.

Illustration 14.7.1 Consider $\{a^m b^m c^m \mid m \geq 0\}$. This is not a CFL. Suppose it is a CFL. Let us derive a contradiction using the CFL Pumping Lemma. According to this lemma, there exists a number n such that given a string w in this language such that $|w| \geq n$, we can

split w into $w = uvwxy$ such that $|vx| > 0$, $|vwx| \leq n$, and for every $i \geq 0$, $uv^iwx^iy \in L(G)$.

Select the string $a^nb^nc^n$ which is in this language. These are the cases to be considered:

- v, w, and x fall exclusively in the region "a".
- v, w, and x fall exclusively in the region "b".
- v, w, and x fall exclusively in the region "c".
- v and x fall in the region "a" and "b", respectively.
- v and x fall in the region "b" and "c", respectively.

In all of these cases, "pumping" takes the string outside of the given language. Hence, the given language is not a CFL.

Illustration 14.7.2 We illustrate the CKY parsing algorithm on string `aabbab` with respect to the following grammar:

```
S -> AB | BA | SS | AC | BD
A -> a  B -> b  C -> SB  D -> SA
```

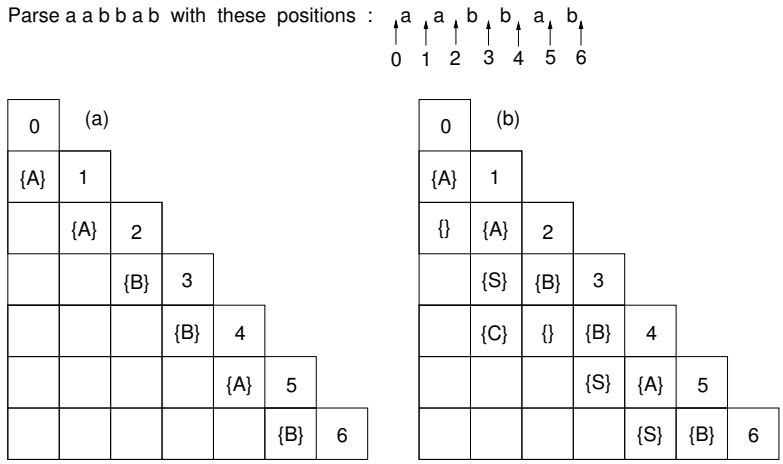


Fig. 14.8. Steps of the CKY parsing algorithm on input aabbab

The basic idea is to subdivide the string into "regions" and apply dynamic programming to "solve" all the shorter regions first, and use that information to solve the "larger" regions. Let us build our table now. The region `01` is generated by the set of non-terminals `A`. We just write `A` below. We write likewise the other non-terminals (Figure 14.8(a)).

The next table is obtained as follows: There is *no* non-terminal that has a right-hand side of the production as `AA`. So we put a $\emptyset$ (`phi`) at `02`. Since `S -> AB`, `A` marks `12`, and B marks `23`, we put an `S` at `13`. We proceed in the same manner for the remaining entries that are similar. Last but not least, we write a `C` at `14`, because `14` is understood to be representing `12-24` or `13-34`. `12-24` is `A phi` (`A` concatenated with the empty language $\emptyset$), and so we ignore it. `13-34` is `SB`, and `C -> SB`; therefore, we write "C" there. We fill the remaining table entries similarly. The results are shown in Figure 14.8(b). The parsing is successful if, in position "06", you manage to write a set of non-terminals that contain "S". Otherwise, the parsing fails.

Illustration 14.7.3 *Prove that any context-free grammar over a singleton alphabet generates a regular language.*

We provide a proof sketch, leaving details to the reader (see [45, page 86] for a full proof). To solve this problem, we can actually use the Pumping Lemma for context-free languages in an unusual way! The CFL Pumping Lemma says that for a long w (longer than some "k"), we can regard $w = uvxyz$ such that $uv^ixy^iz \in L$. Each pump up via i increases the length by $v + y$. However, since $|vxy| \leq k$, there are only a finite number of $v + y$'s we can get. These are the *periodicities* (in the ultimate periodicity sense). If a set is described by a finite number of periods $p_1, p_2, \ldots$, it is easily described by the *product* of these periods. This was the argument illustrated in Section 12.2 when we tackled a regular language Pumping Lemma problem, and chose $0^n\ 1^{n+n!}$ to be the initial string. In that problem, the $n!$ we chose served as the product of all the values possible for $|y|$. For instance, if a set S is such that

- it has strings of a's in it, and
- S is infinite, and for a sufficiently large i,
  - if $a^i \in S$ then $a^{i+4} \in S$ as well as $a^{i-4} \in S$,
  - if $a^i \in S$ then $a^{i+7} \in S$ as well as $a^{i-7} \in S$,

then S is ultimately periodic with period 28.

Therefore, we conclude that any CFL over a singleton alphabet has its strings obeying lengths that form an ultimately periodic set. This means that the language is regular.

14.7.1 Chapter Summary – Lost Venus Probe

In this chapter, we examined many topics pertaining to PDAs and CFGs: notions of acceptance, interconversion, and proofs of correctness. We also examined simple parsing algorithms based on the Chomsky normal form of CFGs. The theory of context-free languages is one

of the pinnacles of achievement by computer scientists. The theoretical rigor employed has truly made the difference between "winging" parsing algorithms—which are highly likely to be erroneous—versus producing highly reliable parsing algorithms that silently work inside programs. For instance, Hoare [55] cites the story of a Venus probe that was lost in the 1960's due to a FORTRAN programming error. The error was quite simple - in hindsight. Paraphrased, instead of typing a "DO loop" as

```
DO 137 I=1,1000
...
137 CONTINUE,
```

the programmer typed

```
DO 137 I=1 1000
...
137 CONTINUE.
```

The missed comma caused FORTRAN to treat the first line as the assignment statement `DO137I=11000` — meaning, an assignment to a newly introduced variable `DO137I`, the value `11000`. The DO statement essentially did not loop 1000 times as was originally intended! FORTRAN's permissiveness was quickly dispensed with when the theory of context-free languages lead the development of "Algol-like" block-structured languages.

Sarcastically viewed, progress in context-free languages has helped us leapfrog into the era of deep semantic errors in programs, as opposed to unintended simple syntactic errors that caused programs to crash. The computation engineering methods discussed in later chapters in this book do help weed out semantic errors, which are even more notoriously difficult to pin down. We hope for the day when even these errors appear to be as shallow and simpleminded as the forgotten comma.

Exercises

14.1. Argue the consistency and completeness of `S` and `W`.

14.2. The following "optimization" is proposed for the PDA of Figure 14.1: merge states q0 and q1 into a new state q01; thus, (i) q01 will now be the start state, and (ii) for any move between q0 and q1 or from q1 to itself, now there will be a q01 to q01 move. Formally argue whether this optimization is correct with respect to the language L_0; if not, write down the language now accepted by the PDA.

14.3. Argue the remaining cases of the proof in Section 14.2, namely the direct q2 to q2 traversal, and the q2 to q3 to q4 to q2 traversal.

14.4. Prove one more case not covered in Exercise 14.3; prove that the language of this PDA cannot go outside the language of regular expressions $a^*\ b^*$.

14.5. Consider the following list of languages, and answer the questions given below the list:

- $L_{a^i b^j c^k} =$

$$\{a^i b^j c^k \mid i, j, k \geq 0 \ and \ if \ odd(i) \ then \ j = k\}.$$

  In other words, if an odd number of a's are seen at first, then an equal number of j's and k's must be seen later.
- $L_{b^j c^k a^i} =$

$$\{b^j c^k a^i \mid i, j, k \geq 0 \ and \ if \ odd(i) \ then \ j = k\}.$$

- $L_{a^i b^j c^k d^l} =$

$$\{a^i b^j c^k d^l \mid i, j, k, l \geq 0 \ and \ if \ odd(i) \ then \ j = k \ else \ k = l\}.$$

- $L_{b^j c^k d^l a^i} =$

$$\{b^j c^k d^l a^i \mid i, j, k, l \geq 0 \ and \ if \ odd(i) \ then \ j = k \ else \ k = l\}.$$

1. Which of these languages are deterministic context-free?
2. Which are context-free?
3. Write the pseudocode of a parsing algorithm for the strings in this language. Express the pseudocode in a tabular notation similar to that in Figure 14.1.
4. For each language that is context-free, please design a PDA and express it in a tabular or graphical notation.
5. For each language that is context-free, please design a CFG.
6. For each of these CFGs, convert each to a PDA using the CFG to PDA conversion algorithm.

14.6. Prove using Floyd's method that the PDA of Figure 14.1 is correct.

14.7. Convert the following PDA to a CFG:

```
delta          contains
----------------------
<p,(,z0>         <q,(z0>
<q,(,(>          <q,((>
<q,),(>          <q,e>
<p,e,z0>         <r,e>
<q,e,z0>         <r,e>
```

14.8.

1. Develop a PDA for the language

$$w \mid w \in \{0,1\}^* \wedge \#_0(w) = 2 \times \#_1(w)\}$$

   In other words, w has twice as many 0's as 1's.
2. Prove this PDA correct using Floyd's method
3. Convert this PDA into a CFG
4. Simplify the CFG
5. Prove the CFG to be correct (consistent and complete)