Finite automata with stack memory

These notes discuss the computational power of FA equipped with a last-in first-out (stack) memory.

1 Pushdown automata (PDA)

We have seen that certain relatively simple languages are beyond the computational capabilities of DFA / NFA. This is a direct result of the lack of general-purpose memory in a DFA or NFA: the only memory available in such machines is the finite state space. We considered the specific non-regular language below:

\[ L = \{0^n1^n | n \in \mathbb{Z}^+\} \]

It is easy to see that the addition of a memory stack to an NFA allows the recognition of this language, \( L \). A stack is a last-in first-out (LIFO) memory, organized like a stack of cards on a table. Items are “pushed” onto the “top” of the stack; only the item at the top of the stack can be accessed directly, and must be “popped” off the stack in order to access items that were pushed onto the stack previously; attempting to pop the top off an empty stack results in an error. The stack memory machine shown in Fig. 1 starts by pushing a 0 onto the stack for each 0 seen at the input, and transitions to the accepting state in which a 0 is popped from the stack for every 1 seen at the input. The computation ends in the accepting state only if the input string is of the form \( 0^n1^m \), with \( n \geq m \). Acceptance also requires that the stack be empty at the end of the computation. This happens only if the string is of the precise form \( 0^n1^n \), for \( n \geq 1 \).

![Figure 1: PDA that accepts the language \( \{0^n1^n | n \in \mathbb{Z}^+\} \).](image-url)
1. **Formal definition of pushdown automaton (PDA)**

A *pushdown automaton (PDA)* is a tuple \( N = (Q, \Sigma, \Gamma, \delta, q_0, F) \), where:

- \( Q \) is a finite set known as the state space
- \( \Sigma \) is a finite set of symbols known as the input alphabet
- \( \Gamma \) is a finite set of symbols known as the stack alphabet
- \( \delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow P(Q \times (\Gamma \cup \{\epsilon\})) \) is the state/stack transition function. Each element of \( \delta(q, a, b) \), where \( q \) is a state, \( a \) is an input symbol or \( \epsilon \), and \( b \) is a stack symbol or \( \epsilon \), is a pair \((q', b')\) that describes an allowable transition from state \( q \) to state \( q' \) in which \( b \) is popped from the stack and \( b' \) is then pushed onto the stack. The empty string \( \epsilon \) in the input slot allows nondeterminism, while \( \epsilon \) in the stack slot is used to define single push and pop operations.
- \( q_0 \) is an element of \( Q \) known as the start state
- \( F \) is a subset of \( Q \) known as the set of accepting (or final) states

1.2 **Computation in a PDA**

In PDA, information about a computation is contained in the state and the stack. We represent the stack contents as a string of the form \( cs \), where \( c \in \Gamma \cup \{\epsilon\} \) is the stack top and \( s \in \Gamma^* \) contains the remaining stack items underneath the stack top. A successful computation of \( N \) on an input string \( w \) is defined as a pair of sequences \( r_0, \ldots, r_m \) in the state space \( Q \) and \( s_0, \ldots, s_m \) in the set of possible stack contents \( \Gamma^* \), such that, for some split of \( w \) as a concatenation \( w_1 \cdots w_m \) of elements of \( \Sigma \cup \{\epsilon\} \), the following conditions hold:

1. \( r_0 = q_0 \) and \( s_0 = \epsilon \)
2. \( s_k = bs \), where \( s_{k-1} = as \) and \((r_k, b) \in \delta(r_{k-1}, w_k, a)\), for each \( k \) with \( 1 \leq k \leq m \)

The first condition states that the computation begins in the start state with an empty stack. The second condition states that each step of the computation is consistent with the transition function. A computation that only succeeds in reading a portion of the input string is said to *fail*.

**Example 1.1.** The following are valid computations of the PDA in Fig. 2 on input 0110. We indicate three items in each step of the computation: the portion of the string that has not yet been read, the state, and the stack contents as a string, with the top at the left end of the stack string.

- \((0110, q_0, \epsilon), (0110, q_1, \epsilon)\) (fails)
- \((0110, q_0, \epsilon), (110, q_0, 0), (110, q_1, 0), (10, q_1, \epsilon)\) (fails)
CS243, Prof. Alvarez

2 PDA AND CONTEXT-FREE GRAMMARS

- (0110, \( q_0 \), 0), (10, \( q_0 \), 10), (10, \( q_1 \), 10), (0, \( q_1 \), 0), (\( \epsilon \), \( q_1 \), \( \epsilon \)) (succeeds)
- (0110, \( q_0 \), \( \epsilon \)), (0, \( q_0 \), 10), (0, \( q_1 \), 110), (0, \( q_0 \), 0110), (\( \epsilon \), \( q_1 \), 10) (fails)
- (0110, \( q_0 \), \( \epsilon \)), (110, \( q_0 \), 0), (10, \( q_0 \), 10), (0, \( q_0 \), 110), (\( \epsilon \), \( q_0 \), \( \epsilon \)) (fails)

Figure 2: Sample nondeterministic PDA. \( a \) represents both 0 and 1.

1.3 Language recognized by a PDA

A PDA \( N \) accepts a string \( w \) over the input alphabet of \( N \) if, and only if, there is a computation of \( N \) on input \( w \) that succeeds in processing all of \( w \) and ends in an accepting state of \( N \) with an empty stack. The language \( L(N) \) recognized by \( N \) is the set of all strings that are accepted by \( N \).

Example 1.2. The PDA in Fig. 2 recognizes the set of all even-length palindromes over \( \{0,1\} \). How would you modify this machine so that it recognizes the set of all palindromes over that alphabet?

2 PDA and context-free grammars

We introduced context-free grammars (CFG) when first discussing sets, relations, and languages. Recall that a CFG is a tuple \( G = (V, \Sigma, R, S) \), where \( V \) is a finite set of variables (also known as nonterminal symbols), \( \Sigma \) is a finite alphabet of terminal symbols, \( R \subseteq V \times (V \cup \Sigma)^* \) is a set of transformation rules, and \( S \in V \) is the start symbol. The language \( L(G) \) generated by a CFG \( G \) is the set of all terminal strings in \( \Sigma^* \) that can be derived from the start symbol of \( G \) by a finite number of applications of the transformation rules of \( G \).

Example 2.1. The language \( L = \{0^n1^n \mid n \in \mathbb{Z}^+\} \) can be generated by the CFG \( G \) that has the two rules shown below.

\[ S \rightarrow 0S1 | \epsilon \]

Example 2.2. The language of even-length palindromes over \( \{0,1\} \) can be generated by the CFG \( G \) that has the three rules shown below.

\[ S \rightarrow 0S0 | 1S1 | \epsilon \]
Example 2.3. The language of Java programs is generated by a context-free grammar. A very small portion of this CFG that corresponds to certain statements is shown below.¹ The notation used here is known as Backus-Naur form (BNF). This notation is very widely used to specify the syntax of programming languages.

\[
<\text{statement}> ::= <\text{statement without trailing substatement}>
| <\text{labeled statement}> | <\text{if then statement}> | <\text{if then else statement}>
| <\text{while statement}> | <\text{for statement}>
\]

\[
<\text{if then statement}>::= \text{if ( } <\text{expression}> \text{ ) } <\text{statement}>
\]

\[
<\text{while statement}>::= \text{while ( } <\text{expression}> \text{ ) } <\text{statement}>
\]

Notice that the context-free languages of the first two examples above are recognized by the PDAs shown in Fig. 1 and Fig. 2, respectively. There is a close relationship between PDA and CFG due to their shared recursive features. In a CFG, recursion occurs through repeated substitution of variables during a derivation. In a PDA, the stack provides a concrete way to implement recursive computation. We will explore this connection more closely. As it turns out, there is a PDA that recognizes the language of valid Java programs, too!

2.1 Implementing a CFG in a PDA

Theorem 2.1. If \( G \) is a CFG, there is a PDA that recognizes the language generated by \( G \).

Proof. Let \( G = (V, \Sigma, R, S) \) be a CFG. We describe a PDA \( N = (Q, \Sigma, \Gamma, \delta, q_0, F) \) that simulates the derivations of \( G \) and that recognizes \( L(G) \). To define this PDA, consider how a derivation of \( G \) proceeds. Initially, only the start symbol \( S \) is derived. At each subsequent stage of the derivation, a string \( x \) of nonterminals and terminals has been derived; if there are no variables in \( w \), then the derivation is complete; if there are variables in \( x \), one of these, call it \( A \), is selected, so that \( x = uAv \), and a rule of the form \( A \rightarrow z \) is applied, yielding the string \( uzv \), in which \( A \) has been replaced by \( z \). The replacement of variables via substitution rules continues recursively.

We are interested in constructing the PDA \( N \) so that it will accept strings that can be derived in the CFG \( G \). This can be done by simulating the derivation of an input string \( w \) and matching the derived terminal symbols to those in \( w \). We can carry out the recursive processing needed in the derivation by keeping the partially derived string \( x \) on the stack during the derivation.

With the above in mind, we can describe the desired operation of the PDA \( N \) as follows. Initially, \( N \) pushes the start symbol \( S \) onto the stack. \( N \) then repeatedly examines the top of the stack. If a terminal symbol occurs there, \( N \) attempts to match it to the next symbol of the input string; the terminal symbol is popped from the stack if a match succeeds. If there is a variable \( A \) at the top of the stack, \( N \) matches it to the left-hand side of a rule \( A \rightarrow w \) of \( G \), then pops \( A \) from the stack and replaces it by \( w \). This process continues until the stack is empty.

¹from http://homepage.cs.uiowa.edu/~fleck/JavaBNF.htm
Formally, we define the PDA $N$ as follows. Multiple push operations will be allowed in a single transition, in order to simplify the description. All such sequences of push operations can be easily implemented using standard single-push transitions, by adding states to the machine.

- The state space $Q$ consists of a two states: the start state $q_0$, and the accepting state $q_f$. Acceptance will also require that the computation end in $q_f$ with an empty stack after processing the entire input string.

- The input alphabet of $N$ is the same set $\Sigma$ as in $G$.

- The stack alphabet of $N$ is the set $V \cup \Sigma$ of all variables and terminals of $G$.

- There is an $\epsilon$ transition from $q_0$ to $q_f$ in $N$ that pushes $S$ onto the stack. All remaining transitions are self-loops at $q_f$. There are two types of transitions at $q_f$:
  - For each terminal symbol $a \in \Sigma$, there is a transition $a, a \rightarrow \epsilon$ that pops $a$ from the stack if $a$ occurs at the input.
  - For each rule $A \rightarrow w$ of $G$, there is a transition $\epsilon, A \rightarrow w$ that replaces $A$ by $w$ at the top of the stack without reading any input. Note that $w$ should be pushed onto the stack in such a way that the left end of $w$ is at the very top. For example, for the rule $S \rightarrow 0S1$, one would first push the 1, then the $S$, then the 0, so that the 0 ends up at the very top of the stack.

With the above construction of the PDA $N$ from the CFG $G$, it is now possible to prove by induction that a string $w \in \Sigma^*$ can be derived in $G$ if, and only if, $w$ is accepted by $N$.

**Example 2.4.** The simulating PDA for the CFG from Example 2.1 is shown in Fig. 3.

Recall that the CFG here is $S \rightarrow 0S1 \mid \epsilon$. Consider the derivation $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$ in this CFG. The corresponding accepting computation in the PDA proceeds as follows, where we indicate the unread portion of the input string, the state, and the stack contents:
at each stage (with the top of the stack at the left end of the stack).

\[0011, q_0, \epsilon, 0011, q_f, S, 0011, q_f, 0S1, 011, q_f, S1, 011, q_f, 0S11, 11, q_f, S11, 11, q_f, 11, 1, q_f, 1, \epsilon, q_f, \epsilon\]

### 2.2 Capturing PDA computation within a CFG

Theorem 2.1 shows that PDA are at least as powerful as CFG in terms of handling language complexity. It turns out that the two are actually equivalent.

**Theorem 2.2.** If \(N\) is a PDA, there is a CFG that generates the language recognized by \(N\).

**Proof.** Let \(N = (Q, \Sigma, \Gamma, \delta, q_0, F)\) be a PDA. We will construct a CFG \(G = (V, \Sigma, R, S)\) such that \(L(G) = L(N)\). In other words, the strings generated by \(G\) will be precisely the strings accepted by \(N\). Note that the terminal alphabet of the CFG \(G\) is just the input alphabet of the PDA \(N\).

The basic idea of the construction is for the CFG to describe the strings that can drive the PDA from a given state to another without changing the contents of the stack. The CFG will include a variable \(V_{p,q}\) for every pair of states \((p,q) \in Q \times Q\). That variable will generate all strings in \(\Sigma^*\) that can drive \(N\) from state \(p\) with an empty stack to state \(q\) with an empty stack. The start symbol \(S\) of the CFG will, accordingly, be chosen to be \(V_{q_0,q_f}\), where \(q_0\) is the start state and \(q_f\) is the accepting state (which can be assumed to be unique, without any loss of generality).

The preceding paragraph describes all formal elements of the CFG \(G\) except for the set of rules, \(R\). We define this set as consisting of the rules described below.

- For each \(q \in Q\), the rule \(V_{q,q} \rightarrow \epsilon\)
- For each triple of states \(p,q,r\), the rule \(V_{p,r} \rightarrow V_{p,q}V_{q,r}\)
- For each pair of matching transitions in \(N\) of the form \(a,\epsilon \rightarrow t\) from state \(p\) to state \(q\) and \(b,t \rightarrow \epsilon\) from state \(r\) to state \(s\) (where the symbols pushed in the first and popped in the second are the same), the rule \(V_{p,s} \rightarrow aV_{q,r}b\)

The final rule type includes rules \(V_{p,q} \rightarrow a\), if the transition from state \(p\) to state \(q\) on input \(a\) does not change the stack (i.e., for transitions of the form \(a,\epsilon \rightarrow \epsilon\)).

We claim that the CFG \(G\) constructed as described above generates the language recognized by the PDA \(N\). This will follow from Lemma 2.3 below.
Lemma 2.3. The construction of the CFG $G$ in Theorem 2.1 has the property that, for all states $p, q$, a string $w \in \Sigma^*$ drives $N$ from state $p$ with empty stack to state $q$ with empty stack if, and only if, the variable $V_{p,q}$ generates $w$.

Proof. We will prove the Lemma by induction in the length of $w$.

- (Basis) If $w = \epsilon$, then $w$ can drive $N$ from $p$ to $q$ without changing the stack only if $p = q$, or else if there is an $\epsilon$ transition in $N$ from $p$ to $q$ of the form $\epsilon, \epsilon \rightarrow \epsilon$. We know that the rule $V_{p,p} \rightarrow \epsilon$ is in $G$, which addresses the former case. For the latter case, take $q = r = s$ in the third rule type to see that the rule $V_{p,q} \rightarrow V_{q,q}$ must be in $G$; since the rule $V_{q,q} \rightarrow \epsilon$ is also in $G$, this shows that $w$ is generated by $V_{p,q}$.

- (Inductive step) Assume, for some natural number $n$, that all strings of length $n$ or less that can drive $N$ between any given pair of states $p, q$ while keeping the stack empty can be generated by the variable $V_{p,q}$ that corresponds to this pair of states. Pick two states $p$ and $q$, and assume that $w$ is a string of length $n + 1$ that can drive $N$ from $p$ with empty stack to $q$ with empty stack. We consider two cases:

1. If $w$ can be split as the concatenation of two nonempty strings $w = uv$ in such a way that $u$ drives $N$ from $p$ with empty stack to an intermediate state $r$ with empty stack, and $v$ drives $N$ from $r$ with empty stack to $q$ with empty stack, then by the induction hypothesis, $V_{p,r}$ generates $u$ and $V_{r,q}$ generates $v$. In this case, since the rule $V_{p,q} \rightarrow V_{p,r}V_{r,q}$ is in $G$, we see that $V_{p,q}$ generates $w$.

2. If $w$ cannot be split as in the preceding case, then the stack will be nonempty during all intermediate stages of the computation of $M$ on input $w$ from $p$ with empty stack to $q$ with empty stack. Therefore, this computation must start with a push move of the form $a, \epsilon \rightarrow t$ from state $p$ to some state $r$, and end with a matching pop move of the same symbol, of the form $b, t \rightarrow \epsilon$ from some state $s$ to state $q$. Since the rule $V_{p,q} \rightarrow aV_{r,s}b$ will then be in $G$, and since the portion of $w$ that excludes its first and last symbols would drive $N$ from state $r$ with empty stack to state $s$ with empty stack, implying by the induction hypothesis that $V_{r,s}$ generates that portion of $w$, it follows that $w$ can be derived from $V_{p,q}$ in $G$.

This completes the proof that each $V_{p,q}$ generates all strings that can drive $N$ from state $p$ with empty stack to state $q$ with empty stack. In particular, the start symbol $V_{q_0,q_f}$ generates all strings that can drive $N$ from state $q_0$ with empty stack to $q_f$ with empty stack, which are precisely the strings accepted by $N$.

This completes the proof of Lemma 2.3 and therefore also of Theorem 2.2.

Example 2.5. For the PDA in Fig. 2, Theorem 2.2 produces the CFG below, where $a$ represents any symbol of the alphabet, $\Sigma$. The symbol $S$ is used instead of $V_{q_0,q_1}$, while $A$ represents $V_{q_0,q_0}$ and $B$ represents $V_{q_1,q_1}$. The variable $V_{q_1,q_0}$ is not used.
The last four rules are not needed, as they do not yield any useful strings other than $S$. Therefore, the CFG reduces to $S \rightarrow aSa | \epsilon$.

**References**


### 3 Exercises

1. Consider the three-symbol push transition $\epsilon, S \rightarrow 0S1$ of the type used to simulate a CFG in Theorem 2.1. Assume that this transition occurs as a self-loop at state $q_0$. Show how to implement this transition by using standard single-symbol push transitions. Include the original and new state transition diagrams. Explain.