ECS 120 Lesson 12 – Pushdown Automata, Pt. 1

Oliver Kreylos

Wednesday, April 25th, 2001

Today we are going to look at a generalization of (nondeterministic) finite state machines that is capable of recognizing context-free languages. To recall, the major limitation of finite automata is their lack of memory: The only way an FSM can store information about its input is by moving from state to state, which led us to stating the Pumping Lemma, the major tool for proving that a given language is not regular.

1 Pushdown Automata (PDAs)

To overcome the limitation of finite automata, we enhance their computing power by providing them with an unlimited amount of memory, accessible in the form of a *stack*, see Figure 1. A stack consists of an unlimited number of cells, where each cell can hold one symbol from some alphabet Γ, the *stack alphabet*. As opposed to normal computer memory, which can be accessed *randomly*, meaning that cells can be read/written in any order, a stack only allows access to its top element at any time, by the following two operations:

- A new character $\gamma \in \Gamma$ can be *pushed* onto the stack, growing its size by one. The new top element of the stack is the pushed character γ.
- If the stack is not empty, the top element of the stack can be *popped* off the stack, reducing its size by one. The new top element of the stack is the element underneath the just popped element.

By this definition, a stack is a *last-in first-out* (LIFO) data structure: Elements are retrieved from the stack in the opposite order they were pushed onto it.

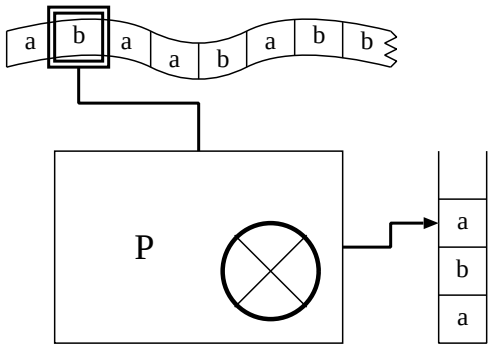

Figure 1: The structure of a pushdown automaton P, consisting of the input tape, the finite state control, the accept signal and the stack. As opposed to the input tape, which can be read only once from left to right, the stack can be modified by push/pop operations during computation.

The finite state control of a PDA is very similar to a nondeterministic finite automaton in that it has a finite set of states, a start state, a set of final states and a transition function. The main difference is, that the PDA can base its decision which state to go to next not only on the next input character, but also on the symbol currently on the top of the stack. In each step of computation, a PDA does all of the following in order:

1. Either ignore the input word, or read a character and include it in the decision to which state to go next,

2. either ignore the current stack contents, or pop off the current top character and include it in the decision to which state to go next, and then

3. either leave the current stack alone, or push a new character onto it.

1.1 Formal Definition of a Stack

Just as a sidenote, a stack is formally defined as an *abstract data type* in the following way: Let Γ be an alphabet. Then Stack(Γ), the class of stacks over the alphabet Γ, is defined as follows:

1. The *empty stack* $s_0 \in \text{Stack}(\Gamma)$ is a stack over Γ.

2. If $s \in \text{Stack}(\Gamma)$ is a stack, then the result of *pushing* a character $\gamma \in \Gamma$ onto s, $\text{Push}(s, \gamma) \in \text{Stack}(\Gamma)$ is a stack over Γ that is not the empty stack: $\text{Push}(s, \gamma) \neq s_0$.

3. If $s \in \text{Stack}(\Gamma) \setminus \{s_0\}$ is a non-empty stack, in other words, $s = \text{Push}(s', \gamma)$ is the result of pushing some character γ onto some other stack $s' \in \text{Stack}(\Gamma)$, then the result of *popping* an element off s is the pair consisting of the stack s' and the pushed character γ: $\text{Pop}(s) = \text{Pop}\big(\text{Push}(s', \gamma)\big) = (s', \gamma) \in \text{Stack}(\Gamma) \times \Gamma$.

2 Importance of Pushdown Automata

We will see later that the class of languages that pushdown automata can accept are exactly the context-free languages. But PDAs are not only a theoretical tool to reason about context-free languages, they are also important in practical applications. The main use of PDAs is in parsers for context-free languages. We have already discussed that a parser is that part of a compiler that analyzes the syntactic structure of a source program. In many cases, the syntax of a programming language will be given as a context-free grammar, and parsing an input word using the grammar directly is possible but prohibitively time-consuming. The naïve algorithm to test all possible derivations from the start symbol takes exponential time to run; whereas the more sophisticated algorithm based on a grammars in Chomsky Normal Form still takes cubic time. Both of these algorithms are too slow to be practical. PDAs, however, make it possible to parse a large subset (not all) of context-free languages in linear time. Therefore, most parser generation utilities, including yacc, will first construct a PDA equivalent to the given grammar (using an algorithm we will discuss later), and then generate `C` code that simulates running that PDA.

3 A PDA for the Language $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$

To write down the definition for a PDA, we will use a transition diagram very similar to that for finite state machines. We will write a bubble for every state, and an arrow for every possible transition. Since a PDA performs three operations for every transition (see above), we will label the transition arrows in the following way, see Figure 2: Let $a \in \Sigma_\epsilon$ be an input character or ϵ,

and let $\alpha, \beta \in \Gamma_\epsilon$ be two stack characters or ϵ. Then a transition label $\xrightarrow{a,\alpha\to\beta}$ will mean that the transition can only be taken when the next input symbol is equal to a, and the symbol on top of the stack is equal to α. When taking this transition, α will be popped off the stack, and β will be pushed onto the stack afterwards. If $a = \epsilon$, no input character will be read; if $\alpha = \epsilon$, the stack will be ignored and no symbol will be popped. If $\beta = \epsilon$, no character will be pushed onto the stack.

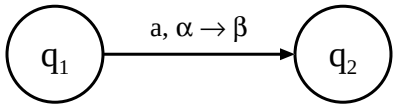

Figure 2: Labeling a transition arrow from state q_1 to q_2. The PDA reads input character $a \in \Sigma$ (or ϵ), pops the stack character $\alpha \in \Gamma$ (or epsilon), and upon taking the transition, pushes stack character $\beta \in \Gamma$ (or ϵ).

To build an automaton that accepts the language L, we use the following algorithm:

1. In the beginning, we push a special symbol \$ onto the stack, the *bottom-of-stack marker*. The reason for this is, that a PDA cannot directly find out if the current stack is empty or not. Using a marker, the PDA knows that the stack is "practically empty" whenever the bottom-of-stack marker is the symbol on top of the stack. The marker will be pushed only once, at the very beginning, and will be popped only once, at the very end of computation.

2. The automaton reads all zeros in the input word, and for every zero it reads, it pushes a zero onto its stack. This way, the PDA can "count" the number of zeros it sees.

3. As soon as the automaton sees a one in the input, it starts popping off one zero for every one it sees, effectively matching zeros and ones. If the stack becomes "empty" (\$ becomes the top symbol) exactly at the end of input, the PDA will have seen exactly as many zeros as ones and will accept the input word by transitioning to a final state.

The transition diagram implementing this algorithm is shown in Figure 3. Similar to an NFA, a PDA's computation will "die" if there is no transition arrow matching the next input character and the top stack symbol. On the

other hand, the computation will branch whenever there are multiple arrows with matching labels.

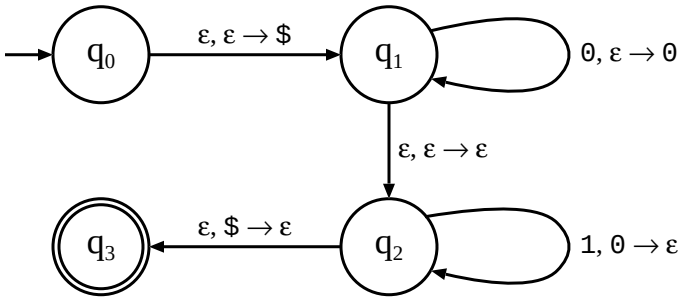

Figure 3: The transition diagram for a PDA accepting the language $L = \left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$.

In Fig. 3, the computation will branch on every step once the machine is in state q_1 – the transition labeled $\epsilon, \epsilon \to \epsilon$ ignores the input word and the stack and can be taken at any time. But as long as there are zeros in the input, any branch of computation that moved on to state q_2 will immediately die, because there are no matching arrows for that situation in state q_2.

Also, though the empty word is part of the language L, it is not necessary to designate the start state q_0 as a final state: Before reading any input, the PDA can transition from q_0 to q_1, pushing `$` onto the stack; then it can transition to q_2 withouth affecting the stack in any way; finally, it can transition to q_3 while popping `$` off the stack. If the input word is empty, the machine will stay in the final state q_3 and accept.

4 Formal Definition of Pushdown Automata

A pushdown automaton P is a six-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $\delta\colon Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function,
- $q_0 \in Q$ is the start state, and

- $F \subset Q$ is the set of final states.

5 Definition of Computation of PDAs

To understand exactly how PDAs work, we have to formalize our intuitive understanding of how a PDA transitions from state to state following its transition function δ. For finite automata, we observed that an automaton's current state is a complete "snapshot" of its computation. Knowing a current state, the transition function δ and the unread rest w of the input word, we could construct the set of all states the automaton can possibly end up in after completing reading its input by defining the extended transition function $\delta^*(q, w)$.

For PDAs, the situation is more complicated. The current state is not enough to determine how the computation will proceed, because a PDA can base its decision where to transition to on the top stack symbol as well. Even symbols not currently on top of the stack must be regarded, since they could become top symbols in later computation steps. Therefore, the complete stack content must be a part of the "computation snapshot." For convenience, we will include the current unread rest of the input word into the snapshot as well.

5.1 Representing Stacks

To encode the complete content of the current stack, we observe that a stack always contains any finite number of stack symbols from the stack alphabet Γ. If we write these stack symbols from top to bottom, they form a word $\gamma \in \Gamma^*$ over the alphabet Γ. We will from now on identify stacks with the words specifying their contents, and define the stack operations in the following way:

- The empty word $\epsilon \in \Gamma^*$ represents the empty stack.
- If $\gamma \in \Gamma^*$ represents a stack, and $\alpha \in \Gamma$ is a stack symbol, then the result of pushing α onto the stack is represented by $\alpha\gamma$.
- If $\gamma \in \Gamma^* \setminus \{\epsilon\}$ represents a non-empty stack, then there exists a word $\gamma' \in \Gamma^*$ and a stack symbol $\alpha \in \Gamma$ such that $\gamma = \alpha\gamma'$. In other words, γ represents the result of pushing α onto γ'. In this case, the

result of popping the top symbol off γ is represented by γ', and the character popped off is α.

5.2 Instantaneous Descriptions

The above observations led us to define an *instantaneous description* (ID), a snapshot of a PDA's computation, as a triple $(q, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*$, where q is the current state, w is the current unread rest of the input word, and γ represents the current stack.

To continue our definition, we will now investigate how a PDA's computation proceeds from one instantaneous description to the next. Instead of defining computation as a chain of states (as we did for NFAs), we will define computation as a chain of instantaneous descriptions.

5.3 The Turnstile Relation

If $d_1, d_2 \in Q \times \Sigma^* \times \Gamma^*$ are two instantaneous descriptions, we say that they are related by the turnstile relation, in symbols $d_1 \vdash d_2$, if and only if the PDA can go from d_1 to d_2 in exactly one step of computation. Formally, let $a \in \Sigma_\epsilon$ be an input character or ϵ, let $w \in \Sigma^*$ be a word such that the current unread input word is aw, let $\alpha, \beta \in \Gamma_\epsilon$ each be a stack symbol or ϵ, and let $\gamma \in \Gamma^*$ be a word such that the current stack is $\alpha\gamma$. Then,

$$(q_1, aw, \alpha\gamma) \vdash (q_2, w, \beta\gamma) \quad \text{if and only if} \quad (q_2, \beta) \in \delta(q_1, a, \alpha)$$

In other words, two IDs are related by the turnstile relation, iff there is a transition arrow from state q_1 to state q_2 that matches the next input character and the top stack symbol, and that pushes stack symbol β onto the stack.

We will now recursively define the extended turnstile relation $\vdash^*$:

1. For any ID $d \in Q \times \Sigma^* \times \Gamma^*$, $d \vdash^* d$.
2. For any IDs $d_1, d_2 \in Q \times \Sigma^* \times \Gamma^*$, $d_1 \vdash^* d_2$ if and only if there exists an ID $d \in Q \times \Sigma^* \times \Gamma^*$ such that $d_1 \vdash d$ and $d \vdash^* d_2$.

Using $\vdash^*$, we can finally define computation of a PDA: A word $w \in \Sigma^*$ is accepted by a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, if and only if $(q_0, w, \epsilon) \vdash^* (q, \epsilon, \gamma)$, where $q \in F$ is a final state and $\gamma \in \Gamma^*$ is any word of stack symbols. In other

words, a word is accepted if there exists a chain of instantaneous descriptions that starts with the machine being in the start state, not having read any input symbols and having an empty stack, and that ends with the machine being in a final state when having read all characters from the input word. We can now define the language accepted by a PDA:

$$L(P) := \left\{ w \in \Sigma^* \;\middle|\; \exists q \in F, \gamma \in \Gamma^* : (q_0, w, \epsilon) \vdash^* (q, \epsilon, \gamma) \right\}$$

When reasoning about pushdown automata and how they compute, the following three properties of the extended turnstile relation are sometimes helpful:

1. If a sequence of IDs (*computation*) is legal for a PDA P, then the computation formed by adding the same additional input string to the end of the input in each ID is also legal. Formally: If $(q_1, w_1, \gamma_1) \vdash^* (q_2, w_2, \gamma_2)$, then for all $x \in \Sigma^* : (q_1, w_1 x, \gamma_1) \vdash^* (q_2, w_2 x, \gamma_2)$.

2. If a computation is legal for a PDA P, and some tail of the input is not consumed during that computation, that tail can be removed from the input word in each ID, and the resulting computation will still be legal. Formally: If $(q_1, w_1 x, \gamma) \vdash^* (q_2, w_2 x, \gamma)$, then $(q_1, w_1, \gamma) \vdash^* (q_2, w_2, \gamma)$

3. If a computation is legal for a PDA P, then the computation formed by adding the same additional stack symbols below the stack in each ID is also legal. Formally: If $(q_1, w_1, \gamma_1) \vdash^* (q_2, w_2, \gamma_2)$, then for all $\omega \in \Gamma^* : (q_1, w_1, \gamma_1 \omega) \vdash^* (q_2, w_2, \gamma_2 \omega)$.

Intuitively, these principles state that data a PDA never looks at cannot influence its computation. Note that properties 1 and 2 are reverses of each other, but the reverse of property 3 may not be true. There could be some things a PDA can do by popping some symbols off ω, and then pushing them back onto the stack, that it could not do without ever looking at ω. Input symbols, on the other hand, can never be restored after they are read.

5.4 Acceptance by Empty Stack

The above definition of acceptance is sometimes referred to as *acceptance by final state*, because a word is accepted when the machine ends in a final state, no matter if the stack is empty or not. Another, equivalent, definition of acceptance works the other way around: A word is accepted when the

machine ends with an empty stack, no matter whether in a final state or not (actually, machines of this type do not have final states). Formally, the definition for the language accepted by a PDA by empty stack would be:

$$L_E(P) := \left\{ w \in \Sigma^* \;\middle|\; \exists q \in Q : (q_0, w, \epsilon) \vdash^* (q, \epsilon, \epsilon) \right\}$$

For the same PDA P, the languages $L(P)$ and $N(P)$ might be different, but for every PDA P there is another PDA P', such that $L(P) = N(P')$. It is easy to construct the machine P': Whenever machine P reaches a final state, the computation branches to a new state that does nothing but empty the stack. The construction can also be reversed: To construct a machine equivalent to P', one adds a transition to a new final state to every state of P' that can be taken whenever the stack becomes empty, i. e., the bottom-of-stack marker becomes the top stack symbol.