ECS 120 Lesson 13 – Pushdown Automata, Pt. 2

Oliver Kreylos

Friday, April 27th, 2001

In the last lecture, we introduced pushdown automata as an alternative means to specify context-free languages. We still have left to prove that they are actually capable of that; in other words, we have to show that the class of languages that can be accepted by pushdown automata is exactly the class of languages that can be generated by context-free grammars. As usual, we will prove this equivalence in two directions. The first direction is a constructive proof that describes how to build a PDA that accepts the language generated by a given grammar. This construction is important in practice, because it shows how to create a parser for a given grammar. The reverse direction, building a grammar from a given PDA, is of lesser practical value.

1 Converting a Context-Free Grammar into a PDA

We have to show that every context-free language is accepted by some PDA. In other words, if L is a context-free language, i. e., there exists a context-free grammar G such that $L = L(G)$, there must exist a PDA P such that $L = L(P)$. The basic idea for the construction of P is to simulate the leftmost derivations in G, and to accept a word iff there exists a leftmost derivation that generates it. The PDA will use its stack to store and manipulate the intermediate strings in the derivation it simulates, and after all variables have been replaced by characters, the resulting string will be compared to the input string. The PDA will use its nondeterminism to "guess" which rule to apply to the leftmost variable in each step. If the input word can be generated

by the grammar, at least one of the resulting branches of computation will succeed in matching the word and the PDA will accept.

When looking at leftmost derivations $S \Rightarrow s_1 \Rightarrow \cdots \Rightarrow s_{n-1} \Rightarrow w$ in G, every intermediate string s_i can be written as $s_i = xA\alpha$, where $A \in V$ is the leftmost variable, $x \in \Sigma^*$ is a string of only characters, and $\alpha \in (V \cup \Sigma)^*$ is a string of variables and characters. Since the PDA needs to access the leftmost variable A in any derivation step, it is not possible to store the entire string s_i on the stack – a PDA can only access the topmost stack symbol. But we know that the symbols to the left of A (the word x) will never change during further derivations, and in the end they have to match the beginning characters in the input string. Therefore, it is not necessary to store the character-only prefix of s_i, but only all symbols from the first variable on. The suffix of an intermediate string including the leftmost variable, $A\alpha$, is called a *tail*. Whenever during derivation the intermediate string has a prefix of only characters, that prefix is going to be matched against input characters immediately. Using this strategy, the leftmost variable of all intermediate strings is always the topmost stack symbol, and can be accessed by the PDA.

As an example, let us consider the grammar for $\mathcal{R}\big(\{\mathtt{0},\mathtt{1}\}\big)$, the language of regular expressions over $\{\mathtt{0},\mathtt{1}\}$, and show a computation for a hypothetical PDA that can accept $\mathcal{R}\big(\{\mathtt{0},\mathtt{1}\}\big)$. As example word, we use $\mathtt{0} \cup \mathtt{1}$. We will write down the computation as a chain of IDs, where the state entries are left blank (we don't know the states yet): $(\cdot, \mathtt{0} \cup \mathtt{1}, U) \vdash^* (\cdot, \mathtt{0} \cup \mathtt{1}, U \cup C) \vdash^* (\cdot, \mathtt{0} \cup \mathtt{1}, C \cup C) \vdash^* (\cdot, \mathtt{0} \cup \mathtt{1}, S \cup C) \vdash^* (\cdot, \mathtt{0} \cup \mathtt{1}, B \cup C) \vdash^* (\cdot, \mathtt{0} \cup \mathtt{1}, \mathtt{0} \cup C) \vdash^* (\cdot, \cup\mathtt{1}, \cup C) \vdash^* (\cdot, \mathtt{1}, C) \vdash^* (\cdot, \mathtt{1}, S) \vdash^* (\cdot, \mathtt{1}, B) \vdash^* (\cdot, \mathtt{1}, \mathtt{1}) \vdash^* (\cdot, \epsilon, \epsilon)$.

The above chain does not contain the complete chain of IDs: For example, between the first two given IDs, the PDA must pop the symbol U off the stack, and push the symbols C, $\cup$ and U onto the stack, in that order. By our definition, a PDA can only push a single symbol in each step; therefore, the PDA must actually have performed three steps between the first two given IDs. It is straightforward, however, to convert a "generalized" PDA that can push multiple symbols per step into a standard one, by introducing intermediate steps pushing one symbol at a time, see Figure 1. In the following construction, we will use this fact to allow a shorthand notation with a modified transition function $\delta\colon Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma^*)$, where the automaton can push any number of symbols in each step.

We now construct the generalized PDA $P = \big(\{q_0, q_1, q_2\}, \Sigma, \Gamma, \delta, q_0, \{q_2\}\big)$ that accepts exactly the words generated by a given grammar $G = (V, \Sigma, R, S)$. P will have three states, the start state q_0, the *loop state* q_1, and the only

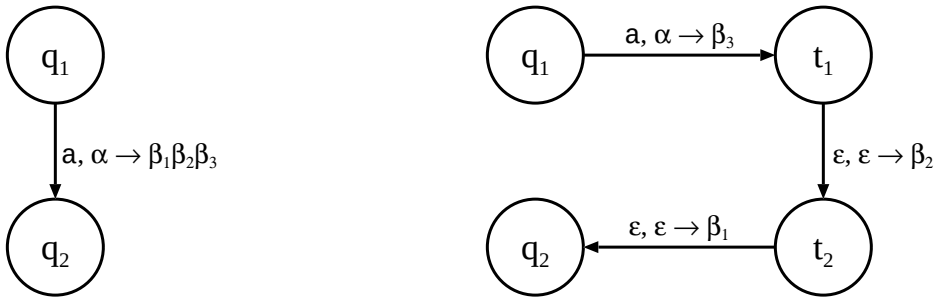

Figure 1: Converting a transition from a generalized PDA (left) to a sequence of transitions in a standard PDA (right). Note that the symbols β_1, β_2 and β_3 have to be pushed in reverse order.

final state q_2. The input alphabet Σ will be the alphabet of G, and the stack alphabet Γ will be the union $\Gamma = V \cup \Sigma \cup \{\$\}$ of G's variables and characters and the special symbol $\$$. Since the PDA has to detect when its stack is empty (this means the derivation is complete), we use the standard trick of initializing the stack with a bottom-of-stack marker $\$$. We also have to push the start symbol S onto the stack. Therefore, the transition function for the start state consists of the only transition $\delta(q_0, \epsilon, \epsilon) = \big\{(q_1, S\$)\big\}$. Note that when extending the generalized PDA to a standard PDA, the two symbols have to be pushed in reverse order: $\$$ first, then S.

The transitions from the loop state q_1 will consist of two types and a special case:

Rule transitions For every rule $A \to x \in R$, where $A \in V$ is some variable and $x \in (V \cup \Sigma)^*$ is a string of variables and characters, there is a transition back to the loop state that pops A off the stack and pushes the right-hand side x onto the stack, if A was the top stack symbol: For all $A \in V : \delta(q_1, \epsilon, A) = \big\{\, (q_1, x) \;\big|\; A \to x \in R \,\big\}$.

Matching transitions For every character $a \in \Sigma$, there is a transition back to the loop state that pops a off the stack if it matches the next input character: For all $a \in \Sigma : \delta(q_1, a, a) = \big\{(q_1, \epsilon)\big\}$.

Final transition When the stack becomes empty, i. e., P sees the bottom-of-stack marker $\$$, it transitions to the accept state: $\delta(q_1, \epsilon, \$) = \big\{(q_2, \epsilon)\big\}$. If the input word has been completely matched at that point, P will

accept; otherwise, since there are no transitions from the accept state, the computation branch will die.

We must now prove that the construction is valid, i. e., that really $L(G) = L(P)$. Again, this involves two directions. The first direction, $L(G) \subset L(P)$, is comparatively easy. Assume that $w \in L(G)$ is some word generated by G. Then there must exist a leftmost derivation $S = x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_n = w$. In the PDA's computation, the intermediate words x_i will be represented partially on the stack, partially by matching them against the input. To show that P accepts w, we have to show that $(q_1, w, S\$) \vdash^* (q_1, y_i, \alpha_i\$)$ for all $i = 1, \ldots, n$, where y_i and α_i are a representation of the intermediate string x_i, i. e., α_i is the tail of x_i, such that $x_i = p_i\alpha_i$, and y_i is the unmatched input, such that $w = p_i y_i$. We show the claim using induction on i:

Induction Basis $x_1 = S$. Thus, $p_1 = \epsilon$, $\alpha_1 = S$ and $y_1 = w$. Therefore, $(q_1, w, S\$) \vdash^* (q_1, w, S\$)$.

Induction Hypothesis Assume $(q_1, w, S\$) \vdash^* (q_1, y_i, \alpha_i\$)$ for some $i \geq 1$.

Induction Step Consider the ID $(q_1, y_i, \alpha_i\$)$. Since α_i is a tail, its first symbol must be a variable $A \in V$. Furthermore, the derivation step $x_i \Rightarrow x_{i+1}$ must replace A by one of its right-hand sides, because A is also the leftmost variable in x_i. By our construction, PDA P is able to replace A in α_i by one of its right-hand sides as well, by using a rule transition, and then to remove leading characters from α_i and y_i by using a sequence of matching transitions. As a result, P can reach the ID $(q_1, y_{i+1}, \alpha_{i+1}\$)$, which represents the next intermediate word x_{i+1}.

To finish the proof, we note that $(q_0, w, \epsilon) \vdash (q_1, w, S\$)$, that $y_n = \alpha_n = \epsilon$, and that $(q_1, \epsilon, \$) \vdash (q_2, \epsilon, \epsilon)$. Therefore, $(q_0, w, \epsilon) \vdash^* (q_2, \epsilon, \epsilon)$, which means that P accepts w.

To show the reverse direction, $L(P) \subset L(G)$, we have to show that every word accepted by P is generated by G. Actually, we need to show something more general: If P executes a sequence of transitions that has the net effect of popping a variable A from its stack, without ever going below A on the stack, then A derives the input string w that was consumed by P in the process. Precisely, if $(q_1, w, A) \vdash^* (q_1, \epsilon, \epsilon)$, then $A \Rightarrow^* w$. We prove this by induction on the number of transitions taken by P.

Induction Basis If $(q_1, w, A) \vdash^* (q_1, \epsilon, \epsilon)$ in exactly one move, that move must have been a rule transition, and must therefore have corresponded to a rule $A \to \epsilon$. In this case, $w = \epsilon$, and we know that $A \Rightarrow \epsilon$.

Induction Hypothesis Assume that $(q_1, w, A) \vdash^* (q_1, \epsilon, \epsilon) \implies A \Rightarrow^* w$, when the computation involves n steps.

Induction Step Suppose P took $n+1$ steps to effectively pop A. The first of these steps must have been a rule transition, because A is a variable. This means that A was replaced on the stack by one of its right-hand sides, say $a_1 a_2 \dots a_k$, where each $a_i \in V \cup \Sigma$ is either a character or a variable. The next n transitions must consume w, and must have the net effect of popping a_1, a_2, …, a_k off the stack, in that order. We can break w into $w = w_1 w_2 \dots w_k$, where w_i is the part of w consumed while popping a_i off the stack. This splitting of w, and its effect on the stack, is illustrated in Figure 2.

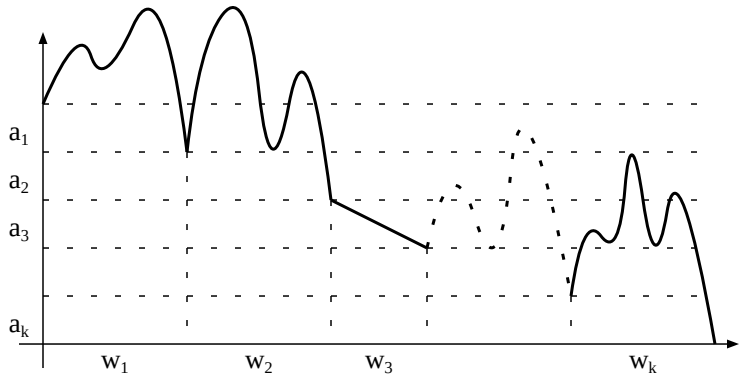

Figure 2: A string $a_1 \dots a_k$ is popped off the stack one symbol at a time, splitting the input word into parts $w_1 \dots w_k$. For illustration, we assume that a_3 is a character. The vertical axis denotes the height of the stack.

Since popping each a_i in turn is a valid computation, we can conclude that $(q_1, w_i w_{i+1} \dots w_k, a_i) \vdash^* (q_1, w_{i+1} \dots w_k, \epsilon)$ for all $i = 1, \dots, k$. Since none of these sequences can involve more than n steps (the total computation to pop $a_1 a_2 \dots a_k$ took exactly n steps), we can apply the induction hypothesis if a_i is a variable: $a_i \Rightarrow^* w_i$. If a_i is a character, then the PDA must have performed a matching transition to pop

it, resulting in removing the same character w_i from the input word. Again, $a_i \Rightarrow^* w_i$, this time involving zero derivation steps. Now we have the derivation $A \Rightarrow a_1 a_2 \dots a_k \Rightarrow^* w_1 a_2 \dots a_k \Rightarrow^* \cdots \Rightarrow^* w_1 w_2 \dots w_k$. Thus, $A \Rightarrow^* w$.

To complete the proof, we set $A = S$ and set $w = x$, where $x \in \Sigma^*$ is a word in $L(P)$. This means that $(q_0, x, \epsilon) \vdash^* (q_2, \epsilon, \epsilon)$, which implies $(q_1, x, S\$) \vdash^* (q_1, \epsilon, \$)$, which we have just proven means $S \Rightarrow^* x$, i. e., $x \in L(G)$.

2 Deterministic Pushdown Automata

The definition of PDA we have used so far is inherently nondeterministic: The transition function can leave a PDA multiple choices as to where to transition to next, and depending on these choices, it might also push/pop different sequences of characters onto/off its stack. Furthermore, we defined that a PDA accepts a word, if there exists at least one chain of instantaneous descriptions that leaves the automaton in a final state.

For finite state machines, we were able to prove that nondeterministic machines are not more powerful than their deterministic counterparts. Every NFA can be simulated by a DFA with a (possibly very much) larger set of states. The reason was, that the only real difference between NFAs and DFAs is that an NFA can be in multiple states (belonging to different branches of computation) at the same time. For PDAs, the situation is more complicated: Not only can it be in multiple states at once, it can also have multiple different stack contents at once. While the multiplicity of states could be resolved in the same way as for finite state machines, the multiplicity of stacks can not – every PDA has exactly one stack, and stacks can exactly store on sequence of stack symbols.

This reasoning hints at standard PDAs being more powerful that deterministic ones. In fact, while there are many context-free languages that can be accepted by deterministic PDAs, there are some that are inherently nondeterministic. One example is the language $\left\{ w \in \Sigma^* \;\middle|\; w \text{ is a palindrome} \right\}$, another one is $\left\{ \mathtt{0}^i \mathtt{1}^j \mathtt{0}^k \;\middle|\; i = j \vee i = k \right\}$. Intuitively, in the first language a PDA has to "guess" when it has seen the first half of the input word, and when it should start matching input characters with the characters stored on the stack. In the second language, a PDA has to guess which of the two possible cases of words it will encounter: $i = j$ or $i = k$. In the first case, it must match zeros with ones; in the second case, it must match zeros with

zeros. A deterministic PDA cannot take any guesses, so it cannot accept either of these languages.

The limited power of deterministic PDAs has a serious consequence: Since a parser for a compiler is usually implemented as a PDA, and a deterministic computer can only directly simulate deterministic PDAs, parsers as generated by the yacc utility can actually only parse a subset of context-free languages. To parse other languages, one must construct a parser in a different way. One possible way is to use the knowledge about grammars in Chomsky Normal Form: A parse tree for a word in such a grammar can be constructed without using PDAs, but it takes time proportional to the third power of the length of the input word.

3 The Fine Structure of CFL

The above examples mean that the class of context-free languages must consist of at least two subclasses: One of them being accepted by deterministic PDAs (called *D-CFL*), the other one being CFL, the class context-free languages itself. We have already seen another splitting of context-free languages: Those that have an unambiguous grammar (called *UA-CFL*), and those that are inherently ambiguous. As it turns out, any language that can be accepted by a deterministic PDA must have an unambiguous grammar; therefore, the class D-CFL is a subset of the class of not inherently ambiguous languages. Altogether, the class of context-free languages has a fine structure as shown in Figure 3.

We have seen that the classes of languages are all non-empty and properly nested:

1. $\left\{ \mathtt{0}^n\mathtt{1}^m \;\middle|\; n, m \geq 0 \right\}$ is in RL.
2. $\left\{ \mathtt{0}^n\mathtt{1}^n \;\middle|\; n \geq 0 \right\}$ is in D-CFL but not in RL.
3. $\left\{ w \in \{\mathtt{0},\mathtt{1}\}^* \;\middle|\; w \text{ is a palindrome} \right\}$ is in UA-CFL but not in D-CFL.
4. $\left\{ \mathtt{0}^i\mathtt{1}^j\mathtt{0}^k \;\middle|\; i = j \vee i = k \right\}$ is in CFL but not in UA-CFL.

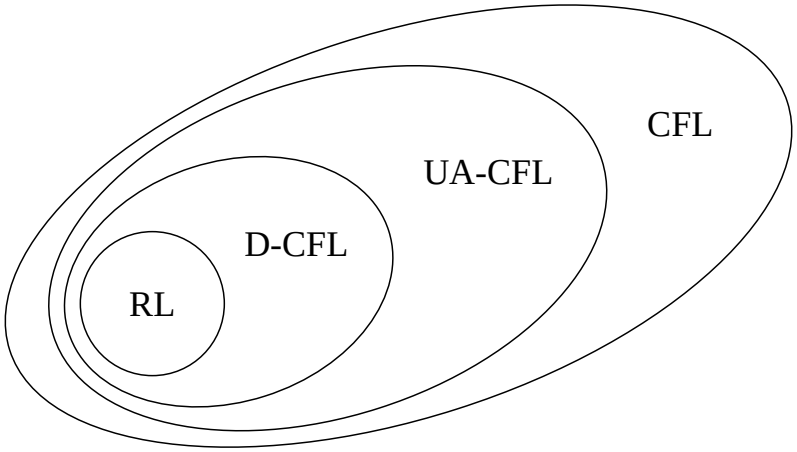

Figure 3: The fine structure of CFL, the class of context-free languages. RL are the regular languages, D-CFL is the class of languages accepted by deterministic PDAs, UA-CFL is the class of languages having at least one unambiguous grammar.