Automata Theory and Formal Grammars: Lecture 8

Pushdown Automata

Pushdown Automata

Last Time

- Chomsky Normal Form
- A Pumping Lemma for CFLs

Today

- Decision procedures for CFLS
- Pushdown Automata
- A Kleene Theorem for CFLs
- Determinism

Decision Procedures for CFGs

Decision Procedures for CFGs

Recall what a decision procedure is: an algorithm for answering a yes/no question.

A several yes/no questions involving CFGs have decision procedures.

1. Given CFG G, is $\varepsilon \in \mathcal{L}(G)$?
2. Given CFG M and $x \in \Sigma^*$, is $x \in \mathcal{L}(M)$?
3. Given FA G, is $\mathcal{L}(M) = \emptyset$?

Answering these questions is harder in the case of CFGs than FAs, but all are decidable.

Deciding Whether $\varepsilon \in \mathcal{L}(G)$

... use “nullability algorithm”!

- Let $G = \langle V, \Sigma, S, P \rangle$.
- Compute $N(G) \subseteq V$, the set of nullable variables (i.e. $A \in N(G)$ if and only if $A \Rightarrow_G^* \varepsilon$).
- Then $\varepsilon \in \mathcal{L}(G)$ if and only if $S \in N(G)$! (Why?)

Deciding Whether $x \in \mathcal{L}(G)$

What we want is an algorithm that, given a CFG G and word x, determines whether or not x can be generated from G.

Our approach will rely on Chomsky Normal Form!

- We'll generate a CNF grammar G_4 from G.
- We'll then use the special properties of CNF grammars to answer the question.

How Does CNF Help?

Consider CFG G given by: $S \longrightarrow \varepsilon \mid 0S1$. Our CNF equivalent G_4 is:

$$\begin{aligned} S &\longrightarrow X_0X_1 \mid X_0Y \\ Y &\longrightarrow SX_1 \\ X_0 &\longrightarrow 0 \\ Y_1 &\longrightarrow 1 \end{aligned}$$

To determine if $S \Rightarrow_{G_4} 001$, do we need to consider derivations beginning

$$S \Rightarrow_{G_4} X_0Y \Rightarrow_{G_4} X_0SX_1 \Rightarrow_{G_4} X_0X_0YX_1 \Rightarrow_{G_4} \cdots?$$

No! $|X_0X_0YX_1| = 4 > 3 = |001|$, and in a CNF grammar only words of length ≥ 4 can be generated from $X_0X_0YX_1$.

A Decision Procedure for $x \in \mathcal{L}(G)$

1. If $x = \varepsilon$, apply previous decision procedure.
2. Otherwise, do following.
   (a) Convert G into CNF, yielding G_4
   (b) Generate all possible derivation sequences whose final configuration has length $|x|$.
   (c) If one derivation sequence leads to x, return "true", else return "false".

Why does this work? Because for CNF grammars, only finitely many appropriate derivation sequences are possible!

Note There are much better algorithms....

Pushdown Automata

Machines for CFLs

Recall our study of regular languages.

- They were defined in terms of regular expressions (syntax).
- We then showed that FAs provide the computational power needed to process them.

We would like to mimic this line of development for CFLs.

- We have a “syntactic” definition of CFLs in terms of CFGs.
- What kind of computing power is needed to “process” (i.e. recognize) CFLs?

Do FAs suffice?

Machines for CFLs

The problem with FAs is that a given FA only has a finite amount of memory.

- States allow you to "store" information about the input seen so far.
- Finite states $=$ finite memory!

However, some CFLs require an unbounded amount of "memory"!

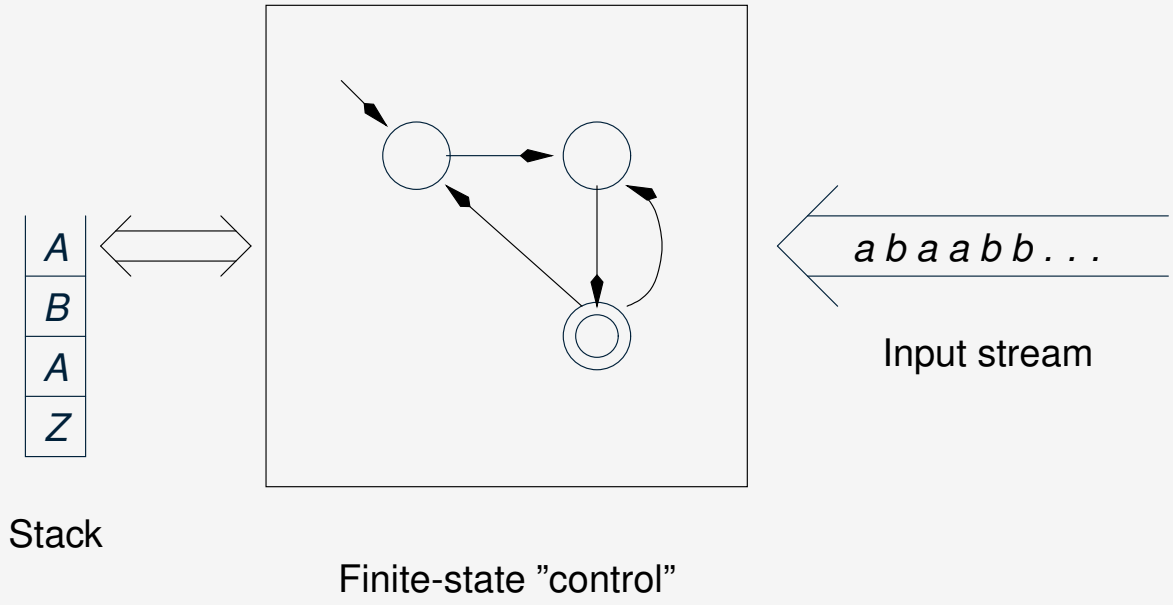
E.g. $L = \{ 0^n 1^n \mid n \geq 0 \}$. To determine if a word is in L, you need to be able to "count" arbitrarily high in order to keep track of the number of initial 0's. This implies a need for an unbounded number of bits of memory. (Why?)

Consequently, we need to have some form of "unbounded memory" in the machines for CFLs.

It turns out that an unbounded *stack*, or *pushdown*, will do!

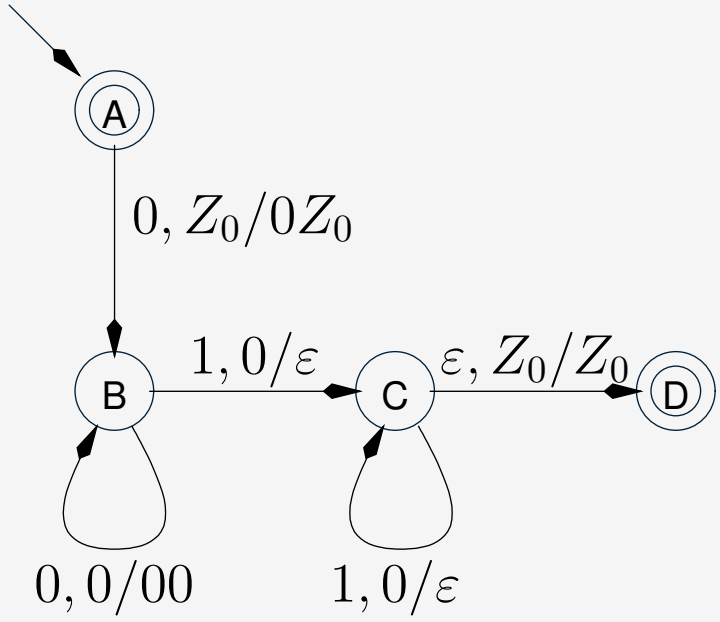
Pushdown Automata (PDAs)

... are (N)FAs with an auxiliary stack.



- State transitions can read inputs *and top stack symbol*.
- When a transition “fires”, new symbols can be *pushed onto stack*.

Example: A PDA for $\{ 0^n 1^n \mid n \geq 0 \}$



Formalizing PDAs

What does a PDA specification need to contain?

- The things we found in FAs: states, input alphabet, start state, transitions, accepting states ...
- ... plus the *stack alphabet* and *initial stack symbol*.
- Also, transitions need to be able manipulate the stack.

Defining PDAs

Definition A *pushdown automaton* (PDA) is a septuple $\langle Q, \Sigma, \Gamma, q_0, Z_0, \delta, A \rangle$ where:

- Q is a finite set of *states*;
- Σ and Γ are the *input* and *stack* alphabets, respectively;
- $q_0 \in Q$ is the *start state*;
- $Z_0 \in \Gamma$ is the *initial stack symbol*;
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow 2^{Q \times (\Gamma^*)}$ is the *transition function*; and
- $A \subseteq Q$ is the set of *accepting states*.

The Transition Function PDA $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, \delta, A \rangle$

δ has type $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow 2^{Q \times (\Gamma^*)}$.

Inputs triples $\langle q, a, X \rangle$.

- q is the *source state* of the transition.
- a can either be an input symbol (element of Σ) or ε, in which case the transition consumes no input when it fires.
- X is the symbol currently *on top of the stack*.

Outputs *sets* of pairs $\langle q', \gamma \rangle$. (Why sets? PDAs can be *nondeterministic*!).

- q' is the *target state* of the transition.
- $\gamma \in \Gamma^*$ is a sequence of symbols pushed onto the stack in place of X.

PDA Transitions (cont.)

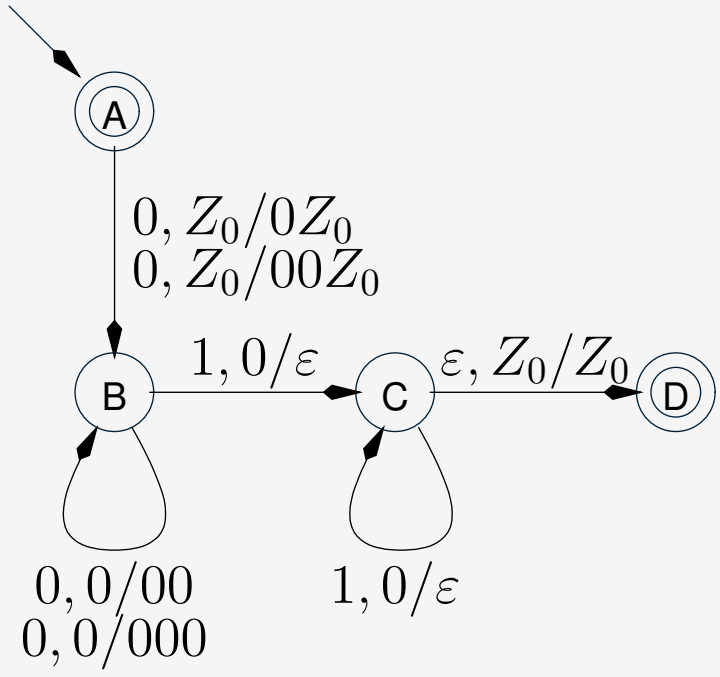
So

if $\langle q', \gamma \rangle \in \delta(q, a, X)$
and the current state is q
and $a \in \Sigma$ and the current input symbol is a
and the current top symbol on the stack is X

then the input symbol is consumed
and the state changes to q'
and X is popped from the stack, with γ then pushed.

(Case for $a = \varepsilon$ is the same, except that input stream is not disturbed.)

Example: PDA for $\{ 0^m 1^n \mid m \leq n \leq 2m \}$



What is $\delta(A, 0, Z_0)$?

What is $\delta(B, 0, Z_0)$?

What is $\delta(C, 1, 0)$?

The Language of a PDA

A PDA M should accept a word w if, starting with the initial stack, M "processes" w and winds up in an accepting state.

What do we need to keep track of to determine if this holds?

- PDA's current state
- Current stack contents
- Remaining input

If we have this information, then we can determine which transitions can "fire" and what the new stack contents and input stream are when a transition takes place!

Formalizing Acceptance in a PDA

We need to define the notions of:

- *Configuration* of a PDA (i.e. a "snapshot" of an executing PDA)
- A one-step *configuration transition relation*, $\vdash_M$

We'll then use these to define the language accepted by a PDA.

In what follows fix PDA $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, \delta, A \rangle$.

Formalizing the Language of a PDA

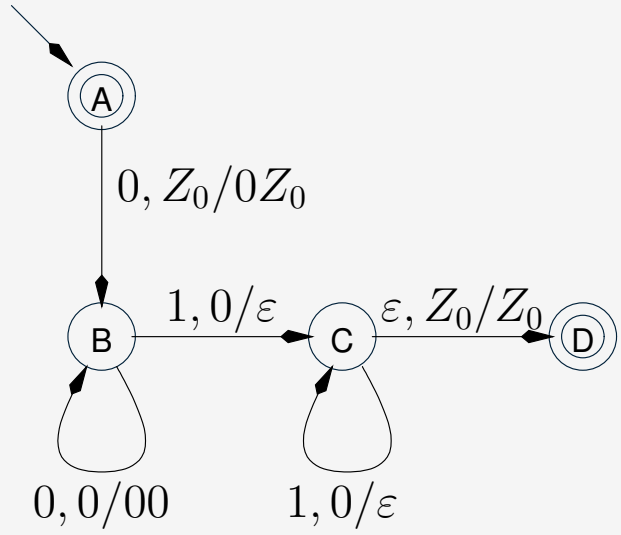
Definition A *configuration* of M is a triple $\langle q, x, \alpha \rangle \in Q \times (\Sigma^*) \times (\Gamma^*)$.

(q is current state, x is remaining input, α is stack, with top element first.)

Definition The *configuration transition relation*, $\vdash_M$, is defined by: $\langle q, x, \alpha \rangle \vdash_M \langle q', x', \alpha' \rangle$ if there exist $a \in \Sigma \cup \{\varepsilon\}$, $X \in \Gamma$, and $\beta, \gamma \in \Gamma^*$ such that:

- $\langle q', \gamma \rangle \in \delta(q, a, X)$.
- $x = a \cdot x'$
- $\alpha = X \cdot \beta$
- $\alpha' = \gamma \cdot \beta$

Example



Configuration: $\langle B, 01, 0Z_0 \rangle$

Sequence of configuration transitions:

$$\langle A, 001, Z_0 \rangle \vdash_M \langle B, 01, 0Z_0 \rangle \vdash_M \langle B, 1, 00Z_0 \rangle \vdash_M \langle C, \varepsilon, 0Z_0 \rangle$$

Formalizing the Language of a PDA (cont.)

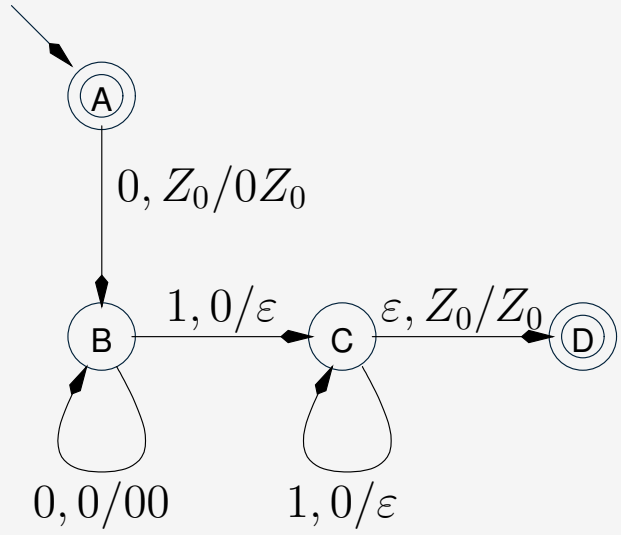
Definition

M *accepts* $x \in \Sigma^*$ if there are configurations $c_0, c_1, ...c_n$ for some $n \geq 0$ such that:

- $c_0 = \langle q_0, x, Z_0 \rangle$;
- $c_n = \langle q, \varepsilon, \alpha \rangle$ for some $q \in A, \alpha \in \Gamma^*$; and
- $c_i \vdash_M c_{i+1}$ all i such that $0 \leq i < n$.

The *language* M is $\mathcal{L}(M) = \{ x \in \Sigma^* \mid M \text{ accepts } x \}$.

Example



M accepts 0011, since

$$\langle A, 0011, Z_0\rangle \vdash_M \langle B, 011, 0Z_0\rangle \vdash_M \langle B, 11, 00Z_0\rangle$$
$$\vdash_M \langle C, 1, 0Z_0\rangle \vdash_M \langle C, \varepsilon, Z_0\rangle \vdash_M \langle D, \varepsilon, Z_0\rangle$$

M does not accept 010, since only possible configuration sequence is: follows.

$$\langle A, 010, Z_0\rangle \vdash_M \langle B, 10, 0Z_0\rangle \vdash_M \langle C, 0, Z_0\rangle \vdash_M \langle D, 0, Z_0\rangle$$

A Kleene Theorem for CFLs

A Kleene Theorem for CFLs

Recall the statement of the Kleene Theorem for regular languages.

> *A language is regular iff it is accepted by some finite automaton.*

This result says that FAs provide the requisite computing power for "processing" regular languages.

We proved one direction by showing how to convert any regular expression into a language-equivalent NFA.

We want to prove a similar connection between CFLs and PDAs:

> *A language is context-free if and only if it is recognized by some PDA.*

How Do We Prove This?

It suffices to prove two things.

1. For every CFG G there is a PDA M with $\mathcal{L}(G) = \mathcal{L}(M)$.
2. For every PDA M there is a CFG G with $\mathcal{L}(M) = \mathcal{L}(G)$.

Why does this suffice?

We will only prove the first direction.

Building PDAs from CFGs

Theorem For any CFG $G = \langle V, \Sigma, S, P \rangle$ there is a PDA M with $\mathcal{L}(G) = \mathcal{L}(M)$.

How do we build M? CFGs and PDAs seem very different (one generates words, the other accepts them). But ...

- CFGs generate words using *derivation* sequences ($\Rightarrow_G$).
- Any variable can be "expanded" at any time: e.g. in AaB either A or B can have a production applied. But expanding one variable does not affect the potential expansions for the others.
- Once a terminal appears, it remains.

Idea Build a PDA that simulates "appropriate" derivations in G.

Leftmost Derivations

In a "leftmost" derivation, the leftmost variable in a sequence of terminals and nonterminals is always worked on "first".

Example Consider G given by:

$$\begin{aligned} S &\longrightarrow AC \\ A &\longrightarrow aAb \mid \varepsilon \\ C &\longrightarrow cC \mid \varepsilon \end{aligned}$$

Here is a leftmost derivation of abc:

$$S \Rightarrow_G AC \Rightarrow_G aAbC \Rightarrow_G abC \Rightarrow_G abcC \Rightarrow_G abc$$

Here is a derivation that is *not* leftmost:

$$S \Rightarrow_G AC \Rightarrow_G AcC \Rightarrow_G aAbcC \Rightarrow_G abcC \Rightarrow_G abc$$

Formalizing Leftmost Derivations

Definition Let $G = \langle V, \Sigma, S, P \rangle$ be a CFG, with $\alpha, \beta \in (V \cup \Sigma)^*$. Then $\alpha \stackrel{\ell}{\Rightarrow}_G \beta$ if there exist $x \in \Sigma^*$, $\alpha', \gamma \in (V \cup \Sigma)^*$, and $A \in V$ such that:

- $\alpha = xA\alpha'$;
- $\beta = x\gamma\alpha'$; and
- $A \longrightarrow \gamma$ is a production in P.

We write $\alpha \stackrel{\ell}{\Rightarrow}^*_G \beta$ if α can be rewritten to β via a sequence of $\stackrel{\ell}{\Rightarrow}_G$ steps .

Lemma For any CFG $G = \langle V, \Sigma, S, P \rangle$ $w \in \mathcal{L}(G)$ iff $S \stackrel{\ell}{\Rightarrow}^*_G w$.

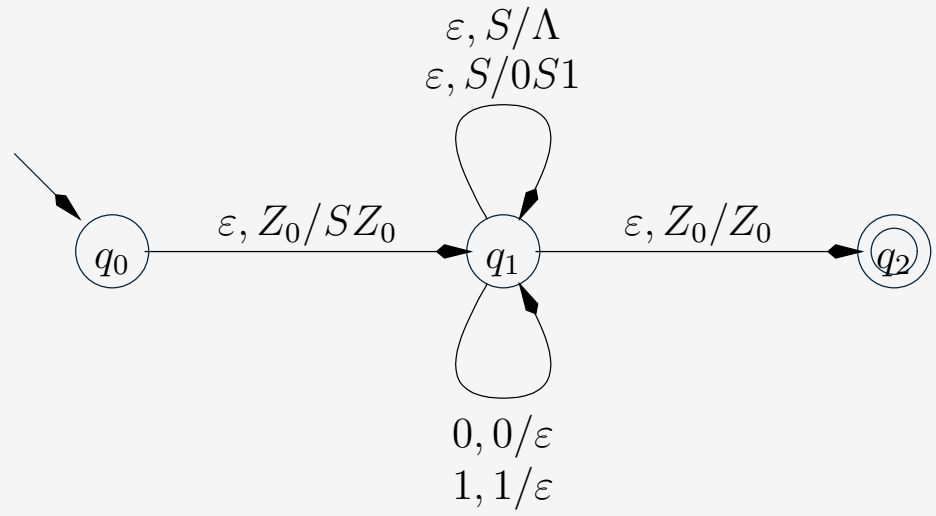
In other words: even though $\stackrel{\ell}{\Rightarrow}_G$ is more restricted than $\Rightarrow_G$, you can still generate the same words!

We Can Build a PDA for Leftmost Derivations!

- Stack contains (part of) current sequence of terminals and nonterminals in derivation, with topmost variable in stack being leftmost variable.
- Variables at top of stack are popped and replaced by right-hand sides of productions.
- Terminals at top of stack are matched against input and popped.

Example

$$S \longrightarrow \varepsilon \mid 0S1$$



Note correspondence:

$$\begin{array}{l|ll} S \stackrel{\ell}{\Rightarrow}_G 0S1 & \langle q_0, 01, Z_0\rangle & \vdash_M \langle q_1, 01, SZ_0\rangle \vdash_M \langle q_1, 01, 0S1Z_0\rangle \\ & & \vdash_M \langle q_1, 1, S1Z_0\rangle \\ \stackrel{\ell}{\Rightarrow}_G 01 & & \vdash_M \langle q_1, 1, 1Z_0\rangle \vdash_M \langle q_1, \varepsilon, Z_0\rangle \\ & & \vdash_M \langle q_2, \varepsilon, Z_0\rangle \end{array}$$

Formalizing the Construction

Let $G = \langle V, \Sigma, S, P \rangle$ be a CFG. We can construct PDA $M_G = \langle Q_G, \Sigma, \Gamma, q_0, Z_0, \delta_G, A_G \rangle$ as follows.

- $Q_G = \{q_0, q_1, q_2\}$
- $\Gamma = V \cup \Sigma \cup \{Z_0\}$ where $Z_0 \notin V \cup \Sigma$ is a new symbol.
- δ_G is defined as follows.

$$
\begin{aligned}
\delta_G(q_0, \varepsilon, Z_0) &= \{\langle q_1, SZ_0 \rangle\} \\
\delta_G(q_1, a, a) &= \{\langle q_1, \varepsilon \rangle\} \text{ if } a \in \Sigma \\
\delta_G(q_1, \varepsilon, A) &= \{\, \langle q_1, \alpha \rangle \mid A \longrightarrow \alpha \in P \,\} \text{ if } A \in V \\
\delta_G(q_1, \varepsilon, Z_0) &= \{\langle q_2, Z_0 \rangle\}
\end{aligned}
$$

- $A_G = \{q_2\}$

Why Does M_G Accept the Language of G?

Recall $w \in \mathcal{L}(M_G)$ iff $\langle q_0, w, Z_0\rangle \vdash^*_{M_G} \langle q_2, \varepsilon, \alpha\rangle$ some α.

Claims

1. For any $q' \in Q_G, w, w' \in \Sigma^*, \alpha \in \Gamma^*_G$,
   if $\langle q_0, w, Z_0\rangle \vdash^*_{M_G} \langle q', w', \alpha\rangle$ then $\alpha = \beta \cdot Z_0$ for some $\beta \in (V \cup \Sigma)^*$.
2. For any $w, w' \in \Sigma^*$, $\alpha \in \Gamma^*$,
   if $\langle q_0, w, Z_0\rangle \vdash^*_{M_G} \langle q_2, w', \alpha\rangle$ then $\alpha = Z_0$.
3. For any $w, w' \in \Sigma^*, \alpha \in (V \cup \Sigma)^*$,
   $\langle q_0, w, Z_0\rangle \vdash^*_{M_G} \langle q_1, w', \alpha \cdot Z_0\rangle$ iff $S \stackrel{\ell}{\Rightarrow}^*_G x \cdot \alpha$, where $x \in \Sigma^*$ is such that $w = x \cdot w'$.

On the basis of these claims, we can prove that $\mathcal{L}(G) = \mathcal{L}(M_G)$.

Determinism

Recall the definition of a PDA

Definition A *pushdown automaton* (PDA) is a septuple $\langle Q, \Sigma, \Gamma, q_0, Z_0, \delta, A \rangle$ where:

- Q is a finite set of *states*;
- Σ and Γ are the *input* and *stack* alphabets, respectively;
- $q_0 \in Q$ is the *start state*;
- $Z_0 \in \Gamma$ is the *initial stack symbol*;
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow 2^{Q \times (\Gamma^*)}$ is the *transition function*; and
- $A \subseteq Q$ is the set of *accepting states*.

A PDA for $L = \{ wcw^R \mid w \in \{a, b\}^* \}$

- It will have two states that correspond to "have not seen the c" and "have seen the c". The former will be the starting state, and the latter will be the final state.
- When in state "have not seen the c", it will push the symbols that it reads onto the stack.
- When it encounters the c it switches states without changing the stack.
- In the state "have seen the c", it compares the current input symbol to the symbol on the top of the stack and advances past both if they match.
- Only valid strings, that is, ones that have matching a's and b's and contain a c in the middle will cause acceptance. Any other string will reach a situation where there is no transition to take.

Definition

Let $M = \langle \{s, f\}, \{a, b, c\}, \{a, b, Z_0\}, s, Z_0, \delta, \{f\} \rangle$, where

1. $\delta(s, a, \gamma) = (s, a\gamma)$
2. $\delta(s, b, \gamma) = (s, b\gamma)$
3. $\delta(s, c, \gamma) = (f, \gamma)$
4. $\delta(f, a, a) = (f, \varepsilon)$
5. $\delta(f, b, b) = (f, \varepsilon)$

Sample accepting computation: $w = abacaba$

State	Unread input	Stack	Transition
s	$abacaba$	ε	—
s	$bacaba$	a	1
s	$acaba$	ba	2
s	$caba$	aba	1
f	aba	aba	3
f	ba	ba	4
f	a	a	5
f	ε	ε	4

Sample rejecting computation: $w = aaaa$

State	Unread input	Stack	Transition
s	$aaaa$	ε	—
s	aaa	a	1
s	aa	aa	1
s	a	aaa	1
s	ε	$aaaa$	1

Now consider the language $L = \{ ww^R \mid w \in \{a, b\}^* \}$.

There is no center marker c to tell us when to switch from the state that pushes input onto the stack into the state that reads input while popping characters off the stack.

We will have to use nondeterminism to "guess" when to make the switch.

1. $\delta(s, a, \gamma) = (s, a\gamma)$
2. $\delta(s, b, \gamma) = (s, b\gamma)$
3. $\delta(s, \varepsilon, \gamma) = (f, \gamma)$
4. $\delta(f, a, a) = (f, \varepsilon)$
5. $\delta(f, b, b) = (f, \varepsilon)$

Sample accepting computation: $w = abba$

State	Unread input	Stack	Transition
s	$abba$	ε	—
s	bba	a	1
s	ba	ba	2
f	ba	ba	3
f	a	a	5
f	ε	ε	4

If there is no way to "guess" correctly, then the string will not be accepted, for example with $w = babaa$

Recall the configuration transition relation

Definition A *configuration* of M is a triple $\langle q, x, \alpha \rangle \in Q \times (\Sigma^*) \times (\Gamma^*)$.

(q is current state, x is remaining input, α is stack, with top element first.)

Definition The *configuration transition relation*, $\vdash_M$, is defined by: $\langle q, x, \alpha \rangle \vdash_M \langle q', x', \alpha' \rangle$ if there exist $a \in \Sigma \cup \{\varepsilon\}$, $X \in \Gamma$, and $\beta, \gamma \in \Gamma^*$ such that:

- $\langle q', \gamma \rangle \in \delta(q, a, X)$.
- $x = a \cdot x'$
- $\alpha = X \cdot \beta$
- $\alpha' = \gamma \cdot \beta$

Determinism

A pushdown automaton is *deterministic* if for each configuration there is at most one configuration that can succeed it in a computation by M.

Question Can we always find an equivalent deterministic pushdown automaton for a given context-free language?

Answer: Unfortunately not.

There are some context-free languages that cannot be accepted by deterministic pushdown automata.

This is a dire result, especially if we actually want to produce a parser for the context-free language.

Some good news: For most programming languages one can construct deterministic pushdown automata that accept all syntactically correct programs.

Determinism and complementation

Theorem The class of deterministic context-free languages is closed under complementation.

This is trickier than it was for DFAs, because of the stack.

Sticking point: a PDA may reject because it never finished reading the input.

This can happen in the following two circumstances:

- M reaches a configuration that has no following configuration;
- M enters a configuration from which it can apply an infinite sequence of configurations that do not consume any input.

Proof idea: add in explicit transitions for these cases, then negate the accepting states.

Consider the language

$L = \{ a^n b^m c^p \mid m, n, p \geq 0, \text{ and } m \neq n \text{ or } m \neq p \}$.

Suppose that L is deterministic. Then $\overline{L}$ is deterministic context-free, and thus, context-free.

So $\overline{L} \cap a^* b^* c^*$ would be context-free since intersection of CFL and RL is a CFL.

But $\overline{L} \cap a^* b^* c^* = \{ a^n b^n c^n \mid n \geq 0 \}$, a language that is not context-free.

Thus, L cannot be deterministic.

Corollary The class of deterministic context-free languages is properly contained in the class of context-free languages.

End result: For pushdown automata, non-determinism is more powerful than determinism.

Ambiguity and Determinism

An ambiguous CFG is one which has more than one derivation for the same string. Note that ambiguity is a property of a grammar, not a language.

An inherently ambiguous CFL is a CFL that is not expressible using an unambiguous CFG. For example $\{ a^n b^m c^p \mid m = n \text{ or } m = p \}$ is an inherently ambiguous CFL.

Every deterministic PDA language is expressible in an unambiguous CFG. The converse is not true. For example, $S \longrightarrow 0S0 \mid 1S1 \mid \varepsilon$ is unambiguous but is accepted by no deterministic PDA.

Proving that a language is inherently ambiguous is tricky. See Herman A. Maurer, "A Direct Proof of the Inherent Ambiguity of a Simple Context-Free Language", Journal of the ACM 16(2), 1969.

http://doi.acm.org/10.1145/321510.321517