

Pushdown automata

Informatics 2A: Lecture 9

Alex Simpson

School of Informatics
University of Edinburgh
als@inf.ed.ac.uk

4 October, 2013

- 1 What are pushdown automata?
- 2 Example
- 3 Formal definition
- 4 Relationship between nondet. pushdown automata and CFLs

Machines for context-free languages

We've seen that **regular** languages can be defined by (det. or non-det.) **finite** automata.

Question: What kinds of machines do **context-free** languages correspond to?

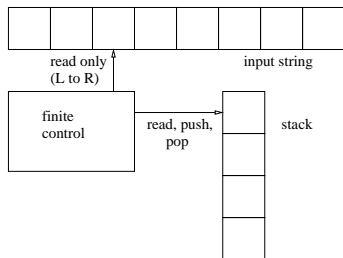
Motivation: Seeing how to **recognize** sentences of a CF language will be a step towards seeing how to **parse** such sentences.

Short answer: NFAs equipped with unlimited memory in the form of a **stack**.

Pushdown automata (PDAs)

As in NFAs, imagine a **control unit** with finitely many possible states, equipped with a **read head** that can (only) read the input string from left to right.

Add to this a **stack** with associated operations of **push**, **pop** and **read** the item on the top of the stack.



Transitions can depend on **both** the current input symbol and the current stack item.

Transitions can also cause items to be pushed to and/or popped from the stack.

Note: the machine **can't read a stack item other than the top one** without first popping (and so losing) all items above it.

Example of a PDA

Consider a PDA with a single state q , **input alphabet** $\Sigma = \{ (,) \}$ and **stack alphabet** $\Gamma = \{ (, \perp \}$. Call \perp the **initial stack symbol**.

Our PDA has four **transitions** $q \rightarrow q$, labelled as follows:

$$^1 (, \perp : (\perp \quad \quad \quad ^2 (, (: ((\quad \quad \quad ^3), (: \epsilon \quad \quad \quad ^4 \epsilon, \perp : \epsilon$$

Meaning. ¹ If current read symbol is $($ and current stack symbol is \perp , may pop the \perp and replace it with $(\perp$. Note that we push \perp first, then $($, so stack grows to the left.

² Similarly with $($ in place of \perp .

³ If current read symbol is $)$ and current stack symbol is $($, may simply pop the $($.

⁴ If current stack symbol is \perp , can just pop it.

Idea: stack keeps track of currently pending $($'s. When stack clears, may pop the initial \perp — this ends the computation.

Clicker question

1 $(, \perp : (\perp$ 2 $(, (: (($ 3 $), (: \epsilon$ 4 $\epsilon, \perp : \epsilon$

Suppose the current configuration of the PDA is

Unread input: $((\))$ Stack: \perp

Which of the following is not a possible successor configuration?

- 1 Unread input: $(\))$ Stack: $(\perp$
- 2 Unread input: $((\))$ Stack: \perp
- 3 Unread input: $((\))$ Stack: ϵ
- 4 They are all possible successor configurations

Sample execution

$1 \quad (, \perp : (\perp$
 $2 \quad (, (: (($
 $3 \quad), (: \epsilon$
 $4 \quad \epsilon, \perp : \epsilon$

	Unread input	Stack state
	((()))	\perp
$\xrightarrow{1}$	(())	(\perp
$\xrightarrow{2}$)())	((\perp
$\xrightarrow{3}$	())	(\perp
$\xrightarrow{2}$))	((\perp
$\xrightarrow{3}$)	(\perp
$\xrightarrow{3}$	ϵ	\perp
$\xrightarrow{4}$	ϵ	ϵ

The machine can empty its stack at the end of the input string **if and only if** the input string is a **well-matched sequence of brackets**. So it acts as a (non-deterministic) recognizer for such sequences.

PDAs: formal definition

A (nondeterministic) pushdown automaton (N)PDA M consists of

- a finite set Q of control states
- a finite input alphabet Σ
- a finite stack alphabet Γ including a start symbol \perp
- a start state $s \in Q$
- a finite transition relation $\Delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$

A string $x \in \Sigma^*$ is accepted by M if there is some run of M on x , starting at control state s with stack \perp , and finishing (at any control state) with empty stack having consumed all of x .

The language accepted by M is

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

PDAs are infinite state machines

A PDA M is a finite description of a machine with infinitely many states.

Define a nondeterministic **infinite state machine** \hat{M} over Σ as follows:

- **Machine states** are pairs $(q, \alpha) \in Q \times \Gamma^*$
- **Transition relation**: $(q, x\alpha) \xrightarrow{a} (q', \beta\alpha)$ where $\alpha \in \Gamma^*$ and $((q, a, x), (q', \beta)) \in \Delta$, and that's all. (Note that a can be ϵ .)
- **Start state**: (s, \perp)
- **Accepting states**: (q, ϵ) for $q \in Q$

Given \hat{M} , we can define the language $\mathcal{L}(\hat{M})$ it accepts exactly as for an ordinary ϵ -NFA (but with infinitely many states).

By definition, $\mathcal{L}(M) = \mathcal{L}(\hat{M})$.

Accepting by empty stack

Our definitions implement **acceptance by empty stack**.

Other acceptance conditions can be found in the literature.

Example variation: Equip M with a set of accepting states, and say a string is accepted if it can trigger a computation ending in an accepting state.

The choice of acceptance condition is not a big deal, For any NPDA of either kind, we can build one of the other kind that accepts the same language. For details see Kozen chapter E.

In Inf2A, we use use empty stack as our default acceptance condition for PDAs.

How powerful are PDAs?

We shall outline the proof of the following theorem:

A language L is context-free if and only if there is an NPDA M such that $L = \mathcal{L}(M)$.

One can also define **deterministic** pushdown automata (DPDAs) in a reasonable way.

;-) **Sadly**, there's no analogue of the NFA 'powerset construction' for NPDAs. In fact, there are context-free languages that can't be recognized by **any** DPDA (example in Lecture 28).

Since DPDAs allow for efficient processing, this prompts us to focus on 'simple' kinds of CFLs that **can** be recognized by a DPDA. (See next lecture on efficient table-driven parsing.)

From CFGs to NPDAs

Given a CFG with nonterminals N , terminals Σ , productions P and start symbol S .

Build an NPDA with a **single state** q , input alphabet Σ and stack alphabet $N \cup \Sigma$. Take S as initial stack symbol.

(**Intuition:** the stack records a 'guess' at a sentential form for the part of the input still to come.)

- For each **production** $X \rightarrow \alpha$ in P , include an **ϵ -transition** $((q, \epsilon, X), (q, \alpha))$.
- For each **terminal** $a \in \Sigma$, include a **transition** $((q, a, a), (q, \epsilon))$.

We'll illustrate this with our grammar for arithmetic expressions:

$$\begin{array}{ll}
 \text{Exp} \rightarrow \text{Var} \mid \text{Num} \mid (\text{Exp}) & \text{Var} \rightarrow x \mid y \mid z \\
 \text{Exp} \rightarrow \text{Exp} + \text{Exp} & \text{Num} \rightarrow 0 \mid \dots \mid 9 \\
 \text{Exp} \rightarrow \text{Exp} * \text{Exp} &
 \end{array}$$

CFGs to NPDAs: example

Consider the input string $(x)^*5$. Our machine can proceed as follows, correctly guessing the rule to apply at each stage.

Transition	Input read	Stack state
	ϵ	Exp
Apply * rule	ϵ	Exp * Exp
Apply () rule	ϵ	(Exp) * Exp
Match ((Exp) * Exp
Apply Var rule	(Var) * Exp
Apply x rule	(x) * Exp
Match x	(x) * Exp
Match)	(x)	* Exp
Match *	(x)*	Exp
Apply Num rule	(x)*	Num
Apply 5 rule	(x)*	5
Match 5	(x) * 5	stack empty!

At each stage, combining 'input read' and 'stack state' gives us a **sentential form**. In effect, the computation traces out a **leftmost derivation** of t in \mathcal{G} . So the computation tells us not just that the string is legal, but how to build a syntax tree for it.

From NPDA's to CFGs: brief sketch

Suppose first M is an NPDA with **just one state** q .

Can turn M into a CFG: almost the reverse of what we just did.

Use M 's stack alphabet Γ as the set of nonterminals of the grammar.

General form for **transitions** of M is $((q, a, X), (q, \alpha))$, where a and/or α might be ϵ . Turn these into **productions** $X \rightarrow a\alpha$.

Now suppose we have an NPDA M with **many states**. Can turn it into an equivalent NPDA with just one state, essentially by storing all 'state information' on the stack.

When pushing multiple stack entries, must nondeterministically 'guess' the intermediate states. For details see Kozen chapter 25.

Summary

We've seen that NPDAs exactly 'match' CFGs in terms of their power for defining languages. Indeed, a **pushdown store** (stack) gives just the computational power needed to deal with **nesting**.

Accepting computations don't just tell us a string is legal — they 'do parsing' (i.e. tell us how to build a syntax tree).

Problem is that computations here are **non-deterministic**: must correctly 'guess' which production to apply at each stage. If only things were deterministic, we'd have an **efficient** parsing algorithm!

Idea: see how far we can get with CFGs for which the corresponding PDA *is* deterministic. Already good enough for many computing applications.

(Later, will look at 'semi-efficient' parsers that work for *any* CFG.)

Reading and prospectus

Relevant reading:

- Core material: Kozen chapters 19, 23, 24.
- Further topics: E, 25, F.

Next time, we look at **LL(1) grammars**: a class of relatively 'simple' CFGs for which very efficient parsing is possible.