PDAs, CFGs, and Transition-based parsing

Tuesday - August 20, 2024

1 Pushdown automata as formal objects

Recall that for every regular language, there exists a finite-state automaton that recognizes that language (and vice versa).

There is the same kind of language–automata correspondence for context-free languages as well: PUSHDOWN AUTOMATA (PDAs). [1]

↪ pushdown automata ↔ context-free grammar

1.1 Informal definition of PDAs

PDAs are more powerful than FSAs because they accept a larger class of languages, which makes sense given that we already know that the regular languages are a subset of the context-free languages (from the Chomsky Hierarchy).

- A PDA is a finite automaton with a **stack** (or **Pushdown store**) on which it may read, write, and erase symbols.
  ↪ the most recently added item is the first one to be removed.
  ↪ Stacks are collections of elements that mandate *last in, first out* (LIFO) access.
  ↪ Stacks allow the following basic operations:
  - PUSH: Add an element to the top of the stack
  - POP: Take an element off the top of the stack
- PDAs read their input tapes from left to right, like an FSA (and its variants).
- The transitions of a PDA allow the top symbol of the stack to be read and removed (i.e. POPPED), added to (i.e. PUSHED), or left unchanged.

Note that a PDA that never uses the stack is effectively an ordinary FSA.

[1] The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafeteria (or used to exist?), since the operations never work on elements other than the top element.

1.2 Formal definition

A pushdown automaton is a six-place tuple $\langle Q, \Sigma, \Gamma, I, F, \Delta \rangle$ where:

1. Q is a finite set of states;
2. Σ, the alphabet, is a finite set of symbols; (symbols allowed on stack)
3. Γ, the stack alphabet, is a finite set of symbols; new each symbol on stack
4. $I \subseteq Q$ is the set of starting states; what we do with
5. $F \subseteq Q$ is the set of ending states; what we do whats on stack currently
6. starting
   $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q \times \Gamma^*$ is the set of transitions.
   symbol transitioning over another transition

For the sake of simplicity, I am assuming that the stack starts out empty. Other formalizations allow the stack to start out with a single symbol in it, which requires a PDA to be a seven-tuple:

(1) $\langle Q, \Sigma, \Gamma, I, Z, F, \Delta \rangle$, where Z $\in \Gamma$ is the initial stack symbol.

A PDA accepts an input if the computation leads to a situation in which all three of the following are simultaneously true:

(2)
(a) the entire input has been read,
(b) the PDA is in a final state, and
(c) the stack is empty.

same amount of a's and b's

Here is an example $L = \{a^n b^n \mid n > 0\}$. Remember that this is non-regular but can be expressed with a CFG.

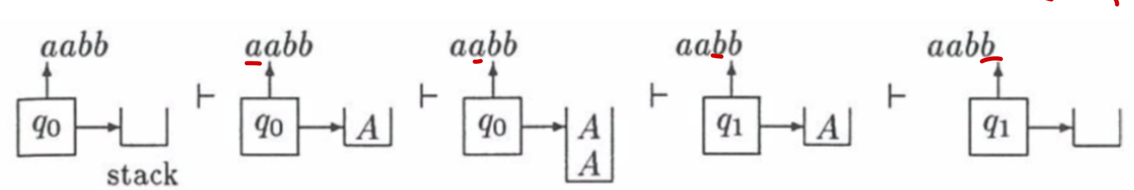
states start end “transition from q0 on symbol a to state q0, don't take anything off the stack, instead put symbol A on stack”

(3) $Q = \{q_0, q_1\}$, $I = \{q_0\}$, $F = \{q_1\}$
$\Delta = \{(q_0, a, \epsilon, q_0, A), (q_0, b, A, q_1, \epsilon), (q_1, b, A, q_1, \epsilon)\}$ transitions

This PDA generates $L = \{a^n b^n | n > 0\}$ by classifying prefixes into unboundedly many categories, identified by the contents of the stack.

For input *aabb*...

accepted ✓



We can also reason about the various inputs that the PDA in (3) would reject (as desired). For instance:

- *ba* is rejected: the PDA blocks in state q0 and cannot read the entire input. → second transition requires you to remove A from stack but stack is empty
- *aaabb* is rejected: the PDA halts in state q1 with A on the stack. A is left on stack at end
- *aabbb* is rejected: the PDA cannot read the last b, as there is no A on the stack.

we can't use a b transition

1.3 Equivalence of PDAs and CFGs

PDAs accept exactly the languages generated by context-free grammars.[2]

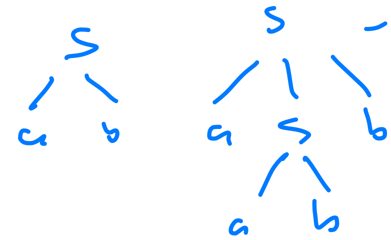
Formally proving the equivalency between PDAs and CFGs is too complex and would take us too far afield; though see Hopcroft & Ullman (1979).[3]

Instead, to get a sense of the proof, let's consider an algorithm for constructing from any given CFG, an equivalent nondeterministic PDA.

Algorithm to convert a CFG to a PDA (Partee et al. 1993, 490-491)[4]

Given a CFG G = (N, Σ_G, I, R), we can construct an equivalent PDA $M = (Q, \Sigma_M, \Gamma, I_M, F_M, \Delta_M)$ as follows: (four tuple

So we can generate aabb



in CFG

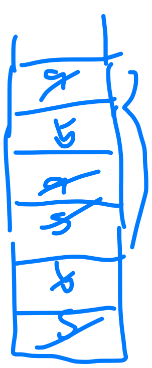
1. $Q = \{q_0, q_1\}$ states
2. $I_M = \{q_0\}$ initial
3. $F_M = \{q_1\}$ Final
4. $\Sigma_M = \Sigma_G$ same alphabet
5. $\Gamma = N \cup \Sigma_G$ Non terminal symbols in CFG — terminal symbols in CFG
6. $\forall S \in I$, $(q_0, \epsilon, \epsilon, q_1, S) \in \Delta$ for every starting non-terminal S
   (effectively: put the start symbol on the stack)
7. For each rule of the grammar $A \to \psi$, $(q_1, \epsilon, A, q_1, \psi) \in \Delta$. delete replace
   (effectively: on the stack, replace A with ψ)
8. For each symbol $a \in \Sigma_G$, $(q_1, a, a, q_1, \epsilon)$.
   (effectively: match a in the input with a on the stack)

The PDAs resulting from this algorithm work by loading the starting nonterminal symbol onto the stack and then simulating a derivation there by manipulations that correspond to the rewriting rules of the CFG.

Convert the CFG below into a PDA:

S→ab
Q = {q0, q1}
I = {q0}
F = {q1}
Sigma = {a, b}
Gamma = {S, a, b}
Delta = {(q0, eps, eps, q1, S),
(q1, eps, S, q1, aSb), (q1, eps, S, q1, ab),
(q1, a, a, q1, eps), (q1, b, b, q1, eps)}

(4) $N = I = \{S\}$, $\Sigma_G = \{a, b\}$
S→ aSb -rule
S→ab -rule



Initial: {q0, empty stack}, string = aabb
1: {q1, S}, string left = aabb, transition = (q0, eps, eps, q1, S)
2: {q1, aSb}, string left = aabb, transition = (q1, eps, S, q1, aSb)
3: {q1, Sb}, string left =abb, transition = (q1, a, a, q1, eps)
4: {q1, abb}, string left = abb, transition = (q1, eps, S, q1, ab)
5: {q1, bb}, string left = bb, transition = (q1, a, a, q1, eps)
6: {q1, b}, string left = b, transition = (q1, b, b, q1, eps)
7: {q1, empty stack}, string left = eps, transition = (q1, b, b, q1, eps)

read: "aabb"

[2]Non-deterministic version; Unlike FSAs, deterministic and nondeterministic PDAs are not equivalent. That is, there is no systematic way to convert every nondeterministic PDA into a deterministic one.
[3]Hopcroft, J. & Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation.* Reading, MA: Addison-Wesley.
[4]Partee, B. H., ter Meulen, A., &Wall, R. (1993). *Mathematical Methods in Linguistics.* Springer

1.4 Local summary

So far, we have introduced two language classes (that is, sets of languages; a language is a set of strings) and different formal objects that can describe these languages:

- regular languages
- context-free languages

For regular languages, we introduced two finite systems that can describe a possibly infinite set of strings: (ϵ-)finite state machines and regular expressions, and showed their equivalence. [5]

For context-free languages, we also introduced two finite systems that can describe a possibly infinite set of strings: context free grammars and push-down automata, and showed their equivalence. [6]

Language class	Algebraic	Automata/Machine	Rules systems
Regular languages	Regular expressions	FSAs and their variants	?
Context-free languages	?	Pushdown automata (FSA + a stack)	Context-free rewrite rules

Even though we said at the beginning of the class that we would not be looking at that much about the algorithmic level of language computation (that is, what specific algorithms people use), now, in the following section, what we are going to see is what it takes if we want to go one step further to make the algorithmic claims. And we are going to focus on context-free languages, specifically.

[5] Regular languages can be characterized by so-called linear rule systems: that is, every rewrite rule in R is of the form $A \to xB$ (right-linear) or $A \to x$ where A and B are in N and x is a terminal string. There is a corresponding notion of a left-linear grammar: every rewrite rule in R is of the form $A \to Bx$ or $A \to x$ where A and B are in N and x is a terminal string.

[6] For its algebraic characterization, essentially, this is regular expressions with some general fixed-point operation. See some references here

2 Embedding and acceptability patterns

The following collection of sentences provides a motivating "test set" for basic theories of human sentence processing.

(5) Left-branching structures

keep going left

a. Mary won
b. Mary 's baby won
c. Mary 's boss 's baby won

(6) Right-branching structures

keep going right

a. John met the boy
b. John met the boy that saw the actor
c. John met the boy that saw the actor that won the award

(7) Center-embedding structures

keep going center

a. the actor won
b. the actor the boy met won
c. the actor the boy the baby saw met won

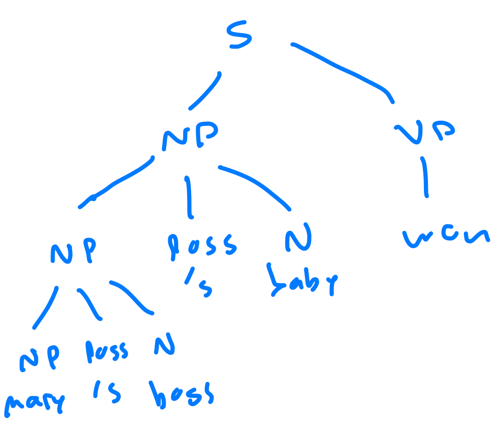
difficult to process

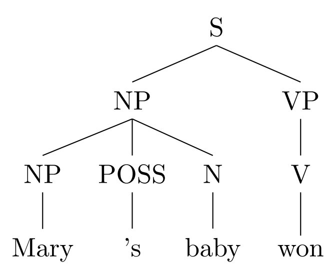
Here's a CFG generating all of the sentences in (5), (6), and (7)

(8)
S → NP VP
S → WHILE S S
NP → NP POSS N
NP → (D) N (PP) (SRC) (ORC)
VP → V (NP) (PP)
PP → P NP
SRC → THAT VP
ORC → NP V

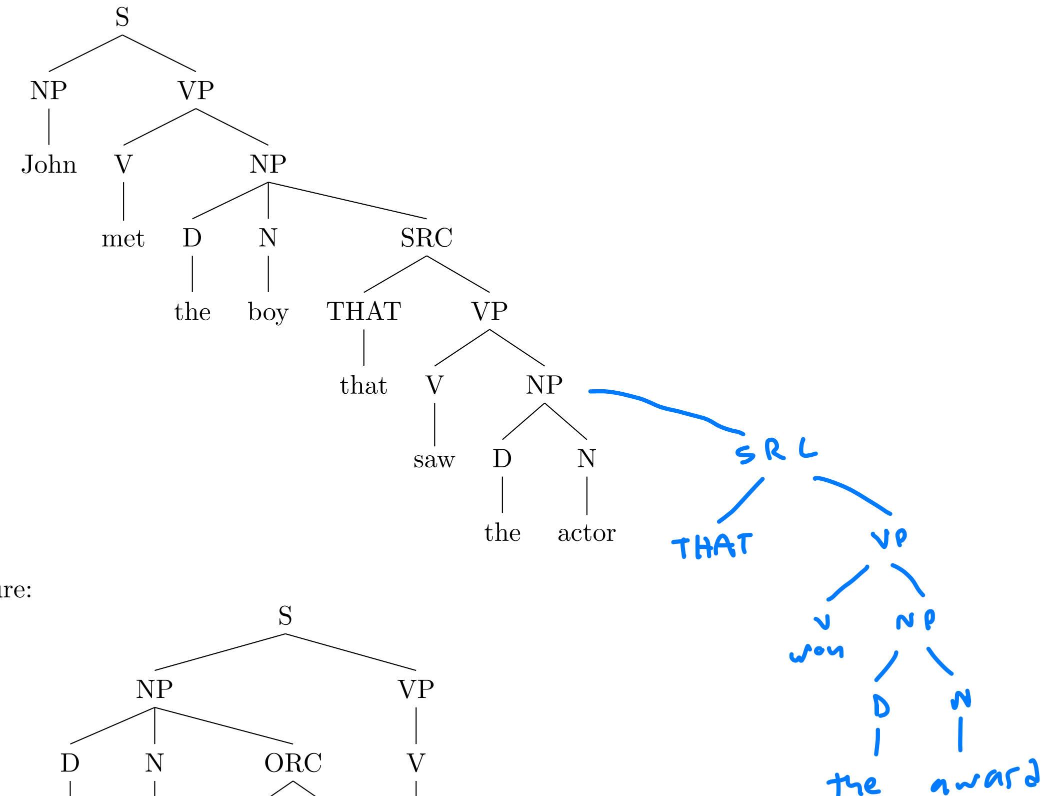
N → baby, boy, actor, spouse, boss, award
NP → Mary, John
V → met, saw, won
D → the
P → on, in, with
THAT → that
POSS → 's
WHILE → while

A **left-branching** structure under this grammar:

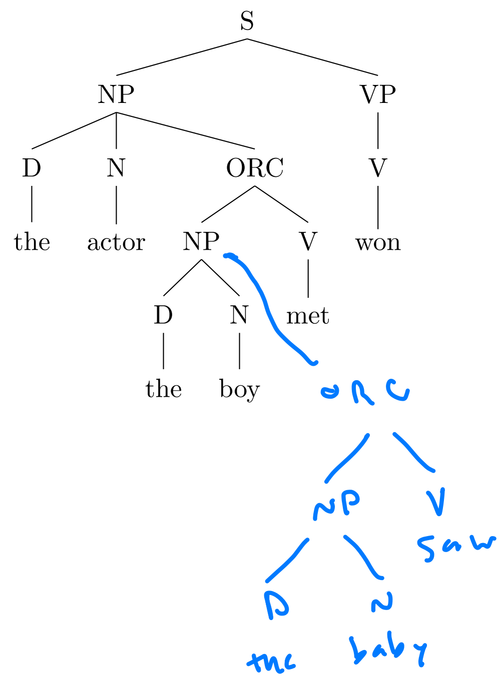




A **right-branching** structure:



A **center-embedding** structure:



3 Transition-based parsing

Let's consider, in the abstract, a type of device that we'll call a "**configuration-transition system**." These are devices for parsing/recognizing symbol-sequences by stepping through them one symbol at a time, from left-to-right. These have the following components:

- a specification of what a **configuration** is: a pair consisting of symbols that are being maintained in memory, and an input buffer containing the input remaining to be parsed
- a specification of what a **starting configuration** is
- a specification of what a **goal configuration** is
- a specification of a **transition relation on configurations** (which I'll write as $\Rrightarrow$)

It's trivial to recast an FSA as one of these kinds of systems:

(9) Given an FSA $M = (Q, \Sigma, I, F, \Delta)$, we can construct a configuration-transition system M' and the same finite alphabet Σ, which recognizes $L(M)$ in the following way: , how much of string left to process

a. A **configuration** is a pair (q, w), such that $q \in Q$, and where w is a string in Σ^*
b. **Starting configuration**: (q_0, w), where q_0 is a start state in I and w is the input string
c. **Goal configuration**: (q_n, ϵ), where q_n is an end state in F
d. **Transition relation** (Consume): $(q_{i-1}, x_i x_{i+1} \dots x_n) \Rrightarrow (q_i, x_{i+1} \dots x_n)$ if $(q_{i-1}, x_i, q_i) \in \Delta$

convert in set of transitions

The consume rule says that if there's a transition in the FSA between q_{i-1} and q_i on x_i, the parser can transition from a configuration consisting of q_{i-1} and a string that starts with x_i, to a configuration consisting of q_i and the rest of the string. This is not particularly exciting, because left-to-right parsing is essentially the same thing as string generation for an FSA.

3.1 Pushdown automata as configuration-transition systems

We can look at **pushdown automata (PDAs)** as configuration-transition systems with memory in the form of a **stack**: a list of symbols that can only be accessed from one end, where the last thing in is the first thing out.

For example, we can ignore the state transitions and just focus on the stack content. $\{a^n b^n \mid n \geq 0\}$. Here's a PDA that can do this:

(10) M_1 has a stack alphabet $\Gamma = \{A, Z, Y\}$, an input alphabet $\Sigma = \{a, b\}$, and the following specification of configurations and transitions:

a. **Configurations**: (Φ, w), where Φ is a string in Γ^* and w is a string in Σ^*
b. **Starting configuration**: (Z, w), where w is the input string
c. **Goal configuration**: (Y, ϵ)
d. Push A step: $(\Phi Z, a x_{i+1} \dots x_n) \Rrightarrow (\Phi A Z, x_{i+1} \dots x_n)$, removing Z from top of stack, then putting a, then putting Z
e. Switch step: $(\Phi Z, b x_{i+1} \dots x_n) \Rrightarrow (\Phi Y, b x_{i+1} \dots x_n)$ - take Z off, then put Y
f. Pop A step: $(\Phi A Y, b x_{i+1} \dots x_n) \Rrightarrow (\Phi Y, x_{i+1} \dots x_n)$

pop Y, pop A, put Y, process b

To parse the string $aaabbb$, M_1 would take the following steps:

	Type of transition	Configuration
0	—	$(Z,\ aaabbb)$
1	Push A	$(AZ, aabbb)$
2	Push A	$(AAZ, abbb)$
3	Push A	$(AAAZ, bbb)$
4	Switch	$(AAAY, bbb)$
5	Pop A	(AAY, bb)
6	Pop A	(AY, b)
7	Pop A	(Y, ϵ)

w concatenated with reverse of w

Here's a PDA that can recognize the palindrome language $\{ww^R \mid w \in \{a,b\}^*\}$:

(11) M_2 has a stack alphabet $\Gamma = \{A, B, Z, Y\}$, an input alphabet $\Sigma = \{a, b\}$, and the following specification of configurations and transitions:

a. Configurations: (Φ, w), where Φ is a string in Γ^* and w is a string in Σ^*
b. Starting configuration: (Z, w), where w is the input string
c. Goal configuration: (Y, ϵ) process a and put a on stack
d. Push A step: $(\Phi Z, ax_{i+1} \dots x_n) \Rrightarrow (\Phi AZ, x_{i+1} \dots x_n)$ we can have abba, babbab
e. Push B step: $(\Phi Z, bx_{i+1} \dots x_n) \Rrightarrow (\Phi BZ, x_{i+1} \dots x_n)$
f. Reverse step: $(\Phi Z, x_i \dots x_n) \Rrightarrow (\Phi Y, x_i \dots x_n)$ – we are halfway through
g. Pop A step: $(\Phi AY, ax_{i+1} \dots x_n) \Rrightarrow (\Phi Y, x_{i+1} \dots x_n)$
h. Pop B step: $(\Phi BY, bx_{i+1} \dots x_n) \Rrightarrow (\Phi Y, x_{i+1} \dots x_n)$ – pop B and process b

To parse the string $aabbaa$, M_2 would take the following steps:

	Type of transition	Configuration
0	—	$(Z,\ aabbaa)$
1	Push A	$(AZ, abbaa)$
2	Push A	$(AAZ, bbaa)$
3	Push B	$(AABZ, baa)$
4	Reverse	$(AABY, baa)$
5	Pop B	(AAY, aa)
6	Pop A	(AY, a)
7	Pop A	(Y, ϵ)

4 CFGs and PDAs

We'll go deeper about the conversion in three ways, by looking at three different "recipes" for converting CFGs to PDAs as *configuration-transition systems*, i.e., purely by focusing on the stack contents.

↪ We aim to determine whether any of these recipes can account for the empirical patterns of acceptability that humans give for left-branching structures, right-branching structures, and center-embedding structures.

Some conventions that we'll adopt:

- A, B, etc. will be placeholders for nonterminal symbols; x_1, x_2, etc. will be placeholders for terminal symbols; and Φ will be a placeholder for a sequence of nonterminals on the stack.
- We'll assume a "modified Chomsky Normal Form," where every right-hand side of a CFG rule has either a single terminal symbol, or a sequence of one-or-more nonterminal symbols.

4.1 Bottom-up parsing

Recipe for a bottom-up parser

Given a CFG (N, Σ, I, R) in modified CNF, we can construct a bottom-up PDA which uses N as its stack alphabet and Σ as its symbol alphabet, and recognizes $L(G)$ in the following way:

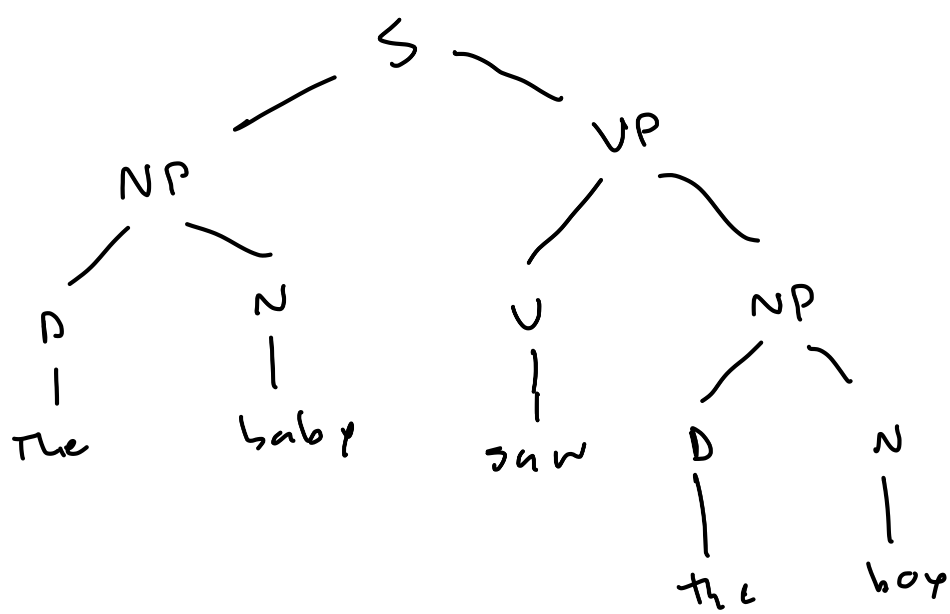
- Starting configuration: $(\epsilon, x_1 \dots x_n)$, where $x_1 \dots x_n$ is the input
- Goal configuration: (A, ϵ), where A is one of the start symbols in I
- SHIFT step: $(\Phi, x_i x_{i+1} \dots x_n) \Rrightarrow (\Phi A, x_{i+1} \dots x_n)$, where there is a rule $A \to x_i$ in R
- REDUCE step: $(\Phi B_1 \dots B_m, x_i \dots x_n) \Rrightarrow (\Phi A, x_i \dots x_n)$, where there is a rule $A \to B_1 \dots B_m$ in R

build tree from bottom up

Here's an example of how a bottom-up parser constructed from the grammar in (8) would parse the string *the baby saw the boy*:

	Type of transition	Rule used	Configuration
0	—	—	(ϵ, the baby saw the boy)
1	SHIFT	D → the	(D, baby saw the boy)
2	SHIFT	N → baby	(D N, saw the boy)
3	REDUCE	NP → D N	(NP, saw the boy)
4	SHIFT	V → saw	(NP V, the boy)
5	SHIFT	D → the	(NP V D, boy)
6	SHIFT	N → boy	(NP V D N, ϵ)
7	REDUCE	NP → D N	(NP V NP, ϵ)
8	REDUCE	VP → V NP	(NP VP, ϵ)
9	REDUCE	S → NP VP	(S, ϵ)

Notice that here, the top of the stack is written on the *right*.



4.2 Top-down parsing

Recipe for a top-down parser

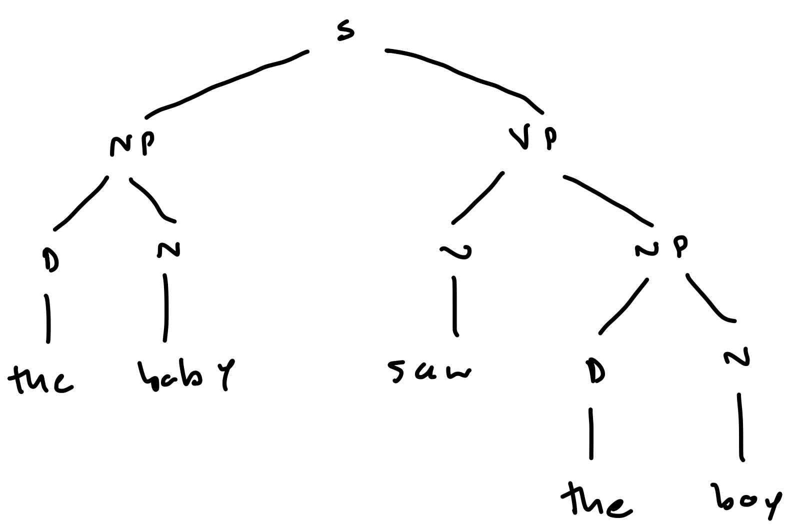
Given a CFG (N, Σ, I, R) in modified CNF, we can construct a top-down PDA which uses N as its stack alphabet and Σ as its symbol alphabet, and recognizes $L(G)$ in the following way:

- Starting configuration: $(A, x_1 \dots x_n)$
  where A is one of the start symbols in I and $x_1 \dots x_n$ is the input
- Goal configuration: (ϵ, ϵ) stack and finished processing string
- PREDICT step: $(A\Phi, x_i \dots x_n) \Rrightarrow (B_1 \dots B_m\Phi, x_i \dots x_n)$
  where there is a rule $A \to B_1 \dots B_m$ in R
- MATCH step: $(A\Phi, x_i x_{i+1} \dots x_n) \Rrightarrow (\Phi, x_{i+1} \dots x_n)$
  where there is a rule $A \to x_i$ in R

Example:

	Type of transition	Rule used	Configuration
0	—	—	(S, the baby saw the boy)
1	PREDICT	S → NP VP	(NP VP, the baby saw the boy)
2	PREDICT	NP → D N	(D N VP, the baby saw the boy)
3	MATCH	D → the	(N VP, baby saw the boy)
4	MATCH	N → baby	(VP, saw the boy)
5	PREDICT	VP → V NP	(V NP, saw the boy)
6	MATCH	V → saw	(NP, the boy)
7	PREDICT	NP → D N	(D N, the boy)
8	MATCH	D → the	(N, boy)
9	MATCH	N → boy	(ϵ, ϵ)

Notice that here, the top of the stack is written on the *left*.



4.3 Left-corner parsing

We'll introduce another convention for left-corner parsing: in addition to the nonterminal symbols in the grammar, the parser's stack alphabet will also include a "barred" version for each of these symbols. We'll use A for a "bottom-up" version of the nonterminal A, and we'll use $\overline{A}$ for a "top-down" version of the nonterminal A.

Recipe for a left-corner parser

Given a CFG (N, Σ, I, R) in modified CNF, we can construct a left-corner PDA with stack alphabet Γ which uses Σ as its symbol alphabet, and recognizes $L(G)$ in the following way:

- Stack alphabet: if $A \in N$, then both $A \in \Gamma$ and $\overline{A} \in \Gamma$
- Starting configuration: $(\overline{A}, x_1 \dots x_n)$
  where A is one of the start symbols in I and $x_1 \dots x_n$ is the input
- Goal configuration: (ϵ, ϵ)

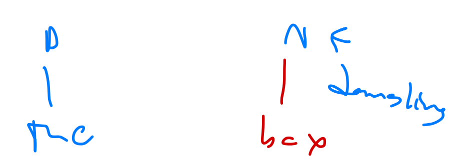
same as top down

- SHIFT step: $(\Phi, x_i x_{i+1} \dots x_n) \Rrightarrow (A\Phi, x_{i+1} \dots x_n)$
  where there is a rule $A \to x_i$ in R
- MATCH step: $(\overline{A}\Phi, x_i x_{i+1} \dots x_n) \Rrightarrow (\Phi, x_{i+1} \dots x_n)$
  where there is a rule $A \to x_i$ in R
- LC-PREDICT step: $(B_1\Phi, x_i \dots x_n) \Rrightarrow (\overline{B_2} \dots \overline{B_m} A\Phi, x_i \dots x_n)$
  where there is a rule $A \to B_1 \dots B_m$ in R
- LC-CONNECT step: $(B_1\overline{A}\Phi, x_i \dots x_n) \Rrightarrow (\overline{B_2} \dots \overline{B_m}\Phi, x_i \dots x_n)$
  where there is a rule $A \to B_1 \dots B_m$ in R

Example:

	Type of transition	Rule used	Configuration — bar means not connected
0	—	—	($\overline{\text{S}}$, the baby saw the boy)
1	SHIFT - new rule	D → the	(D $\overline{\text{S}}$, baby saw the boy)
2	LC-PREDICT	NP → D N	($\overline{\text{N}}$ NP $\overline{\text{S}}$, baby saw the boy)
3	MATCH - connect existing stuff	N → baby	(NP $\overline{\text{S}}$, saw the boy)
4	LC-CONNECT	S → NP VP	($\overline{\text{VP}}$, saw the boy)
5	SHIFT	V → saw	(V $\overline{\text{VP}}$, the boy)
6	LC-CONNECT	VP → V NP	($\overline{\text{NP}}$, the boy)
7	SHIFT	D → the	(D $\overline{\text{NP}}$, boy)
8	LC-CONNECT	NP → D N	($\overline{\text{N}}$, boy)
9	MATCH	N → boy	(ϵ, ϵ)

Shift vs match



Here again, the top of the stack is written on the *left*.