CDM
Parsing and Decidability

Klaus Sutner
Carnegie Mellon Universality

65-parsing 2017/12/15 23:17

The Recognition Problem

Problem:	**Context Free Recognition**
Instance:	A CFG G and a word $x \in \Sigma^\star$.
Question:	Is $x \in \mathcal{L}(G)$?

We will show that this problem is decidable in cubic time. This is cheating, though: for many context free languages much better algorithms are available, no modern compiler could function without them.

Of course, in the RealWorld$^{\text{TM}}$, just checking membership is not enough:

- If x is in the language, we want a parse tree.
- If x is not in the language, we want a reason; e.g., a place in x where there might be a typo.

What Could Go Wrong?

It is obvious that membership is semidecidable: we can simply enumerate all possible derivations, say, in length-lex order.

Is there any way this process could not be decidable? Perhaps a derivation

$$S \Rightarrow \alpha \Rightarrow x \in \Sigma^\star$$

where the intermediate sentential form α is hugely longer than x.

Your gut feeling should tell you that is not an actual problem, we can always clean up the derivation to get a shorter one. The best way to organize this is to streamline the grammar itself.

Cleanup

As defined, context free grammars may contain useless and "awkward" productions. These can be effectively eliminated without changing the language.

In fact, one can convert context free grammars into a number of normal forms that often come in handy when one tries to establish properties of context free languages.

We will only discuss one of these: Chomsky normal form.

Eliminating Unit-Productions

A unit-production is a production of the form $A \to B$.

Lemma

For every CFG G there is an equivalent CFG H without unit productions.

Proof.

Introduce new productions $A \to \beta$ whenever there is a chain of productions

$$A \to B_1 \to B_2 \to \ldots \to B_n \to \beta$$

where $A, B_i \in V$ and $\beta \notin V$.

Then delete all unit productions.

□

So this is quite similar to ε-elimination in FSMs.

ε-Productions

A CFG is ε-free if it has no productions of the form $A \to \varepsilon$.

A variable $A \in V$ is called nullable if $A \Rightarrow \varepsilon$.

Claim

One can easily determine all nullable variables in a grammar.

Proof.

Mark variables with ε-productions, then all with a production $A \to B_1 B_2 \dots B_r$ where the B_i are already marked.

Repeat till a fixed point is reached.

□

Eliminating ε-Productions

Lemma

For any CFG G there is an ε-free CFG H such that $\mathcal{L}(G) - \{\varepsilon\} = \mathcal{L}(H)$.

Proof.

Construct H from G as follows.

Given a production $A \to \alpha = X_1 X_2 \dots X_n$, write α_μ for what is left of α after erasing each X_i with $i \in \mu \subseteq [n]$.

Also, let $\nu \subseteq [n]$ be the set of indices i such that X_i is nullable.

Replace the production $A \to \alpha$ by all productions of the form $A \to \alpha_\mu$ where $\mu \subseteq \nu$.

In the end, remove all ε-productions.

□

Note that $\mathcal{L}(G) = \mathcal{L}(H)$ whenever S is not nullable.

Brute-Force Recognition

In an ε-free grammar we must have

$$\alpha \Rightarrow \beta \qquad \text{implies} \qquad |\alpha| \leq |\beta|$$

Note that we may safely assume that a derivation never contains the same word twice: otherwise we can just remove the loop.

But then, by the lemma, we can solve the recognition problem by a brute force search over all possible derivations: there are only finitely many. Actually, this method is easily primitive recursive, and even elementary.

Of course, the complexity of this algorithm is wildly exponential, so this won't do much good in the real world.

Chomsky Normal Form

A CFG grammar is in Chomsky normal form if all its productions are of the form

$$A \to BC$$
$$A \to a$$

We also allow a special production $S \to \varepsilon$ to deal with languages containing ε. In this case, S is not allowed to appear on the RHS of any production.

This is arguably the most basic normal form. The parse trees are all full binary trees, except at the leaves.

Chomsky Normal Form

Theorem

There is an algorithm to transform a CFG G into an equivalent CFG H in Chomsky normal form.

Proof.

First eliminate all ε-productions and unit productions.

Introduce new variables A_t and new productions $A_t \to t$ for all t in Σ.

In all non-terminal productions replace the terminals by their corresponding variables.

Now we are left with productions of the form $\pi : A \to B_1 \dots B_k$, where $k \geq 3$.

Production π can be replaced by a sequence of $k-1$ productions

$$A \to B_1 C_1 \quad C_1 \to B_2 C_2 \quad \dots \quad C_{k-2} \to B_{k-1} B_k$$

where $C_1, \dots, C_{k-2}$ is a set of new variables.

Deal with ε.

□

The Cocke-Younger-Kasami Algorithm

Suppose $G = \langle V, \Sigma, \mathbb{P}, S \rangle$ is in Chomsky Normal Form and does not contain $S \to \varepsilon$. Then any parse-tree in G is a full binary tree (except for the parents of leaves). Consider a string w over Σ, $n = |w| \geq 1$.

To determine whether w is in $\mathcal{L}(G)$ one can try to reconstruct a parse-tree of w starting at the n leaves labeled $w_1, \dots, w_n$. We can immediately replace label w_i by the set of labels in V: $\{ A \mid A \to w_i \text{ is in } \mathbb{P} \}$. Given two adjacent nodes whose labels contain B and C respectively one can introduce a common parent with label A whenever $A \to BC$ is in $\mathbb{P}$.

Key Problem: we do not know the shape of the tree ahead of time.

Build'em All

To handle this difficulty we will construct all possible trees, implicitly.

For bookkeeping purposes it is convenient to use a matrix representation.

We will show how to construct a so called recognition matrix $T = (t_{i,j})$ of w, $T \in \mathfrak{P}(V)^{n,n}$.

For all $1 \leq i \leq j \leq n$ we explain how to compute $t_{i,j} \subseteq V$ such that:

$$A \in t_{i,j} \quad \text{iff} \quad A \Rightarrow w_i w_{i+1} \dots w_j.$$

Hence, upon completion of the algorithm, $w \in \mathcal{L}(G) \iff S \in t_{1,n}$.

CYK Algorithm

for $i = 1$ to n **do**
  $t_{i,i} = \{ A \mid A \to w_i \}$;

for $d = 1$ to $n-1$ **do**
for $i = 1$ to $n - d$ **do**
  $j = d + i$;
  for $k = i$ to $j - 1$ **do**
    if exists $A \to BC$, $B \in t_{i,k}$, $C \in t_{k+1,j}$
    then add A to $t_{i,j}$;

A classical example of dynamic programming, with running time of $O(n^3)$.

Correctness

Correctness proof is by induction on $d = j - i$.

The case $d = 0$ is taken care of by initialization, so assume $d > 0$, i.e., $i < j$.

Suppose $A \Rightarrow w_i \dots w_j$. Then

$$A \Rightarrow^1 BC \Rightarrow w_i \dots w_j,$$

whence

$$B \Rightarrow w_i \dots w_r \text{ and } C \Rightarrow w_{r+1} \dots w_j$$

for some $i \leq r < j$. By IH $B \in t_{i,r}$ and $C \in t_{r+1,j}$ and thus A is placed in $t_{i,j}$ by definition of the algorithm. The opposite direction entirely similar.

Example I

Consider the following grammar G in CNF:

$$A \to AA \mid AB \mid a$$
$$B \to AA \mid BB \mid b$$

Note that it is not entirely clear what language this grammar describes.

The CYK algorithm on input $u = ababaaa$ and $v = bababbb$ produces the following two recognition matrices, showing that u is in the language generated by G whereas v is not.

Recognition Matrices

A	A	AB	AB	AB	AB	AB
	B	$-$	$-$	B	B	B
		A	A	AB	AB	AB
			B	$-$	B	B
				A	AB	AB
					A	AB
						A

B	$-$	$-$	B	B	B	B
	A	A	AB	AB	AB	AB
		B	$-$	$-$	$-$	$-$
			A	A	A	A
				B	B	B
					B	B
						B

Example II

Consider the ε-free version of the Dyck language D_1.

$$S \to SS \mid a\,S\,\overline{a} \mid a\,\overline{a}$$

A Chomsky Normal Form for this grammar is

$$S \to SS \mid LA \mid LR$$
$$A \to SR$$
$$L \to a$$
$$R \to \overline{a}$$

Note that the CNF is a bit artificial, the original grammar is more natural.

Recognition Matrices

$$\begin{array}{cccccccccc}
L & S & - & S & - & - & - & S & - & S \\
 & R & - & - & - & - & - & - & - & - \\
 & & L & S & - & - & - & S & - & S \\
 & & & R & - & - & - & - & - & - \\
 & & & & L & - & - & S & - & S \\
 & & & & & L & S & A & - & - \\
 & & & & & & R & - & - & - \\
 & & & & & & & R & - & - \\
 & & & & & & & & L & S \\
 & & & & & & & & & R
\end{array}$$

$$\begin{array}{cccccccccc}
L & - & - & - & - & S & - & - & - & S \\
 & L & - & - & S & A & - & - & - & - \\
 & & L & S & A & - & - & - & - & - \\
 & & & R & - & - & - & - & - & - \\
 & & & & R & - & - & - & - & - \\
 & & & & & R & - & - & - & - \\
 & & & & & & L & - & - & S \\
 & & & & & & & L & S & A \\
 & & & & & & & & R & - \\
 & & & & & & & & & R
\end{array}$$

Digression: Generating a Parse

Note that we can label the productions of a grammar, say, 1, 2, ..., m. We can then represent a leftmost derivation by a sequence of numbers in $[m]$: the handle is always uniquely determined as the first variable in the current sentential form.

Given a recognition matrix T we can generate a parse in $O(n^2)$ (how?).

For example, for the grammar

$$\begin{aligned}
S &\to SS \mid LA \mid LR \\
A &\to SR \\
L &\to a \\
R &\to \overline{a}
\end{aligned}$$

and the string $ababaabbab$ we get the parse

$$1, 3, 5, 6, 1, 3, 5, 6, 1, 2, 5, 4, 3, 5, 6, 6, 3, 5, 6$$

Language Decision Problems

Suppose $\mathcal{L} \subseteq \mathfrak{P}(\Sigma^\star)$ is a family of languages where every $L \in \mathcal{L}$ has a representation as a finite data structure.

Classical example: regular languages with DFAs or NFAs.

There are several natural decision problems associated with $\mathcal{L}$.

- $\mathcal{L}$-Emptiness
- $\mathcal{L}$-Finiteness
- $\mathcal{L}$-Equality
- $\mathcal{L}$-Inclusion
- $\mathcal{L}$-Universality

We have seen that these all have efficient solutions for regular languages and DFAs (but NFAs cause problems).

Easy Decision Problems

Problem:	**CFG Emptiness**
Instance:	A CFG G.
Question:	Is $\mathcal{L}(G) = \emptyset$?

Problem:	**CFG Finiteness**
Instance:	A CFG G.
Question:	Is $\mathcal{L}(G)$ finite?

Lemma

Both CFG Emptiness and CFG Finiteness are decidable in linear time.

Proof

Inductively mark the variables of the grammar: first all A such that there is a production $A \to x \in \Sigma^\star$. Then all with productions $B \to \alpha$ where all the variables on the RHS are already marked.

Then the language is non-empty iff S is marked.

Now remove all unmarked variables and associated productions. The language is infinite if there is a derivation

$$A \Rightarrow \alpha\, A\, \beta$$

where $\alpha\beta \neq \varepsilon$ and A appears in some sentential form.

This can be tested easily using standard graph algorithms.

□

Hard Decision Problems

Problem:	**CFG Equality**
Instance:	Two CFGs G_1 and G_2.
Question:	Is $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?

Problem:	**CFG Inclusion**
Instance:	Two CFGs G_1 and G_2.
Question:	Is $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$?

Problem:	**CFG Universality**
Instance:	A CFG G.
Question:	Is $\mathcal{L}(G) = \Sigma^\star$?

Problem:	**CFG Regularity**
Instance:	A CFGs G.
Question:	Is $\mathcal{L}(G)$ regular?

The Bad News

Theorem

For context free languages, Universality, Equality, Inclusion and Regularity are all undecidable.

Proof.

We will only show the argument for Universality.

The idea is to encode accepting computations of a Turing machine as strings

$$\# C_0 \# C_1 \# \ldots \# C_n \#$$

where each C_i codes a single configuration of the TM.

Thus C_i is a string in $\Gamma^\star Q \Gamma^\star$ where Γ is the tape alphabet of the TM.

Call this language Cmp.

Technical Problems

We would like this to be a context free language, but there is no hope: Cmp is too complicated, we have to check that C_i really leads to C_{i+1} in one step for all i.

But Cmp', the complement of Cmp looks more promising: first, we can ditch all strings not of the form

$$(\# \Gamma^\star Q \Gamma^\star)^\star \#$$

a regular language.

Second, we can guess a single position i where things go wrong and then verify that C_i and C_{i+1} are not two consecutive configurations.

Alas, there is a problem: this problem is still too hard for a context free grammar. Try to build one.

Smoke and Mirrors

Use the encoding

$$\# C_0 \# C_1^{\mathsf{op}} \# C_2 \# C_3^{\mathsf{op}} \# \ldots \# C_n^{?} \#$$

So, we reverse every other configuration. This is still a perfectly good representation of a computation, albeit an unconventional one.

But now we can write a CFG that generates Cmp', the collection of all strings that are not of this form.

But Cmp' is universal iff the Turing machine never halts, an undecidable problem.

□

Quoi?

Suppose you have two consecutive configurations of a Turing machine $aab\,p\,baba$ and $aab\,q\,caba$ (overwrite b by c, goto state q).

Then

$$\ldots \# aab\,p\,baba \# abac\,q\,baa \# \ldots$$

is part of a coded computation.

To produce the complement language, we can generate the two configurations starting from the $\#$ and then make a mistake somewhere (say put $q' \neq q$ instead of q).

From Grammars to Machines

The question arises whether there is some natural class of machines that accept exactly the CFLs.

It is clear from our examples that the machines must have some type of unbounded memory. But we have to be careful not to add too much memory:

- $\{ a^i b^i \mid i \geq 0 \}$ is CF
- $\{ a^i b^i c^i \mid i \geq 0 \}$ is not CF

So, simply adding counters will not work. Also, counters do not really help with Dyck language D_k.

This is a white lie. With clever coding, counters can be used to simulate Turing machines, but that's not what we have in mind here.

Stacks

A little experimentation shows that adding a stack to a finite state control might work. So we need to have the finite state control interact with the stack.

This can be done by three types of transitions:

state	action	next state
p	scan a	q
p	pop A	q
p	push A	q

So the first type is just like an ordinary FSM.

Example

For example, for $L = \{ a^i b^i \mid i \geq 0 \}$ we could use a stack symbol A; whenever a symbol a is scanned we push A onto the stack. When the first b is scanned start to pop, one A for each b. Accept iff the stack becomes empty exactly when the last b is scanned. The machine could be described by the following table:

q_0	scan a	q_a
q_a	push A	q_0
q_0	scan b	q_b
q_b	pop A	q
q	scan b	q_b

Formalization

A stack can be modeled by a string over some special alphabet Γ. E.g., a push would look like this:

$$\alpha \rightsquigarrow A\alpha$$

As a practical matter, one usually combines all three operations scan, pop, push into one single step.

Using standard state-splitting one can translate this supper-step into a sequence of our steps.

Pushdown Automata

Definition

A pushdown automaton (PDA) over the alphabet Σ is a quintuple $M = \langle Q, \Sigma, \Gamma, \delta; q_0, F \rangle$ where Q is a finite set, Γ an alphabet, $q_0 \in Q$, $F \subseteq Q$, and

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathfrak{P}_0(Q \times \Gamma^\star)$$

Here $\mathfrak{P}_0(Z)$ denotes the set of finite subsets of Z.

Σ is the input alphabet , Γ the stack alphabet, Q the set of states, q_0 the initial state , F the set of final states , and δ the transition function of M.

Note that in $\delta(p, a, Z)$ both a and Z are allowed to be ε, and $\delta(p, a, Z)$ may have cardinality larger than 1, so these machines are quite nondeterministic.

Acceptance

Intuitively, a PDA M "accepts" a string $x \in \Sigma^\star$ iff starting at the initial state and with empty stack, there is a sequence of transitions that scans the whole string and leaves the machine in a final state and with empty stack.

Here is a precise definition of acceptance. This is déjà vu all over again: we used the same approach for Turing machines, finite state machines, grammars, ...

A configuration of PDA M is an element $\langle q, x, u \rangle$ of $Q \times \Sigma^\star \times \Gamma^*$.

q is the *current state*, x is the *unscanned input* and u is the *current stack content*. Stack $u = u_1 \dots u_k$ indicates that u_1 is on *top* of the stack.

Computations

Define the next configuration relation $\vdash$ as follows:

$$\langle\, p, ax, Xu \,\rangle \vdash^1 \langle\, q, x, vu \,\rangle \quad \text{if } (q, v) \in \delta(p, a, X)$$

Expand to $\vdash^k$ and $\vdash$ as usual.

Unlike with finite automata there are several natural notions of acceptance for PDAs. Let M be a PDA and $x \in \Sigma^\star$.

M accepts x by empty stack and final state if $C_x \vdash \langle\, q, \varepsilon, \varepsilon \,\rangle$ where $q \in F$.

Similarly, M accepts x by final state iff $C_x \vdash \langle\, q, \varepsilon, u \,\rangle$ where $q \in F$, $u \in \Gamma^*$.

The corresponding acceptance languages are defined by:

$$\mathcal{L}(M) = \{\, x \mid M \text{ accepts } x \text{ by empty stack and final state} \,\}$$
$$\mathcal{L}_f(M) = \{\, x \mid M \text{ accepts } x \text{ by final state} \,\}$$

No Difference

Note that in general, for a fixed machine M, the two acceptance languages $\mathcal{L}(M)$ and $\mathcal{L}_f(M)$ are usually different. However, the class of languages defined by the two acceptance methods is the same:

Lemma

The two notions of acceptance coincide in the following sense. Let L be a language over Σ. Then the following are equivalent:

- $L = \mathcal{L}(M_1)$ *for some PDA* M_1
- $L = \mathcal{L}_f(M_2)$ *for some PDA* M_2

Proof. Straightforward simulation. □

Example: Dyck Languages

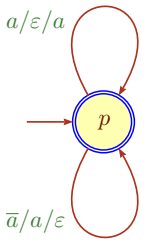
The language $D_k \subseteq (\Sigma \cup \overline{\Sigma})^\star$ is easy to recognize by a PDA.

Let $\Gamma = \Sigma$ and use $Q = F = \{p\}$ where p is the initial state. Transition function:

$$\delta(p, a, \varepsilon) = (p, a)$$
$$\delta(p, \overline{a}, a) = (p, \varepsilon)$$

This machine is deterministic in the sense that for any input there is only one possible computation.

Boring Picture



$a/A/B$ means: scan a, pop A and push B.

Example: Palindromes with Center Marker

$$L_{\text{palcm}} = \{\, x \# x^{\text{op}} \mid x \in \{a, b\}^\star \,\}$$

Here $\Sigma = \{a, b, \#\}$, $\Gamma = \{a, b\}$, $Q = \{p, q\}$, p is initial and q is final. Transition function:

$$\delta(p, \sigma, \varepsilon) = (p, \sigma)$$
$$\delta(p, \#, \varepsilon) = (q, \varepsilon)$$
$$\delta(q, \sigma, \sigma) = (q, \varepsilon)$$

Here $\sigma \in \{a, b\}$.

Again, this machine is deterministic: the $\#$ triggers the transition from storing state p to matching state q.

A Picture

Example: Even Length Palindromes

$$L_{\text{palel}} = \{\, x\, x^{\text{op}} \mid x \in \{a,b\}^\star \,\}$$

This time $\Gamma = \Sigma = \{a,b\}$, Q is as above. Transition function:

$$\begin{aligned}\delta(p,\sigma,\varepsilon) &= (p,\sigma)\\ \delta(p,\varepsilon,\varepsilon) &= (q,\varepsilon)\\ \delta(q,\sigma,\sigma) &= (q,\varepsilon)\end{aligned}$$

This machine is nondeterministic: it has to "guess" when to switch from push to pop mode.

A Similar Picture

Almost the same, but this time the transition p to q is nondeterministic: we have to guess whether to keep on pushing stuff onto the stack, or whether to start popping.

As with finite state machines, we do not know anything about the length of the input ahead of time.

Deterministic PDA

Formally, a PDA is deterministic (DPDA) if it has at most one computation on each input. A CFL is called deterministic if it accepted by a DPDA.

It is not hard to express this in terms of the transition function (exercise).

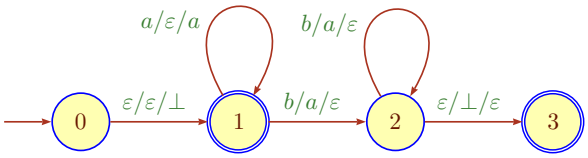
Example

$L = a^\star \cup \{\, a^i b^i \mid i \geq 1 \,\}$ is deterministic.

Here is a DPDA that accepts L by final state (final states are $\{1,3\}$). δ is defined by:

$$\begin{aligned}\delta(0,\varepsilon,\varepsilon) &= (1,\bot)\\ \delta(1,a,X) &= (1,aX)\\ \delta(1,b,a) &= (2,\varepsilon)\\ \delta(2,b,a) &= (2,\varepsilon)\\ \delta(2,\varepsilon,\bot) &= (3,\varepsilon)\end{aligned}$$

The Diagram



A standard trick: place a marker $\bot$ at the bottom of the stack before the actual computation starts.

The Characterization Theorem

Theorem

A language is context free if, and only if, it can be recognized by a pushdown automaton.

Proof.

Given a CFG $G = \langle V, \Sigma, \mathbb{P}, S \rangle$ define $Q = \{q_0, q\}$, $\Gamma = V \cup \Sigma$, $F = \{q\}$. The transition function of M_G is defined by:

$$\begin{aligned}\delta(q_0,\varepsilon,\varepsilon) &= (q,S)\\ \delta(q,\varepsilon,A) &= \{\, (q,\alpha^{\text{op}}) \mid A \to \alpha \text{ in } \mathbb{P} \,\}\\ \delta(q,a,a) &= (q,\varepsilon)\end{aligned}$$

Here $A \in V$ and $a \in \Sigma$.

Proof, Cont'd

Recall that a derivation is leftmost if the next substitution is always performed on the leftmost syntactic variable (this is just a natural way to organize the work).

By induction one can then show that every leftmost derivation of x in G corresponds to an accepting computation of M_G on input x.

$$A \Rightarrow_{\text{left}} w\alpha \text{ iff } \langle q, w, A \rangle \vdash \langle q, \varepsilon, \alpha \rangle .$$

Here $w \in \Sigma^\star$, $A \in V$, $\alpha\Gamma^\star$.

For the opposite direction, suppose we have a PDA $\langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ that accepts some language L.

We need to concoct a grammar that simulates the workings of the PDA.

Proof, Cont'd

Define $V = Q_0 \times Q \cup \{S\}$, where Q_0 are all the target states of push operations.

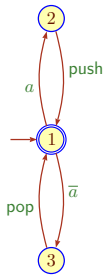
R1	$[p,p] \to \varepsilon$	
R2	$[p,q_2] \to [p,q_1]s$	if q_1 scan s q_2
R3	$[p,q_2] \to [p,p_1][p_2,q_1]$	if p_1 push A p_2, q_1 pop A q_2
R4	$S \to [q_0,q]$	if $q \in F$.

The idea is that variable $[p,q]$ derives all strings obtained by a computation from p to q.

The correctness proof requires tedious inductions, we'll skip.

□

Example: D_1



by R4	$S \to [1,1]$
by R1	$[1,1] \to \varepsilon$
by R3	$[1,1] \to [1,2][1,3]$
by R2	$[1,2] \to [1,1]a$,
by R2	$[1,3] \to [1,1]\overline{a}$

Rename

$$S \to A$$
$$A \to BC \mid \varepsilon$$
$$B \to Aa$$
$$C \to A\overline{a}$$

Looks much better. Further simplification gets us to the unambiguous grammar

$$A \to AaA\overline{a} \mid \varepsilon$$

Consequences

Using the machine model one can now prove the results claimed last time:

Lemma

Suppose L is a CFL and R is regular. Then $L \cap R$ is also context free.

Theorem (Chomsky-Schützenberger)

Every context free language $L \subseteq \Sigma^\star$ has the form $L = h(D \cap R)$ where D is a Dyck language, R is regular and h is a homomorphism.

Proofs

The lemma is easy.

For the theorem, the basic idea is to use terminals of the form

$$(p,a,A,q) \in Q \times \Sigma \times \Gamma \times Q$$

to express push operations, and to have them match with $(p',b,\overline{A},q')$ representing a pop operation. The Dyck language makes sure that every A that is pushed must be popped later, and that we are really using stack discipline.

The regular language controls things like matching states in two consecutive letters, say $(p,a,A,q)(q,b,B,q')$, and handles initial and final conditions.

The homomorphism erases everything except the actual input.

And Stop

It might be tempting to think that one can use some analogue of Rabin-Scott to eliminate nondeterminism from PDA – after all, this works fine for finite state machines and Turing machines.

> But it does not work here: deterministic PDA are strictly weaker than general PDA.

For example, the language of palindromes (without center marker) cannot be recognized by any deterministic PDA. Make sure to think carefully about why determinization fails in this case.

Deterministic CFL

A deterministic CFL is a language that is accepted by some deterministic PDA. These are quite different from arbitrary CFL:

- they are closed under complementation but
- they are not closed under union.

Theorem (Senizergues 2001)

Equality for deterministic CFL is decidable.

The Bible

M. H. Harrison
Introduction to Formal Language Theory
Addison-Wesley 1978

A hard read, and sometimes the proofs are a bit overly formal, but well worth it.