Σ^* : set of all strings obtained by concatenating zero or more symbols from Σ.

- empty string : choose no symbol from Σ, denoted by λ or ϵ or $\perp$
- $|\lambda| = 0$
- $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$
- Although Σ is a finite set, both Σ^+ and Σ^* are infinite sets.

Let $\Sigma = \{a, b\}$

- String reversal : $(abaab)^R = baaba$; Note $(\sigma^R)^R = \sigma$
- String x is a prefix of the string σ if $\exists y$ such that $x \circ y = \sigma$
- Prefixes of 'aab' $= \{a, aa, aab\}$
- String x is a suffix of the string σ if $\exists y$ such that $y \circ x = \sigma$
- Suffixes of 'aab' $= \{aab, ab, b\}$

A grammar G is a quadruple $G = \langle V, T, S, P \rangle$

- V : finite set of variables/nonterminals
- T : set of terminals
- $S \in V$: start symbol
- P : set of 'production rules'

Finite Automata

- A finite automata is a mathematical model of a system with discrete inputs and outputs
- system can have finite no. of internal configs / states

$M = \langle Q, \Sigma, \delta, q, F \rangle$

- Q : finite set of **internal states**
- Σ : finite set of **input alphabet**
- $\delta : Q \times \Sigma \to Q$: is the **transition function**
- $q \in Q$: **initial state**
- $F \subseteq Q$ is the set of **final/accept states**

[illegible]

- A language A is regular iff it is accepted by some FA
- Regular sets are closed under $\cup, \cap, \neg, \circ, ^*$

Product Machines

Given $M_1 = \langle Q_1, \Sigma, \delta_1, q_1, F_1 \rangle$, $M_2 = \langle Q_2, \Sigma, \delta_2, q_2, F_2 \rangle$ let $M_3 = \langle Q_3, \Sigma, \delta_3, q_3, F_3 \rangle$ be defined as follows.

- $Q_3 = Q_1 \times Q_2$,
- $\delta_3 : Q_3 \times \Sigma \to Q_3$ where $\delta_3((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$.
- $F_3 = F_1 \times F_2$
- $q_3 = (q_1, q_2)$

$L(M_3) = L(M_1) \cap L(M_2)$: Prove that $\forall \sigma \in \Sigma^*$, $\forall (p, q) \in Q_3$, $\hat{\delta}_3((p, q), \sigma) = (\hat{\delta}_1(p, \sigma), \hat{\delta}_2(q, \sigma))$ by induction on length of input string $\sigma \in \Sigma^*$. Use $\hat{\delta}_3$ to define $L(M_3)$ as desired.

With $F_3 = (F_1 \times Q_2) \cup (Q_1 \times F_2)$, we have an automata for $\cup$

NFA

Consider $L = \{x \in \{0,1\}^* \mid \text{2nd symbol from the right is 1}\}$

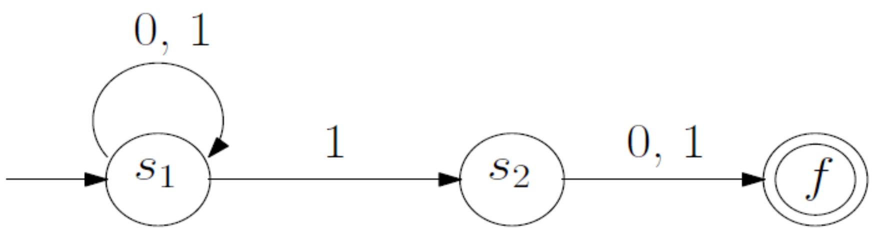


Figure: NFA for L

- the movement from s_1 when input is '1' is nondeterministic
- For an input string, NFA will create a 'computation tree' rather than a 'computation sequence' in case of DFA
- An NFA accepts a string if any one of the paths in the tree leads to some accept state

NFA to DFA

Compute all possible **subsets** :
$\{\{\}, \{s_1\}, \{s_2\}, \{f\}, \{s_1, s_2\}, \{s_2, f\}, \{s_1, f\}, \{s_1, s_2, f\}\}$

- compute single step reachability among subsets for i/p-s 0,1 with same initial state

δ	0	1
$\{\}$	$\{\}$	$\{\}$
$\{s_1\}$	$\{s_1\}$	$\{s_1, s_2\}$
$\{s_2\}$	$\{f\}$	$\{f\}$
$\{f\}$	$\{\}$	$\{\}$
$\{s_1, s_2\}$	$\{s_1, f\}$	$\{s_1, s_2, f\}$
$\{s_1, f\}$	$\{s_1\}$	$\{s_1, s_2\}$
$\{s_2, f\}$	$\{f\}$	$\{f\}$
$\{s_1, s_2, f\}$	$\{s_1, f\}$	$\{s_1, s_2, f\}$

States $\{s_2, f\}$, $\{s_2\}$, $\{f\}$, $\{\}$, are *unreachable*

Reduced transition table with unreachable states removed

δ	0	1
$\{s_1\}$	$\{s_1\}$	$\{s_1, s_2\}$
$\{s_1, s_2\}$	$\{s_1, f\}$	$\{s_1, s_2, f\}$
$\{s_1, f\}$	$\{s_1\}$	$\{s_1, s_2\}$
$\{s_1, s_2, f\}$	$\{s_1, f\}$	$\{s_1, s_2, f\}$

DFA should have 4 states :
$\{a_1, a_2, a_3, a_4\} = \{\{s_1\}, \{s_1, s_2\}, \{s_1, f\}, \{s_1, s_2, f\}\}$
Initial state is same.

NFA formal definition

$(Q, \Sigma, \Delta, S, F)$

- Q : set of states
- Σ : Alphabet
- Δ : transition relation (function) defined as

  $$\Delta : Q \to 2^Q$$

  $2^Q = \{A | A \subseteq Q\}$: captures multiple possible reactions to an input
- $S \subseteq Q$: **set of initial states**

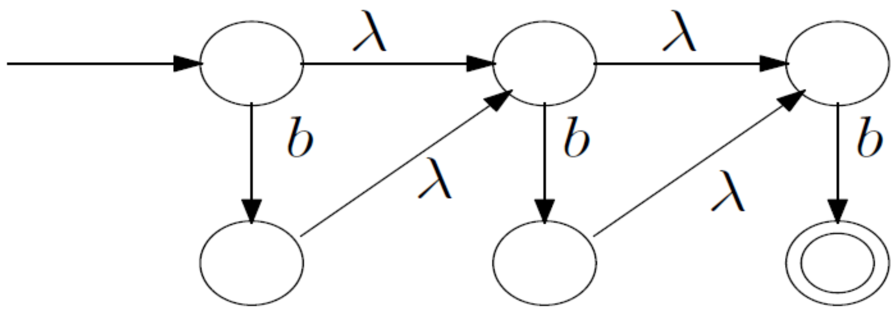
Formalize NFA $\Leftrightarrow$ DFA using $\hat{\Delta}$

Given NFA $N = (Q_N, \Sigma, \Delta_N, S_N, F_N)$, the **equivalent**[1] DFA $M = (Q_M, \Sigma, \delta_M, s_M, F_M)$ is defined as follows.

- $Q_M = 2^{Q_N}$
- $\delta_M(A, a) = \hat{\Delta}_N(A, a)$
- $s_M = S_N$
- $F_M = \{A \subseteq Q_N \mid A \cap F_N \neq \perp\}$

λ transitions

- NFAs can have transitions like $s_1 \xrightarrow{\lambda} s_2$
- The machine can make a move without scanning ANY input.
- NFA + λ transitions $\equiv$ NFA w/o λ transitions, DFA
- $\Delta : Q \times \Sigma \cup \lambda \rightarrow 2^Q$



$L = \{b, bb, bbb\}$. With λ transitions it becomes trivial to construct FA for L^* given FA for L

NFA to DFA : handle λ transitions

λ-closure of set S of states in NFA = collection of states reachable from any state in S using only λ transitions = λ-close(S) say.

- Compute λ-close(S) where S is NFA initial state - this is DFA initial state I
- Keep computing λ-closures for every reachable state

Patterns

A *Pattern* captures a family of strings (like FA) : language of the pattern

- $a \in \Sigma : \; L(a) = \{a\}$: matches a single symbol
- $\epsilon : \; \epsilon = \{\epsilon\}$: matches the null string
- $\phi : \; L(\phi) = \phi$: matches empty set ϕ
- $\sharp : \; L(\sharp) = \Sigma$: any symbol
- $@ : \; L(@) = \Sigma^*$: any string

Above *atomic* patterns can be operated with unary $^*, ^+, \neg$ or connected by $\cup/+, \cap, \circ$ (usually not written, kept silent) to generated other valid patterns

- x matches $\alpha + \beta$ if x matches either α or β : $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$, similarly you have other rules
- $L(\alpha \cap \beta) = L(\alpha) \cap L(\beta)$, $L(\alpha\beta) = L(\alpha)L(\beta)$, $L(\neg\alpha) = \neg L(\alpha) = \Sigma^* - Ł(\alpha)$,
- Can define $L(\alpha^*)$, $L(\alpha^+)$

Regular Grammars

Regular Language (set) $\equiv$ FA $\equiv$ Regular Grammar

- A grammar $G = (V, \Sigma, S, P)$ is **right linear** if all productions are of the form

  $$A \to xB,\ A \to x$$

  where $A, B \in V$, $x \in \Sigma^*$
- A grammar $G = (V, \Sigma, S, P)$ is **left linear** if all productions are of the form

  $$A \to Bx,\ A \to x$$

  where $A, B \in V$, $x \in \Sigma^*$
- A regular grammar is one of the two

Strictly Right Linear Grammars

A grammar $G = (V, \Sigma, S, P)$ is **strictly right linear** if all productions are of the form

$$A \to xB,\ A \to \lambda$$

where $A, B \in V$, $x \in \Sigma \cup \{\lambda\}$

- Any derivation of a word w from S has the form $S \Rightarrow x_1A_1 \Rightarrow x_1x_2A_2 \Rightarrow \cdots \Rightarrow x_1 \cdots x_nA_n \Rightarrow x_1 \cdots x_n$
- some x_i can be λ, connection with NFA ??

♠ For any right-linear grammar G there exists a strictly right-linear grammar H such that $L(G) = L(H)$

Strictly Right Linear Grammars

If G is a strictly right-linear grammar, then $L(G)$ is regular

- Given $G = (V, \Sigma, P, S)$, construct NFA $N = (V, \Sigma, \delta, S, F)$. The set of states is simply the set of non-terminals of G. The start state corresponds to the start variable.
- $\delta(A, x) = \{B \mid A \to xB \in P\}$, $F = \{C \mid C \to \lambda \in P\}$

To show $L(G) = L(N)$

- $L(G) \subseteq L(N)$: by induction on the structure of the derivation
- $L(N) \subseteq L(G)$: by induction on the structure of the computation

♠ A language A is regular **if and only if** there is a strictly right-linear grammar G such that $L(G) = A$

Closure properties of Regular Languages

Regular languages are closed under

- Union
- Intersection
- Set Difference
- Concatenation
- Kleene Closure
- Reversal
- Homomorphism
- Inverse Homomorphism

♠ If a language is not regular then neither is its complement :)
♠ Considering Σ, Γ as two alphabets, a function $h : \Sigma \mapsto \Gamma^*$ is a homomorphism iff $\forall \sigma_1, \cdots, \sigma_n \in \Sigma, \quad h(\sigma_1 \cdots \sigma_n) = h(\sigma_1) \cdots h(\sigma_n)$. Now, if $L \subseteq \Sigma^*$ is regular than so is $h(L)$. Similarly, if $L \subseteq \Gamma^*$ is regular than so is $h^{-1}(L)$.

can be inserted and the resulting string is still accepted.

Theorem 11.1 **(Pumping lemma)** *Let A be a regular set. Then the following property holds of A:*

(P) *There exists $k \geq 0$ such that for any strings x, y, z with $xyz \in A$ and $|y| \geq k$, there exist strings u, v, w such that $y = uvw$, $v \neq \epsilon$, and for all $i \geq 0$, the string $xuv^iwz \in A$.*

For this purpose we usually use it in its contrapositive form:

Theorem 11.2 **(Pumping lemma, contrapositive form)** *Let A be a set of strings. Suppose that the following property holds of A.*

($\neg$P) *For all $k \geq 0$ there exist strings x, y, z such that $xyz \in A$, $|y| \geq k$, and for all u, v, w with $y = uvw$ and $v \neq \epsilon$, there exists an $i \geq 0$ such that $xuv^iwz \notin A$.*

Then A is not regular.

and a demon. You want to show that A is nonregular, and the demon wants to show that A is regular. The game proceeds as follows:

1. The demon picks k. (If A really is regular, the demon's best strategy here is to pick k to be the number of states of a DFA for A.)
2. You pick x, y, z such that $xyz \in A$ and $|y| \geq k$.
3. The demon picks u, v, w such that $y = uvw$ and $v \neq \epsilon$.
4. You pick $i \geq 0$.

You win if $xuv^iwz \notin A$, and the demon wins if $xuv^iwz \in A$.

Here is an algorithm for computing the collapsing relation $\approx$ for a given DFA M with no inaccessible states. The algorithm will mark (unordered) pair of states $\{p, q\}$. A pair $\{p, q\}$ will be marked as soon as a reason is discovered why p and q are *not* equivalent.

1. Write down a table of all pairs $\{p, q\}$, initially unmarked.
2. Mark $\{p, q\}$ if $p \in F$ and $q \notin F$ of vice versa.
3. Repeat the following until no more changes occur:
   If there exists an unmarked pair $\{p, q\}$ such that $\{\delta(p, a), \delta(q, a)\}$ is marked for some $a \in \Sigma$, then mark $\{p, q\}$.
4. When done, $p \approx q$ iff $\{p, q\}$ is not marked.

Myhill-Nerode Relations

Let $R \subseteq \Sigma^*$ be regular with DFA $M = (Q, \Sigma, \delta, s, F)$ for R. M does not have any unreachable states. A relation $\equiv_M$ on Σ^* defined as

- $x \equiv_M y \Leftrightarrow \hat{\delta}(s, x) = \hat{\delta}(s, y)$

$\equiv_M$ is an equivalence relation. Other properties of $\equiv_M$

1. $\forall x, y \in \Sigma^*, a \in \Sigma,\ x \equiv y \Rightarrow xa \equiv ya$: right congruence (show this)
2. $\equiv_M$ refines R : $x \equiv_M y \Rightarrow (x \in R \Leftrightarrow y \in R)$ – every $\equiv_M$-class has either all its elements in R or none of its elements in R, i.e. R is a union of $\equiv_M$-classes
3. The no. of $\equiv_M$ classes is finite ($=$ no. of states in M ?)

Myhill-Nerode Relations

Any equivalence relation on Σ^* which is a right congruence of finite index refining a regular set R is called a Myhill-Nerode Relation

- Just like $M \rightarrow \equiv_M$ we can $\equiv \rightarrow M_\equiv$

Let $\equiv$ be an arbitrary Myhill-Nerode Relation on Σ^* for some $R \subseteq \Sigma^*$, i.e. $\equiv$ is some equivalence Relation on Σ^* which is also right congruence of finite index refining a regular set R

Formally, a *context-free grammar* (CFG) is a quadruple
$G = (N, \Sigma, P, S)$
where

- N is a finite set (the *non-terminal symbols*),
- Σ is a finite set (the *terminal symbols*) disjoint from N,
- P is a finite subset of $N \times (N \cup \Sigma)^*$ (the *productions*), and
- $S \in N$ (the *start symbol*)

It helps to restrict formal grammars to ‘standard forms’. You can write algorithms that work on grammars assuming this standard

- A CFG is in *Chomsky Normal Form* (CNF) if all productions are of the form
  1. $A \rightarrow BC$ or
  2. $A \rightarrow a$

  where $A, B, C \in N$ and $a \in \Sigma$. Note that CNF form grammars cannot generate ϵ
- For any CFG G, there is a CFG G' in Chomsky Normal Form such that

$$L(G') = L(G) - \{\epsilon\}$$

Create grammar for ϵ free language

- For any CFG $G = (N, \Sigma, P, S)$, there is a CFG G' with no ϵ or unit productions $(A \to B)$ such that

$$L(G') = L(G) - \{\epsilon\}$$

.

- Proof: Let $\hat{P}$ be the smallest set of productions containing P and closed under the rules
  1. if $A \to \alpha B \beta$ and $B \to \epsilon$ are in $\hat{P}$; then $A \to \alpha\beta$ is in $\hat{P}$
  2. if $A \to B$ and $B \to \gamma$ are in $\hat{P}$, then $A \to \gamma$ is in $\hat{P}$
- Point to note : $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$

G' to CNF

- For each terminal $a \in \Sigma$ introduce a new nonterminal A_a and production $A_a \to a$ and replace all occurrences of a on the right hand side of old productions (except productions of the form $B \to a$) with A_a
- Now all productions are of the form
  1. $A \to a$ or
  2. $A \to B_1 B_2 \cdots B_k, k \geq 2$

  where the B_i are nonterminals.

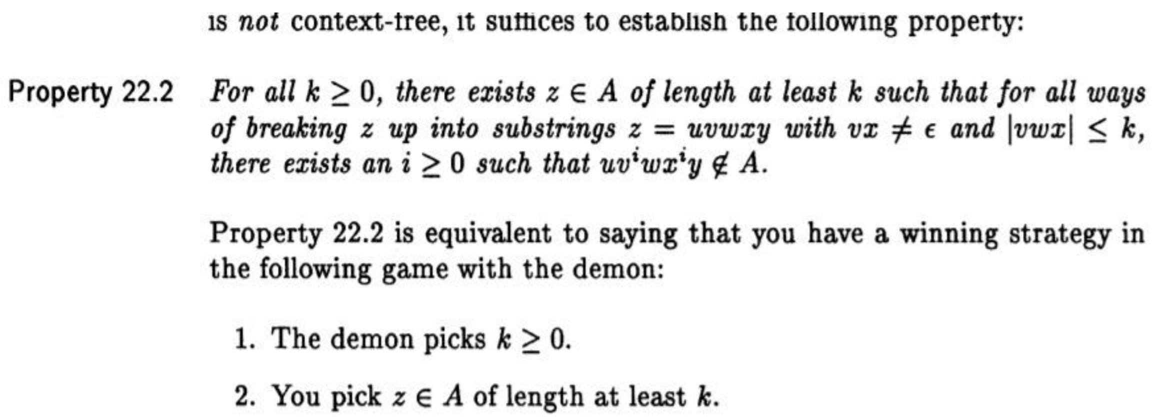
- For any production

$$A \to B_1 B_2 \cdots B_k$$

  with $k \geq 3$, introduce a new nonterminal C and replace this production with the two productions
  1. $A \to B_1 C$ and
  2. $C \to B_2 B_3 \cdots B_k$

  until all right hand sides are of length at most 2.

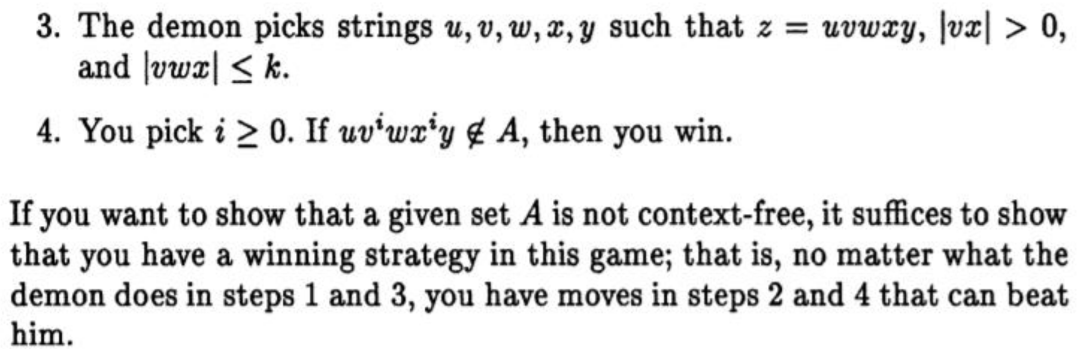
- A grammar G is ambiguous if $\exists w \in L(G)$ for which
  - there are two or more distinct derivation/parse trees, or
  - there are two or more distinct leftmost derivations, or
  - there are two or more distinct rightmost derivations.
- Ambiguity is a property of a grammar, and it is usually (but not always) possible to find an equivalent unambiguous grammar.
- An *inherently ambiguous language* is a language for which no unambiguous grammar exists.


is *not* context-free, it suffices to establish the following property:

Property 22.2 *For all $k \geq 0$, there exists $z \in A$ of length at least k such that for all ways of breaking z up into substrings $z = uvwxy$ with $vx \neq \epsilon$ and $|vwx| \leq k$, there exists an $i \geq 0$ such that $uv^iwx^iy \notin A$.*

Property 22.2 is equivalent to saying that you have a winning strategy in the following game with the demon:

1. The demon picks $k \geq 0$.
2. You pick $z \in A$ of length at least k.



3. The demon picks strings u, v, w, x, y such that $z = uvwxy$, $|vx| > 0$, and $|vwx| \leq k$.
4. You pick $i \geq 0$. If $uv^iwx^iy \notin A$, then you win.

If you want to show that a given set A is not context-free, it suffices to show that you have a winning strategy in this game; that is, no matter what the demon does in steps 1 and 3, you have moves in steps 2 and 4 that can beat him.

$A = \{a^nb^na^n | n \geq 0\}$ is not context free

- Adversary picks k in step 1
- You pick $z = a^kb^ka^k$ such that $z \in A$ and $|z| = 3k \geq k$
- Adversary picks $z = uvwxy$ such that $vx \neq \epsilon$, and $|vwx| \leq k$
- You pick $i = 2$ and in all cases you can ensure $uv^2wx^2y \notin A$
  1. either v or x contain at least one a and at least one b
  2. v and x contains only a's or only b's
  3. one of v or x contains only a's and the other contains only b's

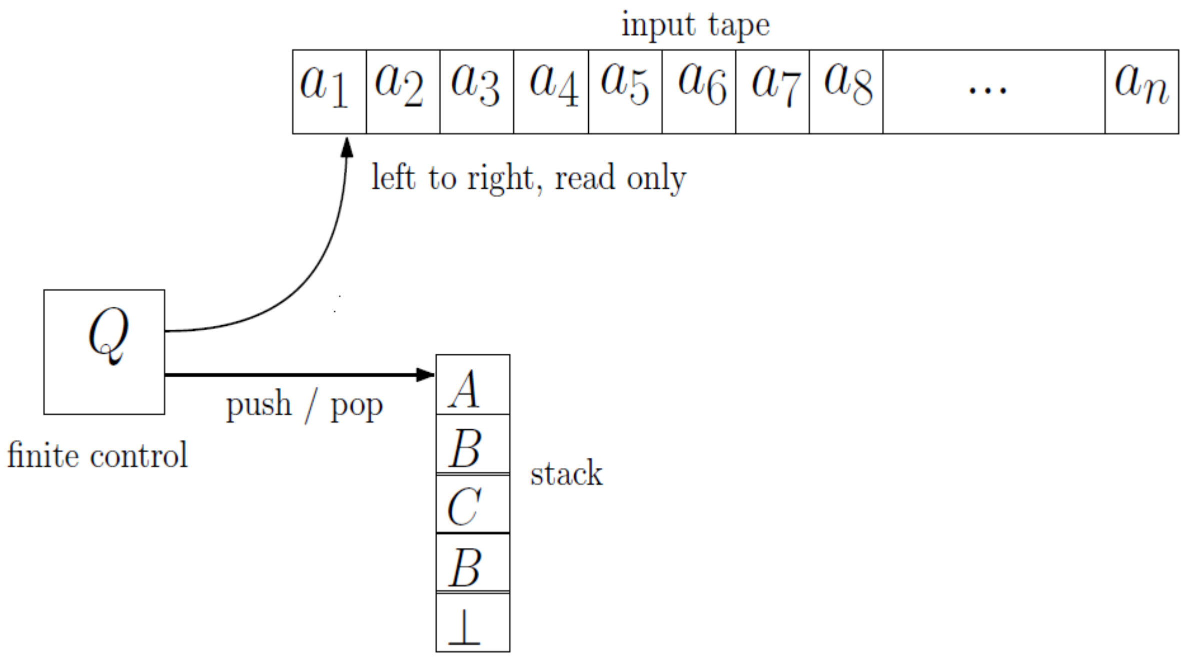


Figure: Non deterministic Pushdown Automata

A non deterministic PDA is a 7 tuple $M = (Q, \Sigma, \Gamma, \delta, s, \perp, F)$,

- Q is a finite set (the states),
- Σ is a finite set (the input alphabet),
- Γ is a finite set (the stack alphabets),
- $s \in Q$ (the start state),
- $\perp \in \Gamma$ (the initial stack symbol), and
- $F \subseteq Q$(the final or accept states),
- $\delta \subseteq (Q \times (\Sigma \cup \varepsilon) \times \Gamma) \times (Q \times \Gamma^*), \delta$: NPDA has ϵ transitions (can move w/o input)

$((p, a, A), (q, B_1 \cdots B_k)) \in \delta$: from a state p while reading some input symbol a with some stack top element A, the PDA moves to another state q, pops A, pushes $B_1 \cdots B_k$ (B_k first)

- The start configuration on input x is $(s, x, \perp)$. That is, the machine always starts in its start state s with its read head pointing to the leftmost input symbol and the stack containing only the symbol $\perp$.
- The next configuration relation $\xrightarrow[M]{1}$ describes how the machine can move from one configuration to another in one step.

Acceptance

- by **final state** : $(s, x, \perp) \xrightarrow[M]{*} (q, \epsilon, \gamma)$: get to some $q \in F$ when string exhausted
- by **empty stack** : $(s, x, \perp) \xrightarrow[M]{*} (q, \epsilon, \epsilon)$: pop stack bottom element when string exhausted

Both kinds of M/Cs can be converted to an equivalent NPDA M that has a single final state t and M can empty its stack after it enters state t.

- For each production $A \rightarrow cB_1B_2.....B_k$ in P
- The transition relation δ is defined as $((q, c, A), (q, B_1B_2....B_k))$
  1. δ has one transition of each production of G
  2. When in state q scanning input symbol c with A on top of the stack, pop A off the stack, push $B_1B_2.....B_k$ onto the stack (B_k first) and enter in q state
  3. For $c = \epsilon$, when in state q with A on top of the stack, without scanning an input symbol, pop A off the stack, push $B_1B_2.....B_k$ onto the stack (B_k first) and enter in q state

Closure Properties of CFL

Context-free languages are closed under the following operations:

1. Union
2. Concatenation
3. Kleene closure
4. Homomorphism
5. Substitution
6. Inverse-homomorphism
7. Reverse

- CFL are closed under intersection with regular sets.
- CFL are not closed under intersection.

Linear Grammars n languages

A linear grammar is a CFG that has at most one nonterminal in the R.H.S. of each of its productions.

- Consider $\{a^i b^i \mid i \geq 0\}$
- $P = \{S \rightarrow aSb, S \rightarrow \epsilon\}$

We call such languages as linear languages. Recall, the special cases

- **left-linear grammars** : a type of linear grammar with R.H.S. nonterminals strictly at the left ends;
- **right-linear grammars** : a type of linear grammar with R.H.S. nonterminals strictly at the right ends;

CYK Algorithm contd.

We'll run the algorithm on the string

$$\begin{array}{ccccccccccccc} | & a & | & a & | & b & | & b & | & a & | & b & | \\ 0 & & 1 & & 2 & & 3 & & 4 & & 5 & & 6 \end{array}$$

For $0 \leq i < j \leq n$, x_{ij} denote the substring of x between lines i and j. Build a table T with $\binom{n}{2}$ entries, one for each pair i, j.

0						
–	1					
–	–	2				
–	–	–	3			
–	–	–	–	4		
–	–	–	–	–	5	
–	–	–	–	–	–	6

$T_{i,j}$ refers to substring $x_{i,j}$.

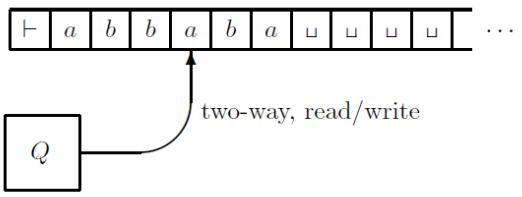
We will fill each entry $T_{i,j} \in T$ with the set of nonterminals of G that produce substring $x_{i,j}$. We start with substring of length 1. For each substring $c = x_{i,i+1}$, if there is a production $X \to c \in G$, we write X in $T_{i,j}$.

0						
A	1					
–	A	2				
–	–	B	3			
–	–	–	B	4		
–	–	–	–	A	5	
–	–	–	–	–	B	6

Informal Description of Turing Machines

We describe here a deterministic, one-tape Turing machine. This is the standard off-the-shelf model. There are many variations, apparently more powerful or less powerful but in reality not. We will consider some of these in §3.

A TM has a finite set of states Q, a semi-infinite tape that is delimited on the left end by an endmarker $\vdash$ and is infinite to the right, and a head that can move left and right over the tape, reading and writing symbols.



The input string is of finite length and is initially written on the tape in contiguous tape cells snug up against the left endmarker. The infinitely many cells to the right of the input all contain a special blank symbol $\sqcup$.

The machine starts in its start state s with its head scanning the left endmarker. In each step it reads the symbol on the tape under its head. Depending on that symbol and the current state, it writes a new symbol on that tape cell, moves its head either left or right one cell, and enters a new state. The action it takes in each situation is determined by a transition function δ. It *accepts* its input by entering a special accept state t and *rejects* by entering a special reject state r. On some inputs it may run infinitely without ever accepting or rejecting, in which case it is said to *loop* on that input.

Formal Definition of Turing Machines

Formally, a *deterministic one-tape Turing machine* is a 9-tuple

$$M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r),$$

where

- Q is a finite set (the *states*);
- Σ is a finite set (the *input alphabet*);
- Γ is a finite set (the *tape alphabet*) containing Σ as a subset;
- $\sqcup \in \Gamma - \Sigma$, the *blank symbol*;
- $\vdash \in \Gamma - \Sigma$, the *left endmarker*;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, the *transition function*;
- $s \in Q$, the *start state*;
- $t \in Q$, the *accept state*; and
- $r \in Q$, the *reject state*, $r \neq t$.

would give the ability to decide the halting problem, which we know is impossible.

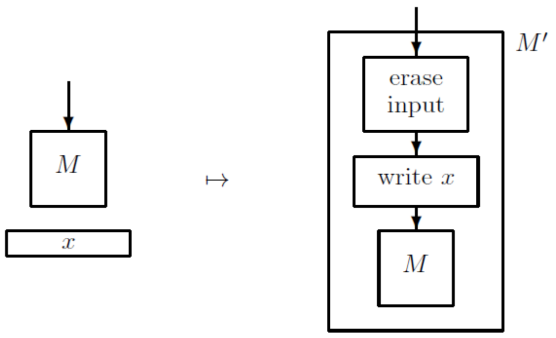
Suppose we could decide whether a given machine accepts ε. We could then decide the halting problem as follows. Say we are given a Turing machine M and string x, and we wish to determine whether M halts on x. Construct from M and x a new machine M' that does the following on input y:

(i) erases its input y;

(ii) writes x on its tape (M' has x hard-wired in its finite control);

(iii) runs M on input x (M' also has a description of M hard-wired in its finite control);

(iv) accepts if M halts on x.



Note that M' does the same thing on all inputs y: if M halts on x, then M' accepts its input y; and if M does not halt on x, then M' does not halt on y, therefore does not accept y. Moreover, this is true for every y. Thus

$$L(M') = \begin{cases} \Sigma^* & \text{if } M \text{ halts on } x, \\ \varnothing & \text{if } M \text{ does not halt on } x. \end{cases}$$

Now if we could decide whether a given machine accepts the null string ε, we could apply this decision procedure to the M' just constructed, and this would tell whether M halts on x. In other words, we could obtain a decision procedure for halting as follows: given M and x, construct M', then ask whether M' accepts ε. The answer to the latter question is "yes" iff M halts on x. Since we know the halting problem is undecidable, it must also be undecidable whether a given machine accepts ε.

6 Reduction

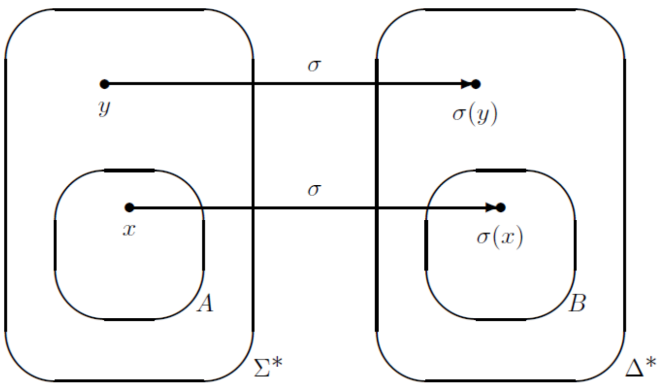
There are two main techniques for showing that problems are undecidable: *diagonalization* and *reductio*
We saw examples of diagonalization in §4 and reduction in §5.

Once we have established that a problem such as HP is undecidable, we can show that another problem B i
undecidable by *reducing* HP to B. Intuitively, this means we can manipulate instances of HP to make the
look like instances of the problem B in such a way that "yes" instances of HP become "yes" instances of
and "no" instances of HP become "no" instances of B. Although we cannot tell effectively whether a give
instance of HP is a "yes" instance, the manipulation preserves "yes"-ness and "no"-ness. If there existed
decision procedure for B, then we could apply it to the disguised instances of HP to decide membership i
HP. In other words, combining a decision procedure for B with the manipulation procedure would give
decision procedure for HP. Since we have already shown that no such decision procedure for HP can exist
we can conclude that no decision procedure for B can exist.

such that for all $x \in \Sigma^*$,

$$x \in A \quad \Leftrightarrow \quad \sigma(x) \in B. \tag{5}$$

In other words, strings in A must go to strings in B under σ, and strings not in A must go to strings not in B under σ.



The function σ need not be one-to-one or onto. It must, however, be *total* and *effectively computable*. This means σ must be computable by a total Turing machine that on any input x halts with $\sigma(x)$ written on its tape. When such a reduction exists, we say that A is *reducible* to B via the map σ, and we write $A \leq_{\mathrm{m}} B$. The subscript m, which stands for "many-one," is used to distinguish this relation from other types of reducibility relations.

Theorem 3 (Rice's theorem)**.** *Every nontrivial property of the r.e. sets is undecidable.*

Yes, you heard right: that's *every* nontrivial property of the r.e. sets. So as not to misinterpret this, let us clarify a few things.

First, fix a finite alphabet Σ. A *property of the r.e. sets* is a map

$$P : \{\text{r.e. subsets of } \Sigma^*\} \to \{\mathbf{1}, \mathbf{0}\},$$

where **1** and **0** represent truth and falsity, respectively. For example, the property of emptiness is represented by the map

$$P(A) \quad = \quad \begin{cases} \mathbf{1} & \text{if } A = \varnothing, \\ \mathbf{0} & \text{if } A \neq \varnothing. \end{cases}$$