MA/CSSE 474 Final Exam Notation and Formulas page Name _____ (turn this in with your exam)

Unless specified otherwise, r,s,t,u,v,w,x,y,z are strings over alphabet Σ; while a, b, c, d are individual alphabet symbols.

DFSM notation: M= (K, Σ, δ, s, A), where:

K is a finite set of ***states***, Σ is a finite ***alphabet***

$s \in K$ is start state, $A \subseteq K$ is set of ***accepting states***

δ: $(K \times \Sigma) \rightarrow K$ is the ***transition function***

Extend δ's definition to δ: $(K \times \Sigma^*) \rightarrow K$ by the recursive definition δ(q, ε)=q,
δ(q, xa) = δ(δ(q, x), a)

M accepts w iff δ(s, w) ∈ A. L(M) = {w ∈ Σ* : δ(s, w) ∈ A}

Alternate notation:

(q, w) is a *configuration* of M. (current state, remaining input)

The *yields-in-one-step* relation: $|-_M$:

$(q, w) \ |-_M \ (q', w')$ iff $w = a\,w'$ for some symbol $a \in \Sigma$, and $\delta(q, a) = q'$

The *yields-in-zero-or-more-steps* relation: $|-_M^*$ is the reflexive, transitive closure of $|-_M$.

A ***computation*** by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε), for some state $q \in K_M$,
- $\forall i \in \{0, 1, \ldots, n\text{-}1\}\ (C_i \ |-_M \ C_{i+1})$

M **accepts** w iff the state that is part of the last step in w is in A.

```
ndfsmtodfsm(M: NDFSM) =
  1. For each state q in K_M do:
       1.1 Compute eps(q).
  2. s' = eps(s)
  3. Compute δ':
       3.1 active-states = {s'}.
       3.2 δ' = ∅.
       3.3 While there exists some element Q of active-states for
           which δ' has not yet been computed do:
             For each character c in Σ_M do:
               new-state = ∅.
               For each state q in Q do:
                 For each state p such that (q, c, p) ∈ Δ do:
                   new-state = new-state ∪ eps(p).
               Add the transition (q, c, new-state) to δ'.
               If new-state ∉ active-states then insert it.
  4. K' = active-states.
  5. A' = {Q ∈ K' : Q ∩ A ≠ ∅ }.
```

Reg. exp. operator precedence (High to Low):
parenthesized expressions, * and +, concatenation, union

A language L is **regular** if L=L(M) for some DFSM M.

In an **NDFSM**, the function δ is replaced by the relation Δ: $\Delta \subseteq (K \times (\Sigma \cup \{\varepsilon\})) \times K$

Equivalent strings relative to a language: Given a language L, two strings w and x in Σ_L^* are *indistinguishable* with respect to L, written $w \approx_L x$, iff $\forall z \in \Sigma^*\ (xz \in L$ iff $yz \in L)$.

[x] is a notation for "the equivalence class that contains the string x".

The construction of a minimal-state DSFM based on $\approx_L$:

$M = (K, \Sigma, \delta, s, A)$, where K contains n states, one for each equivalence class of $\approx_L$.

$s = [\varepsilon]$, the equivalence class containing ε under $\approx_L$,

$A = \{[x] : x \in L\}$,

$\delta([x], a) = [xa]$.

Enumerator (generator) for a language: when it is asked, enumerator gives us the next element of the language. Any given element of the language will appear within a finite amount of time. It is allowed that some may appear multiple times.

Recognizer: Given a string s, recognizer halts and accepts s if s is in the language. If not, recognizer either halts and rejects s or keeps running forever. This is a **semidecision procedure**. If recognizer is guaranteed to always halt and (accept or reject) no matter what string it is given as input, it is a **decision procedure**.

The **regular expressions** over an alphabet Σ are the strings that can be obtained as follows:

1. ∅ is a regular expression.
2. ε is a regular expression.
3. Every element of Σ is a regular expression.
4. If α , β are regular expressions, then so is αβ.
5. If α , β are regular expressions, then so is α∪β.
6. If α is a regular expression, then so is α*.
7. α is a regular expression, then so is α^+.
8. If α is a regular expression, then so is (α).

Functions on languages:

firstchars(L) = $\{w : \exists y \in L\ (y = cx, c \in \Sigma_L, x \in \Sigma_L^*$, and $w \in c^*)\}$

chop(L) = $\{w : \exists x \in L\ (x = x_1 c x_2,\ x_1 \in \Sigma_L^*,\ x_2 \in \Sigma_L^*, c \in \Sigma_L\ |x_1| = |x_2|$, and $w = x_1 x_2)\}$

maxstring(L) = $\{w: w \in L,\ \forall z \in \Sigma^*\ (z \neq \varepsilon \rightarrow wz \notin L)\}$

mix(L) = $\{w: \exists x, y, z\ \ (x \in L,\ x = yz,\ |y| = |z|,\ w = yz^R)\}$

middle(L) = $\{x: \exists y, z \in \Sigma^*\ (yxz \in L)\}$

alt(L) = $\{x: \exists y, n\ (y \in L\ ,\ |y| = n,\ n > 0,\ y = a_1 \ldots a_n, \forall i \leq n\ (a_i \in \Sigma)$, and $x = a_1 a_3 a_5 \ldots a_k$, where k = (if n is even then n-1 else n))$\}$

Recursive formula for constructing a regular expression from a DFSM: r_{ijk} is $r_{ij(k-1)} \cup r_{ik(k-1)}(r_{kk(k-1)})^* r_{kj(k-1)}$

The set of regular languages is closed under complement, intersection, union, set difference, concatenation, Kleene * and +, reverse

Pumping Theorem and its contrapositive:

Formally, if L is regular, then
$\exists k \geq 1$ such that
($\forall$ strings $w \in L$, ($|w| \geq k$ →
($\exists x, y, z$ ($w = xyz$, $|xy| \leq k$, $y \neq \varepsilon$, and
$\forall q \geq 0$ ($xy^q z$ is in L)))))

The contrapositive form:
($\forall k \geq 1$
($\exists$ a string $w \in L$
($|w| \geq k$ *and*
($\forall x, y, z$
(**(**$w = xyz \wedge |xy| \leq k \wedge y \neq \varepsilon$**)** →
$\exists q \geq 0$ ($xy^q z$ is not in L)
)))))
→ L is not regular

CFG definition: G = (V, Σ, R, S)
(vocabulary, terminals, rules, start symbol)
Derivation and language definition

One derivation step: $x \Rightarrow_G y$ iff $\exists \alpha,\beta,\gamma \in V^*, A \in N ((x = \alpha A \beta) \wedge (A \rightarrow \gamma \in R) \wedge (y = \alpha \gamma \beta))$

$\Rightarrow_G^*$ is the reflexive, transitive closure of $\Rightarrow_G$

The **language defined by a grammar**: **L(G)** = $\{w \in \Sigma^* : S \Rightarrow_G^* w\}$
L is ***context-free*** if there is a context-free grammar *G* such that *L* = *L*(*G*).

A ***parse tree***, derived from a grammar *G* = (*V*, Σ, *R*, *S*), is a rooted, ordered tree in which:

Every leaf node is labeled with an element of $\Sigma \cup \{\varepsilon\}$,
The root node is labeled *S*,
Every other node is labeled with an element of *N*, and
If *m* is a non-leaf node labeled *X* and the (ordered) children of *m* are labeled $x_1, x_2, \ldots, x_n$,
then *R* contains the rule $X \rightarrow x_1 x_2, \ldots x_n$.

Chomsky Normal Form, in which all rules are of one of the following two forms:
$X \rightarrow a$, where $a \in \Sigma$, or $X \rightarrow BC$, where *B* and *C* are elements of *V* - Σ.

Greibach Normal Form, in which all rules are of the form $X \rightarrow a\,\beta$, where $a \in \Sigma$ and $\beta \in N^*$.

A grammar is ***ambiguous*** if some string it generates has two different parse trees
Equivalently, two different leftmost derivations, or two different rightmost derivations
A CFL is ***inherently ambiguous*** if *every* CFG that generates it is ambiguous.

PDA definition: *M* = (*K*, Σ, Γ, Δ, *s*, *A*),
states, input alphabet, tape alphabet, transition relation, start state, accepting states
$(q_1, cw, \gamma_1\gamma) \vdash_M (q_2, w, \gamma_2\gamma)$ iff $((q_1, c, \gamma_1), (q_2, \gamma_2)) \in \Delta$.
accepting computation *of M:* $(s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \varepsilon)$, *and* $q \in A$

Top-down PDA from grammar: Production A → XYZ becomes (q, ε, A) → (q, XYZ)
(s, ε, ε) → (q, S) [s is the start state of M). A ={q} . For each terminal, (q, a, a) → (q, ε)

Bottom-up PDA from grammar: The shift transitions: $((p, c, \varepsilon), (p, c))$, for each $c \in \Sigma$.
The reduce transitions: $((p, \varepsilon, (s_1 s_2 \ldots s_n.)^R), (p, X))$, for each rule $X \rightarrow s_1 s_2 \ldots s_n.$ in *G*. .
The finish-up transition: $((p, \varepsilon, S), (q, \varepsilon))$. A = {q}

CFL closure: Union, Concatenation, Kleene Star. Reverse. Intersection with regular language.
Not closed under complement, intersection, set difference.
We have **CFL decision algorithms** for membership, emptiness, finiteness.

Undecidable questions about CFLs: Is $L = \Sigma^*$? Is *L* regular? Is $L_1 = L_2$? Is $L_1 \subseteq L_2$? Is $L_1 \cap L_2 = \varnothing$?
Is the complement of *L* context-free? Is $L_1 \cap L_2 = \varnothing$? Is *L* inherently ambiguous? Is *G* ambiguous?
Deterministic PDA M: Δ_M contains no pairs of transitions that compete with each other, and whenever *M* is in an accepting configuration it has no available moves.
A language *L* is ***deterministic context-free*** iff *L*$ can be accepted by *some* deterministic PDA.

Formal TM definition. A deterministic TM M is (*K*, Σ, Γ, δ, *s*, *H*):

i) *K* is a finite set of states;
ii) Σ is the input alphabet, which does not contain □;
iii) Γ is the tape alphabet, which must contain □ and have Σ as a subset.
iv) $s \in K$ is the initial state;
v) $H \subseteq K$ is the set of halting states;
vi) δ is the transition **function**:
(1) (*K* - *H*) × Γ to *K* × Γ × {→, ←}
non-halting × tape → state × tape × direction to move
state char char (R or L)

Yields. $(q_1, w_1) \vdash_M (q_2, w_2)$ iff (q_2, w_2) is derivable, *via* δ, in one step.
$\vdash_M^*$ is the reflexive, transitive closure of $\vdash_M$.
Configuration C_1 ***yields*** configuration C_2 if: $C_1 \vdash_M^* C_2$.
A ***path*** through *M* is a sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n$.
A **computation** by *M* is a path that halts. If a computation is of *length n* (has *n* steps), we write: $C_0 \vdash_M^n C_n$

---

2) **TMs as language recognizers**. Let *M* = (*K*, Σ, Γ, δ, *s*, {*y*, *n*}).
a) *M* ***accepts*** a string *w* iff $(s, \underline{\sqcup}w) \vdash_M^* (y, w')$ for some string *w*'.
b) *M* ***rejects*** a string *w* iff $(s, \underline{\sqcup}w) \vdash_M^* (n, w')$ for some string *w*'.
c) *M* ***decides*** a language $L \subseteq \Sigma^*$ iff for any string $w \in \Sigma^*$t:
i) if $w \in L$ then *M* accepts *w*, and
ii) if $w \notin L$ then *M* rejects *w*.
d) A language *L* is ***decidable*** iff thewre is a TM M that decides it.
e) We define the set ***D*** to be the set of all decidable languages.
f) *M* ***semidecides*** *L* iff, for any string $w \in \Sigma_M^*$:
i) $w \in L \rightarrow M$ accepts *w*
ii) $w \notin L \rightarrow M$ does not accept *w*. *M* may reject or not halt.
g) A language *L* is ***semidecidable*** iff there is a Turing machine that semidecides it.
h) We define the set ***SD*** to be the set of all semidecidable languages.

Summary of Algorithms

- Compute functions of languages defined as FSMs:
  - Given FSMs M_1 and M_2, construct a FSM M_3 such that $L(M_3) = L(M_2) \cup L(M_1)$.
  - Given FSMs M_1 and M_2, construct a new FSM M_3 such that $L(M_3) = L(M_2)\,L(M_1)$.
  - Given FSM M, construct an FSM M^* such that $L(M^*) = (L(M))^*$.
  - Given a DFSM M, construct an FSM M^* such that $L(M^*) = \neg L(M)$.
  - Given two FSMs M_1 and M_2, construct an FSM M_3 such that $L(M_3) = L(M_2) \cap L(M_1)$.
  - Given two FSMs M_1 and M_2, construct an FSM M_3 such that $L(M_3) = L(M_2) - L(M_1)$.
  - Given an FSM M, construct an FSM M^* such that $L(M^*) = (L(M))^R$.
  - Given an FSM M, construct an FSM M^* that accepts *letsub*($L(M)$).
- Converting between FSMs and regular expressions:
  - Given a regular expression α, construct an FSM M such that: $L(\alpha) = L(M)$
  - Given an FSM M, construct a regular expression α such that: $L(\alpha) = L(M)$
- **Algorithms that implement operations on languages defined by regular expressions**: any operation that can be performed on languages defined by FSMs can be implemented by converting all regular expressions to equivalent FSMs and then executing the appropriate FSM algorithm.
- Converting between FSMs and regular expressions:
  - Given a regular expression α, construct an FSM M such that: $L(\alpha) = L(M)$
  - Given an FSM M, construct a regular expression α such that: $L(\alpha) = L(M)$
- **Algorithms that implement operations on languages defined by regular expressions**: any operation that can be performed on languages defined by FSMs can be implemented by converting all regular expressions to equivalent FSMs and then executing the appropriate FSM algorithm.
  - Given a regular grammar G, construct an FSM M such that: $L(G) = L(M)$
  - Given an FSM M, construct a regular grammar G such that: $L(G) = L(M)$.

contrapositive of CFG Pumping Theorem:
If $\forall k \geq 1$ ($\exists$ a string $w \in L$, where $|w| \geq$
($\forall$ *u*, *v*, *x*, *y*, *z*
(($w = uvxyz$, $vy \neq \varepsilon$, and $|vxy| \leq k$)
implies
($\exists\, q \geq 0$ ($uv^q xy^q z$ is not in *L*)))),
then L is not context-free

Additions since Exam 3:

Notice that the TM's function computes with strings ($\Sigma^* \mapsto \Sigma'^*$), not directly with numbers.

3) **TMs can compute functions**. Let $M = (K, \Sigma, \Gamma, \delta, s, \{h\})$.
   a) $M(w) = z$ iff $(s, \sqcup w) \vdash_M^* (h, \sqcup z)$.
   b) Let $\Sigma' \subseteq \Sigma$ be M's output alphabet, and let f be any function from Σ^* to Σ'^*.
      i) ***M computes f*** iff, for all $w \in \Sigma^*$:
         (1) if w is an input on which f is defined, then $M(w) = f(w)$.
         (2) otherwise $M(w)$ does not halt.
   c) A function f is ***recursive*** or ***computable*** iff there is a Turing machine M that computes it and that always halts.
   d) **Computing numeric functions**:
      i) For any positive integer k, ***value$_k$(n)*** returns the nonnegative integer that is encoded, base k, by the string n.
      ii) TM M computes **a function f from $\mathbb{N}^m$ to $\mathbb{N}$** iff, for some k, $value_k(M(n_1;n_2;...n_m)) = f(value_k(n_1), \ldots value_k(n_m))$.
4) An m-tape TM can be simulated by a 2n-track TM, which can be simulated by a single-track machine.
5) **Encoding a TM** M = $(K, \Sigma, \Gamma, \delta, s, H)$ as a string <M>:
   i) **Encoding the states**: Let i be $\lceil \log2(|K|) \rceil$.
      (1) Number the states from 0 to $|K|$-1 in binary (i bits for each state number):
      (2) The start state, s, is numbered 0; Number the other states in any order.
      (3) If t' is the binary number assigned to state t, then:
         (a) If t is the halting state y, assign it the string yt'.
         (b) If t is the halting state n, assign it the string nt'.
         (c) If t is the halting state h, assign it the string ht'.
         (a) If t is any other state, assign it the string qt'.
   ii) **Encoding the tape alphabet**: Let j be $\lceil \log2(|\Gamma|) \rceil$.
      (1) Number the tape alphabet symbols from 0 to $|\Gamma|$ - 1 in binary.
      (2) The blank symbol is number 0.
      (3) The other symbols can be numbered in any order
   iii) **Encoding the transitions**:
      (1) (state, input, state, output, direction to move)
      (2) Example: (q000,a000,q110,a000,→)
   iv) **Encoding s and H** (already included in the above)
   v) A **special case** of TM encoding
      (1) One-state machine with no transitions that accepts only ε is encoded as (q0)
   vi) **Encoding other T**Ms: It is just a list of the machine's transitions:
      (1) Detailed example on slide
   vii) Consider the alphabet Σ = {(,), a, q, y, n, *h*, 0, 1, comma, →, ←}.
      The following question is decidable:
      (1) Given a string w in Σ^*, is there a TM M such that w = <M> ?
6) We can **enumerate all TMs**, so that we have the concept of "the ith TM".
7) **Specification of U**, the Universal Turing Machine (UTM):
   a) U starts with <M,w> on its input tape, then simulates M's action when it has input w:
   b) U halts iff M halts on w.
   c) If M is a deciding or semideciding machine, then:
      i) If M accepts, U accepts. If M rejects, U rejects.
   d) If M computes a function, then $U(<M, w>)$ must equal $M(w)$.
8) A language is in SD iff it is Turing enumerable.
   A language is in D iff it is lexicographically enumerable.
9) D is closed under complement. SD is not; if L∈SD-D, ¬L∉SD. ¬H is an example.
10) Problem P_1 is **reducible** to problem P_2 (written $P_1 \le P_2$) if there is a Turing-computable
11) function f that finds, for an arbitrary instance I of P_1, an instance f(I) of P_2, and
   a) f is defined such that for every instance I of P_1,
   b) I is a yes-instance of P_1 if and only if f(I) is a yes-instance of P_2.
   c) So $P_1 \le P_2$ means "if we have a TM that decides P_2, then there is a TM that decides P_1.
12) A framework for using reduction to show undecidability. To show language L_2 undecidable:
   a) Choose a language L_1 that is already known not to be in D, and show that L_1 can be reduced to L_2.
   b) Define the reduction *R and show that it can be implemented by a TM.*
   c) Describe the composition C of R with *Oracle* (the purported TM that decides L_1).
   d) Show that C does correctly decide L_1 iff *Oracle* exists.
      We do this by showing that
      C is correct. I.e.,
      i) If $x \in L_1$, then $C(x)$ accepts, and
      ii) If $x \notin L_1$, then $C(x)$ rejects.
13) Rice's Theorem: If P is a non-trivial (Boolean) property of DS languages, it is undecidable.

$\{<M>$: TM M has an even number of states$\}$	D
$H = \{<M, w>$: TM M halts on $w\}$	SD/D
$H_\varepsilon = \{<M>$: TM M halts on $\varepsilon\}$	SD/D
$H_{ANY} = \{<M>$: there exists at least one string on which TM M halts $\}$	SD/D
$H_{ALL} = \{<M>$: TM M halts on $\Sigma^*\}$	¬SD
$A = \{<M, w>$: TM M accepts $w\}$	SD/D
$A_\varepsilon = \{<M>$: TM M accepts $\varepsilon\}$	SD/D
A_{ANY} $\{<M>$: there exists at least one string that TM M accepts $\}$	SD/D
$\neg H = \{<M, w>$: TM M does not halt on $w\}$	¬SD
$A_{ALL} = \{<M> : L(M) = \Sigma^*\}$	¬SD
$EqTMs = \{<M_a, M_b> : L(M_a) = L(M_b)\}$	¬SD
$H_{\neg ANY} = \{<M>$: there does not exist any string on which TM M halts$\}$	¬SD
$\{<M>$: TM M does not halt on input $<M>\}$	¬SD
$TM_{MIN} = \{<M>$: TM M is minimal$\}$	¬SD
$TM_{REG} = \{<M> : L(M)$ is regular$\}$	¬SD
$A_{anbn} = \{<M> : L(M) = A^nB^n\}$	¬SD

In some sense, ≤ means "is no harder than" or "is at least as decidable as"

H_{ANY} is not in D

H = {<*M*, *w*> : TM M halts on input string *w*}

R ↓

(?*Oracle*) H_{ANY} = {<*M*> : there exists at least one string on which TM *M* halts}

R(<*M*, *w*>) =
1. Construct <*M*#>, where *M*#(*x*) operates as follows:
   1.1. Examine *x*.
   1.2. If *x* = *w*, run *M* on *w*, else loop forever.
2. Return <*M*#>.

If *Oracle* exists, then *C* = *Oracle*(*R*(<*M*, *w*>)) decides H:
- *R* can be implemented as a Turing machine.
- *C* is correct: The only string on which *M*# can halt is *w*. So:
  - <*M*, *w*> ∈ H: *M* halts on *w*. So *M*# halts on *w*. There exists at least one string on which *M*# halts. *Oracle* accepts.
  - <*M*, *w*> ∉ H: *M* does not halt on *w*, so neither does *M*#. So there exists no string on which *M*# halts. *Oracle* rejects.

But no machine to decide H can exist, so neither does *Oracle*.

$A_{anbn} = \{<M> : L(M) = A^nB^n\}$ is not SD

R(<*M*, *w*>) reduces ¬H to A_{anbn}:
1. Construct the description <*M*#>:
   1.1. If *x* ∈ A^nB^n then accept. Else:
   1.2. Erase the tape.
   1.3. Write *w* on the tape.
   1.4. Run *M* on *w*.
   1.5. Accept.
2. Return <*M*#>.

If *Oracle* exists and semidecides A_{anbn}, *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H:
M# immediately accepts all strings in A^nB^n. If *M* does not halt on *w*, those are the only strings *M*# accepts. If *M* halts on *w*, *M*# accepts everything:
- <*M*, *w*> ∈ ¬H: *M* does not halt on *w*, so *M*# accepts strings in A^nB^n in step 1.1. Then it gets stuck in step 1.4, so it accepts nothing else. It is an A^nB^n acceptor. ***Oracle* accepts.**
- <*M*, *w*> ∉ ¬H: *M* halts on *w*, so *M*# accepts everything. ***Oracle* does not accept.**

But no machine to semidecide ¬H can exist, so neither does *Oracle*.

A Macro language for Turing Machines

(1) Define some basic machines

You need to learn this simple language. I will use it and I expect you to use it on HW and tests.

- Symbol writing machines

  For each $x \in \Gamma$, define M_x, written as just x, to be a machine that writes x. Read-write head ends up in original position.

- Head moving machines

  R: for each $x \in \Gamma$, $\delta(s, x) = (h, x, \rightarrow)$
  L: for each $x \in \Gamma$, $\delta(s, x) = (h, x, \leftarrow)$

- Machines that simply halt:
  h, which simply halts (don't care whether it accepts).
  n, which halts and rejects.
  y, which halts and accepts.

Checking Inputs and Combining Machines

Next we need to describe how to:

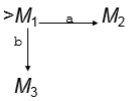
- Check the tape and branch based on what character we see, and
- Combine the basic machines to form larger ones.

To do this, we need two forms:

- $M_1\ M_2$
- $M_1 \xrightarrow{\text{<condition>}} M_2$

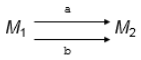
Turing Machines Macros Cont'd

Example:



- Start in the start state of M_1.
- Compute until M_1 reaches a halt state.
- Examine the tape and take the appropriate transition.
- Start in the start state of the next machine, etc.
- Halt if any component reaches a halt state and has no place to go.
- If any component fails to halt, then the entire machine may fail to halt.

More macros



becomes $M_1 \xrightarrow{a,\ b} M_2$

$M_1 \xrightarrow{\text{all elems of } \Gamma} M_2$ becomes $M_1 \rightarrow M_2$ or $M_1 M_2$

Variables

$M_1 \xrightarrow[\text{except a}]{\text{all elems of } \Gamma} M_2$ becomes $M_1 \xrightarrow{x \leftarrow \neg a} M_2$
and x takes on the value of the current square

$M_1 \xrightarrow{a,\ b} M_2$ becomes $M_1 \xrightarrow{x \leftarrow a,\ b} M_2$
and x takes on the value of the current square

$M_1 \xrightarrow{x = y} M_2$
if $x = y$ then take the transition

e.g., $> \xrightarrow{x \leftarrow \neg \sqcup} Rx$ if the current square is not blank, go right and copy it.

Blank/Non-blank Search Machines

Find the first blank square to the right of the current square. $R_{\sqcup}$

Find the first blank square to the left of the current square. $L_{\sqcup}$

Find the first nonblank square to the right of the current square. $R_{\neg\sqcup}$

Find the first nonblank square to the left of the current square $L_{\neg\sqcup}$

More Search Machines

L_a	Find the first occurrence of a to the left of the current square.
$R_{a,b}$	Find the first occurrence of a or b to the right of the current square.
$L_{a,b} \xrightarrow{a} M_1$, $\downarrow b$, M_2	Find the first occurrence of a or b to the left of the current square, then go to M_1 if the detected character is a; go to M_2 if the detected character is b.
$L_{x \leftarrow a,b}$	Find the first occurrence of a or b to the left of the current square and set x to the value found.
$L_{x \leftarrow a,b} Rx$	Find the first occurrence of a or b to the left of the current square, set x to the value found, move one square to the right, and write x (a or b).

Algorithms and decision problems for regular languages

Because there are so many of these algorithms and they have been spread out over several chapters, we present a concise list of them here:

- Algorithms that operate on FSMs without altering the language that is accepted:
  - *Ndfsmtodfsm*: Given an NDFSM M, construct a DFSM M' such that $L(M) = L(M')$.
  - *MinDFSM*: Given a DFSM M, construct a minimal DFSM M', such that $L(M) = L(M')$.
- Algorithms that compute functions of languages defined as FSMs:
  - Given two FSMs M_1 and M_2, construct a new FSM M_3 such that $L(M_3) = L(M_2) \cup L(M_1)$.
  - Given two FSMs M_1 and M_2, construct a new FSM M_3 such that $L(M_3) = L(M_2)L(M_1)$ (i.e., the concatenation of $L(M_2)$ and $L(M_1)$).
  - Given an FSM M, construct a new FSM M' such that $L(M') = (L(M))^*$.
  - Given an FSM M, construct a new FSM M' such that $L(M') = \neg L(M)$.
  - Given two FSMs M_1 and M_2, construct a new FSM M_3 such that $L(M_3) = L(M_2) \cap L(M_1)$.
  - Given two FSMs M_1 and M_2, construct a new FSM M_3 such that $L(M_3) = L(M_2) - L(M_1)$.
  - Given an FSM M, construct a new FSM M' such that $L(M') = (L(M))^R$ (i.e., the reverse of $L(M)$).
  - Given an FSM M, construct an FSM M' that accepts *letsub*$(L(M))$, where *letsub* is a letter substitution function.
- Algorithms that convert between FSMs and regular expressions:
  - Given a regular expression α, construct an FSM M such that $L(\alpha) = L(M)$.
  - Given an FSM M, construct a regular expression α such that $L(\alpha) = L(M)$.
- Algorithms that convert between FSMs and regular grammars:
  - Given a regular grammar G, construct an FSM M such that $L(G) = L(M)$.
  - Given an FSM M, construct a regular grammar G such that $L(G) = L(M)$.
- Algorithms that implement operations on languages defined by regular expressions or regular grammars: Any operation that can be performed on languages defined by FSMs can be implemented by converting all regular expressions or regular grammars to equivalent FSMs and then executing the appropriate FSM algorithm.
- Decision procedures that answer questions about languages defined by FSMs:
  - Given an FSM M and a string w, is w is accepted by M?
  - Given an FSM M, is $L(M) = \varnothing$?
  - Given an FSM M, is $L(M) = \Sigma^*$?
  - Given an FSM M, is $L(M)$ finite (or infinite)?
  - Given two FSMs, M_1 and M_2, is $L(M_1) = L(M_2)$?
  - Given a DFSM M, is M minimal?
- Decision procedures that answer questions about languages defined by regular expressions or regular grammars: Again, convert the regular expressions or regular grammars to FSMs and apply the FSM algorithms.

Algorithms and decision problems for context-free languages

- Algorithms that transform grammars:
  - *removeunproductive*(G: context-free grammar): Construct a grammar G' that contains no unproductive nonterminals and such that $L(G') = L(G)$.
  - *removeunreachable*(G: context-free grammar): Construct a grammar G' that contains no unreachable nonterminals and such that $L(G') = L(G)$.
  - *removeEps*(G: context-free grammar): Construct a grammar G' that contains no rules of the form $X \rightarrow \varepsilon$ and such that $L(G') = L(G) - \{\varepsilon\}$.
  - *atmostoneEps*(G: context-free grammar): Construct a grammar G' that contains no rules of the form $X \rightarrow \varepsilon$ except possibly $S^* \rightarrow \varepsilon$, in which case there are no rules whose right-hand side contains S^*, and such that $L(G') = L(G)$.
  - *converttoChomsky*(G: context-free grammar): Construct a grammar G' in Chomsky normal form, where $L(G') = L(G) - \{\varepsilon\}$.
  - *converttoGreibach*(G: context-free grammar): Construct a grammar G' in Greibach normal form, where $L(G') = L(G) - \{\varepsilon\}$.
  - *removeUnits*(G: context-free grammar): Construct a grammar G' that contains no unit productions, where $L(G') = L(G)$.

Algorithms that convert between context-free grammars and PDAs:

- *cfgtoPDAtopdown*(G: context-free grammar): Construct a PDA M such that $L(M) = L(G)$ and M operates top-down to simulate a left-most derivation in G.
- *cfgtoPDAbottomup*(G: context-free grammar): Construct a PDA M such that $L(M) = L(G)$ and M operates bottom up to simulate, backwards, a right-most derivation in G.
- *cfgtoPDAnoeps*(G: context-free grammar): Construct a PDA M such that M contains no transitions of the form $((q_1, \varepsilon, s_1), (q_2, s_2))$ and $L(M) = L(G) - \{\varepsilon\}$.

Algorithms that transform PDAs:

- *convertPDAtorestricted*(M: PDA): Construct a PDA M' in restricted normal form where $L(M') = L(M)$.

Algorithms that compute functions of languages defined as context-free grammars:

- Given two grammars G_1 and G_2, construct a new grammar G_3 such that $L(G_3) = L(G_1) \cup L(G_2)$.
- Given two grammars G_1 and G_2, construct a new grammar G_3 such that $L(G_3) = L(G_1)L(G_2)$.
- Given a grammar G, construct a new grammar G' such that $L(G') = (L(G))^*$.
- Given a grammar G, construct a new grammar G' such that $L(G') = (L(G))^R$.
- Given a grammar G, construct a new grammar G' that accepts *letsub*$(L(G))$, where *letsub* is a letter substitution function.

Miscellaneous algorithms for PDAs:

- *intersectPDAandFSM* (M_1: PDA, M_2: FSM): Construct a PDA M_3 such that $L(M_3) = L(M_1) \cap L(M_2)$.
- *without$*($M$: PDA): If M accepts L\$, construct a PDA M' such that $L(M') = L$.
- *complementdetPDA*(M: DPDA): If M accepts L\$, construct a PDA M' such that $L(M') = (\neg L)$\$.

- Decision procedures that answer questions about context-free languages:
  - *decideCFLusingPDA*(L: CFL, w : string): Decide whether w is in L.
  - *decideCFLusingGrammar*(L: CFL, w: string): Decide whether w is in L.
  - *decideCFL*(L: CFL, w: string): Decide whether w is in L.
  - *decideCFLempty*(G: context-free grammar): Decide whether $L(G) = \varnothing$.
  - *decideCFLinfinite*(G: context-free grammar): Decide whether $L(G)$ is infinite.