Computation & Formal Systems

Fall 2017

Contents

1 Finite Automata

Definition (Finite Automaton). A finite automaton $M = \langle S, \Sigma, \delta, s_0, F \rangle$ where

- S: finite set of states
- Σ: finite input alphabet
- δ: transition function $S \times \Sigma \to S$
- $s_o \in S$: initial state
- $F \subseteq S$: set of accepting states

An automaton M acting on input $a_0 a_1 \ldots a_k$ produces a *path* (state sequence) $s_0 s_1 \ldots s_k$. The input $a_0 a_1 \ldots a_k$ is called a *label* of $s_0 s_1 \ldots s_k$.

Definition (Language). The language accepted by an automaton M, denoted $L(M)$, is the set of strings accepted by M or equivalently the set of labels of paths to accepting states.

Definition. Let ϵ be the empty string and Σ' the set of finite strings of input characters in Σ. Define $\delta' : S \times \Sigma' \to S$ so that $\delta'(s, \epsilon) = s$ and $\delta'(s, wa) = \delta(\delta'(s, w), a)$ for all $w \in \Sigma'$ and $a \in \Sigma$.

Using this definition we can formally describe the language of a finite automaton M via

$$L(M) = \{x | \delta'(s_0, x) \in F\}.$$

Definition (DFA). In a deterministic finite automaton, for each $s \in S$ and $x \in \Sigma$ there is at most 1 transition from s on x.

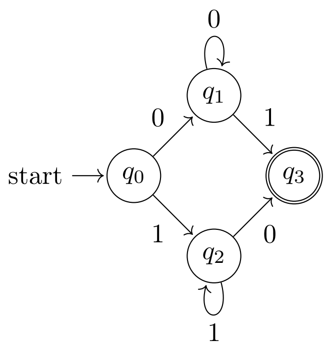


Figure 1: $M = \langle \{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta \text{ (see Table 1)}, q_0, q_3 \rangle$.

δ	0	1
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_3	q_2
q_3	Halt	Halt

Table 1: Transition table for δ in Figure 1.

Definition (NFA). An NFA, or nondeterministic finite automaton, $\langle S, \Sigma, \delta_N, s_0, F\rangle$, has a transition function, $\delta_N : S \times \Sigma \to 2^S$. Hence for each $s \in S$ and $x \in \Sigma$ there can be up to $2^{|S|}$ transitions from s on x.

Thus DFA $\subset$ NFA.

1.1 Equivalence of L(DFA) and L(NFA)

Let $C = \{M_i\}$ and define $L(C) = \{L | L = L(M_i) \text{ for some } M_i \in C\}$.

Theorem 1.1

If $L = L(D)$ for some DFA D, then $L = L(N)$ for some NFA N.

Proof. As previously stated, all DFAs are also NFAs. So the statement holds by definition. ∎

Remark 1.1. Suppose we wanted to simulate an NFA using a DFA. This could be achieved by:

- Enumerating a transition table such that the next state of the DFA is the set of all possible next states for a given input. That is, given current state $s = \{s_i\}$ and input x, we set $s \longleftarrow \bigcap_{s_i \in s} \delta(s_i, x)$.
- Maintaining a set of current possible states.
- Accepting if any possible final state is an accepting state.

1.1.1 Subset Construction

Definition (Equivalence of Automata). We say two automata, A and B, are equivalent if they accept the same set of input strings. That is, if $a_1a_2\cdots a_k$ is any string of symbols, then the following hold:

(i) If there is a path labeled $a_1a_2\cdots a_k$ from the starting state of A to some accepting state of A, then there is a path labeled $a_1a_2\cdots a_k$ from the starting state of B to an accepting state of B.

(ii) Vice versa of (i).

Theorem 1.2

Let N be a nondeterministic finite automaton, and construct the deterministic finite automaton D as follows.

If s_0 is the start state of N then $\{s_0\}$ is the start state of D. Let S be a state of D and x an input symbol. Since S is a state of D, it consists of states of N. Define $T = \{t_i |$ for some $s_i \in S : t_i \in \delta_N(s_i, x)\} = \bigcup_{s_i \in S} \delta_N(s_i, x)$. Then, let T be a state of D and set a transition from S to T on symbol x.

Then, the state $\{s_1, s_2, \ldots, s_n\}$ reached in D by the path labeled $a_1 a_2 \cdots a_k$ from the start state of D is exactly the set of states the are reached from N's start state by following $a_1 a_2 \cdots a_k$. That is, D is equivalent to N.

Proof. We proceed by induction on k, the length of the input sequence.

For $k = 0$, the input is ϵ, and $\delta'(s, \epsilon) = s$. Therefore, s_0 is the start state of N and $\{s_0\}$ is the starting state of D, by the definition of the subset construction.

Suppose the statement for k, and consider an input string $a_1 a_2 \ldots a_k a_{k+1}$. Then the path from the start state of D, $\{s_0\}$, to state T labeled by $a_1 a_2 \ldots a_k a_{k+1}$ goes through some state S before transitioning to T on input a_{k+1}. By the inductive hypothesis, S is exactly the set of states that automaton N reaches from its start state, s_0, along $a_1 a_2 \ldots a_k$. So we must show that T is exactly the set of states N reaches from s_0 along paths labeled $a_1 a_2 \ldots a_k a_{k+1}$.

Now suppose $t \in T$, then there must be some state $s \in S$ such that there is a transition from s to t and its label includes a_{k+1}. By the inductive hypothesis, since $s \in S$ there must be some path from s_0 to s labeled $a_1 a_2 \ldots a_k$. Thus there is a path from the start state of N to t labeled $a_1 a_2 \ldots a_k a_{k+1}$.

Conversely, we must show that if there is a path from s_0 to t labeled $a_1 a_2 \ldots a_k a_{k+1}$ then t is in T. This path must go through some state s before transitioning to t on a_{k+1}. Thus, there is a path from s_0 to s labeled $a_1 a_2 \ldots a_k$. By the induction hypothesis, s is in the set of states of S. Since N has a transition from s to t with a label that includes a_{k+1}, the subset construction applied to the set of states in S and a_{k+1}, requires that t be in T.

Therefore, T is exactly the set of states N reaches from s_0 along paths labeled $a_1 a_2 \ldots a_k a_{k+1}$. So we conclude that the state of the deterministic automaton D reached along the path labeled $a_1 a_2 \ldots a_k$ is always the set of N's states reachable along some path with that label. Since the accepting states of D are those that include an accepting state of N, we conclude that D and N accept the same strings; that is, D and N are equivalent, and the subset construction "works". ■

Corollary 1.3. If $L = L(N)$ for some NFA N, then $L = L(D)$ for some DFA D.

Proof. By Theorem 1.2, given any NFA N, we can construct an equivalent DFA D, i.e. $L(N) = L(D)$. ∎

Corollary 1.4. $L(\text{DFA}) = L(\text{NFA})$.

Proof. This follows directly from Theorem 1.1 and Corollary 1.3. ∎

1.2 Regular Expressions

Definition (Algebra). A formal system in which there are operands and operators from which one builds expressions.

Definition (Algebra of Regular Expressions). The operands of regular expressions are patterns. The atomic operands are:

- c for any input character
- ϵ for the empty string
- $\varnothing$ for the empty set
- Variables, e.g. R, S, ranging over regular expression patterns.

The operators of regular expressions are:

- Concatenation (Then)
- Union (Or)
- Closure (Loop)

Regular expressions denote sets of strings that match a pattern.

Definition.

- $L(\mathsf{c}) = \{\mathsf{c}\}$
- $L(\epsilon) = \{\epsilon\}$
- $L(\varnothing) = \varnothing$

Definition. Let R and S be regular expressions and let $L(R) = \{r_1, \ldots, r_m\}$ and $L(S) = \{s_1, \ldots, s_n\}$. Then,

- The union of R and S is $R|S$, and $L(R|S) = L(R) \cup L(S)$.

- The concatenation of R and S is RS, and $L(RS) = \{r_i s_j : r_i \in L(R) \wedge s_j \in L(S)\}$.
- The closure of R is denoted R^*, and $L(R^*) = \{\epsilon\} \cup L(R) \cup L(RR) \cup \cdots$.

Operator Precedence.

- Closure (highest)
- Concatenation
- Union (lowest)

Example. The language of the regular expression, $\texttt{a}|\texttt{bc}^*\texttt{d}$ is $L(\texttt{a}|\texttt{bc}^*\texttt{d}) = \{\texttt{a}, \texttt{bcd}, \texttt{bccd}, \ldots\}$

Definition (Equivalence of RE). Given two regular expressions R_1 and R_2, we say R_1 is equivalent to R_2, denoted $R_1 \equiv R_2$ if and only if $L(R_1) = L(R_2)$.

Theorem 1.5

Let R, S be regular expressions then the following are equivalent:

- $\varnothing|R \equiv R|\varnothing \equiv R$
- $\epsilon R \equiv R\epsilon \equiv R$
- $\varnothing R \equiv R\varnothing \equiv \varnothing$
- $R|S \equiv S|R$
- $(R|S)|T \equiv R|(S|T)$
- $(RS)T \equiv R(ST)$
- $R(S|T) \equiv RS|RT$
- $(S|T)R \equiv SR|TR$
- $R|R \equiv R$
- $\varnothing^* = \epsilon$
- $RR^* \equiv R^*R$
- $RR^*|\epsilon \equiv R^*$

Example. Prove that $((S|T)R) \equiv (SR|TR)$.

Proof. If x is in $L((S|T)R)$ then $x = yr$ where r in $L(R)$ and y is either in $L(S)$ or in $L(R)$, or both. If y is in $L(S)$ then x is in $L(SR)$, and if y is in $L(T)$ then x is in $L(TR)$. In either case, x is in $L(SR|TR)$. Thus, $L((S|T)R) \subseteq L(SR|TR)$.

Consider, some x in $L(SR|TR)$. Then either $x = sr$ where s is in $L(S)$ and r is in $L(R)$ or $x = tr$ where t is in $L(T)$ and r is in $L(R)$. In either case, x is of the form yr where y is in $L(S|T)$. Hence, x is in $L((S|T)R)$. Thus, $L(SR|TR) \subseteq L((S|T)R)$. ■

1.3 Computing with Regular Expressions

The following section describes how to convert a regular expression, R, with language, $L(R)$, into an NFA that accepts the same language.

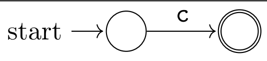
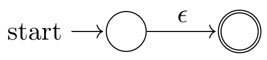
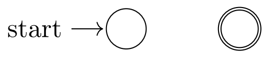
1.3.1 Epsilon Transitions

- Allow arcs to be labeled with ϵ.
- Allow transitions without reading any input.

1.3.2 Regular Expressions to Epsilon NFA

We can convert a regular expression into an ϵ-NFA inductively on the number of operations. Base cases are handled according to Table 2. Inductive steps are handled according to Table 3.

Table 2: Converting RE to NFA

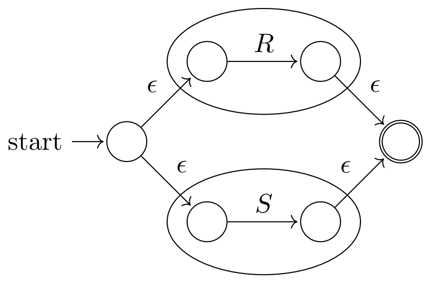
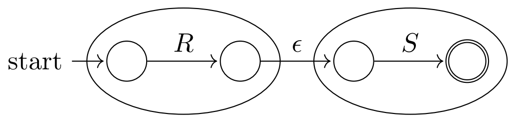
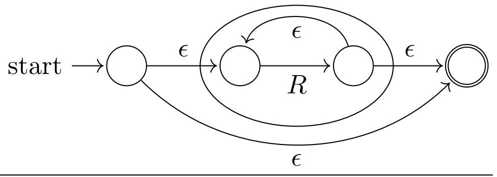
Atomic RE	Figure
`c`	
ϵ	
$\varnothing$	

This conversion is $\mathcal{O}(n)$ in the number of operations.

1.3.3 Converting Epsilon NFAs to NFAs

- Computing ϵ-reachability.
  - Can be achieved via a graph traversal, perhaps depth first search, of an ϵ-NFA with only ϵ-edges.
  - For each state, keep track of which other states are ϵ-reachable.
  - This is $\mathcal{O}(n^2)$ in time and space.
- Important states are states which can be reached by a transition on a real (non-ϵ) symbol, or, the starting state. These are the only states that will be preserved in the conversion to regular NFA.
- The steps for conversion are summarized in Table 4.

Table 3: Converting RE to NFA

Operations	Figure
$R\|S$	
RS	
R^*	

1.4 Converting Finite Automaton to RE

The construction involves elimination of states one by one from the automaton. First, the labels on arcs will be replace by regular expressions. So, if an arc initially has the label, $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$, then its new label will be $\mathtt{x}_1|\ldots|\mathtt{x}_n$. And we can think of the label of a path as the concatenation of the regular expressions along that path. So if the arcs of a path are labeled by $R_1, \ldots, R_n$, then the label of that path becomes $R_1R_2\ldots R_n$.

Note that every state in an automaton can be represented in the form of state u in Figure 2. Where $s_1, \ldots, s_n$ have transitions into u and $t_1, \ldots, t_m$ have transitions from u.

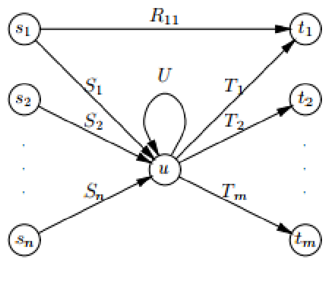
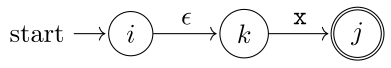
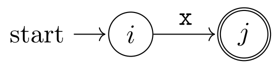
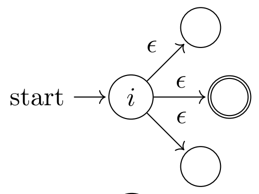
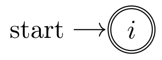


Figure 2: State u should be eliminated.

Let $R_{i,j}$ be the label on the transition from state s_i to state t_j for $1 \leq i \leq n$

Table 4: Converting ϵ-NFA to NFA.

	Figure
Old FA	
New FA	
Old FA	
New FA	

and $1 \leq j \leq m$. Label an arc with $\varnothing$ if there is no possible transition between two given states. Note that the path from s_i through u to t_j is labeled by $S_i U^* T_j$. Then, we can eliminate u by replacing $R_{i,j}$ as the label on the arc from s_i to t_j with $R_{i,j}|S_i U^* T_j$. Repeat this process until only the start state and the accepting states remain. The new automaton will look like Figure 3.

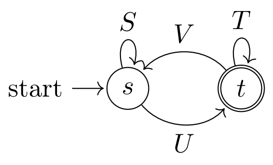


Figure 3: Simplified RE automaton.

Using Figure 3, it is clear that the regular expression matched by the automaton is

$$S^* U (T|VS^*U)^*.$$

If multiple accepting states exist, then the automaton is the union the regular expression, as described above, of each accepting state.

2 Parsing

Definition (Grammar). A grammar $G = \langle N, \Sigma, P, S \rangle$ where

- N is a finite set of non-terminal symbols (syntactic categories).
- Σ is a finite set of terminal symbols disjoint from N.

- P is a finite set of production rules of the form

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*.$$

- $S \in N$ is the start symbol from which strings will be generated.

Definition (Context-free Grammars). A context-free grammar is a grammar in which the left-hand side of each production rule consists of only a single nonterminal symbol.

Definition (Language of a Grammar). Start by assuming that for each syntactic category $<S>$ in the grammar, the language $L(<S>)$ is empty.

Suppose the grammar has a production $<S> \longrightarrow X_1 X_2 \cdots X_N$ where each X_i is either a syntactic category or a terminal. For each $i = 1, 2, \ldots, n$ select a string s_i for X_i as follows:

1. If X_i is a terminal, then we may only use X_i as the string s_i.
2. If X_i is a syntactic category, then select as s_i any string that is already known to be in $L(X_i)$. If several of the X_i's are the same syntactic category, we can pick a different string from $L(X_i)$ for each occurrence.

Then, the concatenation $s_1 s_2 \cdots s_n$ of these selected strings is a string in the language $L(<S>)$. Note that if $n = 0$, then we put ϵ in the language.

Example. The grammar of arithmetic expressions on unsigned integers is defined by the following set of productions:

(1) $<\texttt{Digit}> \longrightarrow 0|1|2|3|4|5|6|7|8|9$

(2) $<\texttt{Number}> \longrightarrow <\texttt{Digit}>$
(3) $<\texttt{Number}> \longrightarrow <\texttt{Number}>\ <\texttt{Digit}>$

(4) $<\texttt{Expression}> \longrightarrow <\texttt{Number}>$
(5) $<\texttt{Expression}> \longrightarrow <\texttt{Expression}> + <\texttt{Expression}>$
(6) $<\texttt{Expression}> \longrightarrow <\texttt{Expression}> - <\texttt{Expression}>$
(7) $<\texttt{Expression}> \longrightarrow <\texttt{Expression}> * <\texttt{Expression}>$
(8) $<\texttt{Expression}> \longrightarrow <\texttt{Expression}>\ /\ <\texttt{Expression}>$
(9) $<\texttt{Expression}> \longrightarrow (<\texttt{Expression}>)$

The terminal symbols are: $\{0, 1, \ldots, 0, +, -, \times, /, (,)\}$, and the syntactic categories are $<\texttt{Expression}>$, $<\texttt{Number}>$, $<\texttt{Digit}>$. The syntactic category, $<\texttt{Expression}>$, will be treated as the start symbol since it generates the strings that we intend to define with the grammar.

The first few iterations to generate $L(<\texttt{Expression}>)$ are shown below:

	$L(<$`Expression`$>)$	$L(<$`Number`$>)$	$L(<$`Digit`$>)$
1	$\varnothing$	$\varnothing$	0-9
2	$\varnothing$	0-9	0-9
3	0-9	0-9, 00, 01, ..., 99	0-9
4	(See below)	1,2,3-digit comb. of 0-9	0-9

On the 4th, iteration, $L(<$`Expression`$>)$ will contain all 1,2-digit combinations of 0-9, as well as any legal arithmetic combination of 1-digit numbers, e.g. $1 + 2$, $3/4$, (5), etc.

2.1 Parse Trees

Definition (Parse Tree). For a given grammar G,

- Leaves are labeled with the terminals of G or ϵ
- Internal nodes a labeled with syntactic categories
- Each internal node and its children correspond to some production in G
- Usually G has an infinite set of parse trees

The *yield* of a parse tree is the string formed by concatenating all the leaves in left to right order.

2.1.1 Constructing Parse Trees

Base Case. For any terminal, x, of the grammar, there is tree with one node labeled x.

Induction. Suppose we have a production $<S> \rightarrow X_1X_2\cdots X_n$, where X_i is either a terminal or a syntactic category. If $n = 0$, then, $<S> \rightarrow \epsilon$, so we can make a tree with root $<S>$ and a single leaf labeled ϵ.

For $n \geq 1$, we can choose a tree T_i for each X_i as follows:

1. If X_i is a terminal choose the 1-node tree labeled X_i
2. Else, we choose any parse tree already constructed that has X_i labeling the root.

Then. we construct a tree with root labeled $<S>$ and give it as children, the roots of the trees selected for each X_i in order, left to right.

Theorem 2.1

The set of all yields of parse trees with root $<X>$ is equal to $L(<X>)$.

Proof. See *FoCS* p. 607-8. ∎

2.2 Ambiguity

Definition ((Un)ambiguous). A grammar, G, is ambiguous if there is more than one parse tree for each string in G. In an unambiguous grammar, any string has no more than one parse tree.

A language for which every grammar is ambiguous is called *inherently ambiguous.*

Remark 2.1. An ambiguous grammar means some string in the grammar has more than one parse tree. However, different parse trees for the same input normally impart different meanings. Thus, it is important to remove ambiguity when defining a grammar. See *FoCS* p. 614 for an unambiguous grammar for arithmetic expressions.

2.3 Recursive-Descent Parsing

Process by which a grammar is replaced by a collection of mutually recursive functions, each corresponding to one of the syntactic categories of the grammar. The goal of the function S that corresponds to $<S>$ is to read a sequence of input characters from a string in $L(<S>)$, and to return a pointer to the root of a parse tree for that string.

Suppose we want to determine if the sequence of terminals $X_1 \cdots X_n$ is a string in $<S>$ and to find the parse tree if so. In an input file, we'd place an *endmarker* at the end of the sequence, $X_1 \cdots X_n$`ENDM`. An *input cursor* marks the terminal to be processed.

For each syntactic category $<S>$, we can design the function S to do the following:

1. Examine the lookahead symbol and decide which production to try. Suppose the chosen production has body $X_1X_2 \cdots X_n$.
2. For $i = 1, 2, \ldots, n$ do the following with X_i
   (a) If X_i is a terminal, check that the lookahead symbol is X_i. If so, advance the input cursor. Otherwise, S fails.
   (b) If X_i is a syntactic category, such as $<T>$, then call the function T corresponding to this syntactic category. If T returns with failure, then the call to S fails. If T returns successfully, store away the returned tree for use later.

If we have not failed then assemble a parse tree to return by creating a new node, with children corresponding to $X_1X_2 \cdots X_n$, in order. If X_i is a terminal, then its child is a newly created leaf with label X_i. Otherwise, the child for X_i is the root of the tree that was returned when a call to the function for X_i was completed.

2.4 Table-Driven Parsing

A *parse table* is a table whose rows correspond to syntactic categories and whose columns correspond to lookahead symbols. The entry in the row for $<S>$ given lookahead X is the number of the production with head $<S>$ if the lookahead is X. Some entries are left blank to indicate a failed parse. A *table driver* keeps a stack of grammar symbols are operates as follows:

1. Initialize a stack with the start symbol, $<S_G>$.
2. If the item on top of the stack is a terminal, `x`, check if the lookahead is `x`. If so, pop `x` and read the next input terminal.
3. If the item on top of the stack is a syntactic category, $<S>$, we satisfy $<S>$ by consulting the entry in row $<S$ and the column for the lookahead symbol of the parse table.
   a) If the entry is blank, the parse fails.
   b) If the entry is i, then pop $<S>$ from the stack and push each of the symbols in the body of production i onto the stack, rightmost first. Note is the body is ϵ then nothing is pushed onto the stack.

Example. Consider the grammar:

$$\begin{array}{ll}
(1) & <S> \to \texttt{w c}<S> \\
(2) & <S> \to \texttt{\{} <T> \\
(3) & <S> \to \texttt{s ;} \\
\hline
(4) & <T> \to <S>\ <T> \\
(5) & <T> \to \texttt{\}}
\end{array}$$

which has the following parse table.

	w	c	{	}	s	;	∅
$<S>$	1		2		3		
$<T>$	4		4	5	4		

where ∅ is the endmarker. Then, the table below shows the first 10 states of the table driver while parsing the string: `{w c s ; s ; }`∅.

	Stack	Lookahead	Remaining Input
1)	$<S>$	`{`	`wcs;s;}`∅
2)	`{`$<T>$	`{`	`wcs;s;}`∅
3)	$<T>$	`w`	`cs;s;}`∅
4)	$<S><T>$	`w`	`cs;s;}`∅
5)	`wc`$<S><T>$	`w`	`cs;s;}`∅
6)	`c`$<S><T>$	`c`	`s;s;}`∅
7)	$<S><T>$	`s`	`;s;}`∅
8)	`s;`$<T>$	`s`	`;s;}`∅
9)	`;`$<T>$	`;`	`s;}`∅
10)	$<T>$	`s`	`;}`∅

The process for building a parse tree is as follows:

1 Initially, the stack contains $<S>$. We initialize the parse tree to have one node labeled $<S>$. The <S> on the stack corresponds to the one node of the parse tree being constructed.

2 If the stack consists of symbols $X_1X_2 \dots X_n$, with X_1 at the top, then the current parse tree's leaves, taken from left to right, have labels that form a string s of which $X_1X_2 \dots X_n$ is a suffix. The last n leaves of the parse tree correspond to the symbols on the stack, so that each stack symbol X_i corresponds to a leaf with label X_i.

3 If $<S>$ is on top of the stack, then we replace $<S>$ by the body of a production $Y_1Y_2 \dots Y_n$. We find the leaf of the parse tree corresponding to this $<S>$ (it will be the leftmost leaf that has a syntactic category for label), and give it n children, labeled $Y_1Y_2 \dots Y_n$, from the left. In the special case that the body is ϵ, we instead create one child, labeled ϵ.

2.5 Making Grammars Parsable

Since left recursion leads to infinite recursion in recursive-descent parsers it must be eliminated. In the case of $<N> \to <N><D> \mid <D>$, this can be changed to $<N> \to <D><N> \mid <D>$ without ill-effect. However, since $<N>$ now has two productions starting with the same symbol, we must use a trick called *left factoring*. We do this by creating a new syntactic category $<T>$ for the tail of the production. Then, we rewrite the production as $<N> \to <D><T>$ where $<T> \to <N>|\epsilon$.

2.6 Pushdown Automaton

Definition. A pushdown automaton is a 7-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z, F\rangle$ where

- Q: Finite set of states
- Σ: Finite input alphabet
- Γ: Finite stack alphabet
- δ: Transition function: $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to 2^{Q\times\Gamma^*}$
- $q_0 \in Q$: Start state
- Z: Start symbol
- $F \subseteq Q$: Accepting states

Thus, a transition in a pushdown automaton is given by $\langle p, a, A, q, \alpha\rangle$, where p is the current state, a is the current input (possibly ϵ), A is the top of the stack, q is the next state, and α is the set of states added to the stack upon popping A. Note: A PDA is deterministic if there is at most one transition on any `c` from a given state/stack configuration, and if there is a transition on ϵ from a state/stack, then there cannot be any transition on another input `c`.

Definition (Language of a PDA). A string w is accepted by $M = \langle Q, \Sigma, \Gamma, \delta, q_0, F\rangle$ if $w = w_1 w_2 \cdots w_m$ for $w_i \in \Sigma \cup \{\epsilon\}$ and there exist $r_0, \ldots, r_m \in Q$ and $s_0, \ldots, s_m \in \Gamma^*$ such that

- $r_0 = q_0$ and $s_0 = \epsilon$
- $\langle r_{i+1}, b\gamma\rangle \in \delta(r_i, w_i, a\gamma)$ for some $\gamma \in \Gamma^*$
- $r_m \in F$.

Remark 2.2. There are languages that can be accepted by non-deterministic PDAs that cannot be accepted by deterministic PDAs. Hence, $L(DPDAs) \subset L(PDAs)$ and nondeterminism adds expressive power. An example is $\{ww^R | w \in (0|1)^*\}$.

Theorem 2.2

A language is context-free if and only if some pushdown automaton recognizes it.

Proof. Summarily, given some grammar G, we can construct a PDA $M = \langle Q, \Sigma, \Gamma, \delta, q_0, F\rangle$ as follows. Let

- $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\}$
- $\Sigma = \Sigma_G$
- $\Gamma = \Sigma_G \cup \Gamma_G \cup \{\$\}$, where Γ_G is the set of syn. categories in G.
- $F = \{q_{\text{accept}}\}$

and define δ as follows. Add the transition $\langle q_{\text{start}}, \epsilon, \epsilon, q_{\text{loop}}, S_G\$\rangle$. For every $A \in \Gamma_G$ and for every production of A, add $\langle q_{\text{loop}}, \epsilon, A, q_{\text{loop}}, Y_1 Y_2 \cdots Y_n\rangle$, where $Y_1 Y_2 \cdots Y_n$ is the body of any production of A. For every $s \in \Sigma$, add $\langle q_{\text{loop}}, s, s, q_{\text{loop}}, \epsilon\rangle$. Finally, add $\langle q_{\text{loop}}, \epsilon, \$, q_{\text{accept}}, \epsilon\rangle$.

Conversely, given some pushdown automaton M, we want to construct a grammar G such that the language of nonterminal $A_{p,q}$ is the set of strings that takes M from state p to state q, leaving the stack unchanged.

- Add production $A_{p,q} \to \mathtt{a}A_{r,s}\mathtt{b}$ if $\langle r, t\rangle \in \delta(p, \mathtt{a}, \epsilon)$ and $\langle q, \epsilon\rangle \in \delta(s, \mathtt{b}, t)$ for all $p, q, r, s \in Q, t \in \Gamma$ and $\mathtt{a}, \mathtt{b} \in \Sigma$ (Stack was never empty).
- Add $A_{p,q} \to A_{p,r}A_{r,q}$ for all $p, q, r \in Q$ (Stack got emptied).
- Add $A_{p,p} \to \epsilon$ for all $p \in Q$.

∎

Therefore, PDAs and CFGs have equal expressive power. For a full proof, see Sipser 2.2.

2.7 CFLs and Regular Expressions

Every regular language is context-free since all NFAs are PDAs that ignore their stack.

We can convert a given RE into a CFG according to the following tables:

Atomic Operands	Grammar
c	$<S> \to$ c
ϵ	$<S> \to \epsilon$
$\varnothing$	Nothing

Let R_1, R_2 be regular expressions with equivalent grammars G_1 and G_2 having start symbols S_1 and S_2, respectively.

Operator	Grammar
$R_1 \mid R_2$	$<S> \to <S_1> \mid <S_2>$
$R_1 R_2$	$<S> \to <S_1><S_2>$
R_1^*	$<S> \to <S_1><S> \mid \epsilon$

Theorem 2.3: Pumping Lemma

Let L be a regular language. Then, there exists a $p \geq 1$ such that for every string $s \in L$ with $|s| > p$, there exists x, y, and z such that:

$$s = xyz \qquad y \neq \epsilon \qquad |xy| \leq p$$

and for every $i \geq 0$, we have $xy^iz \in L$.

Example.

- The language $E = \{0^n 1^n | n \geq 1\}$ is not regular by the pumping lemma.
- The language $F = \{0^n 1^n 2^n | n \geq 1\}$ is not a CFL.

3 Turing Machines

Definition (Turing Machine). A Turing machine is a 7-tuple, $\langle Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}} \rangle$, where

- Q is a finite set of states,
- Σ is a finite input alphabet not containing ϵ,
- Γ is the tape alphabet, with $\epsilon \in \Gamma$ and $\Sigma \subseteq \Gamma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_0 \in Q$ is the start state,

- $q_{\text{accept}} \in Q$ is the accept state, and
- $q_{\text{reject}} \in Q$ is the reject state with $q_{\text{reject}} \neq q_{\text{accept}}$.

A *configuration* of a Turing machine is given by $C = uqv$ where $q \in Q$ is the current state and $u, v \in \Gamma^*$ describe the tape contents. The first symbol of v is the current scanning symbol. For configurations, C_1, C_2 we write $C_1 \vdash C_2$ if C_1 yields C_2 in one step, justified by the transition function. Similarly, we can write $C_1 \vdash^* C_k$ if $C_1 \vdash C_2 \vdash \cdots \vdash C_{k-1} \vdash C_k$. Then, we can write the language of a Turing machine M as

$$L(M) = \{w | q_0 w \vdash^* C_k = uq_{\text{accept}}v\}.$$

We say M halts on input w if $q_0 q \vdash^* C_k$ where C_k is a halting state (i.e. accept or reject).

Definition. A language L is *recognizable* or *recursively enumerable* if some Turing machine M recognizes it. That is, M accepts w iff $w \in L$. Note M may or may not halt if $w \notin L$.

A language L is *decidable* or *recursive* if some Turing machine M decides it. That is, for every input w, M accepts w iff $w \in L$ and M rejects w otherwise.

Note that every decidable language is, by definition, recognizable.

3.1 Variants on the Turing Machine

- Multitape TM: Redefine $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$, where there are k tapes. We can prove that every multitape TM has an equivalent single-tape TM by simulating the MT-TM's behavior. Let $M = \langle Q_M, \Sigma_M, \Gamma_M, \delta_M, q_{M,0}, q_{M,a}, q_{M,r} \rangle$ be a k-tape MT-TM. Let S be the equivalent single-tape TM. Let $\Sigma_S = \Sigma_M$ and $\Gamma_S = \Gamma_M \cup \{\dot{x} : x \in \Gamma_M\} \cup \{\#, \dot{\#}\}$. We simulate the behavior as follows:
  1) On input $w = w_1 w_2 \cdots w_n$, S puts its take into a format that represents all the tapes of M. Initially, $\#\dot{w}_1 w_2 \cdots w_n \# \dot{\llcorner\!\lrcorner}$. The dots mark where the heads of M are on each tape.
  2) S scans its tape from the first to $(k+1)$st $\#$, which marks the end of the input, to determine the position of the virtual heads. Then, it makes a second pass to update the tapes according to M.
  3) S may shift tape contents to the right to extend the tape.

  This shows that MT-TM and single-tape TM have equivalent expressive power.
- Non-determinism: In this case we have: $\delta : Q \to \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$. Given a NTM N, we can construct a tree of all possible configurations of N with branches of the tree representing branches in ND. To simulate N with a deterministic TM D we can perform a BFS of N's tree. D will use three tapes: a fixed input tape, a simulation tape which stores the contents of N's tape, and an address tape that keeps track of D's location in N's tree; in particular it enumerates the tree-paths in lexicographical order.

1) Initialize tape 1 to input w and copy tape 1 to tape 2.
2) Use tape 2 to simulate N on w on one branch of its ND tree given from tape 3. If N enters an accepting configuration then accept, else go to step 3.
3) Increment tape 3, carrying over as required; go to step 2.

Thus NDTMs and DTMs have equivalent expressive power.

- Enumerator: Given an enumerator E we construct a TM M:

  1) Run E. Every time E outputs a string, compare it to input w. If E every outputs w, M accepts w.

  Given a TM M, for $i = 1, 2, 3, \ldots$ an equivalent E will:

  1) Run M for i steps on the first i strings in Σ^*.
  2) If M accepts a string, print it.

  Thus, enumerators and TMs have equal expressive power.

Summary. A language is recognizable iff some

- multitape Turing machine recognizes it
- non-deterministic Turing machine recognizes it
- enumerator enumerates it.

3.2 Church-Turing Thesis

Turing machines and λ-calculus are equally expressive. In fact, any two computational models that satisfy certain reasonable requirements can simulate one another and have equal expressive power. Hence, there is an algorithm for testing membership in a language L iff there is a Turing machine that decides L iff L is decidable. Thus, algorithms are the set of decidable languages.

3.3 Decidability

Given a TM $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r \rangle$ we need a procedure to encode M as a string.

- Choose minimal i, j such that $|Q| \leq 2^i$ and $|\Gamma| \leq 2^j$. Encode states as $\mathtt{q}\{0,1\}^i$. Encode tape symbols as $\mathtt{a}\{0,1\}^j$.
- To encode the transition $\langle q, a, p, b, \{L, R\} \rangle$ we write $(\langle q \rangle, \langle a \rangle, \langle p \rangle, \langle b \rangle, \{0, 1\})$, where $L \mapsto 0$ and $R \mapsto 1$. We encode δ by listing the encoded transitions of δ in state-lexicographical order, starting with $\delta(q_0, \textvisiblespace)$.
- The entire encoding alphabet is $\{\mathtt{q}, \mathtt{a}, 0, 1, \mathtt{,}, (,)\}$.

Theorem 3.1

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle | B \text{ is a DFA that accepts } w\} \text{ is decidable.}$$

M_{DFA}:

1) Simulate B on w
2) M_{DFA} accepts iff B accepts.

$$A_{\mathsf{NFA}} = \{\langle B, w\rangle | B \text{ is a NFA that accepts } w\} \text{ is decidable.}$$

M_{DFA}:

1) Convert B to a DFA C
2) M_{NFA} accepts iff M_{DFA} accepts $\langle C, w\rangle$.

$$A_{\mathsf{REX}} = \{\langle R, w\rangle | R \text{ is a regexp that generates } w\} \text{ is decidable.}$$

M_{REX}:

1) Convert R to an NFA A
2) M_{REX} accepts iff M_{NFA} accepts $\langle A, w\rangle$.

$$E_{\mathsf{DFA}} = \{\langle A\rangle | A \text{ is a DFA and } L(A) = \varnothing\} \text{ is decidable.}$$

M_{EDFA}:

1) Mark the start state of A
2) While at least one new state is marked: Mark any state that has a transition coming into it from a previously marked state
3) M_{NFA} accepts iff no accepting state is marked.

Exercise. $EQ_{\mathsf{DFA}} = \{\langle A, B\rangle | A, B \text{ are DFAs and } L(A) = L(B)\}$ is decidable. *Hint*: Define a new DFA C that accepts the symmetric difference of $L(A)$ and $L(B)$.

Theorem 3.2

$$A_{\mathsf{CFG}} = \{\langle G, w\rangle | G \text{ is a CFG that generates } w\} \text{ is decidable.}$$

M_{CFG}:

1) Convert G to Chomsky Normal Form
2) Consider all derivations of length up to $2|w| - 1$
3) M_{CFG} accepts iff some derivation generates w.

$$E_{\mathsf{CFG}} = \{\langle G\rangle | G \text{ is a CFG and } L(G) = \varnothing\} \text{ is decidable.}$$

M_{ECFG}:

1) Mark all terminals in G
2) While at least one new symbol is marked: Mark any symbol A where G has a rule $A \to U_1 \cdots U_k$ and $U_1, \ldots, U_k$ are already marked
3) M_{ECFG} accepts iff the start variable is unmarked.

Every CFL is decidable.

M_{CFL}: Let L be a CFL. Then there exists a CFG G such that $L = L(G)$.

1) M_{CFL} accepts iff M_{CFG} accepts $\langle G, w\rangle$, where $w \in L$.

3.4 Halting Problem

Definition (Universal Turing Machine). Let U be a 3-tape TM that takes in two arguments: $\langle M\rangle$ an encoding of a TM, and $\langle w\rangle$ an encoding of the string w. We design U such that U accepts $\langle M, w\rangle$ iff M accepts w and U rejects $\langle M, w\rangle$ iff M rejects w.

1) Initialize Tape 1 to $\langle M, w\rangle$.
2) Let Tape 1 store $\langle M\rangle$ and Tape 2 store $\langle w\rangle$.
3) Let Tape 3 store $\langle M\text{'s state}\rangle$.
4) Simulate the steps of M.

Theorem 3.3

Let $A_{\mathsf{TM}} = \{\langle M, w\rangle | M \text{ is a TM that accepts } w\}$. Then A_{TM} is recognizable.

Proof. Define M_{TM} by

1) Use U to simulate M on w.
2) If U accepts, accept. If U rejects, reject.

∎

Theorem 3.4

A_{TM} is undecidable.

Proof. Suppose the exists a machine H that decides A_{TM}. Define $D(\langle M\rangle)$ such that D runs H on $\langle M, \langle M\rangle\rangle$ and if H accepts, then D rejects, and if

H rejects then D accepts. Then $D(\langle D\rangle)$ accepts if D does not accept $\langle D\rangle$ but it rejects if D accepts $\langle D\rangle$. A contradiction, so D and H cannot exist. Thus no TM can decide A_{TM}. ∎

Hence the set of recursive languages is a proper subset of the recursively enumerable languages. Indeed, there are other undecidable languages and we typically prove undecidability via reduction. Hence, if problem A is reducible to problem B and B is decidable, then A is decidable. Alternatively, if A is undecidable, then B is undecidable.

Theorem 3.5

The complement $\bar{A}_{\mathsf{TM}}$ of A_{TM} is not recursively enumerable.

Proof. Since A_{TM} is recognizable but A_{TM} is undecidable, it must be the case that $\bar{A}_{\mathsf{TM}}$ is unrecognizable since a language is decidable iff both it and its complement are recognizable. ∎

4 Databases

Databases are the relational data model. A *relation* is a set of tuples of fixed arity. Individual fields within a tuple are called *attributes*. The *relational schema* of a relation is the set of attributes. For example, we can define the relation of 3-tuples with attributes: Course, StudentID, Grade. The schema for this relation would be {Course, StudentID, Grade}. Typically, we depict relations using tables, whose columns are the attributes of a relation and whose rows are the tuples.

A *database* is simply a collection of relations. The schema of a database is simply the set of schema for its relations. Most all databases support operations for

1) *insert*(t, R). Adds tuple t to relation R if not already present.
2) *delete*(X, R). Deletes all tuples, matching specification X, within R. A wildcard symbol, $*$, is used to say any value is acceptable.
3) *lookup*(X, R). Returns a list of all tuples in R matching specification X.

A *key* for a relation is a set of one or more attributes which uniquely map to a given tuple. For example, StudentID is a key. A *primary index structure* for a relation stores the tuples of a relation. A data structure which, given certain attributes, helps find tuples within the primary that match those attributes is a *secondary index*. A secondary index on attribute A of relation R is a set of pairs (v, p) where v is a value for attribute A and p is a pointer to a tuple in the primary index whose A-component has value v. Secondary indexing facilitates the lookup of commonly executed queries.

4.1 Relational Algebra

The relational algebra consists of operands: the relations and variables ranging over them; and operators: union, intersect, difference, select, project, and join. The union, intersect, and difference operators can only be applied to relations with identical schema.

- Select $\sigma_C(R)$: Returns tuples from R matching condition C.
- Project $\Pi_A(R)$: Returns the attribute(s) A from tuples in R, removing duplicates as required.
- Join $R \bowtie_{A_i=B_j} S$: For every $r \in R$ and $s \in S$, if $A_i \in r$ matches $B_j \in s$, then add the tuple combining r and s. The resultant relation has arity $n+m-1$, if $|R|=n$ and $|S|=m$.

Note that the natural join operator $R \bowtie S$ may be used to join on all common attributes of R and S.

4.1.1 Algebraic Laws

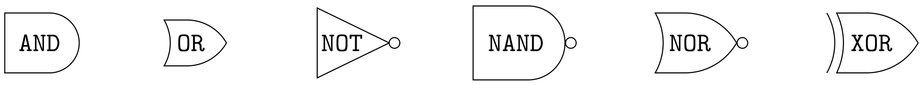
- $R \bowtie S \equiv S \bowtie R$
- $\sigma_{C \texttt{ AND } D}(R) \equiv \sigma_C(\sigma_D(R)) \equiv \sigma_D(\sigma_C(R))$
- $\sigma_C(R \bowtie S) \equiv \sigma_C(R) \bowtie S$ (if all C are in R)
- $\Pi_A(R \cup S) \equiv \Pi_A(R) \cup \Pi_A(R)$
- $\Pi_L(R \bowtie_{A=B} S) \equiv \Pi_L(\Pi_M(R) \bowtie_{A=B} \Pi_N(R))$ where: (1) $M = \{\text{attributes of } L \text{ from } R\} \cup \{A\}$, (2) $N = \{\text{attributes of } L \text{ from } S\} \cup \{B\}$

Algebraic laws of relation algebra help optimize queries. Note all queries can be represented via an expression tree.

Implementation

- Union, Difference, Intersect: Start with relation R, iterate through its tuples and comparing with tuples in S indexing on the key to give constant time comparisons. Expected runtime $\mathcal{O}(|R|+|S|)$.
- Projection: Iterate through tuples of R, hashing to avoid duplicates. Expected runtime $\mathcal{O}(|R|)$.
- Selection: If an index on the conditional attribute is available this is $\mathcal{O}(1)$, otherwise we must iterate through R, for $\mathcal{O}(|R|)$.
- Join: Naively $\mathcal{O}(|R||S|)$. Using an index join, this improves to $\mathcal{O}(m+|R|)$, where m is the size of the join. If no index is available, sort join gives $\mathcal{O}(m + (|R|+|S|)\log(|R|+|S|))$.

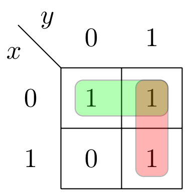
5 Boolean Circuits



A *boolean function* is a function $f : \{0,1\}^n \to \{0,1\}$. There are 2^{2^k} boolean functions of k variables. Let f be an arbitrary boolean function of k variables, having n boolean operators. Then testing whether f is a tautology is $\mathcal{O}(n2^k)$.

A *Karnaugh map* is a graphical method of constructing a boolean expression of a boolean function with up to 4 variables. For example, if $f(A, B)$ is a boolean function, then the Karnaugh map of f is a $2d$-array with the values of A as rows and the values of B as columns and $f(A, B)$ as entries. An *implicant* of a boolean function f is a product x of literals for which no assignment of the variables of f make x true and f false. That is, x implies f. Implicants are said to *cover* the points from which f has value 1.

Example. Suppose $f(x, y) = x \to y$. Then the corresponding Karnaugh map is

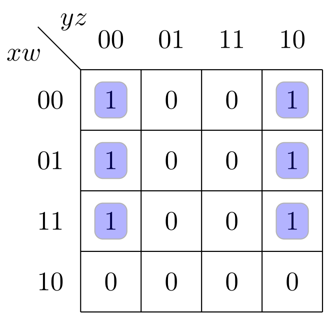


$x \backslash y$	0	1
0	1	1
1	0	1

The implicants cover the values for which $x = 0$ or $y = 1$. Thus, $f(x, y) \equiv \bar{x} + y$. By selecting different implicants we also get $f(x, y) \equiv \bar{x}\bar{y} + y \equiv xy + \bar{x}$.

Note that the dimension of the rectangles are always a power of 2 and are as large as possible. Furthermore, every 1 is covered by at least one rectangle, preferably with minimal overlap. An implicant $x = l_1 l_2 \cdots l_k$ is *prime* if x isn't an implicant whenever any literal l_i is deleted.

A circuit is *combinational* if its graph has no cycles; otherwise, it is *sequential.*



$xw \backslash yz$	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	1	0	0	1
10	0	0	0	0

Figure 4: Using 1×1 implicants is equivalent to the min-term method.
$\bar{x}\bar{y}\bar{z}\bar{w} + \bar{x}y\bar{z}\bar{w} + \cdots + xy\bar{z}w$