CS 3133 Foundations of Computer Science

Christopher Myers

June 13, 2020

Contents

1 Functions

A function f from $A \to B$ is a set of ordered pairs (a, b) such that for every single a in A there exists exactly one b in B. The mapping must be 1-to-many — there can be one output with multiple inputs, but there cannot be one output with multiple inputs. The domain cannot have leftover points, but the codomain can. Mathematically, $\forall a \in A$, $\exists! b \in B$.

For traditional math functions (e.g. $y = x^2$) a good test is the vertical line test: drop a straight vertical line at an arbitrary point on the graph. If the new line intersects the function's line no more than once, it's a function.

1.1 Composition of Functions

If you have a function f $A \to B$ and another function g $B \to C$, the composition of the two functions $g \circ f$ is $A \to B$. Formally, $(g \circ f)(a) = g(f(a))$.

Note that composition is not commutative — that is, $f \circ g \neq g \circ f$, at least not in every case anyways. This is in part because functions do not necessarily share domains or codomains, but even if they do exactly share domains and codomains, the commutative property does not necessarily apply.

1.2 Identity function

The identity function is simple: $id : A \to A$, or $id(a) = a$. id can be used for composition too, in that $f \circ id = f$, as is $id \circ f = f$.

1.3 Injective and Surjective Functions

Injective functions are also called one-to-one, while surjective functions are also called "onto" functions. A function $f : A \to B$ is injective if every element in the domain has excatly one corresponding element in the codomain: $a_1 \neq a_2$ implies $f(a_1) \neq f(a_2)$. Geometrically, this would look like a function where for every y there is exactly one x, so $y = e^x$ is injective but $y = x^2$ is not.

A function $f : A \to B$ is surjective if every element in B is used up: $\forall b \in B$, $\exists a \in A$ such th—*incomplete*. Geometrically this means the whole range must be covered, but there can be y values with multiple x values. A function that is surjective but not injective might be a curvy x^2 function (something with a hump in it).

1.4 Inverses

The inverse of a function $f : A \to B$ would be a function $g : B \to A$, with the property that the composition of the two functions together in either direction results in the identity function.

To build an inverse, suppose you have a function $f : A \to B$ that maps some domain A onto some codomain B. If you want to map B onto A instead, the first thing to try is to define $g : B \to A$ by $g(b) =$ the a such that $f(a) = b$. The problem is the world "the" here — if f isn't injective, there might be multiple a's that result. If f isn't surjective, there might not even *be* a corresponding a.

Now suppose that f *is* surjective, but not injective, i.e. everything in B is used up but some B values have multiple A values. That means that an inverse function will always produce at *least* one A value for each B value — suppose that to resolve this, choose any a such that $f(a) = b$. Under this circumstance, it is true that $f \circ g = id_B$, but it is not true that $g \circ f = id_A$, so this is not an inverse.

What if f is injective but not necessarily surjective? Conceptually this looks like a 1-to-1 mapping from A to B, but not every element in B is used. When building $g(b)$, if there is some a, use $f(a) = b$, use that; otherwise, assign a random a. In this case, it is true that $g \circ f = id_A$, but it is not necessarily true that $f \circ g = id_B$.

Conclusion:

- If f is injective, f has a left inverse, $g \circ f = id_A$.
- If f is surjective, f has a right inverse, $f \circ g = id_B$.
- If f is both injective and surjective (i.e. bijective), f has an inverse.

These can also be reversed: for example, if f has a left inverse, f is injective.

1.5 Characteristic Functions

For some subset S part of the universe U ($S \subseteq U$), the characteristic function of s $ch_s(u)$ returns, for an element u of U, 1 if the element is part of S and 0 otherwise. Really all it answers is the question of whether an element is part of the set or not. For instance, if U is all real numbers and S is all prime numbers, the characteristic function will return 1 if the input number is prime and 0 if it's not prime. Another example is the characteristic function of the empty set, which always returns 0, and the characteristic function of the universe U, which always return 1.

2 Strings & Languages

2.1 Definitions

An alphabet is defined as a finite set of symbols, e.g. $\{0, 1\}$ or the entirety of ASCIIȦlphabets are denoted as Σ. A string is then a finite sequence of symbols from an alphabet, and the set of all strings is written as Σ^*. A language (over an alphabet) E is defined as the set of strings from within the set of all strings that conform to some set of rules. For example, you might write a language as $\{x \in \Sigma^* |$ length of x is even $\}$. These rules can be arbitrarily complex — for instance, you might define a language "Java" as the set of all ASCII strings where a given string is a legal Java program.

To summarize: Σ is an alphabet, a set of all strings from it is Σ^*, and languages are $A \subseteq \Sigma^*$.

2.2 Operations on Strings

Strings can be concatenated together, basically just adding one onto the end of the other. Concatenation is associative ($(xy)z = x(yz)$), but they are not commutative ($xy \neq yx$). Empty string is denoted as λ and is used in the identity function $x\lambda = x = \lambda x$. Finally, the length of a string is expressed using vertical bars, e.g. $|\lambda| = 0$.

2.3 Operations on Languages

If $A, B \subseteq \Sigma^*$, normal logic rules on sets apply. That is, you can do $A \cup B$ to get a union, $A \cap B$ to get an intersection, or even $\bar{A}$ to get a compliment of A. You can also take $AB = \{xy | x \in A, y \in B\}$. $A*$ can be written as $\{x | x$ can be be written $z_1, z_2 \ldots z_k, k \geq 0$ each $z_A \in A\}$. Note that $\lambda \in A*$ for any A. An empty language is not denoted as λ but instead as $\emptyset$.

2.4 Regular Expressions

Regular expressions (or regexes for short) are constructs used to define a language, instead of using something like set notation. Technically, they're a language all on their own, formally, for an alphabet Σ, the base regular expressions on Σ are:

- $\emptyset$ — The empty language
- For $a \in \Sigma$, a denotes $\{a\}$
- λ denotes $\{\lambda\}$
- If E_1 and E_2 are regexes, then $E_1 \cup E_2$ is a regular expression, and $L(E_1) \cup L(E_2)$ (the union of the languages defined) is a language.
- If E_1 and E_2 are regexes, E_1E_2 is as welll and denotes the concatenation of the two, just as $L(E_1)L(E_2)$ denotes the concatenation of their respective languages.
- If E is any regular expression, then $E*$ is a regular expression, and it denotes $(L(E))*$.

These are the base rules of regexes, but more can be done with the various symbols. Suppose you have an alphabet $\Sigma = \{a, b\}$:

- a^* denotes $\{\lambda, a, aa, aaa, \dots\}$ — the finite set of all iterations of that string, including the empty set. The same thing applies to b, naturally.
- $(ab)*$ denotes $\{\lambda, ab, abab, ababab, \dots\}$
- $a^* \cup b^*$ denotes $\{\lambda, a, aa, aaa, \dots, b, bb, bbb \dots\}$.
- $(a \cup b)*$ might look similar to the previous, but instead it denotes all of Σ^*, since you can take any a, b string whatsoever and match the regex to all of its contents.
- $(a \cup b) * b$ denotes all strings that end in b. Similarly, $a(a \cup b)*$ denotes all strings that start with a.
- The set of all strings that begins with a or ends with b is denoted by $a(a \cup b) * \cup (a \cup b) * b$.
- The set of all strings that start with a and end with b is denoted by $a(a \cup b) * b$. Note intersections are not available here, but technically you can achieve their effects usine deMorgan's laws.

3 Finite Automatons

3.1 DFA's: Deterministic Finite Automatons

DFAs are the simplest possible computing machine, but they're not as powerful as Turing machines. DFAs are traditionally referred to as $M = (\Sigma, Q, \delta, s, F)$, where Σ is a finite alphabet, Q is a finite set of states, s is the start state in Q, δ is a function from $Q \times \Sigma \to Q$, and F is a subset of Q that forms the accepting states. δ basically says that if you give it an input state and a symbol, it'll give you the next state of the machine. These are often given in the form of state transition tables. Note that no state can be a "dead end", as the δ function is obligated to always return a state given another state and a character.

The whole input string to the DFA is considered valid if an only if the end state of the DFA is an accepting state. DFAs can therefore be used to define languages — the language of a DFA M is the set of all strings such that the run of the DFA on a given string from the alphabet ends on an accepting state. Furthermore, a language L is said to be **regular** if it can be accepted by some DFA˙

3.2 Intersection and Union of DFAs

Conceptually, the intersection of DFAs happens when you can run through the new machine and it accepts in both machines. Mathematically the machines are represented using the cartesian product o fthe states, an ordered pair of start states, and the cartesian product of the accepting states. δ is $(q_1, q_2) \to^a (r_1, r_2)$ if the transition works in both original DFAs. In practice this means that the intersection of two machines with two states each will have four states, and so on. This means that constructing the intersection of two DFAs is a strictly mechanical process that needs no real intuition to figure out, it's just following an algorithm.

The union of DFAs is idenitcal, except the accepting states aren't given by the intersection of the accepting states, but by the union.

3.3 NFAs: Nondeterministic Finite Automatons

NFA's are DFA's, except each state can have multiple transitions per input. They can be transformed into DFAs, and they only accept regular languages.

An NFA_λ is an NFA + λ transitions, where a λ transition is a silent transition — no input symbol is needed. Some books use λ as a label on the transition to denote that it's a lambda transition (also useful since λ denotes the empty string). Everything else about these NFAs is the same. These might seem more powerful at first, but it turns out that $NFA = NFA_\lambda$, and since NFAs and DFAs are also equivalent, NFA_λ's might be easier to work with but they're ultimately just as limited as DFA's. In fact, the process of converting an NFA_λ to an NFA only involves changing transitions, and not states.

To perform the translation, first construct an intermediary by following three rules. If there are silent transitions from $p \to q \to r$, form a new silent transition $p \to r$. If there are silent transitions $p \to q \to^a r$, form a new transition $p \to^a r$. Finally, if there are silent transitions to an accepting state, the origin state of that transition should also become an accepting state. Repeat these rules until there are no changes left to make, then remove all silent transitions and you're done!

3.4 Finite Automatons and Regular Expressions

Regular expressions can be turned into NFA_λ's with ease by following the expression straight through. Since NFA_λ's can be converted to NFA's and NFA's can be converted to DFA's, regular expressions can therefore be turned into DFA's. There is *also* an algorithm for turning DFA's into regular expressions, closing the loop and making all of these devices interchangeable.

3.4.1 NFA $\to$ Regular Expression

For some NFA M, produce a regex E such that $L(E) = L(M)$. First, think of some equations that define NFA — a state can be written as the union of a set of aX constructs, where a is the input character in the transition and X is the destination state of the transition. States that have no transitions away from them are represented as λ. For example, $S = aP \cup bQ$ represents a state S that transitions to P on a and Q on b. Next, just solve for your start state using basic rules of substitution. For instance, if $Q = aR$ and $R = \lambda$, then you can solve for $Q = a$. By the end of the process, all variables representing states should be resolved and you should be left with a regex.

Naturally, this can get complicated if you have states with transitions that form loops, especially if those loops can include the original state. If in the process of solving by substitutions you can't eliminate a state (i.e. you form a recursive equation), you can use Arden's Lemma. Let A and B be any languages, where λ is not in A (as a technical condition, it's okay to ignore it since we'll never need to check for it here). Under Arden's lemma, the equation $X = AX \cup B$ has the solution A^*B. For instance, you can solve $S = aa^* \cup bS$ as $S = b^*(aa^*)$. It may be helpful to use the commutative property of unions to get the equation into the right form. As another example, $S = a(b^*(aS \cup \lambda)) = ab^*aS \cup ab^*$, which turns into $(ab^*a)^*ab^*$.

If this process seems uncertain at times, there's simple advice: don't be clever. Just apply rules. For instance, if $S = a(aS) \cup b(aS)$, then you can shrink that down to $(aa \cup ba)S$.

4 Regular Languages

Regular languages are those that can be accepted by a DFA/NFA/NFA_λ. The key technique in determining whether a language is regular or not is called *distinguishability.* Suppose you a language K; two strings x, y are K-indistinguishable if for every string z, $xz \in K$ iff $yz \in K$. Alternatively, the two are indistinguishable if there exists a string z such that $xz \in K$ but $yz \notin K$ or vice versa. Note that this naturally depends on the language K.

Examples:

- $K = a^*b^*$, $x = abb$ and $y = aa$. If you take $z =' a'$, you can show that the two strings are K-distinguishable. Note that both $x \in K$ and $y \in K$.
- For the same K, take $x = aba$ and $y = ab$. If you take $z = \lambda$, then $x\lambda \notin K$ and $y\lambda \in k$.
- For the same K, $x = abb$ and $y = ab$. There is no string z that makes one in K but not the other, so these strings are K-indistinguishable.
- For a language of binary strings of even length, $x = 01$ and $y = 1001$, x and y are indistinguishable.

Why is this important? Suppose we consider a regular language $K \subseteq \Sigma^*$. Let M be a DFA accepting K. If x, y are strings and $\hat{\delta}(s, x) = \hat{\delta}(s, y)$ (i.e. x, y go to the same state in M), then $x \equiv_k y$. It follows that $\hat{\delta}(s, xz) = \hat{\delta}(s, yz)$. There are two further observations to make from this. First, if the number of equivalence classes is n, then any DFA for K has at least n states (and in fact it's a fact that the must be a DFA with exactly n states). Second, if there are infinitely many K equivalence classes, there cannot be a DFA for K.

Continuing on the notion of equivalence classes, consider the language $K = a^*b^*$, and the strings $x = \lambda$ and $y = a$. There's nothing you can do to put one in the language while putting the two out, so we call this an equivalence class. You can distinguish b from the first two strings, though, so you call that a new equivalence class. The same goes for ba, which falls into a third equivalence class. For the language a^nb^n, there are infinitely many classes.

This matters because of the following lemma: If K is regular, and M is a DFA, and $L(M) = K$, then whenevever $\hat{\delta}(s,x) = \hat{\delta}(s,y)$, $x \equiv_K y$. Intuitively, if the DFA takes x, y to the same state, the two must be indistinguishable. Suppose q is the state $\hat{\delta}(s,x) = q$ and $\hat{\delta}(s,y)$, and z is any string. Under this, $\hat{\delta}(s,xz) = \hat{\delta}(s,yz)$, so $xz \in K$ iff $yz \in K$. This means that if M is a DFA for K, the number of states in M must be greater than or equal to the number of equivalence classes in K. This is the formal explanation for what was covered two paragraphs ago.

This turns out to be the best way to show that something isn't regular — just prove that there are infinitely many equivalence classes, and therefore infinitely many states (violating the definition of a deterministic **finite** automaton).

5 Context Free Languages

A *grammar* specifies four things: Σ, V, P, and S. Σ is the finite alphabet as usual (although sometimes called the terminal alphabet), V is variables, P is productions (sometimes called rules), and S is the start symbol $\in P$. For example, a grammar where $V = \{S\}, \Sigma = \{a, b\}$, and the rules are $S \to aS$, $S \to Sb$, and $S \to \lambda S$. These rules should be read as "S can be replaced by ...". A *derivation* is a sequence of strings over $E \cup V$, starting with S, where each step is $\alpha X \beta \to a\gamma\beta$ if $X \to \gamma$. Basically, it's just the application of rules in steps to produce a string, usually with only terminals left. It's important to note that the context the variable is in doesn't matter — there's only ever a variable on the left side of the rule. This is why it's called a context-free grammar.

A language can be defined using CFG's — specifically, for a grammar G, $L(G) = \{x \in \Sigma^* | S \to^* x\}$. Note that context free languages can do everything that regular languages can, but not vice versa. In fact, for every regular language there's a corresponding context-free grammar. As an example, the grammar $S \to aSb|\lambda$[1] defines the language a^nb^n, which is impossible in a regular language.

As another example, try $V = \{E, I\}$ and $E = \{+, \times, 0, \ldots, 9\}$, with $S = \{E\}$. The rules are $E \to E + E | E \times E | I$, $I \to 0|1|2| \ldots |9$. This defines a simple language of math expressions on numbers, such as $3 + 2$.

Context free grammars lend themselves well to modularity. Suppose you have a language $\left\{a^nb^nc^k | n, k \geq 0\right\}$. We already have a rule that will do a^nb^n, so just call that $S_1 \to aS_1b|\lambda$. Now define a rule that handles the c's: $S_2 \to cS_2|\lambda$. To stich them together, define $S \to S_1S_2$ and declare the starting symbol to be S.

As another example, consider the language $\left\{a^ib^jc^kd^l | i = j \cup k = l\right\}$. The grammar will be $S \to S_1|S_2$ where S_1 will generate te first case and S_2 will generate the second. For the first case, where $i = j$, we already have the needed rules, so just stitch those together too (the rules are obvious and not shown).

5.1 Parse Trees & Ambiguity

Parse trees are data structures that are used to show derivations in a more structural fashion. Every time a new symbol is generated by the rules of the grammar, new nodes are added below the symbol they were generated from. Eventually all leaf nodes will be terminals, while the root node will be a nonterminal. Different parse trees can be generated for the same resulting string. Being the same derivation is irrelevant, it's the parse tree that matters.

A context-free grammar G is said to be *ambiguous* if there is at least one string $x \in L(G)$ that has more than one parse tree. Ambiguity is generally considered a bad thing in a grammar, since it can result in added complexity. Often you can scrub out the ambiguity from a language, but those that can't be cleaned up are "inherently" or "essentially" ambiguous.

[1]The vertical bar is shorthand for "or"

5.1.1 Heuristics for Removing Ambiguity

The first most important technique is to look at operator precedence, or the order of operations in your language (e.g. multiplication comes before addition). The trick to this heuristic is introduce more "levels" into the language, by saying that some operations have to come before others, and adding one-way nonterminals that you can't return from (e.g. an E that can do either $E * E$ or F, but you can't come back to E from F).

The second type of ambiguity is grouping ambiguity, where all your operators in a given string have the same precedence (perhaps because they are the same?). For instance, in the simple math grammar example given a while back (with subtraction added on), the string $2 - 3 - 5$ would be ambiguous due to grouping ambiguity — is it $2 - (3 - 5)$ or is it $(2 - 3) - 5$? To fix this problem, forbid this doubly-recursive pattern of $E \to E - E$. In this case, you could fix the problem by making the grammar left-recursive only: $E \to E - F$.

5.2 Refactoring Grammars

Grammars define languages, but sometimes it's useful to reverse the process and derive a grammar from a language, for the purpose of refactoring the grammar to clean it up. Let G be a context-free grammar; variable A is reachable if there is some derivation that could possibly lead to something that includes A — formally, if $S \to^* \alpha A \beta$. A is said to be *generating* if there is some derivation that leads to a series of terminals, and A is considerred useless if it is either unreachable or not generating. Naturally, the goal is to eliminate useless variables and rules using them.

There are two algorithms useful here, one to identify which variables are reachable and which variables are generating. The first is the algorithm for reachable variables, which has input $G = \{\Sigma, V, S, P\}$ and an output of a set of reachable variables V'.

- Initialize $V' = S$
- If there is a rule $A \to \alpha$, with $A \in V'$, add all variables in α to V'.
- Repeat the above step until there are no changes.

With this algorithm, you can remove from the grammar any rules involving a non-reachable variable. The second algorithm is for generating variables, which has input $G = \{\Sigma, V, S, P\}$ and the output is the set $V' \subseteq V$ of generating variables:

- Initialize $V' = \{\emptyset\}$.
- If there is a rule $A \to \alpha$ with everything in $\alpha \in (V'\Sigma)$, add A to V'. In other words, if either α is all terminals or it leads to something we already know is generating, add A to V'.
- Repeat the above step until there are no changes.

In general you should eliminate non-generating variables first, then move on to non-reachable, because often in the process of eliminating non-generating variables you'll make some variables unreachable when they were previously reachable.

Rules, too, can be refactored or eliminated to improve the languages. The main two types of undesirable rules are chain rules, where a variable simply produces another rule (e.g. $X \to Y$), and erasing rules, where variable leads to λ (e.g. $X \to \lambda$).

Here's an example language:

$$\begin{aligned} S &\to aS|a|B \\ B &\to bB|b|C \\ C &\to dC|\lambda \end{aligned}$$

A sample derivation in this language might be $S \to B \to bB \to bC \to b$; if you look at this language, it's intuitively obvious that it's an inefficient grammar. To eliminate chain rules, use the following algorithm (with output G' where G' has no change).

- Initialize $P^* = P$
- If P^* has $A \to B$ and $B \to \beta$, add $A \to \beta$ to P^*.
- Repeat the above step until there is no change.
- Return the new grammar with P', where $P' = P^* -$ chain rules.

This algorithm works and terminates because every rule of P^* in the form of $X \to \alpha$ has α as on original right side to a rule, and because there are only finite rules to a grammar.

On to eliminating erasing variables! The idea is that you have $X \to \lambda$ and $A \to \alpha X \beta$, you can just add $A \to \alpha\beta$. This algorithm starts with G, adds rules until you can get rid of erasing rules to get G^*, then reduce it down ot G' by removing the now unneeded erasing rules. There is one subtlety — if you have a chain of rules that could lead to an erasure (e.g. $X \to AB, A \to \lambda, B \to \lambda$), you'd still consider the root rule of the chain to be an erasing rule.

For the algorithm, start by defining variables as *nullable* if $X \to^* \lambda$.

- Initialize $V' = \{X | X \to \lambda is a rule\}$.
- If $A \to \alpha$ is a rule, an each elemnt of α is nullable, add A to V'.
- Repeat until there is no change.
- Identify nullables
- Whenever $A \to \alpha X \beta$ is a rule, and X is nullable, add $A \to \alpha\beta$.
- Erase all $X \to \lambda$ rules.

(the first half of the above might not be entirely accurate, I'm not sure how things are added to V')

5.3 Pushdown Automatons (PDAs)

Regular languages have regular expressions, which are represented by a DFA that accepts the language. Similarly, context-free languages have context-free grammars, which are represented by pushdown automatons (since DFAs are too limited to accept a context-free language). The point of a PDA is therefore to process strings to check if they're part of the language or not.

Suppose you want to define a language $a^n b^n, n \geq 0$. Intuitively, you could scan for a's and push them onto the stack. Once you start seeing b's, however, start popping a's from the stack, and accept the string if the stack is empty — this hihglights a good usage for PDAs. A PDA is basically an NFA_λ (although there are other ways of making them) with a stack for memory. Mathematically, they are:

$$M = (\Sigma, Q, s, \Gamma, \perp, \delta)$$

…where Σ is the alphabet, Q is the set of states, s is the start, Γ is the stack alphabet, $\perp$ is the initial stack symbol, and δ is the "moves" function. δ basically says, given the state, the input, and the symbol at the top of the state, move to a new state with a new stack top (that can in fact be a string). "Push" operations mean replacing the stack top with a new symbol, while "pop" operations mean replacing with λ instead. Another kind of δ move involves the same thing without an input — the idea being that you either scan a symbol or you don't. When these transitions are written on the transition arrows on the diagram, they're represented as $a, a/aa$, where the first term is symbol read, the second is the symbol that must be at the top of the stack, and the third is the string that will pushed to the stack. If the stack is empty, the machine halts as an accepting run has been reached. If there is no transition for the read symbol, the machine blocks.

Formal mathematics: A PDA has a configuration (q, w, γ) at any time, where q is the current state, w is the part of the input string that has not been read yet, and γ is the state of the stack. The one-step transition relation $\vdash$ is defined as: if $\{(p, aB), (q, \beta)\}$ is a move in δ, then for any x, y, $[p, ax, B\gamma], [q, x, \beta\gamma]$. The same applies to λ transitions (without the input symbol, of course). PDAs are *generally* non-deterministic.

5.4 Building PDAs from CFGs

For every context-free grammar, there is also a PDA that accepts strings generated by the grammar. PDAs constructed from CFGs have one state s which is the start state, the input alphabet, the initial stack symbol is the start symbol S from the grammar, and the move relation is as follows:

- For each rule $A \to \alpha$ from P, $((S, A), (s, \alpha))$ is a move in δ.
- For each symbol $a \in \Sigma((s, a, a), (s, \lambda))$ is a move in δ.

In effect you can construct a PDA with only one state!

6 Decidability

Decidability is the problem whether a program can exist that gives a yes/no answer for a given input. For example, the question of whether a program will finish running or not given a select input is undecidable, since no program can be made that will always give a correct answer to that problem. That select problem is called the halting problem, which is of great interest since the principles used in that proof can be applied to other decidability problems.

6.1 Example — The Halting Problem

The halting problem involves a program p and an input x, with the question of whether p halts (i.e. finish running) on input x. To prove that a program giving an always-correct answer cannot exist, we'll use a proof-by-contradiction strategy — that is, assume it *is* possible and derive a contradiction.

Suppose we did have a halt test program that returned 1 if p halts and 0 otherwise, and in C code it was expressed as "haltTest (p,x)". If you were to write a C program that loops infinitely if haltTest returns true, and halts if it returns false, you could call this C program on itself. That would mean your program loops if it's not supposed to loop, or it doesn't loop when it's supposed to. This cannot be, so a contradiction has been derived and therefore the haltTest function cannot exist.

6.2 Decision Problems and Languages

Decision problems can be viewed as languages. Consider the decision problem of whether a number k is prime. Each instance of k is an integer, and instances can be represented as strings (e.g. bit strings), since any finite object can be represented as a string. The set of "yes" answers is now just a set of strings — a language! You could also form a decision problem out of the question of whether a grammar is ambiguous or not, so long as you figure out a way of encoding grammars into a string.

For decision problems like the halting problem, up/down arrow notation is useful: $p[x] \downarrow y$ would denote that for a program p, and input x, the program will halt and the ouput will be y. Similarly, if p odesn't halt on x, it would be $p[x] \uparrow y$. Now, define a program p as a partial function $pf(p) : \Sigma_2^* \to \Sigma_2^*$. Assume that these are batch programs that accept no input while they're running. The language accepted by the program p can then be denoted as $L(p) \hat{=} \{x | p[x] \downarrow 1\}$ A language $L \subseteq \Sigma_2^*$ is said to be decidable if there is a program p such that $x \in L \to p[x] \downarrow 1$ and $x \notin L \to p[x] \downarrow 0$. In english: The program (called a decision procedure) is guaranteed to return a 1 or a zero. That also means that the function computed by the program must be a total function and not a partial function.

6.3 Providing Decidability

The simplest way to prove that a language is decidable is to just write an algorithm that accepts it, often in pseudocode or with an automaton. For instance, to test if a DFA generates an empty language or not, you can't loop over every possible string to check if one of them is accepted, as that could generate an infinite loop. Instead, just look at the DFA as a graph — if there is some path from the start to an accepting state, *then* you say the language isn't empty.

6.4 The Acceptance Problem

The acceptance problem asks whether a program/input pair will halt with a return value of 1: $Acc = \{< p, x > | p[x] \downarrow 1\}$. It's similar to the halting problem, but in actuality it's a subset of the halting problem, and subsets of an undecidable set can be decidable, as can supersets. In this case, the problem is still undecidable, but to prove this we'll need the concept of reducibility.

To prove some language X undecidable, choose some U already known to be undecidable. For the sake of contradiction suppose there was a decision procedure D_X for X (this is what we'll attempt to get a contradiction from). Then, just argue that you can build a decision problem for U if you are allowed to call D_X as a subroutine.

In this case, suppose you have a program that always prints "Hello world" no matter the input. Your program accepts an input for W and tries to decide whether it's in the self halting set. Suppose you have a decision procedure D_{HW}. Now imagine a decision problem for self halt — for an input w, construct the following code: on input W, simulate W on W and print "Hello world". Finally, call D_{HW} on this and return the answer.

(The above might not make sense. . .) The claim is that for any w the code returns 1 if w is in the self halting set and 0 if it isn't. If w is in self halt, the inner code will print "Hello world", and since D_{HW} is assumed correct, the outer code will return 1. If instead w is not in self halt, D_{HW} will recognize that the inner code is not in the "hello world" set, and will print 0.

6.5 The Always Halt Problem

This asks whether, for a given program p, the program will always halt or not. For the sake of contradiction, suppose we had a decision procedure D_{AH}, and build a decision procedure for self halt:

For an input w, construct the following code: For an input x, simulate w on w, and return x. Now call D_{AH} on the code just constructed.

It's the same argument as before — if the outer code goes into an infinite loop, the inner code never halts. If w halts on itself, the inner code actually computes the identity function and does indeed halt. Crucially, this relies on the self halt decision procedure working, which it doesn't, so we know the always-halt problem is undecidable.

6.6 Program Equivalence

This asks whether two programs are equivalent, i.e. for p, q whether $pfn(p) = pfn(q)$. For sake of contradiction, assume we have a D_{equiv} procedure. Use the previous example of the hello-world program undecidability. Let p_0 be equivalent to the program where for input x, print "Hello word". Write a decision program for "hello world", where for input p, you call $D_{equiv}(p, p_0)$. Since this can't exist, neither can D_{equiv}.

What?

6.7 Rice's Theorem

Here are a number of decidability examples:

- Program has an even number of procedures? **Decidable**
- Program computes the identity function? **Undecidable**
- Program accepts an empty string? **Undecidable**
- Program has a length that's a prime number **Undecidable**

The decidability of these statements hinges on whether the property is something about the program's text construction (e.g. prime length) rather than it's computational function (e.g. accepts empty string). As it turns out, any question about the program's text construction is decidable, while anything about their function as programs is undecidable. In short: "if it's a thing about the function the program computes, it's undecidable".

This needs more precision, so we need to make precise the idea "a property of programs is about the functions they compute, not their text". Let $L \subseteq \Sigma_2^*$ — consider L a set of programs. L is a functional set of programs if, whenever p and q compute the same partial function, then $p \in L$ if and only if $q \in L$. So, a set of programs with prime length is not a functional set, while a set of programs that return 1 on λ is.

So, Rice's theorem then states: **Every functional set of programs is undecidable, except $\emptyset$ and Σ^*.** The exception is for trivial cases, where either no program satisfies it or all programs do.

6.8 Semi-Decidability

$L \subseteq \Sigma_2^*$ is some program $L = L(p)$, i.e. if $x \in L$ then $p[x] \downarrow 1$, but if $x \notin L$ then you will **not necessarily** get $p[x] \downarrow 1$. This is different than the qualifier for decidability which requires the program not run forever and return 0 in case $x \notin L$.

A good example of a semi-decidable problem is the question of whether G is an ambiguous context-free grammar. To prove this, just find a semidecision procedure p that meets the requirements. In this case, successively generate all parse trees and keep track of the strings generated. If you ever see a repeat string, return 1. If a repeat string is never encountered, the program will loop infinitely and never return, satisfying the requirements for semi-decidability.

As an immediate from the definition, any L that is decidable is also semidecidable. This is not an if-and-only-if statement. For example, the self halt problem is semi decidable — if the input program halts, return 1. Technically, this is what all operating systems do in one form or another! Also note that if K is decidable, then $\bar{K}$ is decidable too, so K is decidable if and only if K and $\bar{K}$ are both semidecidable (note that decidable problems can be considered a subset of semidecidable problems). This works because you could in theory return the two semidecision programs in parallel and return a value depending on whichever returns first (in fact, this is the proof for it!). This technique of running the two programs in parallel is called "dovetailing".

Since not all languages are decidable this implies that a complement of something semidecidable somewhere is not even semidecidable. For instance, the complement of self halt is not semidecidable (since, if it were, we'd have a decision procedure for self halt by virtue of selfhalt itself being semidecidable), so we say it's completely undecidable. So, generally, if K is either semidecidable or undecidable, then $\bar{K}$ is not semidecidable.

6.9 Closure Properties

The decidable languages are closed under any operation you might want — union, intersection, concatenation, kleene-star, complement, etc., but for semidecidable languages the answer is not the same. The complement of a semidecidable language is not necessarily closed, but all the other operations are and return semidecidable languages.