9 Programs and Computability

9.1 Programs

With this section, we begin to look at what can be computed by general programs. But what is a general program?

A full definition of a general program is involved and takes us into an area, *automata theory*, that we will not explore in this course for lack of time. So let's settle for a less-than-rigorous defininition of a program.

Program "$\{p(x)$: *body*$\}$" is a program or function called p that performs actions indicated by *body*. In the body, a program says **return** r to indicate that the answer is r. Otherwise, the body is written in *psuedo-code* that you can imagine has been translated into your favorite programming language. We use indentation to show program structure.

Technically the input, or parameter, is always a string. But the input might be an integer, written in base 10. The input might also have more than one thing encoded in it. For example, input "(25,400)" describes an ordered pair of integers. So we will allow a program with more than one input, as in "$\{q(x, y)$: $\dots\}$".

Program "$\{a(x_1x_2\dots x_n)$: $\dots\}$" takes a parameter string x = "$x_1x_2\dots x_n$"; in the body, x_i refers to the i-th character of x.

Some examples are shown later in this section.

9.1.1 A Program Is a String

We write a program in quotes because a program is a string. You create a program using a **text-**editor. That point is important for the study of computability. If there are string constants embedded inside the program, I will not write \" for the embedded quotes. There should be no confusion from that.

We refer to program "$\{p(x)$: $\dots\}$" a p. Keep in mind that p is both a program and a string.

9.2 Computability

9.2.1 Computable Functions

Definition 9.1. For our purposes, an *algorithm* is a program that stops and produces an answer for every input. It is not allowed to loop forever, and is not allowed to stop without giving an answer.

Definition 9.2. Suppose that Σ and Γ are alphabets and $f : \Sigma^* \to \Gamma^*$ is a function. Program p *computes* function f provided, for every string $s \in \Sigma^*$, when p is run on input s, it eventually stops and returns string $f(s)$.

Definition 9.3. Function f is *computable* if there exists a program that computes f.

9.2.2 Computable Decision Problems

Definition 9.4. Suppose $A \subseteq \Sigma^*$ is a language over Σ^*. A program p *computes* A provided, for every string $s \in \Sigma^*$, when p is run on input s, it eventually stops and returns 1 if $s \in A$ and returns 0 if $s \notin A$.

If p computes A, we also say that p *solves* A, p *recognizes* A and that p *decides* A.

Definition 9.5. If p is an algorithm, define $L(p)$ to be the set of all strings on which program p stops and returns 1. We say that $L(p)$ is the language that p accepts.

Definition 9.6. Language A is *computable* provided there exists a program that computes A. Equivalently, A is computable if there exists a program p that stops on every input and where $L(p) = A$. Computable decision problems are also said to be *decidable*.

Note that computability is not defined in terms of what you or I are clever enough to do. A function or language is computable if *there exists* a program that computes it, regardless of whether any human is or will ever be able to find such a program.

9.2.3 The Church/Turing Thesis

Each programming language is a *model of computation*. Why can we ignore details like which programming language is chosen (within some limits) in the definition of a computable problem? Because every sufficiently general programming language can solve the same problems, as long as you take away restrictions on the amount of memory that the program can use. That observation is captured in the *Church/Turing Thesis*.

BIG IDEA: The Church/Turing Thesis: the class of computable problems is the same for all sufficiently general models of computation.

You hardly need much to achieve sufficiently general power. A common model of computation is a *Turing machine*, whose memory consists of an infinitely long tape that can store one symbol per cell, and that can only be read and written using a head that can move to the left and right over the tape. That model initially appears to be too simple, but it can solve all of the computational problems that are solvable by other models of computation.

9.2.4 The "Type" of Adjective *Computable*

A language can be computable. A function that takes a string and yields a string can be computable. A function that takes a number and yields a number can be computable.

But a program cannot be computable. It makes no sense to talk about a computable program. So please don't ever to that. Make sure that you know what type of thing you have.

9.3 Examples of Computable Decision Problems

It is easy to come up with computable decision problems.

Theorem 9.7. The empty set is computable.

Proof. Language $\{\}$ is thought of as the following decision problem.

Input. String x
Question. Is $x \in \{\}$?

Of course, the answer to the question is "no" regardless of what x is, and program "{$e(x)$: return 0}" computes {}.

♢

Theorem 9.8. Language {"b", "abb", "$baba$"} is computable.

Proof. The following program t computes language {"b", "abb", "$baba$"}.

```
"{$t(x)$:
   if $x$ == "$b$"
      return 1
   else if $x$ == "$abb$"
      return 1
   else if $x$ == "$baba$"
      return 1
   else
      return 0
}"
```

♢

You should be able to use the idea in the proof of Theorem 9.8 to prove the following.

Theorem 9.9. Every finite set is computable.

Theorem 9.10 shows there is a nonregular language that is computable. That should come as no surprise. General programs have much more power than finite-state machines.

Theorem 9.10. Language $\{a^n b^n \mid n > 0\}$ is computable.

Proof. Suppose that $\Sigma = \{a, b\}$. To compute $\{a^n b^n \mid n > 0\}$, it suffices to (1) check that there does not occur an a after a b, and (2) count the a's, count the b's, and check that the two counts are the same. The following program accomplishes that.

```
"{$p(x_1x_2\dots x_n)$:
   $i = 1$
   $c_a = 0$
   $c_b = 0$
   while $i \le n$ and $x_i ==$ 'a'
      $i = i + 1$
      $c_a = c_a + 1$
   while $i \le n$ and $x_i ==$ 'b'
      $i = i + 1$
      $c_b = c_b + 1$
   if $i == n + 1$ and $c_a == c_b$
      return 1
   else
      return 0
}"
```

$\diamondsuit$

Theorem 9.11. Language $\{n \mid n$ is a prime integer$\}$ is computable.

Proof. The following program tells whether n is prime.

```
"{$p(n)$:
   if $n < 2$
      return 0
   $i = 2$
   while $i < n$
      if $n \bmod i == 0$
         return 0
      $i = i + 1$
   return 1
}"
```

$\diamondsuit$

9.4 Every Regular Language Is Computable

Theorem 9.12. Every regular language is computable.

Proof. Suppose that A is a regular language. That is, there exists a DFA M so that $L(M) = A$. Ask someone else to give us such a DFA $M = (\Sigma, Q, q_0, F, \delta)$. Here is a program $R(x)$ that solves A. It simply runs M on input x.

```
"{$R(x_1x_2 \dots x_n)$:
    $q = q_0$
    $i = 1$
    while $i \le n$
        $q = \delta(q, x_i)$
        $i = i + 1$
    if $q \in F$
        return 1
    else
        return 0
}"
```

$\diamondsuit$

9.5 Computable Questions About DFAs

A program can take a DFA as an input. It is just a matter of encoding the DFA as a string. Suppose that $M = (\{a, b\}, \{1, 2, 3\}, 1, \{2, 3\}, \delta)$ where the transition function δ is as follows.

δ	a	b
1	1	2
2	3	1
3	1	1

A possible encoding of M as a string is

"{a,b}{1,2,3}1{2,3}(1,a:1)(1,b:2)(2,a:3),(2,b:1),(3,a:1)(3,b:1)".

Obviously, many different encodings would work.

9.5.1 Does M Accept x?

Definition 9.13. The *acceptance problem for DFAs* is the following decision problem.

> **Input.** A DFA M (encoded as a string) and a string x.
> **Question.** Does M accept x?

Theorem 9.14. The acceptance problem for DFAs is computable.

Proof. We have seen how to simulate a DFA M on input x. The only difference here is that M is encoded as a string. But that is not a problem; any experienced programmer can write a program that reads the encoding and pulls out all of the features of M.

$\diamondsuit$

9.5.2 Does M Accept All Strings?

Let's look at a more difficult problem.

Definition 9.15. The *everything problem for DFAs* is the following decision problem.

> **Input.** A DFA M (encoded as a string) with alphabet Σ.
> **Question.** Does M accept all strings in Σ^*.

Solving the everything problem for DFAs might at first seem impossible. After all, there are infinitely many strings, and you can't check them all. But that is an illusion; it is actually quite easy to check whether M accepts all strings.

Theorem 9.16. The everything problem for DFAs is computable.

Proof. Suppose $M = (\Sigma, Q, q_0, F, \delta)$. Some DFAs have states that cannot be reached by any input string. M accepts all strings in Σ^* if every state that can be reached is an accepting state. The hardest part is determining the reachable states, and that is actually easy.

Assume that there is a *mark bit* associated with each state of M that a program can set to 0 or 1. (That is easy to arrange. If M's states are $\{1, \ldots, n\}$, all we need is an array of n boolean values to hold the mark bits.)

```
"{everything($M$):
   // Mark all accessible states
   Set the mark bit of every state to 0.
   Set the mark bit of $q_0$ to 1.
   changed = 1
   while changed == 1
      changed = 0
      for each state $q$ of $M$
         if $q$'s mark bit is 1
            for each symbol $a$ in $\Sigma$
               r = $\delta(q, a)$
               if r's mark bit is 0
                  set r's mark bit to 1
                  changed = 1
   // Check if there a marked rejecting state
   for each state $q$ of $M$
      if $q$'s mark bit is 1 and $q \notin F$
         return 0
   return 1
}"
```

$\diamondsuit$

9.5.3 Does M Accept No Strings?

Definition 9.17. The *emptiness problem for DFAs* is language $\{M \mid L(M) = \{\}\}$. That is, it is the following decision problem.

Input. DFA M (encoded as a string).
Question. Is it the case that M does not accept any strings?

Theorem 9.18. The emptiness problem for finite state machines is computable.

Proof. The proof is similar to the preceding proof, but the algorithm checks that each reachable state is a rejecting state.

$\diamondsuit$

9.5.4 Is $L(M) \subseteq L(N)$?

Definition 9.19. The *subset problem for DFAs* is the following decision problem.

> **Input.** Two DFAs M_1 and M_2 (encoded as strings).
> **Question.** Is $L(M_1) \subseteq L(M_2)$? That is, is every string in $L(M_1)$ also in $L(M_2)$?

Once again, a shallow thought process leads one to conclude that the subset problem for DFAs is not computable, since there are infinitely many strings to check. A more careful look shows that it is computable.

Theorem 9.20. The subset problem for DFAs is computable.

Proof. We have seen, in Theorems 5.7 and 5.8, that the class of regular languages is closed under complementation and intersection. It is important that both theorems are proved by constructive proofs. That is,

1. There is an algorithm that, given a DFA M, produces DFA M' so that $L(M') = \overline{L(M)}$.

2. There is an algorithm that, given DFAs M_1 and M_2, produces DFA M' so that $L(M') = L(M_1) \cap L(M_2)$.

For any two sets A and B,

$$A \subseteq B \leftrightarrow A - B = \{\}.$$

But $A - B = A \cap \overline{B}$. The algorithm first builds DFA M_3 so that $L(M_3) = \overline{L(M_2)}$. Then it builds DFA M_4 so that

$$L(M_4) = L(M_1) \cap L(M_3) = L(M_1) \cap \overline{(L(M_2))} = L(M_1) - L(M_2).$$

So $L(M_1) \subseteq L(M_2) \leftrightarrow L(M_4) = \{\}$. But we have an algorithm (Theorem 9.7) to tell if $L(M_4) = \{\}$.

$\diamondsuit$

9.5.5 Are $L(M)$ and $L(N)$ the Same Language?

Definition 9.21. The *equivalence problem for DFAs* is the following decision problem.

> **Input.** Two DFAs M_1 and M_2 (encoded as strings).
> **Question.** Is $L(M_1) = L(M_2)$?

Theorem 9.22. The equivalence problem for DFAs is computable.

Proof. For any two sets A and B, by definition,

$$A = B \quad \leftrightarrow \quad A \subseteq B \wedge B \subseteq A.$$

It suffices to test each of $L(M_1) \subseteq L(M_2)$ and $L(M_2) \subseteq L(M_1)$ separately.

$\diamondsuit$

9.6 Computable Problems About Polynomials

Let's look at problems involving polynomials with integer coefficients, which we simply call polynomials. An input to such a problem might be $5x^2 - 2$ or $x^2 + 1$. A value of x that makes $5x^2 - 2 = 0$ is called a *zero* of polynomial $5x^2 - 2$.

Definition 9.23. The *real-zero problem* takes a polynomial p of variable x as input and asks whether there is a zero of p that belongs to $\mathcal{R}$, the set of real numbers.

For example, polynomial $x^5 - 2x^3 - 16$ has value 0 when $x = 2$, so it is a yes-input to the real-zero problem. Polynomial $4x^2 - 4x + 1$ is also a yes-input, since it has value 0 for $x = 1/2$.

9.6.1 Quadratic Single-Variable Polymomials

A naive first thought might be that the real-zero problem is not computable since an algorithm would have to try every possible number. But it should be clear that the zero problem is computable for quadratic polynomials. The quadratic formula tells you that equation $ax^2 + bx + c = 0$ has a real-valued solution if and only if $b^2 - 4ac \geq 0$.

9.6.2 Arbitrary Degree Single-Variable Polymomials

What if polynomials of x are allowed to have any degree? There are formulas for polynomials of degrees up to 4, but there is no formula for polynomials of degree 5 or higher. (The lack of a formula for degree 5 polynomials is one of the celebrated mathematical results of the nineteenth century.) But we don't need a formula, only an algorithm.

There are algorithms for finding zeros of polymonials of arbitrarily high degree. The details are beyond the scope of this class, but you can get a rough idea of how such an algorithm can work. The coefficient with largest absolute value and the polynomial's degree allow you to compute upper and lower bounds on potential zeros. Outside that range, the polynomial is heading toward ∞ or $-\infty$. An algorithm can cut that range up into small pieces and look for an interval where the polynomial changes sign. The polynomial must cross the x-axis somewhere in that interval.

Although we have not proved it here, the real-zero problem is solvable for arbitrary polynomials of a single variable.

9.6.3 Multivariate polynomials

A *multivariate polynomial*, such as $xy - y^2 + 9z$, can have any number of different variables; it is an expression made using variables, integer constants and only operations of addition, subtraction and multiplication.

A single-variable polynomial of degree k can have no more than k different zeros. But a multivariate polynomial can have infinitely many zeros. Look at equation $x - y = 0$. Obviously, any pair of values (x, y) is a zero if $x = y$.

Although the algorithm is very involved, and well beyond the scope of this class, it turns out that the real-zero problem is computable for arbitrary multivariable polynomials.

prev **next**