8 Programs and Computability

8.1 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* (ah-TOM-a-tah theory; automata is the plural of automaton (ah-TOM-a-tahn)), 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. It might 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 "{$q(x, y)$: ...}".

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

Some examples are shown later in this section.

8.1.1 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 "{$p(x)$: ...}" a p. Keep in mind that p is both a program and a string.

8.2 Computability

8.2.1 Computable functions

Definition 8.1. 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 8.2. 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 $f(s)$. That is, a function is computable if there is an algorithm that computes it.

Definition 8.3. Function f is *computable* if there exists a program that computes f.

8.2.2 Computable decision problems

Definition 8.4. 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 $s \notin A$. That is, language A is computable if there is an algorithm that computes the problem of determining whether a given string x is in A.

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

Definition 8.5. If p is a program, 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 8.6. 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.

8.2.3 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*: the class of computable problems is the same for all sufficiently general models of computation.

One hardly needs 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. One kind of model of computing has only *counters* as memory, where a counter can be incremented, decremented and tested to see whether it is 0. Astoundingly, a machine with only two counters has general power. The input is initially stored in one of the counters, as an integer that represents a string.

8.2.4 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.

8.3 Examples of computable decision problems

It is easy to come up with computable decision problems.

Theorem 8.1. The empty set is computable.

Proof. Recall that 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 8.2. Language {"b", "abb", "$baba$"} is computable.

Proof. The following program t computes {"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 8.2 to prove the following.

Theorem 8.3. Every finite set is computable.

Theorem 8.4 shows that some nonregular languages are computable.

Theorem 8.4. 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 as, count the bs, and check that the two counts are the same. The following program accomplishes that.

```
"{p(x_1 x_2 ... x_n):
   i = 1
```

```
    c_a = 0
    c_b = 0
    while i ≤ n and x_i == 'a'
        i = i + 1
        c_a = c_a + 1
    while i ≤ 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
}"
```

♢———————————————————————————————♢

Theorem 8.5. Language $\{n \mid n \text{ is a prime integer}\}$ 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 mod i == 0
            return 0
        i = i + 1
    return 1
}"
```

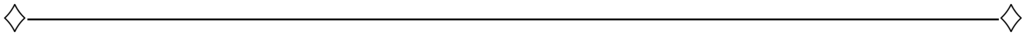
♢———————————————————————————————♢

8.4 Every regular language is computable

Theorem 8.6. Every regular language is computable.

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

```
"{R(x_1 x_2 ... x_n):
   q = q_0
   i = 1
   while i ≤ n
      q = δ(q, x_i)
      i = i + 1
   if q ∈ F
      return 1
   else
      return 0
}"
```



8.5 Computable questions about FSMs

A program can take a FSM as an input. It is just a matter of encoding the FSM 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.

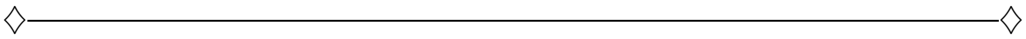
8.5.1 Does M accept x?

Definition 8.7. The *acceptance problem for FSMs* is the following decision problem.

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

Theorem 8.7. The acceptance problem for FSMs is computable.

Proof. We have shown, above, how to simulate a FSM 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.



8.5.2 Does M accept all strings?

Let's look at a more difficult problem.

Definition 8.8. The *everything problem for FSMs* is the following decision problem.

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

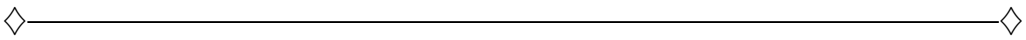
Solving the everything problem for FSMs might at first seem impossible. After all, there are infinitely many strings, and you can't check them all. But that is an illusion; it is actually quite easy to check whether M accepts all strings.

Theorem 8.8. The everything problem for FSMs is computable.

Proof. Suppose $M = (\Sigma, Q, q_0, F, \delta)$. Some FSMs 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 Σ
          r = δ(q, a)
          if r's mark bit is 0
            set r's mark bit to 1
            changed = 1
  // Check is there a marked rejecting state
  for each state q of M
    if q's mark bit is 1 and q ∉ F
      return 0
  return 1
}"
```



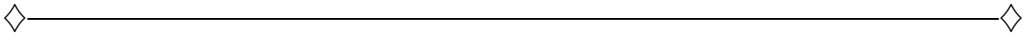
8.5.3 Does M accept no strings?

Definition 8.9. The *emptiness problem for FSMs* is language $\{M \mid L(M) = \{\}\}$. That is, it is the following decision problem.

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

Theorem 8.9. 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.



8.5.4 Is $L(M_1) \subseteq L(M_2)$?

Definition 8.10. The *subset problem for FSMs* is the following decision problem.

> **Input.** Two FSMs 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 FSMs is not computable, since there are infinitely many strings to check. A more careful look shows that it is computable.

Theorem 8.10. The subset problem for FSMs is computable.

Proof. We have seen, in Theorems 5.1 and 5.2, that the class 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 FSM M, produces FSM M' so that $L(M') = \overline{L(M)}$.
2. There is an algorithm that, given FSMs M_1 and M_2, produces FSM 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 FSM M_3 so that $L(M_3) = \overline{L(M_2)}$. Then it builds FSM 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 8.4) to tell if $L(M_4) = \{\}$.

♢———————————————————————————————♢

8.5.5 Are $L(M_1)$ and $L(M_2)$ the same language?

Definition 8.11. The *equivalence problem for FSMs* is the following decision problem.

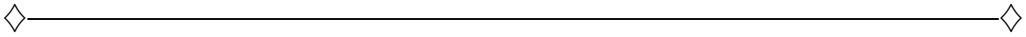
> **Input.** Two FSMs M_1 and M_2 (encoded as strings).
> **Question.** Is $L(M_1) = L(M_2)$?

Theorem 8.11. The equivalence problem for FSMs is computable.

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

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

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



8.6 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 8.12. 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 $x = 1/2$.

8.6.1 Quadratic single-variable polymomials

A naive person's first thought might be that the 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 $b^2 - 4ac \geq 0$.

8.6.2 Arbitrary degree single-variable polymomials

What if polynomials in 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-access somewhere in that interval.

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

8.6.3 Multivariate polynomials

A *multivariate polynomial*, such as $xy - y^2 + 9z$, can have any number of different variables. 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 $x = y$.

The algorithm is very involved, but it turns out that the real-zero problem is computable for arbitrary multivariable polynomials.

prev **next**