20 Models Of Computation

Overview

This chapter gives a very light introduction to the Theory of Computability. Several major concepts and results are presented. However, many proofs are omitted, whereas proofs are the major part of a course in computability.

- Section 20.1 introduces decision problems and finite automata. A finite automaton is a model of a computing machine that has only enough RAM to store one integer (other than the storage space for the program itself).
- Section 20.2 describes grammars and pushdown automata. A grammar is a description of the logic a computing machine uses. A pushdown automaton has enough RAM for one stack of data.
- Sections 20.3-20.4 define a Turing machine, the most general model of a computing machine, and prove that some problems are unsolvable.
- Section 20.5 develops a context-free grammar describing a complete practical programming language J1, which is a simplification of Java.

20.1 Decision Problems And Finite Automata

A fundamental kind of problem to solve is that you are given a non-negative number and you have to say whether it is or is not acceptable according to some criterion. For instance, you may be asked to write a method that tells whether the input number is divisible by 3. Or you may be asked to write a method that tells whether the input number is a prime number.

The general **decision problem** is that you are given a finite string of characters and you have to say whether that string is or is not acceptable according to a specified criterion. The problem should specify what characters are acceptable in the input string. That specified set of characters is called the **alphabet** for the problem. For instance, the numeric decision problem described in the previous paragraph has ten digits as its alphabet. If the decision problem is to decide whether a given string of characters is a compilable Java class, then the alphabet can be considered to contain the characters that have Unicode values 32 through 127, a total of 96 characters.

To solve a decision problem, you (a) write a computer program that inputs a finite string of characters and outputs whether it is acceptable and (b) prove logically that the program is bug-free. The reason for part (b) is that we do not want to have to spend time debugging programs. In fact, part (b) is more important than part (a), because any idiot can write a program, the hard part is to write a correct program.

The technical term for a set of strings of characters over a given alphabet is a **language** over that alphabet. For instance, if the decision problem is to tell whether the input number is divisible by 2, the language for that problem is the set of strings of base-ten digits that end in 0, 2, 4, 6, or 8. If the decision problem is to tell whether the input number is a prime number, the language for that problem is the set of strings of base-ten digits that include 2, 3, 5, 7, 11, 13, 17, 19, 23, etc. If the decision problem is to tell whether a given string of characters is a legal Java class, then one element of the language for that problem is `class Whatever { int x = 4; }`.

Finite Automata

Older computer systems had less RAM. The further back in history you go, the less RAM the computers had. It makes sense to start with the most primitive kind of computing machine, one with just one unit of RAM, capable of storing a single integer. We also restrict the machine to reading through the input string one time only, then making its decision. This is called a **finite automaton**. We will investigate what decision problems such a primitive computer is capable of solving.

The current value of the one unit of RAM is called the **state** of the finite automaton. When a program starts, the state is initialized to zero. All that the machine can do is look at the next available input character (called the **current symbol**) and assign a new value to its state variable. When it comes to the end of the input, RAM should have either an int value that represents acceptance or an int value that represents rejection of the input. If the input contains an illegal character, i.e., not in the permissible alphabet, the machine should say whether it accepts or rejects the input string up to that point.

Acceptance means that the input is a member of the language of the stated decision problem, and rejection means that the input is not. The **FiniteAutomaton** class in Listing 20.1 (see below) codes for what is common to all finite automata. It is abstract because specific subclasses will have different ways of computing the next state. It could be used as follows, if `sam` is a FiniteAutomaton object and `s` is a string of input characters:

```
sam.askOpinion (new Sentence (s))
```

We provide this and related classes because you might not have available an actual physical finite automaton, or yours might be broken. So you can use the Java simulation we develop here if you wish.

It is standard to name the **starting state q0** and the other states that the automaton can be in q1, q2, etc., as well as the standard ACCEPT and REJECT states. The FiniteAutomaton class supplies names only through q9, so if you need larger numbers, e.g. 14, just write them explicitly. We reserve negative numbers as codes for accepting and rejecting the input. These definitions are in the upper part of Listing 20.1.

Listing 20.1 The FiniteAutomaton abstract class

```
public abstract class FiniteAutomaton
{
   public static final int ACCEPT = -1, REJECT = -2;
   public static final int q0 = 0, q1 = 1, q2 = 2, q3 = 3,
           q4 = 4, q5 = 5, q6 = 6, q7 = 7, q8 = 8, q9 = 9;


   public String askOpinion (Sentence input)
   {  return accepts (input)  ?  "accepted"  :  "rejected";
   }  //=====================


   public boolean accepts (Sentence input)
   {  int state = q0;
      while (state >= q0)
         state = nextState (state, input.next());
      return state == ACCEPT;
   }  //=====================


   public abstract int nextState (int state, byte symbol);
}
```

An outside class would create a finite automaton and then pass it a specific input string in the form of a Sentence object, to "ask its opinion" of it -- accept or reject. The `askOpinion` method returns the answer in a printable form.

The `accepts` method is what actually does the work of deciding whether the string of characters is acceptable. It starts by initializing the state variable to the starting state q0. Then it repeatedly gets the next character of the input (obtained by `input.next()`) and uses both it and the current state to calculate the `nextState`. This coding is in the middle part of Listing 20.1. The `nextState` method is of course an abstract method, because its coding depends on the particular decision problem you are trying to solve.

A finite automaton to determine divisibility by 2

We start with a simple example -- a finite automaton (abbreviated **FA**) to find out whether the input string denotes an integer divisible by 2. We can do this by keeping track of the most-recently-read digit. If it is 0, 2, 4, 6, or 8, we switch to state q0; otherwise we switch to another state q1. We arbitrarily decide to treat a string of length zero as denoting an even number. When we come to the end of the input, we report acceptance if we are in state q0, rejection if we are in state q1. So if we are currently in state q1, we change the state to the following value based on the current input symbol (expressed quite compactly using the conditional operator):

```
(symbol == 0) ? q0 : (symbol == 1) ? q1 : (symbol == 2) ? q0
     : (symbol == 3) ? q1 : ... (symbol == 9) ? q1 : REJECT
```

If the input could be any base-ten numeral, this will require a lengthy coding. Just to avoid too much typing, we change the problem to one where the input is a **base-four numeral** (i.e., a string of 0s, 1s, 2s, and 3s only). That gives us the coding in the upper part of Listing 20.2 for the `nextState` method.

Listing 20.2 Finite Automaton to compute divisibility by 2

```
public class Div2FA extends FiniteAutomaton
{
   /** Tell whether the base-four input is divisible by 2. */

   public int nextState (int state, byte symbol)
   {  return state == q0 ? (symbol == 0 ? q0 : symbol == 1 ? q1
                          : symbol == 2 ? q0 : symbol == 3 ? q1
                          : ACCEPT)
           : state == q1 ? (symbol == 0 ? q0 : symbol == 1 ? q1
                          : symbol == 2 ? q0 : symbol == 3 ? q1
                          : REJECT)
           : REJECT;  // should never happen
   }  //======================
}
//#############################################################


class Div2FAApp
{
   public static void main (String[] args)
   {  System.out.println
              (new Div2FA().askOpinion (new Sentence (args[0])));
   }  //======================
}
```

The only difference between what happens for the state of q0 and the state of q1 is when we reach the end of the input or see an illegal symbol (both of which are signaled by some symbol value outside the range 0 through 3). If we are in state q0 when we come to the end of the input, that means that the most-recently-seen digit was even, so the number is even. If we are in state q1, the last digit was odd so the number is odd.

It should be clear from the coding that the state parameter can never be a value other than q0 or q1. So you may think it odd to have the `:REJECT` phrase at the end of the `nextState` coding, because it can never arise. However, that makes the alternatives clearer, so we do it that way. Besides, if someone is silly enough to call the `nextState` method inappropriately, passing in an unreachable state, we rightly reject their input.

The lower part of Listing 20.2 is an application so that we can test this Div2FA class from the command line. It is of course in a separate class so that we are also free to use Div2FA objects from applets and in other contexts. This main method might be called from the command line by one of these two commands:

```
java Div2FAApp 1203011
java Div2FAApp 33221100
```

The Sentence class of objects

The commands above pass in the string of digits to args[0]. The main method creates a new Div2FA object, asks its opinion of the Sentence created from that string of digits, and prints what it says. Listing 20.3 gives an appropriate definition of the Sentence class.

Listing 20.3 The Sentence class

```
public class Sentence
{
   private String itsString;
   private int itsIndex = 0;
   private int itsBase = '0';


   public Sentence (String input)  // the natural constructor
   {  itsString = (input == null)  ?  ""  :  input;
      if (itsString.length > 0 && itsString.charAt (0) >= 'a')
         itsBase = 'a';
   }  //=====================


   public Sentence()
   {  this (javax.swing.JOptionPane.showInputDialog
                     ("Enter a sentence for the FA"));
   }  //=====================


   public Sentence (Sentence given)
   {  this ((given == null)  ?  ""  :  given.itsString);
   }  //=====================


   public byte next()
   {  return (byte) ((itsIndex >= itsString.length)  ?  -1
                  :  itsString.charAt (itsIndex++) - itsBase);
   }  //=====================
}
```

The main method in the earlier Listing 20.2 calls the first constructor in the Sentence class, passing in a string of characters. That string is assigned to an instance variable `itsString` so it can be used later. The current position in the string is `itsIndex`, which is initially zero.

The presumption is that the input string consists of digits, so the `itsBase` Unicode value of '0' is to be subtracted from the Unicode value of the char at a given position to obtain a number in the range 0 through 9. However, we also accommodate strings of lowercase letters -- If the first character is the letter 'a' or higher, we make `itsBase` the Unicode value of 'a', to be subtracted from Unicode values of characters in the input, which gives a number in the range 0 through 25 for each lowercase letter.

The Sentence class offers a convenience constructor with no parameters. It displays a frame to get the string input from the user and passes that over to the natural constructor that has a String parameter. Similarly, if you supply an existing Sentence object as a parameter to a constructor, that invokes the third constructor which passes that Sentence's String value over to the natural constructor that has a String parameter. This is useful when the given Sentence has already been iterated through by some FA and you want to create a new Sentence starting over at `itsIndex = 0`.

The key method of the Sentence class is the `next` method. It returns the next character of the input, "normalized" to produce values of 0 or more. When you have **consumed** all the characters of the input, the `next` method returns -1 to signal this. Reminder: A **byte** value ranges from -128 to 127 inclusive.

A finite automaton to determine divisibility by 3

Okay, divisibility by 2 was extremely easy. Let us try something harder, a FA that tests a base-four numeral for divisibility by 3. We can proceed this way: The state is q0 if the number so far is evenly divisible by 3, q1 if the remainder for dividing the number so far by 3 is 1, q2 if the remainder is 2. If you take the number so far, call it n, and append another digit, call it d, you obtain the value 4*n + d (because that is the definition of base-four). The remainder on dividing 4*n by 3 is the same as the remainder on dividing n by 3 (think about it for a while to verify this). So the remainder on dividing 4*n + d by 3 is the remainder for n added to the remainder for d, except if that comes out to 3, 4, or 5, you subtract 3 from it to get the true remainder of 0, 1, or 2.

This leads to the following expression for the new state when the current state is q1. Similar expressions are needed for the states of q0 and q2:

```
symbol == 0 ? q1 : symbol == 1 ? q2
: symbol == 2 ? q0 : symbol == 3 ? q1 : REJECT
```

This kind of expression is quite wordy, as well as being rather hard to follow due to all the conditional operators in it. It makes more sense to store the information in an array of int values. This array is traditionally called the **delta array**; we call it **`del`** for short. We store in `del[q][sym]` the state to which the finite automaton is to switch when the current state is `q` and the current input symbol is `sym`. The values of `sym` range from 0 through 3. We store in `del[q][4]` either ACCEPT or REJECT, depending on what is supposed to happen when we are in state `q` at the end of the input.

For the divisibility by 3 FA, that means we have an array of 3 rows (one for each state other than ACCEPT or REJECT) and 5 columns (one for each legal symbol plus one for all other cases). The complex conditional-operator expression above corresponds to having the row of index 1 (for state q1) be `{q1, q2, q0, q1, REJECT}`. You can see the entire array in the upper part of Listing 20.4 (see next page).

Listing 20.4 Finite Automaton to compute divisibility by 3

```
public class Div3FA extends FiniteAutomaton
{
   private final int NUM_SYMBOLS = 4;
   private final int NUM_STATES = 3;  // other than ACCEPT/REJECT
   private final int[][] del = { {q0, q1, q2, q0, ACCEPT},   //q0
                                 {q1, q2, q0, q1, REJECT},   //q1
                                 {q2, q0, q1, q2, REJECT} }; //q2


   /** Tell whether the base-four input is divisible by 3. */

   public int nextState (int state, byte symbol)
   {  return (state < 0 || state >= NUM_STATES)
             ? REJECT
             : (symbol < 0 || symbol >= NUM_SYMBOLS)
                    ?  del[state, NUM_SYMBOLS]
                    :  del[state, symbol];
   }  //======================
}
//#############################################################


class Div3FAApp
{
   public static void main (String[] args)
   {  System.out.println
             (new Div3FA().askOpinion (new Sentence (args[0])));
   }  //======================
}
```

The `nextState` method in Listing 20.4 uses the `del` array. Note that any other FA can be written with exactly the same `nextState` coding, as long as you make the appropriate changes in the three instance variables declared at the top of the listing. For instance, the divisibility-by-2 Div2FA class is obtained from Listing 20.4 by just replacing the array declaration by the following and defining NUM_STATES as 2:

```
   private final int[][] del = { {q0, q1, q0, q1, ACCEPT},
                                 {q0, q1, q0, q1, REJECT} };
```

The set of strings that a FA accepts is called the language that it **recognizes**. For instance, Div3FA recognizes all multiples of 3 in base-four notation.

Exercise 20.1 Write out declarations of del and NUM_STATES for a FA that tells whether a given input string contains a base-four number evenly divisible by 4.
Exercise 20.2 Write out declarations of del and NUM_STATES for a FA that tells whether a given input string contains a base-four number that has no two consecutive digits the same.
Exercise 20.3 Write out declarations of del and NUM_STATES for a FA that tells whether a given input string contains a binary number that ends in 01.
Exercise 20.4* Write out declarations of del and NUM_STATES for a FA that tells whether a given input string contains a base-four number evenly divisible by 5.
Exercise 20.5* Write out declarations of del and NUM_STATES for a FA that tells whether a given input string contains a binary number that ends in 01 followed by exactly one digit.
Exercise 20.6* Revise Listing 20.2 so that an input with no symbols is not accepted as being an even number.

20.2 Grammars And Pushdown Automata

You are probably thinking of easy ways to write down the state-transitions of a FA, instead of going to the trouble of declaring a Java array. One way that is popular is illustrated below for the Div2FA of Listing 20.2 (the array declaration is shown below it for comparison). It is called a grammar of the language accepted by this FA:

```
q0 -> 0 q0 | 1 q1 | 2 q0 | 3 q1 | Ë ;  q1 -> 0 q0 | 1 q1 | 2 q0 | 3 q1 ;
del = { {q0, q1, q0, q1, ACCEPT}, {q0, q1, q0, q1, REJECT} };
```

A grammar for the language accepted by the Div3FA of Listing 20.4 is as follows:

```
q0 -> 0 q0 | 1 q1 | 2 q2 | 3 q0 |  Ë ;
q1 -> 0 q1 | 1 q2 | 2 q0 | 3 q1 ;   q2 -> 0 q2 | 1 q0 | 2 q1 | 3 q2 ;
```

In general, a **grammar** is a number of lines, one for each state, each line beginning with that state and the -> symbol. Lines are separated from each other by semicolons and/or ends-of-line characters. That -> symbol is followed by a number of **production items** separated by the | symbol. The meaning is that the state before the -> symbol can be replaced by any one of the production items listed. The **language generated by a grammar** is the set of strings you can get by starting with the grammar's starting state and repeatedly making replacements according to the grammar's production items.

The Ë symbol is special; this **Greek letter lambda** signals that the state before the -> can be replaced by nothing at all and the input is to be accepted if you reach the end of the input in the state for that line. If you reach the end of the input in some state that does not have the Ë as a production item, the input is to be rejected.

The replacement process traces the action of the FA on a given input. For instance, if the input is 120301, a trace of the action of Div2FA is as follows:

```
q0 => 1 q1 => 12 q0 => 120 q0 => 1203 q1 => 12030 q0 => 120301 q1   rejected.
```

This compact notation means that from q0 you consume 1 and go to q1, from which you consume 2 and go to q0, from which you consume 0 and go to q0, etc. When you come to the end of the input, you are in state q1, which is a rejecting state. But if the input is 3012, a trace of the action of Div2FA is as follows:

```
q0 => 3 q1 => 30 q0 => 301 q1 => 3012 q0 => 3012   accepted.
```

Note that in the last step, q0 is replaced by nothing at all to obtain the desired string, as permitted by the production item Ë. These two lines are called **derivations** of the given inputs using the given grammar.

Regular grammars

In grammar rules, the symbols that can appear in the input are called **terminal symbols**. The other symbols that represent states are called **nonterminal symbols**. Production items in general can be any combination of terminals and nonterminals. When all the production items in a grammar (other than Ë) consist of one terminal followed by one nonterminal, the grammar is called a **regular grammar**. As you can see, the given grammars for Div2FA and Div3FA are both regular grammars.

Obviously, if you take a particular grammar or FA and change the names of the states, nothing else, that makes no real difference. In that case, we say that the new grammar or FA is the **same** as the old one. But you can also have two different grammars that generate the same language.

The grammars that make a FA work right require that all production items for a particular state begin with a different terminal. An arbitrary regular grammar may violate that rule. That would mean that the FA has uncertainty as to which to choose in that case. However, you may be able to reengineer the grammar to remove the uncertainty without changing the strings of characters that are or are not acceptable. Then the grammar is different but the language it generates is the same. Some major results of the Theory of Computation are the following (you will see their proofs when you take the whole course):

Kleene's Theorem: Every regular grammar can be reengineered to produce a new grammar that generates the same language but has no uncertainty. So for every regular grammar there is a FA that recognizes the language that the grammar generates (and conversely, of course). The proof describes an algorithm for doing the reengineering.
Lemma: Two strings are **distinguishable** with respect to some language L if there is some string you can append to make one in L and the other not in L. Then any FA that recognizes L must arrive at two different states for any two distinguishable strings.
Theorem: If, for a given language L, you can specify a set of n strings, any two of which are distinguishable with respect to L, then a FA that recognizes L has at least n states.
Theorem: The language of palindromes over the alphabet {0,1} cannot be recognized by a FA, and therefore does not have a regular grammar.
Myhill-Nerode Theorem: For every FA there is a unique FA that recognizes the same language and has fewer states than any different FA that recognizes the same language. That unique FA always arrives in the same state for any two indistinguishable strings.

Connection with programming languages

You should be able to see that you could write a computer program that reads in a regular grammar without uncertainty and outputs a subclass of FiniteAutomaton (similar to Listing 20.4) that recognizes the language generated by the grammar. In other words, the set of regular grammars without uncertainty is a programming language, and your program would be a kind of compiler for that language.

A highly-restricted version of Java can be described as follows:

- A class definition is `public class Name extends Name { MethodGroup }` where the word `public` is optional.
- A method group is zero or more `public void Name() { StatementGroup }` where the word `public` is optional.
- A statement group is zero or more `stmt` words.

This version of Java has a regular grammar. Part is as follows (words not beginning with 'q' are terminals):

q0 -> public q1 | class q2 ; q1 -> class q2 ; q2 -> name q3 ; q3 -> extends q4 ;

Context-free grammars

A more general kind of grammar allows any number and mix of terminal and nonterminal symbols on the right of a production item. Such a grammar is a **context-free grammar**.

Example 1 Consider the language of all palindromes in base 4 that have a 1 in the middle and no 1 anywhere else. This language can be described by the following grammar:

S -> 0 S 0 | 2 S 2 | 3 S 3 | 1 ;

Here the starting state is named S. By tradition, when we write context-free grammars, the nonterminal symbols are represented by capital letters in most cases. The starting state is normally named S.

This grammar means that, if the current symbol is 0 or 2 or 3, you may replace the starting state S by two instances of the current symbol with S in between them. But if the current symbol is 1, you replace S by 1. A derivation that shows that 030212030 is an acceptable palindrome with a unique 1 in the middle is the following:

S => **0 S 0** => 0**3 S 3**0 => 03**0 S 0**30 => 030**2 S 2**030 => 0302**1**2030 accepted.

In this derivation, we try to clarify which production rules are applied where by two notational devices that are not actually part of the official derivation: (a) the symbols that replace a nonterminal are boldfaced, and (b) the nonterminal to be replaced at the following step is separated from other symbols by blanks.

Example 2 Consider the language of all sequences of base 3 digits (0s, 1s, and 2s) that are in ascending order with the same number of 0s as 2s. This language can be described by the following grammar:

S -> 0 S 2 | T ; T -> 1 T | Ë ;

For this grammar, T stands for all sequences of 1s. The Ë production item indicates that T can be replaced by nothing at all at any time. Derivations that show that 000222 and 001122 are in the language are as follows:

S => **0 S 2** => 0**0 S 2**2 => 00**0 S 2**22 => 000 **T** 222 => 000 222 accepted.
S => **0 S 2** => 0**0 S 2**2 => 00 **T** 22 => 00**1 T** 22 => 001**1 T** 22 => 0011 22 accepted.

Example 3 Consider the language of all binary numerals that have the same number of 0s as 1s. This language can be described by the following grammar:

S -> 0 W | 1 Z | Ë ; Z -> 0 S | 1 Z Z ; W -> 1 S | 0 W W ;

For this grammar, Z stands for all strings that have exactly one more 0 than 1s, and W stands for all strings that have exactly one more 1 than 0s. A multi-line derivation that shows that 1000100111 is in the language is as follows:

S => **1 Z** => 1**0 S** => 10**0 W** => 100**0 W W** => 1000**1 S** W => 10001**0 W W**
=> 100010**0 W W**W => 10001001 **S** WW => 10001001 W W => 100010011 **S** W
=> 100010011 W => 1000100111 **S** => 1000100111 accepted.

In this derivation, we sometimes had more than one nonterminal that we could have replaced at that point. We always replace the one furthest left; this is called a **leftmost derivation**. But the result is the same regardless of the order of replacement.

Example 4 Consider the language of all words that are palindromes using only an even number of the letters a, b, and c. This language can be described by the following grammar:

S -> a S a | b S b | c S c | Ë ;

A leftmost derivation that shows that bacbbcab is in the language are as follows:

S => **b S b** => b**a S a**b => ba**c S c**ab => bac**b S b**cab => bacb bcab accepted.

Example 5 A grammar for the structure of the Scheme programming language as described at the beginning of Chapter Sixteen is as follows. S and T are the only nonterminals for the entire programming language. The three words are terminals:

S -> word | text | real | (T ; T -> S T |) ; V -> Ë ;

Ambiguous grammars

An **ambiguous grammar** is a grammar for which some string in the language it generates has two different leftmost derivations. The grammar of Example 6 is ambiguous. An ambiguous grammar can usually (but not always) be replaced by an unambiguous grammar that generates the same language.

Example 6 A grammar for the if-statement in Java could be as follows. Here "if" and "else" are the only terminals:

Stmt -> if Condition Stmt | if Condition Stmt else Stmt ;

Two different leftmost derivations for the same string show that the if-statement introduces an element of ambiguity into the Java language, depending on whether the `else` goes with the first `if` or the second `if`. This is called the dangling-else problem:

Stmt => **if Condition Stmt** => if Condition **if Condition Stmt else Stmt**
Stmt => **if Condition Stmt else Stmt** => if Condition **if Condition Stmt** else Stmt

Pushdown automata

It should be clear that a FA is not adequate to recognize a language generated by most context-free grammars. In fact, each of the six examples just given is beyond the power of a FA. We need a more powerful kind of machine called a **top-down pushdown automaton** (**ToPDA** for short). Its RAM consists of a stack of terminals and nonterminals on which you can store parts of the derivation that have not yet been processed.

The **ToPDA** for a given context-free grammar is a machine that acts as described in the accompanying design block. It has only two states besides the ACCEPT and REJECT states. If and when a ToPDA reaches its ACCEPT state, the string of characters it has processed up to that point is said to be **accepted by the ToPDA**.

SNL DESIGN for the ToPDA constructed from a given context-free grammar

1. The initial state is q0 and the stack initially is empty.
2. Push the start nonterminal S on the stack and switch from state q0 to state q1.
3. Repeat the following until done (remain in state q1 until the end)...
    - 3a. If the stack is empty, then...
        - Switch to ACCEPT and terminate this algorithm.
    - 3b. Otherwise, pop the top element of the stack and then...
        - 3ba. If it is a nonterminal then...
            - Push any one production item for that nonterminal on the stack.
        - 3bb. Otherwise it is a terminal, so...
            - Read (a.k.a. "consume") the next input symbol.
            - If the input symbol is not the same as that terminal then...
                - Switch to REJECT and terminate this algorithm.

Naturally, in step 3ba, you choose to push the production item that is indicated by the next input symbol, where possible. For the earlier Example 1, which is the set of palindromes in base 4 where a 1 occurs only in the middle, you saw the following grammar and the following derivation to show that 030212030 is accepted:

S -> 0 S 0 | 2 S 2 | 3 S 3 | 1 ;
S => **0 S 0** => 0**3 S 3**0 => 03**0 S 0**30 => 030**2 S 2**030 => 0302**1**2030 accepted.

We will trace the action of the ToPDA for this grammar on this string. For this we need some notation: The conventional representation of the empty stack is Z_0. Something like (q1, 03^S30Z_0) means that we are currently in state q1 (or whatever is before the comma), we have consumed 03 in that order (or whatever is before the caret), and the stack contains S30 reading from the top down (because Z_0 marks the bottom of the stack). The preceding derivation implies that the ToPDA's action is as follows:

(q0, ^Z_0) by Step 1, then	(q1, ^SZ_0) by Step 2.
(q1, ^0S0Z_0) by Step 3ba, then	(q1, 0^S0Z_0) by Step 3bb, consuming 0.
(q1, 0^3S30Z_0) by Step 3ba, then	(q1, 03^S30Z_0) by Step 3bb, consuming 3.
(q1, 03^0S030Z_0) by Step 3ba, then	(q1, 030^S030Z_0) by Step 3bb, consuming 0.
(q1, 030^2S2030Z_0) by Step 3ba, then	(q1, 0302^S2030Z_0) by Step 3bb, consuming 2.
(q1, 0302^12030Z_0) by Step 3ba, then	(q1, 03021^2030Z_0) consuming 1.
(q1, 030212^030Z_0) consuming 2, then	(q1, 0302120^30Z_0) consuming 0.
(q1, 03021203^0Z_0) consuming 3, then	(q1, 030212030^Z_0) consuming 0.

At this point the stack is empty ("contains" only Z_0), so we ACCEPT 030212030.

For a **pushdown automaton** in general (**PDA** for short), you still start in state q0 with the stack containing only a special element Z_0 which is never removed or pushed (so Z_0 on the top means it is effectively empty). But you may have any finite number of states, not just q0 and q1. The PDA allows a finite number of operations of the following structure:

> If the state is ___ and the stack top is ___ and the next input is ___ then...
> Replace the stack top by a particular sequence of zero or more symbols.
> Either do or do not consume the input symbol.
> Either do or do not change the current state to a particular state.

If you get to a point where the stack is empty (i.e., only the Z_0 bottom-marker), you can accept the input consumed so far. If you get to a point where no operation can be performed given the current state and stack top and input symbol, you reject the input. The major theorems on PDAs are as follows:

Theorem: The language of all strings that can be accepted by a ToPDA is precisely the language that is generated by the context-free grammar used in the ToPDA algorithm. That is, for every context-free grammar there is a ToPDA that accepts its language.
Theorem: For every PDA there is a ToPDA that accepts the same language. In particular, the language of any PDA has a context-free grammar.

Exercise 20.7 Write out a regular grammar for a FA that tells whether a given input string contains a base-four number that has no two consecutive digits the same.
Exercise 20.8 Write out a regular grammar for a FA that tells whether a given input string contains a binary number that ends in 01.
Exercise 20.9 Write out a context-free grammar for the language of all binary numerals that have more 0s than 1s.
Exercise 20.10 Trace the action of the ToPDA for the grammar of Example 2 on the input 00122.
Exercise 20.11* Write out a context-free grammar for the language of all binary numerals that are palindromes of any length.
Exercise 20.12* Write out a context-free grammar for the language of differences of numbers in variables x and y, e.g., x - y, y - x - y, x - x - y - y.
Exercise 20.13* Trace the action of the ToPDA for the grammar of Example 3 on the input 000222.
Exercise 20.14* Trace the action of the ToPDA for the grammar of Example 4 on the input abaaba.

20.3 Turing Machines

A **Turing machine** is a more complex automaton than either of the two already discussed in this chapter. It also takes a string of symbols as input and produces an answer of either accept or reject (though it may go into an infinite loop and produce neither). At any given time during its computation, it is in one of a finite number of different states. It works with an unlimited amount of RAM. You can think of this RAM area as a number of memory locations numbered from 0 on up, each one capable of storing a single symbol.

The Turing machine starts in a state conventionally named q0, positioned at index 0 in the RAM. First the input is loaded into RAM starting at index 1, with all other RAM positions left blank (in particular, index 0 is left as the blank symbol). The Turing machine is not limited to a single look at input symbols as they are read; it can keep them stored in RAM so it can re-check them when needed.

The historical description of a Turing machine imagined it as having a long tape in place of RAM, with a tapehead positioned at one point on the tape. So we will use `head()` to indicate the symbol at the current position in RAM. The machine repeatedly carries out instructions of the following form until it terminates with either ACCEPT or REJECT:

> If the current state is ___ and the value of head() is the symbol ___ then
> 1. Replace the symbol at the current position by the symbol ___ (optional).
> 2. Move one position to the right or one position to the left (optional).
> 3. Change to another state ___ (or maybe keep the same state).

Implementing a Turing machine in code

We could code this in Java with the following `accepts` method. It is the same as the one for Finite Automata in Listing 20.1 except that the first two lines of the body of the method do the initial loading of the input into RAM. So in the body of the while-loop, the call of `input.next()` is replaced by the call of `head()`.

```
public boolean accepts (Sentence input)
{  // load the input into RAM starting at index 1
   thePos = 0;  // initial position of the tapehead in RAM
   int state = q0;
   while (state >= q0)
      state = nextState (state, head());
   return state == ACCEPT;
}  //=====================
```

This means that a main method could operate a particular Turing machine (**TM** for short) and print the result with a statement such as the following, exactly as in Listing 20.3 (here Palindrome is a TM we will code next):

```
System.out.println
      (new Palindrome().askOpinion (new Sentence (args[0])));
```

We could store the particular instructions that drive the TM in a 2-dimensional array indexed by state and symbol, as in Listing 20.3. Many authors develop TMs that way. But it is quite useful to think of each state as a single Java method with a very limited structure, returning a boolean value of either true (for ACCEPT) or false (for REJECT). This leads to the abstract TuringMachine class in Listing 20.5 (see next page). Study the upper part of that listing now. The call of `start()` corresponds to making the state q0.

Listing 20.5 The TuringMachine abstract class

```
public abstract class TuringMachine
{
   public final boolean ACCEPT = true, REJECT = false;
   private final byte[] theRam = new byte[10000];
   private int thePos = 0;


   public String askOpinion (Sentence input)
   {  return accepts (input)  ?  "accepted"  :  "rejected";
   }  //=====================


   public boolean accepts (Sentence input)
   {  if (input != null)
      {  theRam = new byte[10000];  // start with fresh input
         int k = 1;
         byte symbol = input.next();
         while (symbol >= 0)
         {  theRam[k++] = symbol;
            symbol = input.next();
         }
         thePos = 0;
      }
      return start();
   }  //=====================


   public byte head()
   {  return theRam[thePos];
   }  //=====================


   public static void moveOn()
   {  thePos++;
   }  //=====================


   public static void backUp()
   {  thePos--;
   }  //=====================


   public static void moveOn (byte replacement)
   {  theRam[thePos++] = replacement;
   }  //=====================


   public static void backUp (byte replacement)
   {  theRam[thePos--] = replacement;
   }  //=====================


   public abstract boolean start();
}
```

Some exercises address the need for increasing the size of the array if more room is needed, and for returning REJECT if the machine tries to make `itsPos` negative.

This listing has an abstract method `start` that will vary from one Turing machine to another, depending on what it is supposed to do. The methods in the middle of Listing 20.5 are the four basic operations: `moveOn` moves one position to the right in RAM (i.e., increments the index in RAM by 1); if it has a parameter, the given symbol replaces the current value in RAM before it moves to the right. For instance, `moveOn(1)` writes 1 in the current position before moving right. Similarly, `backUp` does exactly the same except it goes to the left (i.e., decrements the index in RAM by 1) instead of to the right.

A method that represents one state of the TM is restricted to (a) looking at the current `head()` value and then, depending on that value, (b) executing at most one of the `moveOn` and `backUp` methods, and finally (c) returning a boolean value of ACCEPT or REJECT or the value returned by some other state of the TM (or possibly of the same state). This means that, when the non-blank symbols can only be 0 or 1, the coding for a TM state should be the following multi-way selection statement or its equivalent:

```
if (head() == 0)
{    moveOn/backUp, possibly with a replacement value for the symbol
     return ACCEPT/REJECT/otherState
}else if (head() == 1)
{    moveOn/backUp, possibly with a replacement value for the symbol
     return ACCEPT/REJECT/otherState
}else if (head() == -1)   // representing a blank
{    moveOn/backUp, possibly with a replacement value for the symbol
     return ACCEPT/REJECT/otherState
}
```

If you think for a while, you can see that the first statement in such a method could be:

```
while (head() == 1)
     moveOn/backUp, possibly with a replacement value for the symbol
```

As long as the rest of the coding of that method is the multi-way selection statement shown above, but with the alternative for 1 omitted, this is the equivalent of having the alternative for 1 end with `return theCurrentMethod()`. Similarly, you could start with this while-statement

```
while (head() == 0 || head() == 1)
     moveOn/backUp, possibly with a replacement value for the symbol
moveOn/backUp, possibly with a replacement value for the symbol
return ACCEPT/REJECT/otherState
```

because it is the equivalent of calling the current method for 0 and 1, but doing the part after the while-loop for -1.

Think of the Turing machine as having two light bulbs on it marked ACCEPT and REJECT. If and when the Turing machine halts, one of the two light bulbs comes on. You can also think of whatever is **left on the tape** (i.e., in RAM) as an **output** of the machine.

A Turing machine to test for palindromes

The standard multi-way selection statement is the body of the `isPalindrome` method in the middle part of Listing 20.6 (see next page). The `start` method simply moves to the right from the blank at position 0 and calls the `isPalindrome` method to test whether the input is a palindrome (reads the same backwards as forwards). This Turing machine accepts the input if it is a palindrome and it rejects the input if not.

Listing 20.6 The Palindrome-testing TuringMachine subclass

```
public class Palindrome extends TuringMachine  // 6 states
{
   public boolean start()
   {  if (head() == -1)                                          //1
      {  moveOn();                                               //2
         return isPalindrome();                                  //3
      }                                                          //4
      else // the current position has 0 or 1                   //5
         return REJECT;                                          //6
   }  //=====================


   public boolean isPalindrome()
   {  if (head() == 0)                                           //7
      {  moveOn (-1);                                            //8
         return findZeroAtOtherEnd();                            //9
      }else if (head() == 1)                                     //10
      {  moveOn (-1);                                            //11
         return findOneAtOtherEnd();                             //12
      }else                                                      //13
         return ACCEPT;                                          //14
   }  //=====================


   public boolean findZeroAtOtherEnd()
   {  while (head() == 0 || head() == 1)                         //15
         moveOn();                                               //16
      backUp();                                                  //17
      return shouldSeeZero();                                    //18
   }  //=====================


   public boolean shouldSeeZero()
   {  if (head() == 0)                                           //19
      {  backUp (-1);                                            //20
         return goToFront();                                     //21
      }                                                          //22
      else                                                       //23
         return (head() == 1) ? REJECT : ACCEPT;                 //24
   }  //=====================


   public boolean goToFront()
   {  while (head() == 0 || head() == 1)                         //25
         backUp();                                               //26
      moveOn();                                                  //27
      return isPalindrome();                                     //28
   }  //=====================


   // the following two are left as exercises
   public boolean findOneAtOtherEnd()
   public boolean shouldSeeOne()
}
```

The `isPalindrome` method repeatedly removes one symbol from each end of the input, making sure they are the same symbol, until the entire input has been processed. So it returns ACCEPT if it can remove all symbols this way. If the `isPalindrome` method detects a 0 as the first symbol, it writes a blank in its place and calls the `findZeroAtOtherEnd`, but if it detects a 1 as the first symbol, it calls the analogous `findOneAtOtherEnd` (which is left as an exercise).

The `findZeroAtOtherEnd` method loops to the right past all the symbols until it gets to the blank at the other end (in effect, calling itself repeatedly). Then it backs up one to make sure there is a 0 on the other end. If there is only a blank left, the `shouldSeeZero` method accepts this palindrome of odd length. If there is a 0 on the other end, it replaces it by a blank and then moves all the way to the left to start over.

The reason for allowing a restricted kind of while-statement instead of using an equivalent if-statement with recursion is not to save a few lines of code or to be more efficient. The Turing machine should execute a goto statement for each method, which avoids storing all the active method calls on an internal stack. But Java is too advanced a language to have a goto statement; method calls allow recursion. Serious use of the TuringMachine class could involve a stack of thousands of method calls, which can lead to a stack-overflow condition. The while-loop greatly postpones the problem.

We allow just a few variations from the standard multi-way selection statement, for convenience, and only as long as it is very clear what the standard multi-way selection statement would actually be. A strict coding of a TM would use only standard multi-way selection statements.

Exercise 20.15 Revise Listing 20.5 so that the `accepts` method catches any attempt to move below index 0 in the array and returns the appropriate value.
Exercise 20.16 Write the `findOneAtOtherEnd` method for Listing 20.6.
Exercise 20.17* Revise Listing 20.5 so that `theRam` doubles in size when an attempt is made to increase `itsPos` too high.
Exercise 20.18* Write the `shouldSeeOne` method for Listing 20.6.

20.4 Unsolvable Problems And Uncomputable Functions

The primary purpose of Turing machines is to allow the study of what computers can and cannot do. The **Church-Turing Thesis** is that a Turing machine can perform any computation that a modern computer can do, only much more slowly. This assertion is not provable logically, but its truth is generally accepted by computer scientists. So if we prove that certain problems cannot be solved by a TM or that certain functions cannot be computed by a TM, it follows that no one can program a modern computer to solve those problems or compute those functions.

A TM is a finite sequence of characters. If you replace each character by its Unicode value (0 to 255, 8 bits each), and treat the sequence of characters as one long number in base two, you get a single number for each TM. For instance, if it takes 1000 characters to write in Java a strictly-coded TM (confined to multi-way selection statements), that TM corresponds to an 8000 digit number in base 2. Different TMs have different numbers. Call this number the **binary encoding of the Turing machine**.

You could even write a Turing machine program to take as input a sequence of 1s and 0s and accept that input if it is the binary encoding of some Turing machine, rejecting the input otherwise. In other words, the binary encoding of a TM can be the input for a TM. You could write a TM that tells whether its input is the binary encoding of a TM that has less than 10 states. But there are some problems of this type you cannot solve.

Universal Turing Machines

A 1936 paper by Alan Turing developed the coding for a **Universal Turing Machine** (**UTM** for short). This is a Turing machine that takes as input the binary encoding of a TM followed by a particular sequence of 1s and 0s, and then produces as output whatever that TM would have produced on the input of that sequence of 1s and 0s. That is, if and when the UTM terminates, its ACCEPT or REJECT state is the same as the given TM would have produced, and whatever is in RAM is what would have been there if the given TM had been run with the given input.

This paper therefore described the idea of a stored-program computer. Early computers hard-wired their programs; if you wanted to change what action the computer took, you had to move some wires around. Later computers functioned by first reading in a specification of the program to be executed and then running it on various inputs. Of course, those early computers required that the program be in some numeric code (machine code). That is essentially what the binary encoding of a Turing machine is.

Henceforth, we will often say "takes a TM as input" when it would be more precise to say "takes the binary encoding of a TM as input". This has the advantage of making several assertions easier to state and understand.

The NonSelfAccepting problem

In the town of Seville, all men shave regularly. Some shave themselves; anyone who does not shave himself goes to the barber for shaves. Key fact: The barber shaves all men, and only those men, who do not shave themselves. Question: Who shaves the barber?

You probably figured out the answer -- the barber is not a man. She cannot be a man, because if she were, then he would either shave himself (contradicting the second part of the key fact) or he would not shave himself (contradicting the first part of the key fact). So she must be a woman.

When you execute a Turing machine on a particular input, either it says the input is acceptable or it does not. You could execute any TM on the input that is its own binary encoding, and it will either return ACCEPT or it will not. Let us say a TM is SelfAccepting if it accepts its own binary encoding as input, and it is NonSA if it does not accept its own binary encoding as input.

The **NonSelfAccepting enumeration problem** is the problem of writing a Turing machine that accepts any NonSA TM (and does not accept any other). Note that the Turing machine you write is allowed to go into an infinite loop for any TM that is not NonSA. We will explain later in this section why it is called an "enumeration" problem.

Theorem 1 No Turing machine can be the answer to this NonSelfAccepting enumeration problem.

Proof by contradiction If X were a TM that accepts only the NonSA TMs, then X would have to either (a) return ACCEPT when given its own binary encoding, or (b) not return ACCEPT when given its own binary encoding. In case (a), X is SelfAccepting, which X does not accept. In case (b), X is NonSA, which X does accept. Conclusion: X cannot be a TM.

The Church-Turing Thesis then allows us to conclude that, if anyone ever claims to have written a computer program that will read in a TM and say whether it is NonSA, you can be absolutely guaranteed that the program has a bug in it. It cannot be done. **Every TM is either NonSA or it is not, but no one can write a correct program to tell whether any given TM is NonSA.**

You can obviously write a Turing machine that takes some TM as input and very often tells whether it is or is not NonSA. The theorem just says that such a Turing machine will not be able to tell for every TM.

The SelfAccepting problem

The **SelfAccepting enumeration problem** is the problem of writing a Turing machine that accepts any SelfAccepting TM and does not accept any other TM.

Theorem 2 A Turing machine exists that is the answer to the SelfAccepting enumeration problem.

Proof by construction Let E be the binary encoding of a given TM. Apply the Universal Turing Machine to the input consisting of E followed by E. If the given TM produces ACCEPT given its encoding as input, then the UTM will also produce ACCEPT given EE. If the given TM does not ACCEPT itself, the UTM will not ACCEPT its EE either.

A **decision problem** is the problem of accepting all things of a particular type that satisfy a certain property and rejecting all things of that type that do not satisfy the property. For instance, the SelfAccepting decision problem is the problem of writing a Turing machine that accepts a SelfAccepting TM and rejects all other TMs. Here the particular type of thing is a binary encoding of a TM. By contrast, a solution to an enumeration problem can loop infinitely on an input that it does not accept, rather than stop by rejecting.

Theorem 3 The SelfAccepting decision problem is not solvable.

Proof by contradiction If X were a Turing machine that accepts all SelfAccepting TMs and rejects all NonSelfAccepting TMs, then you could write a new Turing machine as follows: The `start` method of the new machine calls the `start` method of X and returns the opposite of what X's `start` method returns. That would give you a TM that accepts all NonSA TMs, which Theorem 1 proved to be impossible.

Note that the difficulty here is that any TM that accepts all SelfAccepting TMs and no others, must go into an infinite loop for some TMs that are not SelfAccepting. So the enumeration problem has an answer but the decision problem does not. In other words, any computer program that correctly spots SelfAccepting TMs must go into an infinite loop on some inputs. In yet other words, the SelfAccepting problem is enumerable but not decidable.

An **enumeration problem** is the problem of accepting all things of a particular type that satisfy a certain property and either rejecting or going into an infinite loop on all things of that type that do not satisfy the property. It is so-called because of a theorem that constructs an Enumerating Turing Machine (ETM for short). An ETM takes as input any TM and produces as output (on the tape, i.e., in RAM) all the inputs that that TM accepts, one at a time. They will not necessarily be in any particular order. But every input that the TM accepts will eventually (after a finite amount of time) appear on the tape.

Therefore, given a TM X that is the answer to some enumeration problem, the ETM acting on X will enumerate (list) all the inputs that X accepts. We do not explain here how to construct an ETM; we leave that for a course in Theory of Computation.

The Accepting Problem

Theorem 4 No Turing machine can be the solution to the **Accepting decision problem**, which is the problem of deciding (accept or reject) whether any given Turing machine will accept a given input.

Proof by contradiction Assume X is a Turing machine that takes as input some TM followed by a particular string of symbols S, and X always returns ACCEPT if that TM accepts S and X returns REJECT if that TM does not accept S. Then you could apply X to the input consisting of E followed by E, where E is the binary encoding of a given TM. That would give you a solution of the SelfAccepting decision problem, contradicting Theorem 3.

Theorem 5 A Turing machine exists that solves the **WritesSomething decision problem**, which is the problem of deciding whether any given TM writes any symbol at all when its input is an all-blank tape.

Proof by construction Have the UTM execute the given TM T for n+1 instructions, where n is the number of states that T has. If T has not written anything on the tape yet, T never will. The reason is that, at the end of executing n+1 instructions, T will be in a state that it has previously been in, and the tape will be in the same configuration it was at the time T was in that previous same state, namely, all blanks, so T will repeat that sequence of operations that writes nothing over and over.

The **Halting decision problem** is to find a Turing machine that tells whether or not any given TM halts on a given input (accepts those that do, rejects those that do not). Clearly, we already have a Turing machine that returns ACCEPT for any given TM T that halts on a given input, namely, the UTM. It calls T's `start` method; if and when T accepts or rejects, the UTM returns ACCEPT. But the UTM goes into an infinite loop if T does. So the UTM is an answer to the Halting enumeration problem.

Theorem 6 No Turing machine can be the solution to the Halting decision problem.

Proof by contradiction Assume X is a Turing machine that takes as input some TM T and some input P and always returns ACCEPT if T does not go into an infinite loop on input P, but returns REJECT otherwise. Then you easily write a Turing machine Y that (a) calls X's `start` method applied to the input consisting of E followed by E, where E is the binary encoding of a given TM; (b) returns REJECT if X's `start` method returns REJECT (signalling an infinite loop), otherwise executes the given TM on the same input and returns whatever it returns. Then Y would be a solution of the SelfAccepting decision problem, contradicting Theorem 3.

By the Church-Turing Thesis, you cannot write a computer program that can test any TM to tell correctly whether it does or does not halt. In other words, any attempt to write such a computer program will have a bug in it.

If you are given a TM to work with, you can always write a program that tells whether or not that particular TM halts. This may seem startling, but it is actually trivial. After all, you just write two programs, one that accepts it and another that rejects it, and one of those will be the one you wanted. Unfortunately, you will often not know which one.

Computable functions

When you execute a Turing machine, it leaves certain symbols on the tape (i.e., in RAM) if and when it terminates. That is its output. Say you have a TM that, whenever its input is a number of 1s, terminates with a (usually different) number of 1s on the tape. Then you could consider it to be computing a function of non-negative integers for which the answer is always a non-negative integer.

For instance, you could have a TM that moves to the right until it sees a blank (after index 0), then writes a 1 and quits. So if it started with 17 1s, it would end with 18 1s, and if it started with 200 1s, it would end with 201 1s. It is computing the function f(n) = n+1. Or you could have a TM that copies its input to the position directly after the input. So if it started with 17 1s, it would end with 34 1s, and if it started with 200 1s, it would end with 400 1s. It is computing the function f(n) = 2*n.

TMs can compute functions that have very large values. For instance, it is possible to write a TM that replaces an input of 1 with 1, replaces an input of 2 with 2^2 (which is 4), replaces an input of 3 with 3-to-the-3^3 power (which is about 7 trillion), etc. In general, an input of n has an output of n-to-the-nth-to-the-nth-to-the... for n levels. This "pyramid" value gets to be really big really fast.

A **computable function** is a function that a TM can be written to compute in the sense described above. The interesting thing is that there are some functions you can define but you cannot compute.

Define the **Busy Beaver function b(n)** as follows for TMs where 0 and 1 are the only non-blanks allowed on the tape. ACCEPT and REJECT are the **halting states**:

- b(1) is the largest number that a TM with just 1 non-halting state and an input of 1 can leave as output and terminate (going into an infinite loop does not count).
- b(2) is the largest number a TM with 2 non-halting states and an input of 11 can leave as output and terminate.
- b(3) is the largest number a TM with 3 non-halting states and an input of 111 can leave as output and terminate.
- b(4) is the largest number a TM with 4 non-halting states and an input of 1111 can leave as output and terminate.
- etc.

Theorem 7 No Turing machine can compute the Busy Beaver function.

Proof by contradiction Assume you have a Turing machine T that computes the Busy Beaver function. Replace each return of ACCEPT or REJECT in T by a call of another method (state) that does the following:

```
public boolean addOne()
{  if (head == 1)
   {  moveOn();
      addOne();
   }else  // it is 0 or blank
   {  moveOn (1);
      return ACCEPT;
   }
}
```

Let n denote the number of states this revised machine has. Then this revised machine will output b(n)+1 1s when the input is n 1s. That contradicts the definition of the Busy Beaver function b(n).

Note that this function is unambiguously defined, but not in a "constructive" way -- It does not tell you how to compute it. It cannot, as the theorem shows -- No one can write a computer program that computes b(n) for every n.

This theorem is a "non-existence" theorem -- it proves that something does not exist. If you prove something does exist, it does not always help. For instance, it is true that, for any n you choose, there exists a TM that computes b(n). But it is trivial and it is useless. First choose a positive number n. Then write a program that prints a TM that prints one 1, then a TM that prints two 1s, then a TM that prints three 1s, etc., going on without limit. Eventually it will print a TM that computes that particular b(n). So such a TM exists. But you do not know which one it is.

Exercise 20.19 The pyramid function evaluated at 4 produces 4 to some power. Choose the closest answer for the power: 10^5, 10^{15}, 10^{50}, 10^{150}.
Exercise 20.20* Explain why no Turing machine can tell whether any given TM whatsoever ever writes the symbol 0 on its tape.

Part B Enrichment And Reinforcement

20.5 Extended Example Of A Context-Free LL(1) Grammar

In this section we describe a complete new programming language, mainly in terms of its context-free grammar. Java is a language that was developed for professional use, so it has a number of drawbacks for purposes of teaching beginning programming. We want a mild simplification of Java that (a) is as good as possible for teaching beginning programming, subject to (b) is so much like Java that the equivalent "real" Java can be learned in an hour or so.

One problem with languages intended for professional use is that an advanced feature useful to people who know what they are doing can be accidentally invoked by a beginner. This can totally confuse the beginner. It is better if the compiler tells the beginning student that the coding is simply wrong.

Lisp is a language used in artifical intelligence applications. Scheme is a language based on Lisp but modified to be highly suitable to teaching beginners. One of its features that we will adopt here is to have several levels of the language; You learn Scheme#1 to start with, then Scheme#2 adds more features to Scheme#1, then Scheme#3 adds yet more. A feature the compiler flags as an error at Level 1 is accepted at Level 2.

We call our beginner's language J1 (since it is Level 1 of a simplification of Java). J1 does not allow interfaces or throwing/catching Exceptions. It allows inheritance and late binding (where the overriding of a method call cannot be determined until runtime) but not private methods. J2 allows all of these, plus nesting of one class inside another.

All classes in the J1 language are considered as being in the same package; J3 allows packages and imports. J1 has only one kind of number, Java's float (32-bits, whole number values exact to plus or minus 8 million, up to plus or minus 10^{80}). J3 adds the byte, short, int, long, and double numeric types. We do not need the char type if we have the methods `s.at(k)` for the Unicode of the kth character of s and `System.chr(u)` for the 1-character String whose character has Unicode u.

The J1 interpreter

Another problem for beginners in Java is that getting a small program to compile and run is a lot of trouble. You have to write a complete class definition in a text file using some text editor; save the file on disk; run the compiler to produce the compiled form; and then execute the compiled file. A line-by-line interpreter is far more intuitive for beginners.

Our J1 language will have a compiler, of course, invoked by a command similar to javac. But it will also come with an interpreter program that accepts lines written in J1 and gives an immediate response, similar to Java's DrJava at http://drjava.sourceforge.net/. If the input is an expression rather than a J1 statement, the interpreter prints the expression's value.

We start with some examples of student interactions with the interpreter. The interpreter gives an `input>` prompt and responds to each line typed with the information shown at the right in the following example:

```
input> 2 + 3                    5
input> 4 + (7 * 2.1)            18.7
input> 5 / 2                    2.5
input> 7 / * 2.1                ERROR: * is not allowed there
```

Grammar rules for basic expressions

A number of items added together could be defined with the following grammar rules (where Ë indicates you may have nothing there):

```
ManyTerms --> Term  MaybeMoreTerms
MaybeMoreTerms --> AddOp  Term  MaybeMoreTerms  |  Ë
AddOp --> +  |  -
```

Programming languages typically have many situations in which you can write one or more of something. We use a special shorthand notation for such cases: **Braces** around a part of a rule means that part can occur as many times as you want at that point. So the first two lines above can be summarized as follows:

```
ManyTerms --> Term {AddOp  Term}
```

This braces shorthand allows for eliminating most uses of the lambda symbol Ë (standing for the empty string) from the grammar rules. Of course, it could be ambiguous if the language allowed braces as symbols, but J1 syntax does not have braces.

A **Literal** is an element in a computer program that represents an explicit and unchanging value. In the grammar, "Literal" is considered to be a terminal, since it is not defined by grammar rules. There are four kinds of Literals in J1, defined as follows:

1. A string literal is quotes around anything not involving quotes or returns except \" .
2. A numeric literal is a sequence of one or more digits that may have a dot between two digits, optionally followed by an exponent base 10, along the lines of 42e+21 or 3.704e-06 (implemented as a 32-bit float value).
3. A boolean literal is one of the two keywords `true` and `false`.
4. The only object literal is the keyword `null`.

Grammar rules for expressions

We now look at the grammar rules that define a general **Expression** in J1. These allow for boolean expressions using the conditional operator and relational operators. J1 uses keywords (`not`, `and`, and `or`, boldfaced in grammar rules) instead of Java's symbols !, &&, and || for booleans, since beginners find words easier. The conditional operator is easier to recognize with an initial keyword; for instance, `x = when y > 2 ? 7 : 5` is easier to comprehend than `x = y > 2 ? 7 : 5`.

```
1.  Expression  -->  Term  MaybeMore
        |  when  Expression  ?  Expression  :  Expression
2.  MaybeMore -->  or  Term  {or  Term}  |  and  Term  {and  Term}
        |  +  Term  {  +  Term}  |  *  Term  {  *  Term}
        |  <  Term  |  >  Term  |  <=  Term  |  >=  Term  |  ==  Term  |  !=  Term
        |  -  Term  |  /  Term  |  %  Term  |  Ë
3.  Term  -->  ClassName  {  [  ]  }  Primary  |  float  {  [  ]  }  Primary
        |  boolean  {  [  ]  }  Primary  |  not  Primary  |  -  Primary  |  Primary
4.  Primary  -->  VariableOrMethodCall  |  Literal  |  (  Expression  )
```

J1 avoids the difficulties with operator precedence that languages cause for some beginners, by simply not allowing two or more binrary operators in sequence unless they are the kind where the order makes no difference (`+` `*` or and), unless the programmer uses parentheses to indicate the order of operations.

LL(1) grammars

A grammar is **unambiguous** when you can always tell which of several rules to apply by looking at the next symbol in the coding to be processed. This is obviously true when a nonterminal is defined with only one rule (the definition of ManyTerms) or with several rules each beginning with a different terminal (the definition of SpecialOp).

When a definition has several rules of which all but one begin with a different terminal, and the remaining rule begins with a nonterminal defined so that it must begin with some other terminal, the definition is unambiguous. You will see shortly that a VariableOrMethodCall must begin with a word other than `true`, `false`, or `null`, so definition 4 is unambiguous. The word at the beginning of a VariableOrMethodCall cannot be a ClassName, `float`, `boolean`, or `not`, so definition 3 is unambiguous. And the word also cannot be `when`, so definition 1 is unambiguous.

When a definition has several rules of which all but one begin with a different terminal, and the other rule is the empty string Ë, that definition is unambiguous if the nonterminal being defined can never be followed by one of those terminals in the language. Definition 2 is therefore unambiguous: an Expression can never be directly followed by MaybeMore, and no other definition contains one of the 13 terminals for MaybeMore.

For purposes of this section, let us define a **perfectly-clear nonterminal** to be one for which every rule in its definition begins with a different terminal (and no rule is the empty string). For instance, AddOp is perfectly-clear. Let us define the **selectors** for a definition to be (a) whatever terminals occur at the beginning of the various rules for that definition, plus (b) the selectors for any perfectly-clear nonterminal that occurs at the beginning of some rule for that definition. For instance, Expression has one selector `when`, and MaybeMore has 13 selectors.

In general, a basic **LL(1) grammar** is a grammar for which each definition's selectors are all different, and each rule in that definition begins with a terminal except possibly one rule, and moreover, that one exceptional rule (when it exists) expands to:

a) a non-empty string that cannot begin with one of the selectors, or
b) the empty string Ë, in which case the nonterminal being defined is never followed in any grammar rule by something that begins with one of the selectors.

A special requirement applies for the shorthand braces notation: If a production rule contains {X}, that is shorthand for some Y defined by Y --> X Y | Ë. The grammar is still LL(1) if (a) X begins with a terminal that can never follow {X}, or if (b) X begins with a perfectly-clear nonterminal none of whose selectors can follow {X}, or if (c) {X} is followed by a terminal that cannot begin X. So the braces in definitions 2 and 3 are unambiguous by (a). You will see examples of (b) and (c) later in this section.

Variables and method calls

Just as in Java, there are two kinds of construction in J1, illustrated by `new Turtle()` and by `new Turtle[5][n]`. The latter can also be used with the primitive types `float` and `boolean`, but the former cannot. These could be described as follows:

```
Construction --> new Type RestOfConstruction
RestOfConstruction --> Parameters | [ Expression ] { [ Expression ] }
```

However, those two definitions do not forbid having `new float(...)`. So they have to be made more precise. The grammar rules in the following block define a **VariableOrMethodCall** in terms of Expression. Reminder: if xxx is some method name, then `super.xxx` invokes the method named xxx in the superclass, even when xxx is defined in the current class.

5. VariableOrMethodCall --> Variable { . MethodName Parameters } | Message
   | SingletonName . Message | **new** Construction
6. Variable --> **this** | **super** | VariableName { [Expression] }
7. Message --> MethodName Parameters { . MethodName Parameters }
8. Parameters --> (RestOfParameters | Ë
9. RestOfParameters --> Expression { , Expression }) |)
10. Construction --> **float** [Expression] { [Expression] }
    | **boolean** [Expression] { [Expression] }
    | ClassName RestOfConstruction
11. RestOfConstruction --> Parameters | [Expression] { [Expression] }

Note that J1 does not require empty parentheses for a constructor call or method call with no parameters. This complicates the grammar, but it really reduces irritation for beginning programmers. Also, J1 has no compound variables such as x.length, but that is not really a problem, and it makes the language easier for beginners to learn.

Definitions 6, 7, and 10 maintain the LL(1) property because they are perfectly-clear. Definition 5 is okay because none of the three terminals that can begin a Variable can begin the other three alternatives for definition 10. Definition 9 is okay because an Expression cannot start with a right parenthesis. A search of all the definitions in this grammar will show that Parameters cannot be directly followed by a left parenthesis or a left bracket. This shows that definitions 8 and 11 maintain the LL(1) property.

The braces in definitions 5-11 must also be checked: The one in definition 9 obviously maintains the LL(1) property. The ones in definitions 6, 10, and 11 are okay because a left bracket cannot follow a Construction or a Variable. The ones in definitions 5 and 7 are okay because a period cannot follow a VariableOrMethodCall.

These LL(1) conditions are rather complicated, but they make it easy to write a compiler that looks ahead one token and chooses the rule with that as its selector. If no rule has the next token as its selector, it choose the one rule that is empty or not perfectly-clear.

Assignment statements and local declarations

A variable that is local to a method is both declared and initialized with the := symbol. So in J1, no beginner can declare a local variable and forget to initialize it. You can later re-assign a new value to the variable using the simple = assignment symbol. Of course, a variable must be declared before any point where it is used or re-assigned. Here are some more examples of interpreter interactions with a beginning programmer:

input> x := 7 % 2	x initialized to 1
input> x / 3	0.3333333
input> x = 3 + 4	x assigned 7
input> x / 3	2.333333
input> x += 5	x assigned 12
input> x / 3	4

If the first input had been x = 7 instead of x := 7, the interpreter would have responded with `ERROR: x has not been declared`. And if the third input had been x := 3 + 4 instead of x = 3 + 4, the interpreter would have responded with `ERROR: x cannot be declared again`. J1 uses x += 1 and x -= 1 instead of ++ x and -- x, to simplify the language while keeping the ability to increment or decrement efficiently.

Many beginners find it puzzling that Java requires statements such as the following:

```
float x = 7.2;
Turtle y = new Turtle();
```

They ask, "Why do you have to say what type it is? The compiler can see what the type is from what is assigned to it." Also, they have trouble remembering when to put semicolons at the end, particularly since anyone can see where the statement ends without the semicolon. Moreover, they do not see why you should write parentheses that contain nothing on method calls. So in J1, we write these two statements as follows (the empty parentheses and semicolons are however, optional, if you prefer to put them in):

```
x := 7.2
y := new Turtle
```

What is lost by using the := symbol instead of stating the type as in Java? One thing lost is the ability to distinguish between different numeric types, as in `int x = 4` and `long x = 4`. But other numeric types besides float do not occur until J3, so that is not a problem in J1. It will be solved there by the device described in the next paragraph.

One other thing is lost: `x := null` does not tell what type of variable x is. So J1 require a typecast in this case, as in `x := Turtle null`. The compiler signals an ambiguity error if you leave out this typecast. A typecast to a superclass, as in Java's `Object x = new Turtle()` is expressed in J2 as `x := Object new Turtle` (parentheses around the typecast would make it harder for the grammar to be LL(1)). J1 requires Java-like type specifications for parameters and field variables, since their values are never assigned to them at the point where they are declared. Local variables and parameters cannot be named the same as field variables or methods in the class.

Modules and class declarations

A confusion for beginners in Java is that a class can be either of two quite different things: A module that describes a class of objects that can be constructed versus a module that contains a `public static void main` method. Beginning students also have trouble with distinguishing between class methods and instance methods, between class variables and instance variables, etc.

J1 eliminates the confusion by allowing two kinds of **modules**, a **class** and a **singleton**. A class contains only instance methods and instance variables; a singleton contains only static methods and static variables. A compilable file in J1 consists of one or more modules, each of which can be a class or a singleton. J1 does not use the `static` keyword; all members of a singleton module are static but no members of a class module are static. If you want a class of objects to have static variables as well as instance variables, J2 will let you put a singleton module nested inside the class module, privately if you wish, to hold those static variables; it accomplishes the same thing.

A singleton module should be thought of as a description of a single object that is created when the program begins execution. For instance, you could think of Java's Math class as describing an object named Math with "instance method" `Math.abs()` and "instance variable" `Math.PI`. We define a module declaration as follows:

ModuleDeclaration --> **class** Name **extends** ClassName {Declaration} **end**
| **singleton** Name {Name Definition} **end**

The `end` keyword marks the end of the module definition, so we do not need braces. In the grammar, "Name" is treated as a terminal, since it is not defined by grammar rules. It is an **identifer**: a sequence of letters, digits, and underscores, beginning with a letter, that is not a keyword and is not declared elsewhere as a class name or singleton name. The compiler will treat the Name following `class` or `singleton` as a ClassName or SingletonName, respectively, later on in the compilation. So other grammar rules necessarily treat ClassName and SingletonName as terminals. Note: "Declaration" includes constructor declarations, which are not allowed in singleton modules.

Grammar for declarations of modules, methods, and field variables

The complete definition of **ModuleDeclaration**, in terms of Expression, Parameters, and Statement, is in the following block. The compiler considers a Name declared at the beginning of a method heading (i.e., with a Definition that starts with parentheses) to be a MethodName anywhere else in that module. That Name can be previously declared in the same module as a MethodName, as long as it is with a different number of parameters (to allow some overloading).

12. ModuleDeclaration --> **class** Name **extends** ClassName {Declaration} **end**
| **singleton** Name {Name Definition} **end**
13. Declaration --> Name Definition
| ClassName (VariableDeclarations) **super** Parameters {Statement} **end**
14. Definition --> := Expression
| (VariableDeclarations) **returns** ReturnType {Statement} **end**
15. ReturnType --> Type | **nothing**
16. VariableDeclarations --> Type Name { , Type Name } | Ë
17. Type --> ClassName { [ ] } | **float** { [ ] } | **boolean** { [ ] }

The compiler considers a Name declared in a VariableDeclaration of a method heading to be a VariableName in the rest of that method. A Name declared with a Definition beginning with := outside any method is a **field variable**; the compiler considers it to be a VariableName elsewhere in that module. A Name of a field variable cannot be declared as the name of a method or of another variable in the same module. The compiler will process all method headings in a class before analyzing the body of any method.

J1 has method headings begin with the method name because that makes it easier to find one method amongst several pages of methods. What Java denotes as a void method, J1 denotes as `returns nothing` (making it clearer to beginners).

In J1, all methods are public by default and all variables are private by default, since those are the most common choices in programming. In J2, we will have the keywords `private`, `public`, and `final` to modify declarations. The other new keywords added in J2 are `interface`, `instanceof`, `try`, `throw`, and `catch`.

The {X} phrase in a grammar rule maintains the LL(1) property if the {X} is followed by a terminal that cannot be the beginning of X (the `end` in definitions 17, 18, and 19), or if X begins with a terminal that cannot follow the {X} (the comma in definition 21 and the left bracket in definition 22).

Definitions 17, 18, 19, and 22 maintain the LL(1) property because their rules start with different terminals. Definition 20 is okay because a Type does not start with `nothing`. And definition 21 is okay because a Type is perfectly-clear and only a right parenthesis can directly follow VariableDeclarations.

The Math module in J1 would look something like the following, in part. The runtime system evaluates all variables in a singleton module when the module is loaded for use. The method call `Math.abs(x)` returns the absolute value of x. The PI method is provided for public use, because field variables such as pi are inherently private in J1.

```
singleton Math
   pi := 3.141593
   randomSeed := System.currentTimeMillis()//for Math.random()

   abs (float x) returns float
      return when (x >= 0)  ?  x  :  -x
   end
```

```
      PI () returns float
         return pi
      end
   end
```

This grammar requires that every field variable be initialized by an assignment, even though the constructors will often re-assign the correct value based on the parameters of the constructor. But this is what Java itself does (initializing everything to null, zero, or false). You may omit constructors if the default initialization suffices; just as in Java, the compiler supplies an empty constructor when you do not give any. The first statement in a constructor must be super(...); there is no default for this as in Java. The following class creates a data structure capable of storing five or more strings of characters:

```
   class ListOfStrings extends Object
      size := 0
      item := String[] null

      ListOfStrings (float cap)  // constructor
         super()    // first invoke the superclass's constructor
         item = new String [when (cap > 5) ? cap : 5]
      end

      isEmpty () returns boolean
         return size == 0
      end

      get (float index) returns String
         return when (index < 0) or (index >= size)
                     ?  null  :  item[index]
      end

      add (String given) returns nothing
         if size < item.length then
            item[size] = given
            size += 1
         endif
      end  // of add method
   end  // of ListOfStrings class
```

The instructor of a beginning programming class can encourage students to use semicolons at the ends of statements, but the compiler will not require them. Making such semicolons optional, and empty parentheses optional for method calls, and parentheses optional around if-conditions, eliminates perhaps half of the simple but irritating errors that beginners make.

Statements

Java has the notoriously ambiguous if-else statement. We adopt a feature of some dialects of BASIC to solve the problem. An example of a J1 if-else statement is:

```
   if x > 3 then
      y += x
      stack.push (z)
   elsif x < -3 then
      y -= x
   else
      y = 0
      z = stack.pop()
   endif
```

The general idea is that you do not need braces because you always put `endif` at the end to mark where it finishes. You can put as many statements as you want in each part. You can have several `elsif` parts or none at all. Also, the `else` part is optional. Note that it is necessary to distinguish `elsif` from `else if`; the latter indicates the beginning of a separate `if` statement inside the if-structure. The following lines show a basic ambiguous Java statement and two ways it can be expressed in J1:

```
Java: if (x > 2)    if (y > 3)    z = 2;    else z = 1;
J1:   if x > 2 then if y > 3 then z = 2 endif else z = 1 endif
J1:   if x > 2 then if y > 3 then z = 2 else z = 1 endif endif
```

Much of the time, `if` statements in coding have just one statement for the then-part and have no else-part. J1 provides an alternative shorthand form for this common case, omitting `then` to unambiguously signal this shorthand form (FORTRAN-66 does this). So the following two statements mean the same:

```
if x > 2 then z = 2 endif
if x > 2 z = 2
```

J1 also has two looping statements: repeat... until x > 4 is a form that is substantially more intuitive than Java's while (x <= 4) .... The repeat statement is illustrated by the following method to find the positive whole-number logarithm of a given number. This class method could be in the Math module:

```
logInt (float base, float target) returns float // in Math
   if (target <= 1) or (base <= 0) return 0
   toReturn := 0  // a local variable declared and initialized
   repeat
      target = target / base
      toReturn += 1
   until target <= 1
   return toReturn
end
```

There is a special looping form for cases where you declare a loop-control variable. It is illustrated by the following two methods that could be in the ListOfStrings class:

```
contains (String given) returns boolean  // in ListOfStrings
   for (k := 0;  k < size;  k += 1)
      if item[k].equals (given) return true
   endfor
   return false
end

toArray () returns String[]  // in ListOfStrings
   toReturn := new String [size]
   for (k := 0;  k < size;  k += 1)
      toReturn[k] = item[k]
   endfor
   return toReturn
end
```

The grammatical rules for a **Statement** are in the following block, defined in terms of Expression, Message, and Construction. The Name that occurs in the first rule for a BasicStatement must not be otherwise defined within the method or defined anywhere within the module as a method or field variable name; the compiler considers that Name to be a VariableName for the rest of the method. J1 has no `break`, `continue` or `switch` statement. The {;} part just means that semicolons are optional at the ends of statements in J1.

18. Statement --> BasicStatement {;}
    | **return** Expression {;}
    | **if** Expression Alternatives {;}
    | **repeat** {Statement} **until** Expression {;}
    | **for** (Optional ; Expression ; Optional) {Statement} **endfor** {;}
19. Optional --> BasicStatement | Ë
20. BasicStatement --> Name := Expression | Variable Action | Message
    | SingletonName . Message | **new** Construction
21. Action --> = Expression | += Expression | -= Expression | . Message
22. Alternatives --> Statement | **then** {Statement} RestOfIf
23. RestOfIf --> **elsif** {Statement} RestOfIf | **else** {Statement} **endif** | **endif**

These additional grammar rules maintain the LL(1) property: Rules 25, 26, 27, and 28 all do so obviously (a Statement cannot begin with `then`). Rule 23 is okay because a BasicStatement cannot begin with any of `return`, `if`, `repeat`, or `for`. Rule 24 is okay because a BasicStatement cannot begin with a semicolon. The braces are okay because each {Statement} part is followed by a terminal that cannot occur at the beginning of a Statement: `until`, `endfor`, `elsif`, `else`, or `endif`.

These rules finish the full LL(1) grammar for a fairly nice programming language in only 28 definitions with 25 keywords. Then you simply add a moderate number of standard library modules to allow things such as the following:

```
System.println (someString)   // has the System object print information.
System.readln()   // has the System object get the next line of input as a String.
System.readNumber()   // has the System object parse the next line of input as a float.
```

Although it is not expressed by the grammar, J1 also omits typecasting in cases where you have generic Objects in a collection. In Java, a statement such as String x = (Object) stack.pop() requires the cast, but the compiler also puts in instructions to the runtime system to verify that the Object popped is in fact a String. J1's compiler does not require the cast from Object, although it still provides the runtime check. Specifically, assigning a value of type Object to any variable of a subclass of Object is allowed without the cast.

Exercise 20.21 Write a method to be added to Math to return the value of its non-negative parameter truncated to the next-lower whole number (leave unchanged if already a whole number). Hint: Add 1 until you get a result larger than the parameter.
Exercise 20.22 Revise the grammar to make the returns nothing phrase optional.
Exercise 20.23 Revise the grammar to allow a pound sign # to replace the keyword end wherever desired (# often indicates end-of-entry for automated telephones).
Exercise 20.24 Write a ListOfStrings method `indexOf` that returns the earliest index where a given String appears in the current list. Return -1 if the String is not there.
Exercise 20.25 Revise the grammar to require empty parentheses () after a method call or constructor call that has no parameters. Simplify any rules that can be simplified.
Exercise 20.26 Essay Explain why we cannot replace definitions 14 and 15 by Parameters --> (Arguments) and Arguments --> Expression { , Expression } | Ë.
Exercise 20.27* Essay Explain why you need the distinction between := and = in J1.
Exercise 20.28* For the ambiguous Java statement that is shown written two ways in J1, write it two more ways using the short if-statement.
Exercise 20.29** Add another shorthand notation to the grammars: { }^{1} around a phrase mean that phrase is optional (note the superscript, meaning "1 or none"). Then rule 15 can be omitted, if you put { (Arguments) }^{1} wherever Parameters appears in the rules. Restate all grammar rules to use this shorthand, thereby omitting all uses of the Ë symbol. Brackets [...] are normally used in grammars for this purpose, but for J1 that conflicts with the array brackets.
Exercise 20.30 Essay** Explain how the grammar could easily be re-adjusted to be LL(1) if we simply omitted the `when` keyword.

20.6 Comparing Programming Languages -- Smalltalk

Smalltalk is an object-oriented language that has been around since the late 1970's. It is even more object-oriented than Java, in a sense. This section describes language features that Smalltalk has that Java does not. The purpose is to provide some experience comparing different ways computer languages have for communicating the same thing.

When you have a method call such as Java's str.length(), the length() part is a **message** that you send to the object referred to by str. That object str is the **receiver** of the message, and it returns an answer (the number of characters in the string). The expression str.charAt(k) is a message sent to the receiver str with a parameter. The heart of object-oriented programming is that you think in terms of sending messsages to receivers rather than in terms of writing statements that perform a particular operation or procedure using certain values.

1: Smalltalk does not put unary operators such as ! and the negative sign in front of a value. Instead, unary operators are placed after the operand. Also, unary operators are words instead of symbols. So Smalltalk would have what is shown in the left-hand column below, to mean what Java writes as in the right-hand column:

```
hasNext not       ! hasNext
b negated         - b
```

2: Method calls that do not have any parameters are written without punctuation. For instance, the left-hand column below is Smalltalk for the Java values at the right:

```
coll isEmpty      coll.isEmpty()       // for a Collection
str size          str.length()         // for a String
stack pop         stack.pop()
z asString        z.toString()
```

3: At this point, you might observe that there seems to be no difference between, on the one hand, unary operators on primitive values such as numbers and booleans, and on the other hand, no-parameter method calls on object values. The expression `hasNext not` looks as much like sending a message to an object as does `str size.`

In Smalltalk, all unary operators are simply no-parameter method calls on object values. In fact, `Number and Boolean` are subclasses of the Object class. There are no primitive values in Smalltalk -- all values are object values. Even classes are object variables, e.g., String is an object to which you send messages defined by class methods in String. This is the sense in which Smalltalk is more object-oriented than Java.

4: Binary operators in Smalltalk evaluate left-to-right in all cases (just like method calls). For instance, 3 + 4 * 2 is 7 times 2, not 3 plus 8. But you can write 3 + (4 * 2) when you want multiplication done first. Actually, 3 + 4 is treated as a one-parameter method call, where 3 is the receiver, + is the name of the method, and 4 is the parameter.

5: Some examples of one-parameter method calls are in the left-hand column below, with the corresponding Java expressions in the right-hand column:

```
str at: k            str.charAt (k)       // for a String
str equals: t        str.equals (t)       // for a String
coll add: obj        coll.add (obj)       // for a Collection
coll includes: obj   coll.contains (obj)  // for a Collection
x raisedTo: 5        Math.pow (x, 5)      // for a Number
```

The correspondence is direct, except that since numbers are objects, the class method Math.pow is replaced by an instance method of class Number, sending a message to the number itself. In general, a one-parameter method call has simply a colon after the method name rather than the connecting dot and parentheses that Java has.

Some examples of two- and three-parameter method calls are below, with the corresponding Java explained in a comment:

```
str  at: k  put: ch
      // This is sort of str.charAt (k) = ch, except of
      // course you cannot change a string object in Java
File  rename: str  to: t
      // Class method File.rename (str, t) in Java.
      // The file named str is renamed as t
aFile  writeBuffer: str  ofSize: siz  atPosition: pos
      // Could be aFile.write (str, siz, pos) in Java
Prompter  prompt: 'Enter line'  default: 'none'
      // Return the user's input in response to "Enter line"
      // if the user declines, return the empty string.
      // String literals are delimited by apostrophes in
      // Smalltalk; quotes are used to delimit comments
```

In general, a message with one or more parameters is indicated by one "key" word for each parameter, with a colon between each key and its argument.

Encapsulation and type-checking in Smalltalk

6: Only single identifiers can represent variables in Smalltalk. That is, there is no equivalent of a compound variable reference such as x.len or this.len. Say you have a class X with instance variable len. When you are coding an instance method of class X, you can refer to just plain len, which is the len of the receiver for that method. But if that instance method has a parameter of type X, you cannot refer to that parameter's len. In Java you could, even if the instance variables were declared as private.

In Java, a private instance variable of class X can be accessed anywhere inside X, but not anywhere outside X, not even in a subclass of X. In Smalltalk, an instance variable of X can be accessed inside any instance method of X or its subclasses, though only for the receiver for that instance method (the object that Java calls "this").

Thus Smalltalk has no encapsulation: Access to instance variables or class variables cannot be limited to the class where they are declared. Class variables are accessible only within the class and its subclasses, simply because the grammar does not provide a compound way to refer to class or instance variables such as Math.PI or anArray.length.

You might think that of course an exception would be made for arrays: Java allows you to use the expression item[k], a variable that represents one component of the array named item. You would be wrong. item[k] is not a variable in Smalltalk. However, every array variable has two instance methods that allow you to accomplish what you need to do with arrays. The following illustrates the use of these instance methods for assigning a value to an array component and for retrieving a value from an array component. Note that Smalltalk uses := instead of = for assignment to a variable:

```
item at: k  put: obj          item[k] = obj
x := item at: k               x = item[k]
```

7: Smalltalk has no type-checking at compile time. The compiler considers every variable to be just an Object variable. At the beginning of a series of statements in a method, you have to declare all local variables, which you do by listing them between vertical bars -- no declaration of their type is made. A series of statements is separated

by periods. So the following could be at the beginning of the body of an instance method where an array named item is an instance variable and there are two local variables:

```
| k obj |  k := 0.  obj := item at: k.
```

This lack of type-checking means you can skip all the typecasting that you have to do in Java, but it also means that many logic errors that Java detects at compile time are not detected in Smalltalk until runtime.

Smalltalk's requirement that you declare all local variables for a method before you have any statements in the method is typical of older languages such as Pascal, C, and FORTRAN. Modern languages such as Java allow you to postpone the declaration of a local variable until you actually are ready to use it. However, at runtime, all space for local variables is allocated when the method begins to execute, as in other languages.

Basic method calls that require blocks

8: Smalltalk has a boolean operator that tells whether both of two expressions are true: `x > 0 & (6 / x > 2)` is true if x is positive and less than 3; it is false otherwise. However, what if x is zero? That expression will throw an exception. This is exactly what the & operator does in Java, which is why we do not use & much in Java.

Pascal and FORTRAN have only a single and-operator that does the same as &, but Java and Smalltalk have the much more useful short-circuit-evaluation operator. In Java, with && in place of & in that expression, if the first part, `x > 0`, is false, then the second part is not evaluated. So the result is false rather than throwing an exception.

Smalltalk also has a short-circuiting and-message: `x > 0 and: [6 / x > 2]` evaluates as false when x is zero instead of throwing an exception. This is a method call, a message sent to the Boolean receiver that is the value of x > 0. But what, you ask, is that pair of brackets doing around the parameter of the and: message?

In Smalltalk, as in Java, the parameters of a method call are always evaluated before the method call is executed. For instance, the method call `sam.check(6 / x > 2)` in Java, with a boolean parameter, would crash before the method is called if x were zero. Smalltalk's solution to this problem is a **block**. A block is a segment of code inside brackets. The value of the block is a pointer to a segment of code in memory which can be executed. That is, the code segment is sent to the method, not its evaluation; the statements within the method can then choose to evaluate it if needed.

The and: message throws an exception if its parameter is a non-block expression, just as the multiplication operator * throws an exception if the second operand is a string rather than a number. But if the parameter is a block, then the and: message does not evaluate it unless the receiver of the message (the Boolean value before the and:) is true.

9: Similarly, the or: message requires a block as its parameter, and it uses short-circuiting: It evaluates the block if and only if the receiver (before the or:) is false. For instance, the following is a definition of a class method that tells whether a given positive number is an exact power of 2. This method is defined in a class named Check and has no local variables. `n even` tells whether n is even; note that unary and binary operators have precedence over method calls formed with key words, and the caret symbol ^ is used where Java uses "return":

```
isPowerOf2 : n  | |
   ^ n <= 1 or: [ n even and: [ Check isPowerOf2:  n / 2 ] ]
```

10: As you would expect, Smalltalk has an analog of Java's if-statement. It is, however, another method call. Here is an example of its use. It assigns 7 to x if `x < y` is true and assigns 4 to z if `x < y` is false:

```
x < y  ifTrue: [ x := 7 ] ifFalse: [ z := 4 ]
```

This is the ifTrue:ifFalse: message sent to a Boolean object. The two parameters have to be blocks, because only one should be evaluated. We would not want the assignment to x executed if x is not less than y, so we cannot use a non-block parameter after the ifTrue: part. And we would not want z assigned a value if x is less than y, so we cannot use a non-block parameter after the ifFalse: part. Note that this Smalltalk expression has the same meaning as Java's `x < y ? x = 7 : z = 4`.

Smalltalk also has a one-parameter method to correspond to a simple if-statement without an else-part. For instance, the following could be used as part of the logic for finding the minimum of a sequence of values.

```
x < minimum  ifTrue: [ minimum := x ]
```

The ifTrue: message is actually the same as the and: message. There is also a one-parameter ifFalse: message that is the same as the or: message.

In effect, a block is an on-the-fly definition of a class without a name and with one method. That (anonymous) class is passed as a parameter of the Boolean ifTrue: message. If the receiver is true, it executes the class's method, but if false it does not.

11: Smalltalk also has a method call that is the analog of Java's while-statement. It is a one-parameter message named whileTrue: and it requires that both the receiver and the parameter be blocks. For instance, the following method computes the sum of the integers less than a given limit:

```
sumTo: givenLimit      |  sum  i  |
   sum := 0.  i := 0.
   [ i < givenLimit ]
         whileTrue:  [ sum := sum + i.  i := i + 1 ].
   ^ sum
```

Why, you may ask, does the receiver (before the whileTrue:) have to be a block? Because the method needs to evaluate the condition each time through the loop. So it needs access to the coding to be evaluated for loop continuation, as well as the coding to be executed in the body of the loop.

Constructors

#12: Even the constructors in Smalltalk are ordinary methods, normally named `new`. Here is an example of a constructor for a Place class that has two instance variables named city and state. You could then create a new Place object with the expression `Place new: 'Boston' state: 'MA'` (which corresponds to the Java expression `new Place ("Boston", "MA")`).

```
new: givenCity  state: givenState  | |
   ^ super new  initialize: givenCity  state: givenState

initialize: givenCity  state: givenState
   city := givenCity.
   state := givenState.
   ^ self
```

The new:state: method is a class method in the Place class. It calls the superclass's new: method (i.e., sends the new: message to the Object class) to get a new object, on which it calls the initialize:state: instance method. That instance method is allowed to refer to the instance variables of the object. It then returns the object itself.

Note that Smalltalk uses "self" where Java uses "this"; the five global variables accessible in all classes are self, super, true, false, and nil. All variables are automatically initialized to nil (which corresponds to Java's null).

Cascades

For convenience in coding, you may send a sequence of two or more messages to the same object, separated by semicolons. For instance, some versions of Smalltalk have a a Pen class of objects used for drawing pictures. It provides facilities similar to the Logo programming language. The following is a sequence of messages you could send to a turtle object once it is defined as `turtle := Pen new`:

```
turtle home.      "place the turtle in the middle of the screen facing to the right."
turtle go: 200.   "move the turtle 200 pixels forward in the direction it is facing."
turtle turn: 60.  "turn the turtle 60 degrees to the left."
turtle down.      "put the drawing pen down."
turtle gray.      "make the drawing color gray."
turtle go: 50.    "move the turtle 50 pixels forward, drawing a gray line."
turtle up.        "put the drawing pen up."
turtle go: 40.    "move the turtle 40 pixels forward, without leaving a trace."
```

Since all of the messages go to the same object, you can use cascading as a sort of shorthand. That is, the following single Smalltalk statement does the same thing as the above eight statements:

```
turtle home; go: 20; turn: 60; down; gray; go: 50; up; go: 40.
```

An unambiguous grammar

The following grammar describes only the subset of Smalltalk that we have discussed in this section. It has only twelve rules. The vertical bar "|" used to list local variables is inside quotes in the rules because we are already using | as a meta-symbol meaning "choose one of these alternatives". In the definition of Expression, multiple KeyNames must be all part of one method call. Definition 3 says that a single statement can assign the same value to several different variables at once. Reminder: {X} means that you may have as many occurrences of X as you like, i.e., X occurs zero or more times.

1. MethodDefinition --> Name "|" {Name} "|" StatementSeries
   | Name : Name {Name : Name} "|" {Name} "|" StatementSeries
2. StatementSeries --> {Statement . } Statement | {Statement . } ^ Expression
3. Statement --> {VariableName :=} Expression
4. Expression --> Primary | Primary CompoundMessage { ; Message }
5. Primary --> VariableName | **self** | **super** | Literal | [StatementSeries]
   | (Expression)
6. Message --> UnaryName | BinOpMessage | KeyMessage
7. BinOpMessage --> BinaryOperator Factor
8. Factor --> Primary {UnaryName}
9. BinaryOperator --> + - * / < <= > >= ==
10. KeyMessage --> KeyName : Term {KeyName : Term}
11. Term --> Factor {BinOpMessage}
12. CompoundMessage --> UnaryName {UnaryName}
    | {UnaryName} BinOpMessage {BinOpMessage}
    | {UnaryName} {BinOpMessage} KeyMessage

This grammar is unambiguous because only one alternative from each definition can possibly apply in any syntactically correct program. But the grammar is not LL(1), because you have to look ahead several tokens to determine which alternative to apply for definitions 1, 2, 3, and 4. VariableName, KeyName, and UnaryName are all identifiers (a letter followed by letters and digits) that have previously been declared, and Name is an identifier that is being declared by the rule. But the grammar would be unambiguous even if we did not know about previous declarations. The name of a class is considered a VariableName, which is accessible in any method of any class.

The grammar for this restricted Smalltalk is so short in part because it has no rules for class definitions or instance or class variable definitions; it only has rules for a method. This is because Smalltalk implementations normally come with an environment where you create a new class by bringing up a window with a form you fill in and/or menus you choose from, including specifying the instance variables and class variables. You often create individual methods as well with a pop-up form, including the decision as to whether they are instance methods or class methods.

The primary features of Smalltalk that the restricted grammar leaves out are (a) the use of parameters of blocks, and (b) the ability to define more binary operators, even on non-numeric objects. Every Smalltalk method returns a value. If there is no caret ^ on the last statement of a method, then the method returns the value of that last statement. The value of an assignment statement is the value being assigned to the variable(s).

The lack of encapsulation (no private methods or variables) and the lack of type-checking at compile-time are serious drawbacks of Smalltalk. Many experienced programmers will find the lack of if- and for-statements and += rather irritating, as well as the requirement that all local variables be declared at the beginning of a method. But overall, the simplicity and power of Smalltalk has much to recommend it, even compared with Java.

More information, including a tutorial, a manual, and a free downloadable Smalltalk compiler, is available at www.gnu.org/software/smalltalk.

Exercise 20.31** Rewrite the twelve rules for Smalltalk to make it an LL(1) grammar.

20.7 Review Of Chapter Twenty

- This chapter describes various automata studied in a normal course in the Theory of Computability and ways of describing them. A regular grammar describes a finite automaton, and a context-free grammar describes a pushdown automaton.
- A Turing Machine is an even more general kind of automaton. The Church-Turing Thesis is that a Turing Machine embodies all the computing capabilities of modern computers, though it is of course slower.
- There are problems that can be unambiguously defined but cannot be solved by a Turing Machine. For instance, no one can write a program that analyzes any given program to tell whether it has an infinite loop (the Halting Problem).

Answers to Selected Exercises

20.1 private final int NUM_STATES = 4;
private final int[][] del = { {q0, q1, q2, q3, ACCEPT}, {q1, q2, q3, q0, REJECT},
{q2, q3, q0, q1, REJECT}, {q3, q0, q1, q2, REJECT} };

20.2 private final int NUM_STATES = 5;
private final int[][] del = { {q1, q2, q3, q4, ACCEPT}, {REJECT, q2, q3, q4, ACCEPT},
{q1, REJECT, q3, q4, ACCEPT}, {q1, q2, REJECT, q4, ACCEPT}, {q1, q2, q3, REJECT, ACCEPT} };

20.3 private final int NUM_STATES = 3;
private final int[][] del = { {q1, q0, REJECT}, {q1, q2, REJECT}, {q1, q0, ACCEPT} };

20.7 S -> 0 A | 1 B | 2 C | 3 D; A -> 1 B | 2 C | 3 D; B -> 0 A | 2 C | 3 D; C -> 0 A | 1 B | 3 D;
D -> 0 A | 1 B | 2 C;

20.8 S -> 0 T | 1 S; T -> 0 T | 1 U; U -> 0 T | 1 S | Ë;

20.9 S -> 1 S S | 0 T; T -> T 1 S | S 1 | S | Ë; // T has at least as many 0s as 1s.

20.10 A derivation is S => 0 S 2 => 00 S 22 => 00 T 22 => 001 T 22 => 001 22, so a trace of the ToPDA is:
(q0, ^Z_0) by Step 1; (q1, ^S Z_0) by Step 2; (q1, ^0S2 Z_0) by Step 3ba; (q1, 0^S2 Z_0) by Step 3bb
consuming 0; (q1, 0^0S22 Z_0) by Step 3ba; (q1, 00^S22 Z_0) by Step 3bb consuming 0;
(q1, 00^T22 Z_0) by Step 3ba; (q1, 00^1T22 Z_0) by Step 3ba; (q1, 001^T22 Z_0) by Step 3bb
consuming 1; (q1, 001^22 Z_0) by Step 3ba; (q1, 0012^2 Z_0) by Step 3bb consuming 2;
(q1, 00122^ Z_0) by Step 3bb consuming 2; the stack empty, so we accept 00122.

20.15 Add the following statement at the end of both backUp() and backUp(byte replacement):
if (thePos < 0) throw new RuntimeException("");
Replace the last statement of accepts(Sentence input) by the following:
try { return start(); } catch (RuntimeException e) { return REJECT; }

20.16 Code it the same as findZeroAtOtherEnd() except the last statement should be:
return shouldSeeOne();

20.19 4-to-the-4th is 256; 4 to that power is 2^{512}, which is about 10^{154} (as 2^{10} is about 10^3).
So the pyramid function produces 4 to the power of approximately 10^{154}.

20.21
```
trunc (number x) returns number
    below := 0
    repeat
        below += 1
    until below > x
    return below - 1
end
```

20.22 Rule 20: Definition --> := Expression | (VariableDeclarations) ReturnPart {Statement} end
Rule 20a: Returnpart --> returns ReturnType | Ë

20.23 In rules 18, 19, and 20, replace "end" by "EndMarker", and add the rule EndMarker --> end | #

20.24
```
indexOf (String given) returns number
    for k := 0; k < this.size; k += 1
        if this.item[k].equals (given) return k
    endfor
    return -1
end
```

20.25 Omit rule 14 and replace "Parameters" in rules 11, 13, 17, and 19 by "(RestOfParameters".

20.26 Expression is not perfectly-clear, so the compiler cannot decide between the two rules for Arguments simply by looking at the selectors for Expression. So Arguments does not fit our LL(1) definition.