13 Modeling Computation

Introduction

In this chapter, we will use *Mathematica* to study theoretical models of computation. We will see how to generate elements of a language from a type 2 phrase-structure grammar and how to implement finite-state machines with and without output. We will also examine the Wolfram Language's support for regular expressions, and we will implement Turing machines.

13.1 Languages and Grammars

We will write a function to generate elements of a language from a type 2 phrase-structure grammar. Recall that a type 2 grammar has productions only of the form $w_1 \to w_2$ with w_1 a single nonterminal symbol.

Our strategy for generating the language will be as follows. We initialize a list L to the empty list. In this list, we will store all words, that is, strings consisting only of terminal symbols. A list A is initialized to the list consisting of the starting symbol.

We process an element of A by removing it from the list and applying all possible productions to it. The results of the productions are either added to L if they consist solely of terminal symbols, or placed in A to be processed further.

In order to prevent the time taken from becoming excessive, we will impose a time limit using the `TimeConstrained` function. This limit will be an argument to the function.

Representation

We first need to determine how we will model the elements of the grammar in the Wolfram Language.

We will generally represent terminal symbols as lower case letters stored as characters (strings). Nonterminal symbols will be upper case letters, also entered as strings.

Strings containing nonterminal symbols and words will be stored as strings. Productions will be stored in an `Association`. The keys will be the nonterminal symbols (recall that we are considering only type 2 grammars). The value associated to a nonterminal symbol will be the list of all products derivable from that symbol.

In Example 12 in the textbook, $S \to \mathrm{AB}$ is the only derivation from the starting symbol, so {"AB"} will be the entry associated to S in the indexed variable. On the other hand, $B \to \mathrm{Ba}$, $B \to \mathrm{Cb}$, and $B \to b$ are all productions from B. Thus, {"Ba", "Bc", "b"} would be the entry associated to B.

Here are the productions for Example 12.

```
In[1]:= ex12productions=<|"S"→{"AB"},"A"→{"Ca"},
          "B"→{"Ba","Cb","b"},"C"→{"cb","b"}|>
Out[1]= <|S→{AB},A→{Ca},B→{Ba,Cb,b},C→{cb,b}|>
```

Our function will require the following arguments: the set `V` defining the vocabulary, the set `T` of terminal symbols, the starting symbol `S`, the association of productions `P`, and the limit on the time, in seconds, `timelimit`. Note that, with the exception of the time limit, this is the same information that makes up a grammar.

Implementation

The function begins by initializing `L` to the empty set and `A` to the list containing the starting string as the sole element. Recall that `L` and `A` will store the words that have been produced and the list of strings with nonterminal symbols that still require processing, respectively.

After the initializations are complete, we begin a `While` loop controlled by the condition that `A` is nonempty. Within the loop, set `curString` (the "current string") equal to the first element of `A` and remove it from `A`.

We need to find all the strings that are directly derivable from `curString`. We do this as follows. First, determine the nonterminal symbols `N` by computing the complement of the terminal symbols `T` relative to the vocabulary `V`. Also initialize a list `D` (for derivations) to the empty list. We will store all the strings derived from `curString` in this list and then later determine which should be added to `L` and which to `A`.

Remember that `curString` is represented as a string. We can use `StringPosition` to determine whether a particular nonterminal ***symbol*** appears in a ***string*** by evaluating the expression `StringPosition[`***string***`,`***symbol***`]`. The output will be a list of lists with each inner list specifying the location of an occurrence of ***symbol***.

```
In[2]:= StringPosition["AbcAb","A"]
Out[2]= {{1,1},{4,4}}
```

Note that the positions are given as ranges. This is because `StringPosition` is often used to find substrings of more than one character, so the function is returning a list of ranges.

```
In[3]:= StringPosition["abccbabca","ab"]
Out[3]= {{1,2},{6,7}}
```

Note that the result when the target string is not found is the empty list.

```
In[4]:= StringPosition["AbcAb","X"]
Out[4]= {}
```

For a given `curString`, we will loop over the nonterminal symbols. For any nonterminal symbols that are found, we look the symbol up in the production table `P`. For each associated production, we perform a substitution.

An example may be helpful to explain this step. Suppose we are processing the string "cBbaBa" as part of the grammar given in Example 12 of Section 13.1.

In[5]:=
```
curString="cBbaBa"
```
Out[5]=
```
cBbaBa
```

First, we check for the nonterminal symbol “A”.

In[6]:=
```
StringPosition[curString,"A"]
```
Out[6]=
```
{}
```

Since “A” is not present, we move on to “B”.

In[7]:=
```
StringPosition[curString,"B"]
```
Out[7]=
```
{{2,2},{5,5}}
```

We see that “B” does occur in the string, so we look up “B” in the production table.

In[8]:=
```
ex12productions["B"]
```
Out[8]=
```
{Ba,Cb,b}
```

We have two occurrences of the nonterminal symbol “B” and three productions. Applying each production to each location will produce six new strings, each of which has one of the occurrences of “B” replaced. We use a `Do` loop with two loop specifications: one over the productions and one over the list of positions. We will be using the `{`***variable***`,`***list***`}` form of the loop specifications. Note that if ***list*** is the empty list, then no iteration will occur.

We apply the derivation with the `StringReplacePart` function. This function requires three arguments. The first is the original string, in this case `curString`. The second argument is the new string, in this case the element from the list of productions. The third argument is the location being replaced, in the same format as output from `StringPosition`. For example, below we replace “xyz” with “d”.

In[9]:=
```
StringReplacePart["abcxyzefg","d",{4,6}]
```
Out[9]=
```
abcdefg
```

These elements combine to give the following code.

In[10]:=
```
Do[Print[StringReplacePart[curString,p,l]],
  {p,ex12productions["B"]},
  {l,StringPosition[curString,"B"]}
]
```
```
cBabaBa

cBbaBaa

cCbbaBa

cBbaCba

cbbaBa

cBbaba
```

In our function, instead of printing the productions, we will Sow them and enclose the loop in a Reap. We will also enclose the loop illustrated above within another Do loop over all of the nonterminal symbols. The resulting list of derived strings is stored as D.

Once curString has been completely processed, we turn to deciding whether each element we placed in D is a word or not. The most straightforward way to approach this is to consider whether or not it contains any nonterminal symbols. We can do this by using StringContainsQ. This has similar syntax to StringPosition, but returns true or false. With a list of strings as the second argument, StringContainsQ results in true if any member of the list appears in the string in the first argument. If the output is true, then the string has nonterminal symbols and need further processing. Otherwise it is a word.

```
In[11]:= StringContainsQ["babaaa",{"S","A","B","C"}]

Out[11]= False
```

Here is the function.

```
In[12]:= formWords[V_,T_,S_,P_,timelimit_]:=
           Module[{L={},A={"S"},N,curString,D,s,d},
             N=Complement[V,T];
             TimeConstrained[
               While[A≠{},
                 curString=A[[1]];
                 A=Delete[A,1];
                 D=Reap[
                   Do[
                     Do[Sow[StringReplacePart[curString,p,l]],
                       {p,P[s]},
                       {l,StringPosition[curString,s]}
                     ]
                   ,{s,N}]][[2,1]];
                 Do[If[StringContainsQ[d,N],
                   AppendTo[A,d],
                   AppendTo[L,d]],
                   {d,D}
                 ]
               ],timelimit];
             DeleteDuplicates[L]
           ]
```

We use our function on the grammar defined by Example 12, up to one tenth of a second.

```
In[13]:= formWords[{"a","b","c","A","B","C","S"},
           {"a","b","c"},"S",ex12productions,.1]
```

Out[13]=

```
{cbab,bab,cbaba,baba,cbacbb,cbabb,bacbb,babb,cbabaa,
  babaa,cbacbba,cbabba,bacbba,babba,cbabaaa,babaaa,
  cbacbbaa,cbabbaa,bacbbaa,babbaa,cbabaaaa,babaaaa,
  cbacbbaaa,cbabbaaa,bacbbaaa,babbaaa,cbabaaaaa,
  babaaaaa,cbacbbaaaa,cbabbaaaa,bacbbaaaa,babbaaaa,
  cbabaaaaaa,babaaaaaa,cbacbbaaaaa,cbabbaaaaa,
  bacbbaaaaa,babbaaaaa,cbabaaaaaaa,babaaaaaaa,
  cbacbbaaaaaa,cbabbaaaaaa,bacbbaaaaaa,babbaaaaaa,
  cbabaaaaaaaa,babaaaaaaaa,cbacbbaaaaaaa,cbabbaaaaaaa,
  bacbbaaaaaaa,babbaaaaaaa,cbabaaaaaaaaa,babaaaaaaaaa,
  cbacbbaaaaaaaa,cbabbaaaaaaaa,bacbbaaaaaaaa,babbaaaaaaaa,
  cbabaaaaaaaaaa,babaaaaaaaaaa,cbacbbaaaaaaaaa,
  cbabbaaaaaaaaa,bacbbaaaaaaaaa,babbaaaaaaaaa,
  cbabaaaaaaaaaaa,babaaaaaaaaaaa,cbacbbaaaaaaaaaa,
  cbabbaaaaaaaaaa,bacbbaaaaaaaaaa,babbaaaaaaaaaa,
  cbabaaaaaaaaaaaa,babaaaaaaaaaaaa,cbacbbaaaaaaaaaaa,
  cbabbaaaaaaaaaaa,bacbbaaaaaaaaaaa,babbaaaaaaaaaaa}
```

13.2 Finite-State Machines with Output

Example 4 in Section 13.2 describes a finite-state machine with five states and with input and output alphabets both equal to $\{0, 1\}$. Example 6 describes how to implement addition of integers using their binary expressions with a finite-state machine with output. Here, we will model those two finite-state machines. We will represent strings in the language as lists.

A First Example

Recall from Definition 1 in Section 13.2 that a finite-state machine consists of six objects: a set S of states, an input alphabet I, an output alphabet O, a transition function f, an output function g, and an initial state s_0.

We will write a function that, given data defining a finite-state machine and an input string, will return the associated output string. Specifically, we will give as an argument to the function a list of members of the input alphabet, and the function will return a list of members of the output alphabet such that the ith element in the output list is the output associated with the ith member of the input list.

Representation

As is typical, we must first describe how we will represent the necessary objects using the Wolfram Language.

For simplicity, the states will be represented by nonnegative integers. For example, in Example 4, the states will be $\{0, 1, 2, 3, 4\}$. We will assume, for the sake of simplicity, that the initial state will always be state 0. Neither S nor s_0 are therefore required as arguments to the function.

The input and output alphabets, I and O can be represented by lists of objects but will not be required arguments to the function, as they can be inferred from the transition and output function. In Example 4, these are both equal to the set $\{0, 1\}$.

The transition function and output function will be represented by a single `Association`. This will have the benefit of making the definition of the functions less cumbersome. The keys will be pairs {***state***, ***input***} where ***state*** is a nonnegative integer and ***input*** will be a member of I. The values of the variable will be pairs {***newState***, ***output***}, where ***newState*** is the state transitioned to and ***output*** is the output corresponding to the original state and the input.

Here is the definition of the transition-output table for Example 4. (Refer to Table 3 of Section 13.2 as the source of the values in the table.)

```
In[14]:= ex4Table=<|{0,0}→{1,1},{0,1}→{3,0},{1,0}→{1,1},
           {1,1}→{2,1},{2,0}→{3,0},{2,1}→{4,0},
           {3,0}→{1,0},{3,1}→{0,0},{4,0}→{3,0},
           {4,1}→{4,0}|>

Out[14]= <|{0,0}→{1,1},{0,1}→{3,0},
           {1,0}→{1,1},{1,1}→{2,1},{2,0}→{3,0},
           {2,1}→{4,0},{3,0}→{1,0},{3,1}→{0,0},
           {4,0}→{3,0},{4,1}→{4,0}|>
```

Observe that the keys for the transition-output table consist of every possible state-input pair.

The Machine Modeling Function

The function we create will accept as arguments the association representing the transition-output table and the input string. It will produce the output string.

The function is fairly straightforward. Initialize the current state of the machine, stored in `curState`, to 0, since we are insisting that 0 represent the starting state. Also initialize the output string, `outString`, to the list of all 0s of the same length as the input list. (It is more efficient, when the length of a list is known in advance, to initialize it to the correct length than it is to build it one element at a time.)

Begin a `For` loop from 1 to the length of the input string. For each index, look up the pair consisting of `curState` and the element in the input string in the transition-output table. The second element in the result is placed in the output string at the correct position, and the first element is used to update `curState`. Once the loop is complete, the output list is returned.

Here is the function.

```
In[15]:= machineWithOutput[transTable_Association,inString_List]:=
          Module[{curState=0,outString,i,newo,news},
            outString=ConstantArray[0,Length[inString]];
            For[i=1,i≤Length[inString],i++,
              {news,newo}=transTable[{curState,inString[[i]]}];
              outString[[i]]=newo;
              curState=news
            ];
            outString
          ]
```

Example 4 asks for the output string when the input is 101011.

```
In[16]:=  machineWithOutput[ex4Table,{1,0,1,0,1,1}]

Out[16]=  {0,0,1,0,0,0}
```

A Finite-State Machine for Addition

Example 6 in Section 13.2 describes how a finite-state machine with output that adds two integers using their binary expansions can be designed. Figure 5 in the main text gives a diagram illustrating the machine.

The input alphabet for this machine are the four bit pairs: 00, 01, 10, and 11. We will represent the pairs as strings. As described by the text, we assume that the most significant bits, x_n and y_n, are both 0.

As an example, consider adding $7 = 0111_2$ and $6 = 0110_2$. We input these two numbers as pairs and in reverse order. Thus the input string will be $\{10, 11, 11, 00\}$.

The transition-output table is obtained from the diagram shown in Figure 5.

```
In[17]:=  addTable=<|{0,"00"}→{0,0},{0,"01"}→{0,1},
            {0,"10"}→{0,1},{0,"11"}→{1,0},{1,"00"}→{0,1},
            {1,"01"}→{1,0},{1,"10"}→{1,0},{1,"11"}→{1,1}|>

Out[17]=  <|{0,00}→{0,0},{0,01}→{0,1},
            {0,10}→{0,1},{0,11}→{1,0},{1,00}→{0,1},
            {1,01}→{1,0},{1,10}→{1,0},{1,11}→{1,1}|>
```

Applying the `machineWithOutput` function to this table and the input produces the sum of the integers.

```
In[18]:=  machineWithOutput[addTable,{"10","11","11","00"}]

Out[18]=  {1,0,1,1}
```

This corresponds to $1101_2 = 13$.

13.3 Finite-State Machines with No Output

In this section, we will see how to represent finite-state automata in the Wolfram Language and to perform language recognition.

Kleene Closure

We begin this section by writing functions to compute the concatenation of two sets of strings and the partial Kleene closure of a set of strings. Again, we will model a string as a list.

Given two lists of strings (themselves represented as lists), we can form all possible concatenations by using `Table` and `Join` to concatenate each pair. In order to simplify the appearance of input, particularly to enter single-element strings as a simple number, this function will wrap any nonlists into a list structure so that single-term strings can be given to the function without braces.

```
In[19]:= setCat[A_,B_]:=Module[{a,b},
           Flatten[Table[Which[
             Head[a]===List&&Head[b]===List,Join[a,b],
             Head[a]===List&&Head[b]=!=List,Join[a,{b}],
             Head[a]=!=List&&Head[b]===List,Join[{a},b],
             Head[a]=!=List&&Head[b]=!=List,Join[{a},{b}]],
             {a,A},{b,B}],1]
         ]
```

Note that `Flatten` is used since `Table` with more than one loop specification produces a nested list. The argument 1 prevents `Flatten` from flattening the list beyond the highest level of nesting.

Applying this function to the sets from Example 1 produces the same output as in the solution to that example.

```
In[20]:= listA={0,{1,1}}

Out[20]= {0,{1,1}}

In[21]:= listB={1,{1,0},{1,1,0}}

Out[21]= {1,{1,0},{1,1,0}}

In[22]:= setCat[listA,listB]

Out[22]= {{0,1},{0,1,0},{0,1,1,0},
           {1,1,1},{1,1,1,0},{1,1,1,1,0}}
```

Given a set A, recall that A^0 is defined to be the set containing only the empty string, and that for $n > 0$, $A^{n+1} = A^n A$. Also recall that the Kleene closure of A is $A^* = \bigcup_{k=0}^{\infty} A^k$. We define the partial Kleene closure to level n by $A^{[n]} = \bigcup_{k=0}^{n} A^k$.

We write a function to produce the powers of A. The function is modeled on the recursive definition given in the text.

```
In[23]:= setPow[A_,k_]:=
           If[k==0,{{}},setCat[setPow[A,k-1],A]];
```

For example, with $B = \{1, 10, 110\}$, we can compute B^3 as follows:

```
In[24]:= setPow[listB,3]

Out[24]= {{1,1,1},{1,1,1,0},{1,1,1,1,0},{1,1,0,1},
           {1,1,0,1,0},{1,1,0,1,1,0},{1,1,1,0,1},
           {1,1,1,0,1,0},{1,1,1,0,1,1,0},{1,0,1,1},
           {1,0,1,1,0},{1,0,1,1,1,0},{1,0,1,0,1},
           {1,0,1,0,1,0},{1,0,1,0,1,1,0},{1,0,1,1,0,1},
           {1,0,1,1,0,1,0},{1,0,1,1,0,1,1,0},
           {1,1,0,1,1},{1,1,0,1,1,0},{1,1,0,1,1,1,0},
```

```
{1,1,0,1,0,1},{1,1,0,1,0,1,0},
{1,1,0,1,0,1,1,0},{1,1,0,1,1,0,1},
{1,1,0,1,1,0,1,0},{1,1,0,1,1,0,1,1,0}}
```

To form the partial Kleene closure $A^{[n]}$, we must find the union of $A^0, A^1, \ldots, A^n$. Iteratively building the A^k while taking unions is more efficient than using `setPow`.

```
In[25]:= kleene[A_,n_]:=Module[{K={{}},x,Ak,i},
           Do[K=Union[K,{{x}}],{x,A}];
           Ak=K;
           For[i=2,i≤n,i++,
             Ak=setCat[Ak,A];
             K=Union[K,Ak]
           ];
           K
         ]
```

We compute the Kleene closure up to level 3 of $\{0, 1\}$.

```
In[26]:= kleene[{0,1},3]

Out[26]= {{},{0},{1},{0,0},{0,1},{1,0},
           {1,1},{0,0,0},{0,0,1},{0,1,0},{0,1,1},
           {1,0,0},{1,0,1},{1,1,0},{1,1,1}}
```

Extended Transition Function for a Finite-State Automaton

Now, we will create a function that serves as the extension of the transition function of a finite-state automaton, as described following Example 4 in Section 13.3 of the textbook.

As in Section 13.2, we will model the transition function as an association. The keys will be the pairs consisting of the current state of the automaton and the input. The corresponding value will be the next state of the automaton.

For example, the transition function of the finite-state automaton M_1 in Example 5 is as follows.

```
In[27]:= ex51Table=<|{0,0}→1,{0,1}→0,{1,0}→1,{1,1}→1|>

Out[27]= <|{0,0}→1,{0,1}→0,{1,0}→1,{1,1}→1|>
```

To model the extended function that takes a pair consisting of a state and a member of the Kleene closure of the alphabet and returns the final state, we write a function, `extendedTransition`. The arguments of this function will be a state number, a list representing the input string, and the transition function.

We will not use the recursive definition provided in the text, but will instead use an iterative approach. Begin by initializing the current state to the input state. Then, loop through the list representing the input string and apply the transition function to update the current state. Once the loop is concluded, return the state.

```
In[28]:= extendedTransition[state_,input_,transFunc_]:=
           Module[{curState=state,i},
             For[i=1,i≤Length[input],i++,
```

```
        curState=transFunc[{curState,input[[i]]}]
      ];
      curState
    ]
```

We can use this function to see that applying the automaton M_1 from Example 5 to the input $\{1, 0, 1, 1, 0\}$ with initial state 0 ends in state 1.

```
In[29]:= extendedTransition[0,{1,0,1,1,0},ex51Table]

Out[29]= 1
```

Language Recognition with Finite-State Automata

Recall that a string x is recognized by a finite-state automaton if the extended transition function applied to the initial state and the string x results in a final state.

We will write a function that, given the transition function for a finite-state automaton with initial state `init`, the set of `final` states, and the string `x`, will return `True` or `False` indicating whether or not the string is recognized by the machine.

The function only needs to apply `extendedTransition` to the state 0, the transition table, and string, and then check to see whether or not the result is in the set of final states.

```
In[30]:= recognizedQ[x_,transFunc_,init_,final_]:=
          Module[{endState},
            endState=extendedTransition[init,x,transFunc];
            MemberQ[final,endState]
          ]
```

The solution to Example 5 indicated that the only strings accepted by M_1 are those consisting of consecutive 1s.

```
In[31]:= recognizedQ[{1,1,1,1,1},ex51Table,0,{0}]

Out[31]= True

In[32]:= recognizedQ[{1,1,0,1},ex51Table,0,{0}]

Out[32]= False
```

Using the `kleene` function from the beginning of this section, we can partially determine the language recognized by a machine.

Given the transition table, the initial state, the set of final states, a set A, and a positive integer n, the following function will calculate the subset of $A^{[n]}$ recognized by the finite-state automaton defined by the transition table and set of final states.

This function operates by brute force; applying `kleene` and then using `recognizedQ` to check each element of $A^{[n]}$.

```
In[33]:= findLanguage[transFunc_,init_,final_,A_,n_]:=
          Module[{An,x,L={}},
            An=kleene[A,n];
```

```
    Do[If[recognizedQ[x,transFunc,init,final],
      AppendTo[L,x]]
      ,{x,An}];
    L
  ]
```

Applying this function to our M_1 machine and $\{0,1\}^{[10]}$, we see that the only strings in that set recognized by the finite-state automaton are those consisting only of 1s.

```
In[34]:=  findLanguage[ex51Table,0,{0},{0,1},10]

Out[34]=  {{},{1},{1,1},{1,1,1},
            {1,1,1,1},{1,1,1,1,1},{1,1,1,1,1,1},
            {1,1,1,1,1,1,1},{1,1,1,1,1,1,1,1},
            {1,1,1,1,1,1,1,1,1},{1,1,1,1,1,1,1,1,1,1}}
```

Nondeterministic Finite-State Automata

We conclude this section with an implementation of the constructive proof of Theorem 1 of Section 13.3. Given a nondeterministic finite-state automaton, our function will produce a deterministic finite-state automaton.

In particular, given the transition function (indexed variable) for a nondeterministic automaton, its input alphabet, its starting state, and its set of final states, the function will produce the transition function for a deterministic automaton, its starting state, and its set of final states.

For a nondeterministic automaton, we will represent the transition function in the same way as for the deterministic automaton earlier, except the values will be sets of states, rather than individual states.

For example, here is the transition function for the nondeterministic automaton described in Example 10, which has final states 0 and 4.

```
In[35]:=  ex10Table=<|{0,0}→{0,2},{0,1}→{1},{1,0}→{3},
            {1,1}→{4},{2,0}→{},{2,1}→{4},{3,0}→{3},
            {3,1}→{},{4,0}→{3},{4,1}→{3}|>

Out[35]=  <|{0,0}→{0,2},{0,1}→{1},{1,0}→{3},
            {1,1}→{4},{2,0}→{},{2,1}→{4},{3,0}→{3},
            {3,1}→{},{4,0}→{3},{4,1}→{3}|>
```

To determine the deterministic automaton's transition table, its starting state, and final states, we follow the proof of Theorem 1. The deterministic automaton's states are sets of states of the nondeterministic automaton.

We begin with the set consisting of the nondeterministic automaton's starting state. This is the starting state for the deterministic automaton. Given any state of the deterministic automaton, and any input, the deterministic transition is the union over all members of the state of the results of applying the nondeterministic automaton's transition with that input value.

In our function, we will create an association. We will also create two sets S and T. The set S will be initialized to the empty set and at the conclusion of the procedure will be the set of all states of the deterministic automaton. The set T will be initialized to $\{\{s_0\}\}$, the set containing the initial state of the deterministic automaton.

As long as T is non-empty, we will move one of its members from T to S and apply the nondeterministic automaton's transition function with all possible input values. The results are the entries in the deterministic transition table and those that are not already members of S are added to T for further processing.

The final states of the deterministic automaton are those states which contain a final state of the nondeterministic automaton. That is, the final states are those whose intersection with the set of the original final states is nonempty. Before exiting, the function calculates the set of final states for the deterministic automaton.

Here is the function. Note that the function returns a list containing the new starting state, the set of final states, and the association representing the deterministic automaton's transition function.

```
In[36]:= makeDeterministic[transFunc_,alphabet_,init_,final_]:=
          Module[{S={},T={{init}},state,i,s,x,newfinal,
             newTable=Association[]},
            While[T≠{},
              state=T[[1]];
              T=Delete[T,1];
              AppendTo[S,state];
              Do[x={};
                Do[x=Union[x,transFunc[{s,i}]],{s,state}];
                x=Union[x];
                newTable[{state,i}]=x;
                If[!MemberQ[S,x],T=Union[T,{x}]]
              ,{i,alphabet}]
            ];
            newfinal={};
            Do[If[Intersection[state,final]≠{},
              newfinal=Union[newfinal,{state}]]
              ,{state,S}];
            {{init},newfinal,newTable}
          ]
```

Applying this function to the Example 10 information produces the following:

```
In[37]:= {ex10Dinit,ex10Dfinal,ex10DTable}=
          makeDeterministic[ex10Table,{0,1},0,{0,4}]

Out[37]= {{0},{{0},{4},{0,2},{1,4},{3,4}},
          <|{{0},0}→{0,2},{{0},1}→{1},{{1},0}→{3},
            {{1},1}→{4},{{3},0}→{3},{{3},1}→{},
            {{},0}→{},{{},1}→{},{{4},0}→{3},{{4},1}→{3},
            {{0,2},0}→{0,2},{{0,2},1}→{1,4},{{1,4},0}→{3},
            {{1,4},1}→{3,4},{{3,4},0}→{3},{{3,4},1}→{3}|>}
```

You can confirm that this agrees with Figure 8 in Section 13.3.

We use the output as the arguments to `findLanguage`.

```
In[38]:= findLanguage[ex10DTable,ex10Dinit,ex10Dfinal,{0,1},10]

Out[38]= {{},{0},{0,0},{0,1},{1,1},{0,0,0},{0,0,1},
           {0,1,1},{0,0,0,0},{0,0,0,1},{0,0,1,1},
           {0,0,0,0,0},{0,0,0,0,1},{0,0,0,1,1},
           {0,0,0,0,0,0},{0,0,0,0,0,1},{0,0,0,0,1,1},
           {0,0,0,0,0,0,0},{0,0,0,0,0,0,1},
           {0,0,0,0,0,1,1},{0,0,0,0,0,0,0,0},
           {0,0,0,0,0,0,0,1},{0,0,0,0,0,0,1,1},
           {0,0,0,0,0,0,0,0,0},{0,0,0,0,0,0,0,0,1},
           {0,0,0,0,0,0,0,1,1},{0,0,0,0,0,0,0,0,0,0},
           {0,0,0,0,0,0,0,0,0,1},{0,0,0,0,0,0,0,0,1,1}}
```

This list of strings suggests that the language recognized by this automaton are those strings consisting of a positive number of 0s followed by no more than two 1s, together with the empty string and the string 11.

13.4 Language Recognition

In this section, we will introduce the Wolfram Language's support for regular expressions for working with strings. We will also develop a function for calculating the concatenation of two nondeterministic automata.

Regular Expressions

In the Wolfram Language, a regular expression is enclosed in the `RegularExpression` head. It is typically used in the second argument to a string information or manipulation function, such as `StringMatchQ`, `StringReplace`, `StringCases`, or `StringSplit`. We will illustrate the Wolfram Language syntax for regular expressions using the `StringMatchQ` function. This function takes a string as the first argument and a regular expression, enclosed in `RegularExpression`, as the second argument. It returns `True` if the regular expression matches the string.

Perhaps the most basic form of a regular expression is the concatenation of elements of the set. For example, "01" is a regular expression. This expression matches itself, of course.

```
In[39]:= StringMatchQ["01",RegularExpression["01"]]

Out[39]= True
```

The output indicates that yes, the string "01" matches the regular expression "01".

Kleene Closure

The asterisk is a symbol used in a regular expression to represent the Kleene closure.

For example, the regular expression "10*" will match a 1 followed by any number of 0s.

```
In[40]:= StringMatchQ["10000000",RegularExpression["10*"]]

Out[40]= True
```

```
In[41]:= StringMatchQ["1",RegularExpression["10*"]]
Out[41]= True
In[42]:= StringMatchQ["0111000",RegularExpression["10*"]]
Out[42]= False
```

As in the text, parentheses can be used to group symbols. For example “(10)*” matches any number of copies of “10”.

```
In[43]:= StringMatchQ["1010101010101010",RegularExpression["(10)*"]]
Out[43]= True
In[44]:= StringMatchQ["101010101",RegularExpression["(10)*"]]
Out[44]= False
```

The Wolfram Language, like most languages that support regular expressions, also recognizes “+“ and “?”. These are used like “*” but with different meaning. The expression “A+“ is used to match one or more copies of “A”. Essentially, it is the Kleene closure minus the empty string. For example, “1*0+“ matches any number of 1s followed by at least one 0.

```
In[45]:= StringMatchQ["1111000",RegularExpression["1*0+"]]
Out[45]= True
In[46]:= StringMatchQ["00",RegularExpression["1*0+"]]
Out[46]= True
In[47]:= StringMatchQ["111",RegularExpression["1*0+"]]
Out[47]= False
```

The “A?” expression is used to match 0 or 1 copies of “A”. For example, “1*0?” matches any number of 1s which may be followed by at most one 0.

```
In[48]:= StringMatchQ["111111",RegularExpression["1*0?"]]
Out[48]= True
In[49]:= StringMatchQ["1111110",RegularExpression["1*0?"]]
Out[49]= True
In[50]:= StringMatchQ["11111100",RegularExpression["1*0?"]]
Out[50]= False
```

Union

To represent union, the vertical line is used. A “|” placed between two expressions will match either of them. The “|” can take the place of the “∪” symbol in an expression such as “0(0∪1)*”.

```
In[51]:= StringMatchQ["011010",RegularExpression["0(0|1)*"]]
Out[51]= True
In[52]:= StringMatchQ["1011010",RegularExpression["0(0|1)*"]]
Out[52]= False
```

This can also be done in more complicated expressions. For example, "2((10)*∪(01)*)2" describes the set of strings beginning and ending with 2s with an alternating sequence of 0s and 1s in between.

```
In[53]:= StringMatchQ["21010102",
           RegularExpression["2((10)*|(01)*)2"]]
Out[53]= True
In[54]:= StringMatchQ["201012",
           RegularExpression["2((10)*|(01)*)2"]]
Out[54]= True
In[55]:= StringMatchQ["210012",
           RegularExpression["2((10)*|(01)*)2"]]
Out[55]= False
```

In some circumstances, union can be replaced by character classes. By placing characters within a pair of brackets, you indicate that any of the characters inside the brackets are allowed. For example, "0(0∪1)*" can be expressed as follows:

```
In[56]:= StringMatchQ["011010",RegularExpression["0[01]*"]]
Out[56]= True
```

Note that this is only allowed when the options are single characters.

Character classes can also be used to specify a range of characters with a hyphen. For example, "(0∪1∪2∪3∪4)*" can be specified as follows.

```
In[57]:= StringMatchQ["4213442101",RegularExpression["[0-4]*"]]
Out[57]= True
```

Character classes can be complemented. By beginning a character class with a caret, you indicate that any character other than those specified are allowed. For example, in the following, the regular expression matches all strings beginning with 1, ending with 0, and which include no other 1s nor 0s.

```
In[58]:= StringMatchQ["169jwq0",RegularExpression["1[^01]*0"]]
Out[58]= True
In[59]:= StringMatchQ["169j1wq0",RegularExpression["1[^01]*0"]]
Out[59]= False
```

There are also several defined character classes: you enter “\\d” for a digit, “\\D” for a nondigit, “\\s” for space, including newline and tab, “\\S” for any non-whitespace character, “\\w” for a word character (letters, digits, and underscores), and “\\W” for a nonword character.

```
In[60]:= StringMatchQ["126qb",RegularExpression["\\d*\\w\\w"]]
Out[60]= True
In[61]:= StringMatchQ["b32xy",RegularExpression["\\d*\\w\\w"]]
Out[61]= False
```

The special character dot, “.”, is used to match any character. For example, “1...0” will match any string beginning with a 1, followed by any three characters and ending with a 0.

```
In[62]:= StringMatchQ["12340",RegularExpression["1...0"]]
Out[62]= True
In[63]:= StringMatchQ["1230",RegularExpression["1...0"]]
Out[63]= False
In[64]:= StringMatchQ["1234567890",RegularExpression["1...0"]]
Out[64]= False
```

Regular expressions in the Wolfram Language are extremely flexible. The interested reader is referred to the tutorial page on regular expressions for more information.

Concatenation of Automata

We will write a function that concatenates two nondeterministic finite-state automata, as described in the proof of Theorem 1 of the text.

Two Automata

We begin by defining two example automata that our function will concatenate.

The first automata is the result of Example 3, for recognizing “1*∪01”. Our implementation is based on the simple form shown in Figure3b.

Note that the diagram in the text omits the results of transitioning from certain states via certain input values. For example, it does not show the result of the transition from state s_1 with input 0. This makes for a simpler and cleaner diagram, but the transition table will need to include this information. It will be assumed that all such omissions correspond to a transition to the state { }.

Here is the transition table corresponding to the automaton shown in Figure 3b.

```
In[65]:= atable=<|{0,0}→{2},{0,1}→{1},{1,0}→{},
           {1,1}→{1},{2,0}→{},{2,1}→{3},{3,0}→{},
           {3,1}→{}|>
Out[65]= <|{0,0}→{2},{0,1}→{1},{1,0}→{},{1,1}→{1},
           {2,0}→{},{2,1}→{3},{3,0}→{},{3,1}→{}|>
```

The final states for this automaton are $\{0, 1, 3\}$. We can confirm that it recognizes "1*∪01" by applying `makeDeterministic` and `findLanguage`.

```
In[66]:= {aDinit,aDfinal,aDtable}=
           makeDeterministic[atable,{0,1},0,{0,1,3}]

Out[66]= {{0},{{0},{1},{3}},
           <|{{0},0}→{2},{{0},1}→{1},{{1},0}→{},
             {{1},1}→{1},{{},0}→{},{{},1}→{},{{2},0}→{},
             {{2},1}→{3},{{3},0}→{},{{3},1}→{}|>}

In[67]:= findLanguage[aDtable,aDinit,aDfinal,{0,1},10]

Out[67]= {{},{1},{0,1},{1,1},{1,1,1},
           {1,1,1,1},{1,1,1,1,1},{1,1,1,1,1,1},
           {1,1,1,1,1,1,1},{1,1,1,1,1,1,1,1},
           {1,1,1,1,1,1,1,1,1},{1,1,1,1,1,1,1,1,1,1}}
```

As you can see, the language recognized by this machine includes the string "01" as well as "1*".

The second automaton we create will recognize the language "101".

```
In[68]:= btable=<|{0,0}→{},{0,1}→{1},{1,0}→{2},
           {1,1}→{},{2,0}→{},{2,1}→{3},{3,0}→{},
           {3,1}→{}|>

Out[68]= <|{0,0}→{},{0,1}→{1},{1,0}→{2},{1,1}→{},
           {2,0}→{},{2,1}→{3},{3,0}→{},{3,1}→{}|>
```

The only final state is 3. We confirm that this models that machine that recognizes 101.

```
In[69]:= {bDinit,bDfinal,bDtable}=
           makeDeterministic[btable,{0,1},0,{3}]

Out[69]= {{0},{{3}},<|{{0},0}→{},{{0},1}→{1},{{},0}→{},
           {{},1}→{},{{1},0}→{2},{{1},1}→{},{{2},0}→{},
           {{2},1}→{3},{{3},0}→{},{{3},1}→{}|>}

In[70]:= findLanguage[bDtable,bDinit,bDfinal,{0,1},10]

Out[70]= {{1,0,1}}
```

Concatenating the Machines

Our concatenation function will require the following arguments, for both machines: the transition table, the starting state, and the final states. It will also require that the two machines have a common input alphabet but that alphabet does not need to be an argument.

Recall the following elements of the construction of the concatenation as described in the proof of Theorem 1 of Section 13.4:

1. The states of the concatenation is the union of the states of the original machines, which are assumed to be disjoint.
2. The starting state of the concatenation is the starting state of the first of the two machines.

3. The final states of the concatenation include the set of final states of the second machine.
4. The final states of the concatenation also include the starting state if the empty string is a member of both languages.
5. All transitions of the original machines are transitions of the new machine.
6. Additionally, for every transition in the first machine leading to a final state, we add a transition in the concatenation to the starting state of the second machine.
7. Finally, if the starting state of the first machine is final, then for every transition from the starting state of the second machine, we add a transition from the starting state of the new machine.

The assumption that the states of the original two machines are disjoint means that we will need to make them so. There are a variety of ways in which we could do this. Since we assume that states are designated by nonnegative integers, we can make the states distinct by multiplying each state by 10 and adding 1 if it is in the first machine and 2 if it is in the second machine.

Therefore, the starting state of the concatenation is found by $10 \cdot s_A + 1$ where s_A is the starting state of the first machine. In our example, this will be equal to $10 \cdot 0 + 1 = 1$.

Next, we find the final states of the concatenation. Let `finalA` and `finalB` be the sets of final states for the original two machines. According to point 3 above, the final states of the concatenated machine include the final states of the second machine. We only need to update the names.

The final states of the machines we defined above are as follows:

```
In[71]:= finalA={0,1,3}
Out[71]= {0,1,3}
In[72]:= finalB={3}
Out[72]= {3}
```

We can obtain the final states of the concatenation by applying the function $s \rightarrow 10s + 2$ to the set of final states of the second machine.

```
In[73]:= Map[(10*#+2)&,finalB]
Out[73]= {32}
```

Item 4 asserts that the starting state of the concatenated machine is a final state if and only if the empty string is a member of both languages. Another way to put this is that the starting state of the concatenated machine is a final state when both of the original machines have their own starting states as final states. This is not the case in our example. We will include this possibility in our general function by checking to see if the starting states are members of the sets of final states.

To form the transitions of the new machine, we begin by making the original two transition functions disjoint. The function `KeyValueMap` accepts a function of two variables and an association and applies the function to the keys and corresponding values. Applying `KeyValueMap` with the undefined symbol `f` as the function illustrates the result.

```
In[74]:= KeyValueMap[f,<|1→"a",2→"b",3→"c"|>]
Out[74]= {f[1,a],f[2,b],f[3,c]}
```

Note that `#1` refers to the key and `#2` refers to the value. For an association representing a transition function, the keys will be pairs consisting of a state and an input, so `#1[[1]]` refers to the state. As discussed above, we will multiply the state by 10 and add 1 to obtain the new state names for the first machine. The values in the association are sets of states. We can make use of the fact that arithmetic automatically threads over lists of change all of the states without individually accessing the elements. For example:

```
In[75]:= 10*{1,4}+1

Out[75]= {11,41}
```

The result of `KeyValueMap` is a list, so we also need to apply the `Association` head. The following, then, produces the transition rules for the first machine on the disjoint state names.

```
In[76]:= Association@@
           KeyValueMap[{10*#1[[1]]+1,#1[[2]]}→10*#2+1&,atable]

Out[76]= <|{1,0}→{21},{1,1}→{11},{11,0}→{},{11,1}→{11},
           {21,0}→{},{21,1}→{31},{31,0}→{},{31,1}→{}|>
```

For the second machine, we do the same thing except adding 2 instead of 1.

```
In[77]:= Association@@
           KeyValueMap[{10*#1[[1]]+2,#1[[2]]}→10*#2+2&,btable]

Out[77]= <|{2,0}→{},{2,1}→{12},{12,0}→{22},{12,1}→{},
           {22,0}→{},{22,1}→{32},{32,0}→{},{32,1}→{}|>
```

Thus, we begin forming the transition function for the combined machine by joining those two associations.

```
In[78]:= abtable=
           Join[Association@@
             KeyValueMap[{10*#1[[1]]+1,#1[[2]]}→10*#2+1&,
               atable],
            Association@@
             KeyValueMap[{10*#1[[1]]+2,#1[[2]]}→10*#2+2&,
               btable]]

Out[78]= <|{1,0}→{21},{1,1}→{11},{11,0}→{},{11,1}→{11},
           {21,0}→{},{21,1}→{31},{31,0}→{},{31,1}→{},
           {2,0}→{},{2,1}→{12},{12,0}→{22},{12,1}→{},
           {22,0}→{},{22,1}→{32},{32,0}→{},{32,1}→{}|>
```

Next, we must add transitions between the two components. As item 6 instructs, for each transition in the first of the two machines that leads to a final state, we must add a transition in the concatenated machine to the starting state of the second machine.

We loop through the keys of `atable`, checking whether the value contains any states that are final for machine A. If so, we will add the transition to state 2 (the name of the starting state in the second machine in the concatenation). (Note that we must update the value in the `abtable` rather than replace it.)

```
In[79]:= Do[If[Intersection[atable[i],finalA]≠{},
           abtable[{10*i[[1]]+1,i[[2]]}]=
             Union[abtable[{10*i[[1]]+1,i[[2]]}],{2}]]
           ,{i,Keys[atable]}]
```

We can see that this has added transitions that lead to state 2.

```
In[80]:= abtable

Out[80]= <|{1,0}→{21},{1,1}→{2,11},{11,0}→{},
           {11,1}→{2,11},{21,0}→{},{21,1}→{2,31},
           {31,0}→{},{31,1}→{},{2,0}→{},{2,1}→{12},
           {12,0}→{22},{12,1}→{},{22,0}→{},
           {22,1}→{32},{32,0}→{},{32,1}→{}|>
```

Finally, since the starting state of the first machine is final, we must add transitions from the starting state of the concatenated machine for each of the transitions from the starting state of the second machine. The starting state of the second machine in this example is 0, and the starting state of the concatenation is 1.

```
In[81]:= Do[If[i[[1]]==0,
           abtable[{1,i[[2]]}]=
             Union[abtable[{1,i[[2]]}],Map[(10*#+2)&,btable[i]]]]
           ,{i,Keys[btable]}]
```

Inspect the table again.

```
In[82]:= abtable

Out[82]= <|{1,0}→{21},{1,1}→{2,11,12},{11,0}→{},
         {11,1}→{2,11},{21,0}→{},{21,1}→{2,31},
         {31,0}→{},{31,1}→{},{2,0}→{},{2,1}→{12},
         {12,0}→{22},{12,1}→{},{22,0}→{},
         {22,1}→{32},{32,0}→{},{32,1}→{}|>
```

Note that this modified the value associated to key $\{1, 1\}$. (Recall that state 1 is the starting state for the combined machine.) Before, $\{1, 1\}$ was associated with $\{2, 11\}$, the starting state of the second machine and state 1 of the first machine. Now, the value for $\{1, 1\}$ also includes 12, state 1 of the second machine.

That $\{1, 1\}$ is associated with $\{2, 11, 12\}$ means that from the starting state of the concatenation and input 1, there are three options. First, going to state 2, the starting state of the second machine, corresponds to recognizing the string 1 followed by a string recognized by the second machine. Second, going to state 11, state 1 of the first machine, corresponds to building a string of all 1s, which is recognized by the first machine. Third, going to state 12, state 1 of the second machine, corresponds to the first machine contributing the empty string followed by 1 as the first character of a string recognized by the second machine.

Implementation as a Function

Here is the complete function based on the example above.

In[83]:=
```
catAutomata[atable_,astart_,afinal_,btable_,bstart_,
  bfinal_]:=Module[{abstart,abfinal,abtable,i},
  abstart=10*astart+1;
  abfinal=Map[(10*#+2)&,bfinal];
  If[MemberQ[afinal,astart]&&MemberQ[bfinal,bstart],
    abfinal=Union[abfinal,{abstart}]];
  abtable=
    Join[Association@@
      KeyValueMap[{10*#1[[1]]+1,#1[[2]]}→10*#2+1&,
        atable],Association@@
      KeyValueMap[{10*#1[[1]]+2,#1[[2]]}→10*#2+2&,
        btable]];
  Do[If[Intersection[atable[i],afinal]≠{},
    abtable[{10*i[[1]]+1,i[[2]]}]=
      Union[abtable[{10*i[[1]]+1,i[[2]]}],{2}]]
    ,{i,Keys[atable]}];
  If[MemberQ[afinal,astart],
    Do[If[i[[1]]==0,
      abtable[{1,i[[2]]}]=
        Union[abtable[{1,i[[2]]}],
          Map[(10*#+2)&,btable[i]]]]
      ,{i,Keys[btable]}]
  ];
  {abstart,abfinal,abtable}
  ]
```

Applying this to our examples and passing the results on to `makeDeterministic` and `findLanguage` shows us that the result does indeed recognize “(1*∪01)101”.

In[84]:=
```
{cstart,cfinal,ctable}=
  catAutomata[atable,0,{0,1,3},btable,0,{3}]
```

Out[84]=
```
{1,{32},<|{1,0}→{21},{1,1}→{2,11,12},
  {11,0}→{},{11,1}→{2,11},{21,0}→{},
  {21,1}→{2,31},{31,0}→{},{31,1}→{},
  {2,0}→{},{2,1}→{12},{12,0}→{22},{12,1}→{},
  {22,0}→{},{22,1}→{32},{32,0}→{},{32,1}→{}|>}
```

In[85]:=
```
{cDstart,cDfinal,cDtable}=
  makeDeterministic[ctable,{0,1},cstart,cfinal]
```

Out[85]=
```
{{1},{{32}},<|{{1},0}→{21},{{1},1}→{2,11,12},
  {{21},0}→{},{{21},1}→{2,31},{{},0}→{},
  {{},1}→{},{{2,31},0}→{},{{2,31},1}→{12},
  {{12},0}→{22},{{12},1}→{},{{22},0}→{},
```

```
  {{22},1}→{32},{{32},0}→{},{{32},1}→{},
  {{2,11,12},0}→{22},{{2,11,12},1}→{2,11,12}|>}
```

```
In[86]:= findLanguage[cDtable,cDstart,cDfinal,{0,1},10]
```

```
Out[86]= {{1,0,1},{1,1,0,1},{0,1,1,0,1},
  {1,1,1,0,1},{1,1,1,1,0,1},
  {1,1,1,1,1,0,1},{1,1,1,1,1,1,0,1},
  {1,1,1,1,1,1,1,0,1},{1,1,1,1,1,1,1,1,0,1}}
```

13.5 Turing Machines

In this section, we will explore the Wolfram Language's `TuringMachine` function. We will then create our own model of a Turing machine to help you better understand this important concept in detail.

`TuringMachine`

To illustrate the Wolfram Language's built-in function, we will use Example 1 from Section 13.5. This Turing machine is defined by seven tuples: $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, (s_0, B, s_3, B, R), $(s_1, 0, s_0, 0, R)$, $(s_1, 1, s_2, 0, L)$, (s_1, B, s_3, B, R), and $(s_2, 1, s_3, 0, R)$.

The first argument of `TuringMachine` will be that data, but in the form of `Rule`s (`->`) of the form `{`***state***`,`***entry***`}->{`***newstate***`,`***newentry***`,`***move***`}`, where ***state*** and ***entry*** are the current state of the machine and the value seen by the head and ***newstate*** and ***newentry*** are the next state and the value to be written on the tape. The ***move*** is an integer representing how the head is to move, with +1 representing right and −1 left. Thus, the machine of Example 1 is described by the set of rules below.

```
In[87]:= example1Rules={{0,0}→{0,0,1},{0,1}→{1,1,1},{0,""}→{3,"",1},
  {1,0}→{0,0,1},{1,1}→{2,0,-1},{1,""}→{3,"",1},{2,1}→{3,0,1}}
```

```
Out[87]= {{0,0}→{0,0,1},{0,1}→{1,1,1},
  {0,}→{3,,1},{1,0}→{0,0,1},{1,1}→{2,0,-1},
  {1,}→{3,,1},{2,1}→{3,0,1}}
```

The second argument to `TuringMachine` specifies the initial condition of the tape. It is a list containing two members. The first element of the initial condition list will be the initial state of the machine. The second element of the initial condition is a list with two members, the first being a list representing a finite portion of the tape and the second specifying the value appearing at every position of the infinite tape outside the finite area.

In our example, the machine will begin in state 0. The tape is initially {0,1,0,1,1,0} with blanks outside that range. Therefore, the second argument to `TuringMachine` will be

```
In[88]:= example1Init={0,{{0,1,0,1,1,0},""}}
```

```
Out[88]= {0,{{0,1,0,1,1,0},}}
```

Applying `TuringMachine` to these two elements produces the following output:

```
In[89]:= TuringMachine[example1Rules,example1Init]
```

```
Out[89]= {{0,2,1},{{0,1,0,1,1,0},}}
```

This output represents the result of one step of the Turing machine. It is of the form

{{***state***,***pos***,***distance***},{***tape***,***rest***}}

where ***state*** is the new state of the machine, ***tape*** is the current state of the finite segment of tape with ***rest*** filling the rest of the infinite tape, ***pos*** is the position of the head relative to the list ***tape***, and ***distance*** is how far the head has moved from its starting position. So, the output above indicates that the machine is still in state 0 but has moved one position to the left.

Note that you can initialize a machine with a position argument similar to this output, but without the distance. The following will start the machine at the final 1 of the tape:

```
In[90]:= TuringMachine[example1Rules,
           {{0,5},{{0,1,0,1,1,0},""}}]

Out[90]= {{1,6,1},{{0,1,0,1,1,0},}}
```

Note that the machine has moved one position to the left and changed to state 1.

An optional third argument allows you run the machine more than one step.

```
In[91]:= TuringMachine[example1Rules,example1Init,5]

Out[91]= {{{0,1,0},{0,1,0,1,1,0}},{{0,2,1},{0,1,0,1,1,0}},
           {{1,3,2},{0,1,0,1,1,0}},{{0,4,3},{0,1,0,1,1,0}},
           {{1,5,4},{0,1,0,1,1,0}},{{2,4,3},{0,1,0,1,0,0}}}
```

Note that the output is a list of lists representing each step along the way. The final element indicates that after 5 steps, the machine is in state 2 at position 4.

For a machine with a terminal state, we can run it to completion with a `While` loop as below. Note that `TuringMachine` allows its initialization argument to include the distance parameter, so that we can feed its output back to it.

```
In[92]:= machinestate={{0,1},{{0,1,0,1,1,0},""}};
         While[machinestate[[1,1]]≠3,
           machinestate=TuringMachine[example1Rules,machinestate]
         ];
         machinestate

Out[94]= {{3,5,4},{{0,1,0,0,0,0},}}
```

Note that this agrees with the result of Example 1 in the textbook.

Creating a Turing Machine Function

In our model, the tape will be represented by a list, with the assumption that all elements to the left and right of the bounds of the list are blanks. The blank symbol will be represented by the symbol `B` and left and right by the symbols `L` and `R`. We ensure that these have not been assigned values by applying `Clear`.

```
In[95]:= Clear[B,L,R]
```

The Partial Function

The text uses the convention that the partial function that controls the operation of the Turing machine is defined by a set of five-tuples. It will be more convenient for our functions to represent the partial function as an association whose keys are pairs $\{s, x\}$ and whose values are triples $\{s', x', d\}$.

We create a function that will transform the set of 5-tuples representation into the association representation.

In[96]:=
```
tuplesToIndexed[S_]:=Module[{x},
  Association@@Table[x[[{1,2}]]→x[[{3,4,5}]],{x,S}]
]
```

Applying this function to the set of tuples given in Example 1 provides us with an example of a partial function to work with.

In[97]:=
```
ex1=tuplesToIndexed[
  {{0,0,0,0,R},{0,1,1,1,R},{0,B,3,B,R},
  {1,0,0,0,R},{1,1,2,0,L},{1,B,3,B,R},
  {2,1,3,0,R}}]
```

Out[97]=
```
<|{0,0}→{0,0,R},{0,1}→{1,1,R},
  {0,B}→{3,B,R},{1,0}→{0,0,R},{1,1}→{2,0,L},
  {1,B}→{3,B,R},{2,1}→{3,0,R}|>
```

Note that `B`, `L`, and `R` must all be unassigned symbols, otherwise they will be evaluated within the set of 5-tuples and will produce unexpected results.

The Turing Machine Function

Our Turing machine function will accept as input an association representing the partial function, a list representing the status of the tape before running the machine, and the initial state. It will return the final tape and the final state.

When the function begins, we initialize the symbol `pos` to 1, indicating that the control head is positioned at the leftmost element in the tape. We set the `state` of the machine to the initial state and copy the `tape` from the argument as well. We also compute the `domain` of the partial function using the `getIndices` function we created in the previous section. This will make it easier to check whether we have reached a halt.

The main work of the function will take place within a `While` loop controlled by the condition that the domain of the function includes the pair consisting of the current state and the entry on the tape at the current position.

Within the loop, we first obtain the values of the new state, new tape entry, and direction from the partial function. We then set the `state` to the new state, change the entry on the `tape`, and update the position `pos`. Note that when changing the position of the control head, we must take care not to exceed the bounds of the list representing the tape. If the previous position was location 1 in the list and the direction is left, then instead of changing the position, we extend the list by adding a blank on the left with the `PrependTo` function. On the other hand, if the previous position was the right end of the tape and the direction is right, then we increase the position and extend the tape to the right via `AppendTo`.

Here is the function.

```
In[98]:= Turing[f_,t_,init_]:=
           Module[{pos=1,state=init,tape=t,domain,y},
             domain=Keys[f];
             While[MemberQ[domain,{state,tape[[pos]]}],
               y=f[{state,tape[[pos]]}];
               state=y[[1]];
               tape[[pos]]=y[[2]];
               Which[pos==1&&y[[3]]===L,PrependTo[tape,B],
                 pos==Length[tape]&&y[[3]]===R,AppendTo[tape,B];
                 pos++,
                 y[[3]]===L,pos--,
                 y[[3]]===R,pos++];
             ];
             {tape,state}
           ]
```

We use the function to run the Turing machine from Example 1 on the tape shown in Figure 2a.

```
In[99]:= Turing[ex1,{0,1,0,1,1,0},0]

Out[99]= {{0,1,0,0,0,0},3}
```

Observe that this agrees with Figure 2 from Section 13.5 in the textbook.

We create a verbose version of this function as well. The operation of the verbose version is identical to `Turing`, but it displays the status of the machine at every step.

```
In[100]:= verboseTuring[f_,t_,init_]:=
            Module[{pos=1,state=init,tape=t,domain,y,
               displayTape},
              domain=Keys[f];
              displayTape=t;
              displayTape[[pos]]="→"<>ToString[tape[[pos]]];
              Print[displayTape,state];
              While[MemberQ[domain,{state,tape[[pos]]}],
                y=f[{state,tape[[pos]]}];
                state=y[[1]];
                tape[[pos]]=y[[2]];
                Which[pos==1&&y[[3]]===L,PrependTo[tape,B],
                  pos==Length[tape]&&y[[3]]===R,AppendTo[tape,B];
                  pos++,
                  y[[3]]===L,pos--,
                  y[[3]]===R,pos++];
                displayTape=tape;
                displayTape[[pos]]="→"<>ToString[tape[[pos]]];
                Print[displayTape,state];
```

```
    ];
    {tape,state}
  ]
```

In[101]:=
```
verboseTuring[ex1,{0,1,0,1,1,0},0]
```

```
{→0,1,0,1,1,0}0
{0,→1,0,1,1,0}0
{0,1,→0,1,1,0}1
{0,1,0,→1,1,0}0
{0,1,0,1,→1,0}1
{0,1,0,→1,0,0}2
{0,1,0,0,→0,0}3
```

Out[101]=
```
{{0,1,0,0,0,0},3}
```

Applications of Turing Machines

We now apply our Turing machine function to two applications: recognizing strings in a language and computing functions.

Recognizing Sets

We will implement the Turing machine for recognizing $\{0^n 1^n \mid n \geq 1\}$.

The partial function was given in the solution to Example 3. To be safe, we again clear all the symbols used.

In[102]:=
```
Clear[M,B,L,R];
ex3=tuplesToIndexed[
  {{0,0,1,M,R},{1,0,1,0,R},{1,1,1,1,R},
    {1,M,2,M,L},{1,B,2,B,L},{2,1,3,M,L},
    {3,1,3,1,L},{3,0,4,0,L},{3,M,5,M,R},
    {4,0,4,0,L},{4,M,0,M,R},{5,M,6,M,R}}]
```

Out[103]=
```
<|{0,0}→{1,M,R},{1,0}→{1,0,R},{1,1}→{1,1,R},
  {1,M}→{2,M,L},{1,B}→{2,B,L},{2,1}→{3,M,L},
  {3,1}→{3,1,L},{3,0}→{4,0,L},{3,M}→{5,M,R},
  {4,0}→{4,0,L},{4,M}→{0,M,R},{5,M}→{6,M,R}|>
```

To determine whether or not a string is in the language, we only have to apply the Turing machine to the string and check the exit state.

In[104]:=
```
Turing[ex3,{0,0,0,0,1,1,1,1},0]
```

Out[104]=
```
{{M,M,M,M,M,M,M,M,B},6}
```

The fact that the machine halted in state 6, the final state, indicates that it recognizes the string. On the other hand,

In[105]:=
```
Turing[ex3,{0,0,0,1,1},0]
```
Out[105]=
```
{{M,M,M,M,M,B},2}
```

halted in state 2, indicating that the string is not in the language.

Adding Nonnegative Integers

Example 4 describes how to use Turing machines to perform addition.

The machine is described by the following tuples:

In[106]:=
```
adder=tuplesToIndexed[
  {{0,1,1,B,R},{1,"*",3,B,R},{1,1,2,B,R},
    {2,1,2,1,R},{2,"*",3,1,R}}]
```
Out[106]=
```
<|{0,1}→{1,B,R},{1,*}→{3,B,R},
  {1,1}→{2,B,R},{2,1}→{2,1,R},{2,*}→{3,1,R}|>
```

We add two numbers a and b by using the unary representation tape consisting of $a + 1$ ones followed by an asterisk and then $b + 1$ ones. We create a small function to create the tape given a and b.

In[107]:=
```
unaryTape[a_,b_]:=
  Join[ConstantArray[1,{a+1}],{"*"},
    ConstantArray[1,{b+1}]]
```

The tape used to add 3 and 4 is shown below.

In[108]:=
```
unaryTape[3,4]
```
Out[108]=
```
{1,1,1,1,*,1,1,1,1,1}
```

Performing addition is accomplished by applying `Turing` to the transition function and the tape.

In[109]:=
```
Turing[adder,unaryTape[3,4],0]
```
Out[109]=
```
{{B,B,1,1,1,1,1,1,1,1},3}
```

You can see that this contains a string of 8 ones, indicating a result of 7.

Using the verbose form of `Turing`, you can see how the Turing adder operates.

In[110]:=
```
verboseTuring[adder,unaryTape[3,4],0]
```
```
{→1,1,1,1,*,1,1,1,1,1}0

{B,→1,1,1,*,1,1,1,1,1}1

{B,B,→1,1,*,1,1,1,1,1}2

{B,B,1,→1,*,1,1,1,1,1}2

{B,B,1,1,→*,1,1,1,1,1}2

{B,B,1,1,1,→1,1,1,1,1}3
```
Out[110]=
```
{{B,B,1,1,1,1,1,1,1,1},3}
```

Solutions to Computer Projects and Computations and Explorations

Computer Projects 8

Given the state table of a nondeterministic finite-state automaton and a string, decide whether this string is recognized by the automaton.

Solution: One solution to this problem, the solution used earlier in this chapter, is to find the deterministic automaton that recognizes the same language and use it to decide whether the string is recognized or not. This is what we have been doing when we apply `findLanguage` to the result of `makeDeterministic`.

Here, we will take a direct approach. For deterministic machines, we created two functions: `extendedTransition` and `recognizedQ`. The `recognizedQ` function merely called `extendedTransition` and checked whether the result was a final state or not. The `extendedTransition` function took a state, an input string, and a transition table, and determined the state of the machine following the processing of the input.

Our approach for nondeterministic machines will be similar. We will create two functions: `extendedTransitionND` and `recognizedNDQ`. The main difference between the deterministic machines and nondeterministic machines is that with nondeterministic machines, given the initial state and an input, we do not know the next state. Instead, there is a set of possible states.

`extendedTransitionND` will therefore take a set of possible states, an input, and a transition table as its arguments. For each member of the input string, it will apply the transition table to each of the possible states, producing a new set of possible states. It will return the set of possible states after processing each element in the input string.

```
In[111]:= extendedTransitionND[states_,input_,transFunc_]:=
            Module[{curStates,i,s,newStates},
              curStates=states;
              For[i=1,i≤Length[input],i++,
                newStates={};
                Do[newStates=Union[newStates,
                  transFunc[{s,input[[i]]}]]
                 ,{s,curStates}];
                curStates=newStates
              ];
              curStates
            ]
```

A nondeterministic machine recognizes a string if the result of running the machine from the starting state with the input string results in a set of possible ending states that includes at least one final state. We write `recognizedNDQ` to call `extendedTransitionND` and check to see if the result intersects the set of final states.

In[112]:=
```
recognizedNDQ[x_,transFunc_,init_,final_]:=
  Module[{endStates},
    endStates=extendedTransitionND[{init},x,transFunc];
    Intersection[endStates,final]≠{}
  ]
```

With `recognizedNDQ` in hand, we can create `findLanguageND`. This is effectively identical to `findLanguage`.

In[113]:=
```
findLanguageND[transFunc_,init_,final_,A_,n_]:=
  Module[{An,x,L},
    An=kleene[A,n];
    L={};
    Do[If[recognizedNDQ[x,transFunc,init,final],
      L=Union[L,{x}]]
     ,{x,An}];
    L
  ]
```

Applying this function to the machine defined by transition function `ctable`, starting state 1, final state $\{32\}$, and alphabet $\{0,1\}$, which was produced by `catAutomata`, we see that the result is the same as when we applied `findLanguage` and `makeDeterministic` in Section 13.4.

In[114]:=
```
findLanguageND[ctable,1,{32},{0,1},10]
```

Out[114]=
```
{{1,0,1},{1,1,0,1},{0,1,1,0,1},
  {1,1,1,0,1},{1,1,1,1,0,1},
  {1,1,1,1,1,0,1},{1,1,1,1,1,1,0,1},
  {1,1,1,1,1,1,1,0,1},{1,1,1,1,1,1,1,1,0,1}}
```

Computations and Explorations 1

Solve the busy beaver problem for two states by testing all possible Turing machines with two states and alphabet $\{1,B\}$.

Solution: The busy beaver problem, described in the preface to Exercise 31 in Section 13.5, asks: what is the maximum number of ones that a Turing machine with n states on the alphabet $\{1,B\}$ may print on an initially blank tape? This exercise asks us to solve the busy beaver problem with a brute force approach for $n=2$.

We will construct all possible Turing machines on 2 states with the given alphabet. For each possible Turing machine, we will allow it to run until either it halts, or until it has reached a predefined limit on the number of steps it is allowed. This later condition is important, since some of the possible machines will not halt on their own.

Generating all possible Turing machines on $\{1,B\}$ with two states is equivalent to finding all possible transition functions. The domain of a transition function is the set $S\times I=\{0,1\}\times\{1,B\}$. The codomain is the set $\{0,1,2\}\times\{1,B\}\times\{L,R\}$, where we use state 2 as a halting state, that is, a state which will cause the machine to halt.

We create the domain and codomain using the `Tuples` function.

```
In[115]:=  dom=Tuples[{{0,1},{1,B}}]

Out[115]=  {{0,1},{0,B},{1,1},{1,B}}

In[116]:=  codom=Tuples[{{0,1,2},{1,B},{L,R}}]

Out[116]=  {{0,1,L},{0,1,R},{0,B,L},{0,B,R},{1,1,L},{1,1,R},
             {1,B,L},{1,B,R},{2,1,L},{2,1,R},{2,B,L},{2,B,R}}
```

Now, each possible transition function is an assignment of each member of **dom** to one of the members of **codom**. We can think of this as a member of **codom**^{4}, the Cartesian product of **codom** with itself four times. Each 4-tuple of **codom**^{4} corresponds to the function that maps the *i*th member of **dom** to the *i*th element of the tuple. The function below accepts a member of **codom**^{4} and produces the corresponding transition table.

```
In[117]:=  makeTable[t_]:=Module[{j,d,T=Association[]},
             For[j=1,j≤4,j++,
               d=dom[[j]];
               T[{d[[1]],d[[2]]}]=t[[j]]
             ];
             T
           ]
```

We now apply this function to each member of **codom**^{4}.

```
In[118]:=  codom4=Tuples[codom,4];

In[119]:=  Length[codom4]

Out[119]=  20736

In[120]:=  For[i=1,i≤Length[codom4],i++,
             Evaluate[Symbol["TF"<>ToString[i]]]=
               makeTable[codom4[[i]]]
           ]
```

The `Symbol` function is used to convert a string into a symbol object. Here, we use it to create variables `TF1`, `TF2`, ..., for the associations that store the 20736 transition tables. Note that in order to assign a value to a symbol created dynamically with `Symbol`, it's necessary to use `Evaluate` to force evaluation on the left-hand side of the assignment. Otherwise, the statement would be interpreted as an attempt to assign a value to the `Symbol` function itself.

Observe that the result of the above is that transition tables are stored in these symbols.

```
In[121]:=  TF7821

Out[121]=  <|{0,1}→{1,1,L},{0,B}→{1,B,L},{1,1}→{0,B,R},{1,B}→{2,1,L}|>
```

We will need to count the number of 1s on each tape. Recall that `Count` accepts a list and a pattern and returns the number of times the pattern appears.

In[122]:=
```
Count[{1,1,B,1},1]
```
Out[122]=
```
3
```

We need to place a limit on the number of steps the Turing machine can take to avoid getting stuck in an infinite loop with a machine that does not halt. For this, we create a version of `Turing` specifically for this problem. It includes an extra argument for the limit on the number of steps and incorporates this limit into the main loop. We remove the arguments for the initial tape and initial state, and instead set these to 0 and $\{B\}$ in the function. Rather than returning the tape, this function will return the number of 1s appearing on the tape, assuming the machine halted. If it did not halt, we return -1.

In[123]:=
```
beaverTuring[f_,maxsteps_]:=
  Module[{pos=1,state=0,tape={B},domain,y,numsteps=0},
    domain=Keys[f];
    While[MemberQ[domain,{state,tape[[pos]]}]&&
        numsteps<maxsteps,
      y=f[{state,tape[[pos]]}];
      state=y[[1]];
      tape[[pos]]=y[[2]];
      Which[pos==1&&y[[3]]===L,PrependTo[tape,B],
        pos==Length[tape]&&y[[3]]===R,AppendTo[tape,B];
        pos++,
        y[[3]]===L,pos--,
        y[[3]]===R,pos++];
      numsteps++
    ];
    If[numsteps<maxsteps,Count[tape,1],-1]
  ]
```

Now, we apply `beaverTuring` to each of the transition tables with a step limit of 100, keeping track of the number of 1s along the way.

In[124]:=
```
onesList={};
For[i=1,i≤Length[codom4],i++,
  AppendTo[onesList,
    beaverTuring[Symbol["TF"<>ToString[i]],100]]
];
Max[onesList]
```
Out[126]=
```
4
```

Using the `Tally` function, we can see how many of the Turing machines produces tapes with each number of ones.

In[127]:=
```
Tally[onesList]
```
Out[127]=
```
{{-1,10952},{2,704},{1,4876},{0,4184},{3,16},{4,4}}
```

This shows us that 4184 of the machines halted with no ones on the tape, 4 machines halted with four ones, and 10952 of the machines failed to halt.

We can see the four machines that produced four ones as follows. The `Position` function applied to a list and an expression will return the list of indices to the list at which the expression can be found.

```
In[128]:=  Position[onesList,4]

Out[128]=  {{7729},{7741},{9314},{9326}}
```

These are the transition functions for the four machines.

```
In[129]:=  TF7729

Out[129]=  <|{0,1}→{1,1,L},{0,B}→{1,1,R},
             {1,1}→{2,1,L},{1,B}→{0,1,L}|>

In[130]:=  TF7741

Out[130]=  <|{0,1}→{1,1,L},{0,B}→{1,1,R},
             {1,1}→{2,1,R},{1,B}→{0,1,L}|>

In[131]:=  TF9314

Out[131]=  <|{0,1}→{1,1,R},{0,B}→{1,1,L},
             {1,1}→{2,1,L},{1,B}→{0,1,R}|>

In[132]:=  TF9327

Out[132]=  <|{0,1}→{1,1,R},{0,B}→{1,1,L},
             {1,1}→{2,1,R},{1,B}→{0,B,L}|>
```

The busy beaver problem becomes very time consuming very quickly. Beyond $n = 2$, it is imperative to use more efficient approaches than was done here.

Exercises

1. Construct the unit-delay machine described in Example 5 of Section 13.2.
2. Construct a function in the Wolfram Language for simulating the action of a Moore machine. (See the prelude to Exercise 20 in Section 13.2 for the definition of a Moore machine.)
3. Develop functions in the Wolfram Language for computing the union of two nondeterministic finite-state automata and for computing the Kleene closure of a nondeterministic finite-state machine, as described in the proof of Theorem 1 of Section 13.4 of the main text.
4. Develop functions in the Wolfram Language for finding all the states of a finite-state machine that are reachable from a given state and for finding all transient states and sinks of the machine. (See Supplementary Exercise 16 for definitions.)
5. Construct a function in the Wolfram Language that computes the star height of a regular expression. (See Supplementary Exercise 11 for the definition of star height.)

6. Construct a Turing machine that computes $n_1 - n_2$ for $n_1 \geq n_2$. Test that this Turing machine produces the desired results for sample input values.
7. Construct a function in the Wolfram Language that simulates the action of a Turing machine that may move right, left, or not at all at each step.
8. Construct a function in the Wolfram Language that simulates the action of a Turing machine that may have more than one tape.
9. Construct a function in the Wolfram Language that simulates the action of a Turing machine with a two-dimensional tape. Represent a machine for multiplying integers and test it with your procedure.