13 Modeling Computation

Introduction

In this chapter, we will use Maple to create implementations of 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 Maple's support for regular expressions, and we will implement Turing machines.

13.1 Languages and Grammars

In this section, we write a procedure 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 $w_1 \rightarrow w_2$ with w_1 a single nonterminal symbol.

Our strategy for generating the language will be as follows. We initialize a set L to the empty set. In this set, we will store all words, that is, strings consisting only of terminal symbols. A list A is initialized to the set 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 placed in L if they consist solely of terminal symbols, or placed at the end of A to be processed further.

In order to prevent the time taken from becoming excessive, we will stop processing elements of A either if A becomes empty of if a set number of words have been produced. This limit will be an argument to our procedure.

Representation

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

We insist that terminal symbols be represented as characters (strings), so that the words produced can be presented as strings. Nonterminal symbols can also be represented as strings, but, for convenience, we will use unassigned Maple names instead.

Strings containing nonterminal symbols will be represented as lists, but words will be presented as strings. Note that the **cat** (concatenation) command accepts any number of strings and concatenates them. Given a list consisting entirely of strings, we apply **op** followed by **cat** to obtain a single string.

```
> cat(op(["a", "b", "c", "d"]))
      "abcd"                                                    (13.1)
```

Productions will be stored in a **table**. The indices of the table will be the nonterminal symbols (recall that we are considering only type 2 grammars). The entry associated to a nonterminal symbol will be the set of all products derivable from that symbol.

In Example 12, $S \rightarrow AB$ is the only derivation from the starting symbol, so $\{[A, B]\}$ will be the entry associated to S in the table. On the other hand, $B \rightarrow Ba$, $B \rightarrow Cb$, and $B \rightarrow b$ are all productions from B. Thus, $\{[b], [B, a], [C, b]\}$ would be the entry associated to B.

Here is the production table for Example 12. We begin by ensuring the names used as the nonterminal symbols do not store values.

```
> unassign('S', 'A', 'B', 'C') :

> Ex12productions := table( ) :

> Ex12productions[S] := {[A, B]} :

> Ex12productions[A] := {[C, “a”]} :

> Ex12productions[B] := {[“b”], [B, “a”], [C, “b”]} :

> Ex12productions[C] := {[“b”], [“c”, “b”]} :

> eval (Ex12productions)
      table ([A = {[C, “a”]} , C = {[“b”], [“c”, “b”]} , S = {[A, B]} ,
         B = {[“b”], [B, “a”], [C, “b”]}])                                  (13.2)
```

Our procedure will require the following arguments: the set **V** defining the vocabulary, the set **T** of terminal symbols, the starting symbol **S**, the table of productions **P**, and the limit on the number of words to generate, **wordlimit**. Note that, with the exception of the limit on the number of words, this is the same information that makes up a grammar. If you wish, you could apply the **timelimit** function to limit the run of the procedure by time rather than number of words, but the procedure will run so quickly that imposing a limit on the number of words generated also controls the length of the output.

Implementation

The procedure begins by calculating **N**, the nonterminal symbols. Then, **L** is initialized to the empty set and **A** is initialized to the list **[S]**. Recall that these will store the words that have been produced and the list of strings with nonterminal symbols that still require processing. We also initialize **count** to 0. This will count the number of words that have been produced, which is more efficient that repeatedly calculating the size of **L**.

After the initializations are complete, we begin a while loop controlled by the conditions that **count** does not exceed the limit on the number of words and that **A** is nonempty. Within the while loop, we set **curString** (the “current string”) equal to the first member 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, 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 list. Loop over the entries of **curString**. For each element, check to see if it is a member of **N**, the nonterminal symbols. If not, we move on to the next element. If the symbol is nonterminal, then 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 $[c, b, a, B, a]$ as part of the grammar given in Example 12.

```
> curString := [“c”, “b”, “a”, B, “a”]
    curString := [“c”, “b”, “a”, B, “a”]                                (13.3)
```

After determining that the first three entries are terminal, we look at **curString[4]** and see that it is nonterminal. We obtain the derivations associated with it from the table.

```
> Ex12productions[curString[4]]
    {[“b”], [B, “a”], [C, “b”]}                                          (13.4)
```

For each of these derivations, we will substitute the derivation into **curString** in place of the fourth position. We use **subsop** to perform the substitution within a loop over the elements of the set obtained from the derivations table.

```
> for s in Ex12productions[curString[4]] do
    subsop(4 = op(s), curString)
  end do
    [“c”, “b”, “a”, “b”, “a”]
    [“c”, “b”, “a”, B, “a”, “a”]
    [“c”, “b”, “a”, C, “b”, “a”]                                         (13.5)
```

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 the set of elements in the string is a subset of the terminal symbols.

```
> {op([“c”, “b”, “a”, “b”, “a”])} subset {“a”, “b”, “c”}
    true                                                                 (13.6)
```

```
> {op([“c”, “b”, “a”, B, “a”, “a”])} subset {“a”, “b”, “c”}
    false                                                                (13.7)
```

Those that are words are concatenated into strings and added to **L** (and **count** is updated). Those that are not words are added to **A**.

Here is the procedure.

```
FormWords := proc(V,T,S,P,wordlimit)
  local N, L, A, count, curString, D, i, s, d;
  N := V minus T;
  L := {};
  A := [[S]];
  count := 0;
  while count <= wordlimit and A <> [] do
    curString := A[1];
    A := A[2..-1];
    D := [];
    for i from 1 to nops(curString) do
      if curString[i] in N then
        for s in P[curString[i]] do
          D := [op(D),subsop(i=op(s),curString)];
```

```
                end do;
            end if;
        end do;
        for d in D do
            if {op(d)} subset T then
                L := L union {cat(op(d))};
                count := count + 1;
            else
                A := [op(A),d];
            end if;
        end do;
    end do;
    return L;
end proc:
```

We use our procedure on the grammar defined by Example 12, up to 20 words.

```
> FormWords({"a", "b", "c", A, B, C, S}, {"a", "b", "c"},
    S, Ex12productions, 20)
    {"bab", "baba", "babb", "bacbb", "cbab", "cbaba",
      "cbabb", "cbacbb"}                                  (13.8)
```

Note that the procedure did not return a set of size 20. This is because **count** is incremented every time a word is derived, not every time a *new* word is derived. Since there is more than one way to derive the same word, we obtain fewer than 20 words.

13.2 Finite-State Machines with Output

Example 4 from 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 use Maple to model those two finite-state machines.

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 procedure 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 procedure a list of members of the input alphabet, and the procedure 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 in Maple.

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 required as arguments to the procedure.

The input and output alphabets, I and O, can be represented by sets of Maple objects but will not be required arguments to the procedure. In Example 4, these are both equal to the set $\{0, 1\}$.

The transition function and output function will be represented by a single table. This will have the benefit of making the definition of the functions less cumbersome. The indices to the table will be pairs [*state*, *input*] where *state* is a nonnegative integer and *input* will be a member of I. The entries of the table 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.)

```
> Ex4Table := table():

> Ex4Table[0, 0] := [1, 1]:

> Ex4Table[0, 1] := [3, 0]:

> Ex4Table[1, 0] := [1, 1]:

> Ex4Table[1, 1] := [2, 1]:

> Ex4Table[2, 0] := [3, 0]:

> Ex4Table[2, 1] := [4, 0]:

> Ex4Table[3, 0] := [1, 0]:

> Ex4Table[3, 1] := [0, 0]:

> Ex4Table[4, 0] := [3, 0]:

> Ex4Table[4, 1] := [4, 0]:
```

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

The Machine Modeling Procedure

The procedure we create is to accept as arguments the transition-output table and the input string. It will produce the output string.

The procedure 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 procedure.

```
MachineWithOutput := proc(transTable::table, inString::list)
   local curState, outString, i;
   curState := 0;
   outString := [0 $ nops(inString)];
   for i from 1 to nops(inString) do
      outString[i] := transTable[curState, inString[i]][2];
      curState := transTable[curState, inString[i]][1];
   end do;
   return outString;
end proc:
```

Example 4 asks to find the output string when the input is 101011.

```
> MachineWithOutput(Ex4Table, [1, 0, 1, 0, 1, 1])
     [0, 0, 1, 0, 0, 0]                                              (13.9)
```

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 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 initial 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, 0].

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

```
> addTable := table():
> addTable[0, "00"] := [0, 0]:
> addTable[0, "01"] := [0, 1]:
> addTable[0, "10"] := [0, 1]:
> addTable[0, "11"] := [1, 0]:
> addTable[1, "00"] := [0, 1]:
> addTable[1, "01"] := [1, 0]:
> addTable[1, "10"] := [1, 0]:
> addTable[1, "11"] := [1, 1]:
```

Applying the **MachineWithOutput** procedure to this table and the input produces the sum of the integers.

> *MachineWithOutput*(*addTable*, [“10”, “11”, “11”, “00”])

$[1, 0, 1, 1]$ **(13.10)**

This corresponds to $1101_2 = 13$.

13.3 Finite-State Machines with No Output

In this section, we will see how to use Maple to represent finite-state automata and to perform language recognition.

Kleene Closure

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

Given two lists **listA** and **listB**, we can concatenate them as follows: **[op(listA),op(listB)]**. For example,

> *listA* := [1, 2, 3]

listA := [1, 2, 3] **(13.11)**

> *listB* := [“a”, “b”, “c”]

listB := [“a”, “b”, “c”] **(13.12)**

> [*op*(*listA*), *op*(*listB*)]

[1, 2, 3, “a”, “b”, “c”] **(13.13)**

Note that **op** applied to a number or a string has no effect.

> *op*(5)

5 **(13.14)**

> *op*(“abc”)

“abc” **(13.15)**

Given two sets of strings, we can form all possible concatenations by using two for loops to concatenate each pair.

```
SetCat := proc(A::set, B::set)
   local C, x, y;
   C := {};
   for x in A do
      for y in B do
         C := C union {[op(x),op(y)]};
      end do;
   end do;
   return C;
end proc:
```

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

```
> listA := {0, [1, 1]}
    listA := {0, [1, 1]}                                                    (13.16)

> listB := {1, [1, 0], [1, 1, 0]}
    listB := {1, [1, 0], [1, 1, 0]}                                         (13.17)

> SetCat(listA, listB)
    {[0, 1], [0, 1, 0], [1, 1, 1], [0, 1, 1, 0], [1, 1, 1, 0], [1, 1, 1, 1, 0]}   (13.18)
```

Given a set A, recall that A^0 is defined to be the set of 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 the following procedure to produce the powers of A. The procedure is modeled on the recursive definition given in the text.

```
SetPow := proc(A::set, k::nonnegint)
    if k=0 then
        return {[]};
    else
        return SetCat(SetPow(A,k-1),A);
    end if;
end proc:
```

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

```
> SetPow(listB, 3)
    {[1, 1, 1], [1, 0, 1, 1], [1, 1, 0, 1], [1, 1, 1, 0], [1, 0, 1, 0, 1], [1, 0, 1, 1, 0],
       [1, 1, 0, 1, 0], [1, 1, 0, 1, 1], [1, 1, 1, 0, 1], [1, 1, 1, 1, 0], [1, 0, 1, 0, 1, 0],
       [1, 0, 1, 1, 0, 1], [1, 0, 1, 1, 1, 0], [1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 1, 0],
       [1, 1, 1, 0, 1, 0], [1, 0, 1, 0, 1, 1, 0], [1, 0, 1, 1, 0, 1, 0], [1, 1, 0, 1, 0, 1, 0],
       [1, 1, 0, 1, 1, 0, 1], [1, 1, 0, 1, 1, 1, 0], [1, 1, 1, 0, 1, 1, 0], [1, 0, 1, 1, 0, 1, 1, 0],
       [1, 1, 0, 1, 0, 1, 1, 0], [1, 1, 0, 1, 1, 0, 1, 0], [1, 1, 0, 1, 1, 0, 1, 1, 0]}   (13.19)
```

To form the partial Kleene closure $A^{[n]}$, we must find the union of 1, A, A^2, .., A^n. Building the A^k iteratively is more efficient than using **SetPow**.

```
Kleene := proc(A::set, n::posint)
    local K, x, Ak;
    K := {[]};
    for x in A do
        K := K union {[op(x)]};
    end do;
    Ak := K;
    from 2 to n do
      Ak := SetCat(Ak,A);
      K := K union Ak;
    end do;
    return K;
end proc:
```

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

```
> Kleene({0, 1} , 3)
      {[ ], [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]}                                    (13.20)
```

Extended Transition Function for a Finite-State Automaton

We now create a procedure 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 text.

As in Section 13.2 of this manual, we model the transition function as a table. The indices to the table will be the pairs consisting of the current state of the automaton and the input. The corresponding entries 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.

```
> Ex51Table := table([(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 procedure, **ExtendedTransition**. The arguments of this procedure 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 to designing the procedure. 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.

```
ExtendedTransition := proc(state,input,transFunc::table)
    local curState, i;
    curState := state;
    for i from 1 to nops(input) do
      curState := transFunc[curState,input[i]];
    end do;
    return curState;
end proc:
```

We can use this procedure to see that applying the automaton M_1 from Example 5 to [1, 0, 1, 1, 0] from initial state 0 ends in state 1.

```
> ExtendedTransition(0, [1, 0, 1, 1, 0], Ex51Table)
      1                                                                                  (13.21)
```

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 write a procedure that, given the transition table 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 procedure 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.

```
IsRecognized := proc(x,transFunc::table,init,final::set)
    local endState;
   endState := ExtendedTransition(init,x,transFunc);
   return evalb(endState in final);
end proc:
```

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

```
> IsRecognized([1, 1, 1, 1, 1], Ex51Table, 0, {0})
    true                                                        (13.22)
```

```
> IsRecognized([1, 1, 0, 1], Ex51Table, 0, {0})
    false                                                       (13.23)
```

Using the **Kleene** procedure 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 procedure will calculate the subset of $A^{[n]}$ recognized by the finite-state automaton defined by the transition table and set of final states.

This procedure operates by applying **Kleene** and then using **IsRecognized** to check each element of $A^{[n]}$. Note that we extract those members of $A^{[n]}$ that are recognized by applying the **select** command. This command requires its first argument be a procedure that returns true or false and the second argument a set, list, or other expression. The result is the operands in the expression given as the second argument, in this case, the members of the set, for which the procedure returns true. When the procedure given as the first argument requires more than one argument, additional arguments to the procedure can be given as the third and subsequent arguments of **select**. For example, below we use **select** to obtain the members of the list that are less than the number 5.

```
> select((a, b) → evalb(a < b), [3, 9, 5, 2, 1, 5, 6, 0, 3, 8], 5)
    [3, 2, 1, 0, 3]                                             (13.24)
```

The members of the list are substituted in for *a*, while 5 is used for *b*.

Here is the procedure.

```
FindLanguage :=
   proc(transFunc::table,init,final::set,A::set,n::posint)
   local An;
   An := Kleene(A,n);
   return select(IsRecognized,An,transFunc,init,final);
end proc:
```

Applying this procedure 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.

```
> FindLanguage(Ex51Table, 0, {0}, {0, 1}, 10)
    {[ ], [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]}                                        (13.25)
```

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 procedure will produce a deterministic finite-state automaton.

In particular, given the transition table for a nondeterministic automaton, its input alphabet, its starting state, and its set of final states, the procedure will produce the transition table 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 entries in the table will be sets of states, rather than individual states.

For example, here is the transition table for the nondeterministic automaton described in Example 10.

```
> Ex10Table := table():

> Ex10Table[0, 0] := {0, 2}:

> Ex10Table[0, 1] := {1}:

> Ex10Table[1, 0] := {3}:

> Ex10Table[1, 1] := {4}:

> Ex10Table[2, 0] := {}:

> Ex10Table[2, 1] := {4}:

> Ex10Table[3, 0] := {3}:

> Ex10Table[3, 1] := {}:
```

```
> Ex10Table[4, 0] := {3} :
```

```
> Ex10Table[4, 1] := {3} :
```

The final states are 0 and 4.

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 procedure, we will create a table initialized to the empty table. 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 nonempty, 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 procedure calculates the set of final states for the deterministic automaton.

Here is the procedure. Note that the procedure returns the sequence of the new transition table, the starting state, and the set of final states.

```
MakeDeterministic :=
    proc(transFunc::table, Iset::set, init, final::set)
   local newTable, S, T, state, i, s, x, newfinal;
   newTable := table();
   S := {};
   T := {{init}};
   while T <> {} do
      state := T[1];
      T := T minus {state};
      S := S union {state};
      for i in Iset do
         x := {};
         for s in state do
            x := x union transFunc[s,i];
         end do;
         newTable[state,i] := x;
         if not(x in S) then
```

```
            T := T union {x};
         end if;
      end do;
   end do;
   newfinal := {};
   for state in S do
      if state intersect final <> {} then
         newfinal := newfinal union {state};
      end if;
   end do;
   return newTable,{init},newfinal;
end proc:
```

Applying this procedure to the Example 10 information produces the following.

```
> Ex10DTable, Ex10Dinit, Ex10Dfinal :=
     MakeDeterministic(Ex10Table, {0, 1}, 0, {0, 4})
     Ex10DTable, Ex10Dinit, Ex10Dfinal := newTable, {0},
        {{0}, {4}, {0, 2}, {1, 4}, {3, 4}}                        (13.26)
```

We can inspect the transition table by applying **eval** to **Ex10DTable**.

```
> eval(Ex10DTable)
     table([({0, 2}, 0) = {0, 2}, ({1, 4}, 1) = {3, 4}, ({3}, 0) = {3},
        ({0, 2}, 1) = {1, 4}, ({3}, 1) = ∅, (∅, 0) = ∅, ({4}, 1) = {3},
        ({3, 4}, 1) = {3}, ({0}, 1) = {1}, ({4}, 0) = {3}, ({0}, 0) = {0, 2},
        ({3, 4}, 0) = {3}, ({1}, 1) = {4}, ({1}, 0) = {3}, ({1, 4}, 0) = {3},
        (∅, 1) = ∅])                                              (13.27)
```

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

We use the output as the arguments to **FindLanguage**.

```
> FindLanguage(Ex10DTable, Ex10Dinit, Ex10Dfinal, {0, 1}, 10)
     {[ ], [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]}                           (13.28)
```

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 Maple's support for regular expressions for working with strings. We will also develop a procedure for calculating the concatenation of two nondeterministic automata.

Regular Expressions

Maple's commands related to regular expressions lie in the **StringTools** package, which we load now.

```
> with(StringTools) :
```

The command **RegMatch** is used to determine whether or not a given string can be matched to a given pattern. When using **RegMatch**, the first argument is a string describing the regular expression or pattern, and the second argument is the string you are attempting to match against.

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.

```
> RegMatch(“01”, “01”)
      true                                                    (13.29)
```

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

Anchors

The next example illustrates a significant difference between regular expressions as described in the text and regular expressions in Maple.

```
> RegMatch(“01”, “so42301kkj91”)
      true                                                    (13.30)
```

It is clear that the string “so42301kkj91” is not a member of the regular set specified by the regular expression “01”, since the only member of that regular set is the string “01”. Maple returned true because the pattern “01” matched a substring of the second argument.

Maple, and most programming languages, use regular expressions primarily to search for patterns within strings and they interpret regular expressions as loosely as possible to make them flexible. To use regular expressions in Maple in the more formal sense described in the text, we will need to specify that the pattern we give as the first argument is to match the entire string, not part of it.

To do this, we use two special characters, **^** and **$**. These are referred to as anchors and they match the beginning and end of a string, respectively. Including them in the regular expression will ensure that we match only those strings completely described by the regular expression and that we do not match substrings.

```
> RegMatch(“^01$”, “01”)
      true                                                    (13.31)
```

```
> RegMatch(“^01$”, “so42301kkj91”)
      false                                                   (13.32)
```

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.

```
> RegMatch(“^10*$”, “10000000”)
      true                                                    (13.33)

> RegMatch(“^10*$”, “1”)
      true                                                    (13.34)

> RegMatch(“^10*$”, “0111000”)
      false                                                   (13.35)
```

Note that without the **ˆ** and **$**, the final example would have also returned true.

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

```
> RegMatch(“^(10)*$”, “10101010101010”)
      true                                                    (13.36)

> RegMatch(“^(10)*$”, “1010101”)
      false                                                   (13.37)
```

Maple, and 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.

```
> RegMatch(“^1*0+$”, “1111000”)
      true                                                    (13.38)

> RegMatch(“^1*0+$”, “00”)
      true                                                    (13.39)

> RegMatch(“^1*0+”, “111”)
      false                                                   (13.40)
```

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.

```
> RegMatch(“^1*0?$”, “111111”)
      true                                                    (13.41)

> RegMatch(“^1*0?$”, “1111110”)
      true                                                    (13.42)

> RegMatch(“^1*0?$”, “11111100”)
      false                                                   (13.43)
```

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 $\cup$ symbol in an expression such as $0\,(0 \cup 1)\,*$.

```
> RegMatch(“^0(0|1)*$”, “011010”)
      true                                                    (13.44)
```

```
> RegMatch(“^0(0|1)*$”, “1011010”)
      false                                                   (13.45)
```

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

```
> RegMatch(“^2((10)*|(01)*)2$”, “21010102”)
      true                                                    (13.46)
```

```
> RegMatch(“^2((10)*|(01)*)2$”, “201012”)
      true                                                    (13.47)
```

```
> RegMatch(“^2((10)*|(01)*)2$”, “210012”)
      false                                                   (13.48)
```

In some circumstances, union can be replaced by “character classes.” By placing characters within a set of brackets, you indicate that any of the characters inside the brackets are allowed. For example, $0\,(0 \cup 1)\,*$ can be expressed as follows.

```
> RegMatch(“^0[01]*$”, “011010”)
      true                                                    (13.49)
```

Note that this cannot be used in more complicated expressions such as **(13.46)** and **(13.48)**. It is only available when the options are single characters.

Ranges can also be used within a character class by separating the beginning and end of a range of characters with a hyphen. For example, the following matches strings beginning at least one lowercase letter and ending with a digit between 6 and 9.

```
> RegMatch(“^[a-z]+[6-9]$”, “test9”)
      true                                                    (13.50)
```

```
> RegMatch(“^[a-z]+[6-9]$”, “Test9”)
      false                                                   (13.51)
```

Observe that character classes are case sensitive, but multiple ranges can be used within a class.

```
> RegMatch(“^[a-zA-Z]+[6-9]$”, “Test9”)
      true                                                    (13.52)
```

Character classes have a negated option. 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.

```
> RegMatch("^1[^01]*0$", "169jwq0")
      true                                              (13.53)
```

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.

```
> RegMatch("^1...0$", "12340")
      true                                              (13.54)
```

```
> RegMatch("^1...0$", "1230")
      false                                             (13.55)
```

```
> RegMatch("^1...0$", "1234567890")
      false                                             (13.56)
```

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

Concatenation of Automata

We will write a procedure 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 automata that our procedure will concatenate.

The first automata is the result of Example 3, for recognizing $1 * \cup 01$. Our implementation is based on the simple form shown in Figure 3b.

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.

```
> Atable := table( ):
```

```
> Atable[0, 0] := {2}:
```

```
> Atable[0, 1] := {1}:
```

```
> Atable[1, 0] := { }:
```

```
> Atable[1, 1] := {1}:
```

```
> Atable[2, 0] := { }:
```

```
> Atable[2, 1] := {3} :

> Atable[3, 0] := {} :

> Atable[3, 1] := {} :
```

The final states for this automaton are $\{0, 1, 3\}$. We can confirm that it recognizes $1 * \cup 01$ by applying **MakeDeterministic** and **FindLanguage**.

```
> FindLanguage (MakeDeterministic (Atable, {0, 1} , 0, {0, 1, 3}) , {0, 1} , 10)
    {[ ], [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]}                                            (13.57)
```

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.

```
> Btable := table( ) :

> Btable[0, 0] := {} :

> Btable[0, 1] := {1} :

> Btable[1, 0] := {2} :

> Btable[1, 1] := {} :

> Btable[2, 0] := {} :

> Btable[2, 1] := {3} :

> Btable[3, 0] := {} :

> Btable[3, 1] := {} :
```

The only final state is state 3.

Applying **FindLanguage**, we confirm that this models that machine that recognizes 101.

```
> FindLanguage (MakeDeterministic (Btable, {0, 1} , 0, {3}) , {0, 1} , 10)
    {[1, 0, 1]}                                                               (13.58)
```

Concatenating the Machines

Our concatenation procedure 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 $10 \cdot \cdot \cdot_A + 1$ where s_A is the starting state of the first machine. In our case, this will be equal to $10 \cdot 0 + 1 = 1$.

Next, we find the final states of the concatenation. Let **Afinal** and **Bfinal** 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.

```
> Afinal := {0, 1, 3}
    Afinal := {0, 1, 3}                                        (13.59)
```

```
> Bfinal := {3}
    Bfinal := {3}                                              (13.60)
```

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

```
> map( f → 10 f + 2, Bfinal)
    {32}                                                       (13.61)
```

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 procedure 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 with an empty table.

```
> ABtable := table( ):
```

Item 5 tells us that all of the original transitions are transitions in the new machine. Thus, for both of the original tables, we need to add corresponding entries to this table. Keep in mind that the state names are updated by multiplying by 10 and adding 1 or 2.

We proceed as follows. For **Atable**, use **indices** to obtain the list of all indices in the table. For each index **i**, the index in **ABtable** will be **[10*i[1]+1,i[2]]**. This computes the appropriate state name in the concatenated machine and keeps the same input value. The associated entry will be obtained by using **map** and the functional operator **f->10*f+1** applied to the previous entry. Note that **op** must be applied to **i** before it can be used as an index to **Atable**, since the **indices** command wraps indices in lists.

```
> for i in indices(Atable) do
      ABtable[10 ∗ i[1] + 1, i[2]] := map( f → 10 ∗ f + 1, Atable[op(i)])
  end do
```

$$
\begin{aligned}
&ABtable_{11,1} := \{11\}\\
&ABtable_{21,1} := \{31\}\\
&ABtable_{1,1} := \{11\}\\
&ABtable_{11,0} := \emptyset\\
&ABtable_{31,1} := \emptyset\\
&ABtable_{31,0} := \emptyset\\
&ABtable_{1,0} := \{21\}\\
&ABtable_{21,0} := \emptyset
\end{aligned}
\tag{13.62}
$$

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

```
> for i in indices(Btable) do
      ABtable[10 ∗ i[1] + 2, i[2]] := map( f → 10 ∗ f + 2, Btable[op(i)])
  end do
```

$$
\begin{aligned}
&ABtable_{12,1} := \emptyset\\
&ABtable_{22,1} := \{32\}\\
&ABtable_{2,1} := \{12\}\\
&ABtable_{12,0} := \{22\}\\
&ABtable_{32,1} := \emptyset\\
&ABtable_{32,0} := \emptyset\\
&ABtable_{2,0} := \emptyset\\
&ABtable_{22,0} := \emptyset
\end{aligned}
\tag{13.63}
$$

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 will again loop through the indices of **Atable**, this time checking whether the image 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 entry in the **ABtable** rather than replace it.)

```
> for i in indices(Atable) do
      if Atable[op(i)] intersect finalA ≠ { } then
          ABtable[10 ∗ i[1] + 1, i[2]] := ABtable[10 ∗ i[1] + 1, i[2]] union {2}
      end if
  end do
```

We can see the current transition table by applying **eval**.

```
> eval(ABtable)
    table([(32, 1) = ∅, (1, 1) = {2, 11}, (11, 0) = ∅, (2, 1) = {12},
       (22, 1) = {32}, (1, 0) = {21}, (21, 0) = ∅, (2, 0) = ∅, (31, 0) = ∅,
       (21, 1) = {2, 31}, (31, 1) = ∅, (32, 0) = ∅, (22, 0) = ∅, (12, 1) = ∅,
       (11, 1) = {2, 11}, (12, 0) = {22}])                                  (13.64)
```

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.

```
> for i in indices(Btable) do
    if i[1] = 0 then
      ABtable[1, i[2]] := ABtable[1, i[2]] union map(f → 10 * f + 2, Btable[op(i)])
    end if
  end do
```

Apply **eval** again.

```
> eval(ABtable)
    table([(32, 1) = ∅, (1, 1) = {2, 11, 12}, (11, 0) = ∅, (2, 1) = {12},
       (22, 1) = {32}, (1, 0) = {21}, (21, 0) = ∅, (2, 0) = ∅, (31, 0) = ∅,
       (21, 1) = {2, 31}, (31, 1) = ∅, (32, 0) = ∅, (22, 0) = ∅, (12, 1) = ∅,
       (11, 1) = {2, 11}, (12, 0) = {22}])                                  (13.65)
```

Note that this modified the entry for $(1,\ 1)$. (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 entry 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. 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. 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. And 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 Procedure

Here is the complete procedure.

```
CatAutomata :=
    proc(Atable,Astart,Afinal,Btable,Bstart,Bfinal,Iset)
   local ABtable, ABstart, ABfinal, i;
   ABstart := 10*Astart + 1;
   ABfinal := map(f->10*f+2,Bfinal);
   if (Astart in Afinal) and (Bstart in Bfinal) then
      ABfinal := ABfinal union {ABstart};
   end if;
```

```
    ABtable := table();
    for i in indices(Atable) do
      ABtable[10*i[1]+1,i[2]] := map(f->10*f+1,Atable[op(i)]);
    end do;
    for i in indices(Btable) do
      ABtable[10*i[1]+2,i[2]] := map(f->10*f+2,Btable[op(i)]);
    end do;
    for i in indices(Atable) do
      if Atable[op(i)] intersect Afinal <> {} then
        ABtable[10*i[1]+1,i[2]] := ABtable[10*i[1]+1,i[2]] union
           {Bstart*10+2};
      end if;
    end do;
    if (Astart in Afinal) then
      for i in indices(Btable) do
        if i[1] = 0 then
          ABtable[1,i[2]] := ABtable[1,i[2]] union
             map(f->10*f+2,Btable[op(i)]);
        end if;
      end do;
    end if;
    return ABtable,Iset,ABstart,ABfinal;
end proc:
```

Applying this to our examples and passing the results on to **MakeDeterministic** and **FindLanguage** shows us that the result does indeed recognize $(1 * \cup 01)\,101$.

```
> Ctable, CI, Cstart, Cfinal :=
    CatAutomata(Atable, 0, {0, 1, 3}, Btable, 0, {3}, {0, 1})
    Ctable, CI, Cstart, Cfinal := ABtable, {0, 1}, 1, {32}                    (13.66)

> FindLanguage(MakeDeterministic(Ctable, CI, Cstart, Cfinal), {0, 1}, 10)
    {[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]}           (13.67)
```

13.5 Turing Machines

In this section, we will create a model of a Turing machine. 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.

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 procedures to represent the function as a table from pairs (s, x) to triples (s', x', d).

We create a procedure that will transform the set of five-tuples representation into the table representation.

```
TuplesToTable := proc(S::set)
   local T, x;
   T := table();
   for x in S do
      T[x[1],x[2]] := x[3..5];
   end do;
   return T;
end proc:
```

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

```
> unassign('B', 'R', 'L')

> Ex1 := TuplesToTable({[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]})
    Ex1 := T                                                        (13.68)

> eval(Ex1)
    table([(0, B) = [3, B, R], (1, 1) = [2, 0, L], (2, 1) = [3, 0, R],
       (0, 1) = [1, 1, R], (1, 0) = [0, 0, R], (1, B) = [3, B, R],
       (0, 0) = [0, 0, R]])                                         (13.69)
```

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

The Turing Machine Procedure

Our Turing machine procedure will accept as input a table 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 state of the machine upon exit.

When the procedure begins, we initialize the name **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 by applying **indices** to the table. This will make it easier to check whether we have reached a halt.

The main work of the procedure 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 syntax **[B,op(tape)]**. 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 **[op(tape),B]**.

Here is the procedure.

```
Turing := proc(f::table,T::list,init)
  local pos, state, tape, domain, Y;
  pos := 1;
  state := init;
  tape := T;
  domain := {indices(f)};
  while [state,tape[pos]] in domain do
    Y:= f[state,tape[pos]];
    state := Y[1];
    tape[pos] := Y[2];
    if pos=1 and Y[3]='L' then
      tape := ['B',op(tape)];
    elif pos=nops(tape) and Y[3]='R' then
      tape := [op(tape),'B'];
      pos := pos + 1;
    elif Y[3]='L' then
      pos := pos - 1;
    else
      pos := pos + 1;
    end if;
  end do;
  return tape,state;
end proc:
```

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

```
> Turing(Ex1, [0, 1, 0, 1, 1, 0], 0)
    [0, 1, 0, 0, 0, 0], 3                                    (13.70)
```

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

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

```
VerboseTuring := proc(f::table,T::list,init)
  local pos, state, tape, domain, Y, displayTape;
  pos := 1;
  state := init;
  tape := T;
  domain := {indices(f)};
  displayTape := tape;
  displayTape[pos] := cat(`*`,tape[pos]);
  print(displayTape,state);
  while [state,tape[pos]] in domain do
```

```
        Y:= f[state,tape[pos]];
        state := Y[1];
        tape[pos] := Y[2];
        if pos=1 and Y[3]='L' then
          tape := ['B',op(tape)];
        elif pos=nops(tape) and Y[3]='R' then
          tape := [op(tape),'B'];
          pos := pos + 1;
        elif Y[3]='L' then
          pos := pos - 1;
        else
          pos := pos + 1;
        end if;
        displayTape := tape;
        displayTape[pos] := cat(‘*‘,tape[pos]);
        print(displayTape,state);
    end do;
    return tape, state;
end proc:
```

```
> 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
     [0, 1, 0, 0, 0, 0], 3                                   (13.71)
```

Applications of Turing Machines

We now apply our Turing machine procedure 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.

```
> Ex3 := TuplesToTable({[0, 0, 1, M, R], [1, 0, 1, 0, R], [1, 1, 1, 1, R],
    [1, B, 2, B, L], [1, M, 2, M, L], [2, 1, 3, M, L], [3, 0, 4, 0, L], [3, 1, 3, 1, L],
    [3, M, 5, M, R], [4, 0, 4, 0, L], [4, M, 0, M, R], [5, M, 6, M, R]})
     Ex3 := T                                                (13.72)
```

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.

```
> Turing(Ex3, [0, 0, 0, 0, 1, 1, 1, 1], 0)
      [M, M, M, M, M, M, M, M, B], 6                                  (13.73)
```

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

```
> Turing(Ex3, [0, 0, 0, 1, 1], 0)
      [M, M, M, M, M, B], 2                                           (13.74)
```

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.

```
> adder := TuplesToTable({[0, 1, 1, B, R], [1, 1, 2, B, R], [1, "*", 3, B, R],
    [2, 1, 2, 1, R], [2, "*", 3, 1, R]})
      adder := T                                                      (13.75)
```

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

```
UnaryTape := proc(a::nonnegint, b::nonnegint)
   return [1$(a+1), "*", 1$(b+1)];
end proc:
```

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

```
> UnaryTape(3, 4)
      [1, 1, 1, 1, "*", 1, 1, 1, 1, 1]                                (13.76)
```

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

```
> Turing(adder, UnaryTape(3, 4), 0)
      [B, B, 1, 1, 1, 1, 1, 1, 1, 1], 3                               (13.77)
```

You can see that this contains a string of eight 1s, indicating a result of 7.

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

```
> 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
      [B, B, 1, 1, 1, 1, 1, 1, 1, 1], 3                               (13.78)
```

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 procedures: **ExtendedTransition** and **IsRecognized**. The **IsRecognized** procedure merely called **ExtendedTransition** and checked whether the result was a final state or not. The **ExtendedTransition** procedure 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 procedures: **ExtendedTransitionND** and **IsRecognizedND**. 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.

```
ExtendedTransitionND := proc(states,input,transFunc)
   local curStates, i, s, newStates;
   curStates := states;
   for i from 1 to nops(input) do
      newStates := {};
      for s in curStates do
         newStates := newStates union transFunc[s,input[i]];
      end do;
      curStates := newStates;
   end do;
   return curStates;
end proc:
```

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 **IsRecognizedND** to call **ExtendedTransitionND** and check to see if the result intersects the set of final states.

```
IsRecognizedND := proc(x,transFunc,init,final)
   local endStates;
   endStates := ExtendedTransitionND({init},x,transFunc);
   return evalb(endStates intersect final <> {});
end proc:
```

With **IsRecognizedND** in hand, we can create **FindLanguageND**. This is effectively identical to **FindLanguage**.

```
FindLanguageND := proc(transFunc,init,final,A,n)
   local An, x, L;
   An := Kleene(A,n);
   L := {};
   for x in An do
      if IsRecognizedND(x,transFunc,init,final) then
         L := L union {x};
      end if;
   end do;
   return L;
end proc:
```

Applying this procedure to the machine defined by transition function **Ctable**, starting state 1, final state {32}, and alphabet {0,1} that was produced by **CatAutomata**, we see that the result is the same as when we applied **FindLanguage** and **MakeDeterministic** in **(13.67)**.

```
> FindLanguageND(Ctable, 1, {32}, {0, 1}, 10)
    {[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]}                                      (13.79)
```

```
> evalb(% = (13.67))
    true                                                                    (13.80)
```

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 1s 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 $n = 2$. (Note: Several of the steps in this solution take a few seconds to complete; therefore, except for the procedure definitions, none of the input lines in this solution will autoexecute.)

We will construct all possible Turing machines on two 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 **cartprod** command from **combinat**.

```
> dom := []
    dom := []                                                        (13.81)

> StimesI := combinat[cartprod]([[0, 1], [1, B]]) :

> while not StimesI[finished] do
    dom := [op(dom), StimesI[nextvalue]( )]
  end do :

> dom
    [[0, 1], [0, B], [1, 1], [1, B]]                                 (13.82)

> codom := []
    codom := []                                                      (13.83)

> StimesItimesLR := combinat[cartprod]([[0, 1, 2], [1, B], [L, R]]) :

> while not StimesItimesLR[finished] do
    codom := [op(codom), StimesItimesLR[nextvalue]( )]
  end do :

> codom
    [[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]]         (13.84)
```

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 ith member of *dom* to the ith element of the tuple. The procedure below accepts a member of $codom^4$ and produces the corresponding transition table.

```
MakeTable := proc(t::list)
  local T, j;
  T := table();
  for j from 1 to 4 do
    T[op(dom[j])] := t[j];
  end do;
  return T;
end proc:
```

We now apply this procedure to each member of $codom^4$.

```
> allTFs := []
    allTFs := []                                                     (13.85)

> codom4 := combinat[cartprod]([codom $ 4]) :
  while not codom4[finished] do
    allTFs := [op(allTFs), MakeTable(codom4[nextvalue]( ))]
  end do :
```

```
> nops(allTFs)
    20736                                                        (13.86)
```

The list **allTFs** now stores all 20 736 potential transition tables.

Recall, from Chapter 12, that the **Occurrences** command in **ListTools** can be used to count the number of 1s that appear on a tape.

```
> ListTools[Occurrences](1, [1, B, B, B, 1, 1, 1, 0, 1])
    5                                                            (13.87)
```

We need to place a limit on the number of steps the Turing machine can take and avoid getting stuck in an infinite loop because of 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 while loop. We remove the argument for the initial tape and initial state, and instead set these to 0 and **[B]** in the procedure. Rather than returning the tape, this procedure will return the number of 1s appearing on the tape, assuming the machine halted. If it did not halt, we return -1.

```
BeaverTuring := proc(f::table,maxstep)
   local pos, state, tape, domain, Y, numsteps;
   pos := 1;
   state := 0;
   tape := [’B’];
   domain := {indices(f)};
   numsteps := 0;
   while [state,tape[pos]] in domain and numsteps < maxstep do
      Y:= f[state,tape[pos]];
      state := Y[1];
      tape[pos] := Y[2];
      if pos=1 and Y[3]=’L’ then
         tape := [’B’,op(tape)];
      elif pos=nops(tape) and Y[3]=’R’ then
         tape := [op(tape),’B’];
         pos := pos + 1;
      elif Y[3]=’L’ then
         pos := pos - 1;
      else
         pos := pos + 1;
      end if;
      numsteps := numsteps + 1;
   end do;
   if numsteps < maxstep then
      return ListTools[Occurrences](1,tape);
   else
      return -1;
   end if;
end proc:
```

Now, we apply **BeaverTuring** to each of the transition tables in **allTFs** with a step limit of 100, keeping track of the number of 1s along the way.

```
> onesList := [ ]
    onesList := [ ]                                                        (13.88)
```

```
> for i to nops(allTFs) do
    onesList := [op(onesList), BeaverTuring(allTFs[i], 100)]
  end do :
```

```
> max (onesList)
    4                                                                      (13.89)
```

Using the **Tally** command from the **Statistics** package, we can see how many of the Turing machines produced tapes with four 1s.

```
> Statistics[Tally] (onesList)
    [−1 = 10 952, 0 = 4184, 1 = 4876, 2 = 704, 3 = 16, 4 = 4]              (13.90)
```

This shows us that 4184 of the machines halted with no 1s on the tape, 4 machines halted with four 1s, and 10 952 of the machines failed to halt.

We can see the four machines that produced four 1s as follows. The **SearchAll** command in **ListTools** will, given an element and a list, return the sequence of indices in the list that contain the element.

```
> ListTools[SearchAll] (4, onesList)
    7729, 7741, 9314, 9326                                                 (13.91)
```

These are the transition functions for the four machines.

```
> for i in[ListTools[SearchAll](4, onesList)] do
    eval(allTFs[i])
  end do
    table ([0, B = [1, 1, R], 1, 1 = [2, 1, L], 0, 1 = [1, 1, L], 1, B = [0, 1, L]])
    table ([0, B = [1, 1, R], 1, 1 = [2, 1, R], 0, 1 = [1, 1, L], 1, B = [0, 1, L]])
    table ([0, B = [1, 1, L], 1, 1 = [2, 1, L], 0, 1 = [1, 1, R], 1, B = [0, 1, R]])
    table ([0, B = [1, 1, L], 1, 1 = [2, 1, R], 0, 1 = [1, 1, R], 1, B = [0, 1, R]])    (13.92)
```

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

Exercise 1. Construct the unit-delay machine described in Example 5 of Section 13.2.

Exercise 2. Construct a Maple procedure 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.)

Exercise 3. Develop Maple procedures 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 text.

Exercise 4. Develop Maple procedures 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.)

Exercise 5. Construct a Maple procedure that computes the star height of a regular expression. (See Supplementary Exercise 11 for the definition of star height.)

Exercise 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.

Exercise 7. Construct a Maple procedure that simulates the action of a Turing machine that may move right, left, or not at all at each step.

Exercise 8. Construct a Maple procedure that simulates the action of a Turing machine that may have more than one tape.

Exercise 9. Construct a Maple procedure 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.