CHAPTER 9

# Turing Computable Functions

In the preceding chapter Turing machines provided the computational framework for accepting languages. The result of a computation was determined by final state or by halting. In either case there are only two possible outcomes: accept or reject. The result of a Turing machine computation can also be defined in terms of the symbols written on the tape when the computation terminates. Defining the result in terms of the halting tape configuration permits an infinite number of possible outcomes. In this manner, the computations of a Turing machine produce a mapping between input strings and output strings; that is, the Turing machine computes a function. When the strings are interpreted as natural numbers, Turing machines can be used to compute number-theoretic functions. We will show that several important number-theoretic functions are Turing computable and that computability is closed under the composition of functions. In Chapter 13 we will categorize the entire family of functions that can be computed by Turing machines.

The current chapter ends by outlining how a high-level programming language could be defined using the Turing machine architecture. This brings Turing machine computations closer to the computational paradigm with which we are most familiar—the modern-day computer.

## 9.1 Computation of Functions

A function $f : X \rightarrow Y$ is a mapping that assigns at most one value from the set Y to each element of the domain X. Adopting a computational viewpoint, we refer to the variables of $f$ as the input of the function. The definition of a function does not specify how to obtain

$f(x)$, the value assigned to $x$ by the function $f$, from the input $x$. Turing machines will be designed to compute the values of functions. The domain and range of a function computed by a Turing machine consist of strings over the input alphabet of the machine.

A Turing machine that computes a function has two distinguished states: the initial state $q_0$ and the halting state $q_f$. A computation begins with a transition from state $q_0$ that positions the tape head at the beginning of the input string. The state $q_0$ is never reentered; its sole purpose is to initiate the computation. All computations that terminate do so in state $q_f$ with the value of the function written on the tape beginning at position one. These conditions are formalized in Definition 9.1.1.

**Definition 9.1.1**

A deterministic one-tape Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ computes the unary function $f : \Sigma^* \to \Sigma^*$ if

i) there is only one transition from the state $q_0$ and it has the form $\delta(q_0, B) = [q_i, B, R]$;

ii) there are no transitions of the form $\delta(q_i, x) = [q_0, y, d]$ for any $q_i \in Q$, $x, y \in \Gamma$, and $d \in \{L, R\}$;

iii) there are no transitions of the form $\delta(q_f, B)$;

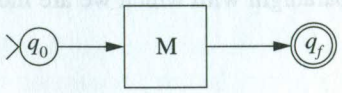
iv) the computation with input $u$ halts in the configuration $q_f BvB$ whenever $f(u) = v$; and

v) the computation continues indefinitely whenever $f(u) \uparrow$.

A function is said to be **Turing computable** if there is a Turing machine that computes it. A Turing machine that computes a function $f$ may fail to halt for an input string $u$. In this case, $f$ is undefined for $u$. Thus Turing machines can compute both total and partial functions.

An arbitrary function need not have the same domain and range. Turing machines can be designed to compute functions from $\Sigma^*$ to a specific set R by designating an input alphabet $\Sigma$ and a range R. Condition (iv) is then interpreted as requiring the string $v$ to be an element of R.

To highlight the distinguished states $q_0$ and $q_f$, a Turing machine M that computes a function is depicted by the diagram

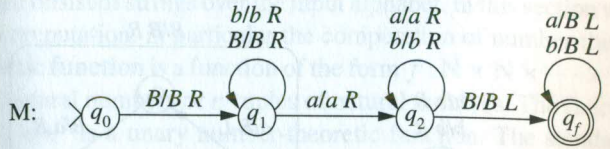



Intuitively, the computation remains inside the box labeled M until termination. This diagram is somewhat simplistic since Definition 9.1.1 permits multiple transitions to state $q_f$ and transitions from $q_f$. However, condition (iii) ensures that there are no transitions from $q_f$ when the machine is scanning a blank. When this occurs, the computation terminates with the result written on the tape.



**Example 9.1.1**

The Turing machine



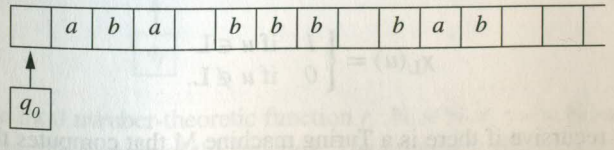

computes the partial function $f$ from $\{a, b\}^*$ to $\{a, b\}^*$ defined by

$$f(u) = \begin{cases} \lambda & \text{if } u \text{ contains an } a \\ \uparrow & \text{otherwise.} \end{cases}$$

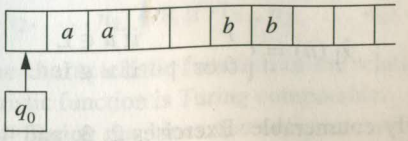
The function $f$ is undefined if the input does not contain an $a$. In this case the machine moves indefinitely to the right in state $q_1$. When an $a$ is encountered, the machine enters state $q_2$ and reads the remainder of the input. The computation is completed by erasing the input while returning to the initial position. A computation that terminates produces the configuration $q_f BB$ designating the null string as the result. □

The machine M in Example 9.1.1 was designed to compute the unary function $f$. It should be neither surprising nor alarming that computations of M do not satisfy the requirements of Definition 9.1.1 when the input does not have the anticipated form. A computation of M initiated with input $BbBbBaB$ terminates in the configuration $BbBbq_f B$. In this halting configuration, the tape does not contain a single value and the tape head is not in the correct position. This is just another manifestation of the time-honored "garbage in, garbage out" principle of computer science.

Functions with more than one argument are computed in a similar manner. The input is placed on the tape with the arguments separated by blanks. The initial configuration of a computation of a ternary function $f$ with input $aba$, $bbb$, and $bab$ is



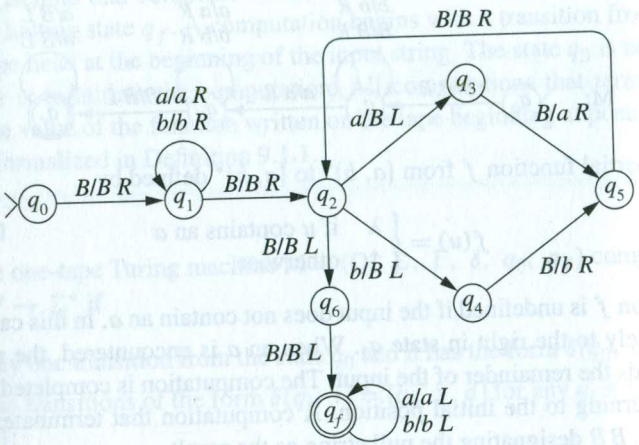

If $f(aba, bbb, bab)$ is defined, the computation terminates with the configuration $q_f Bf(aba, bbb, bab)B$. The initial configuration for the computation of $f(aa, \lambda, bb)$ is




The consecutive blanks in tape positions three and four indicate that the second argument is the null string.

**Example 9.1.2**

The Turing machine




computes the binary function of concatenation of strings over $\{a, b\}$. The initial configuration of a computation with input strings $u$ and $v$ has the form $q_0BuBvB$. Either or both of the input strings may be null.

The initial string is read in state $q_1$. The cycle formed by states $q_2$, $q_3$, $q_5$, $q_2$ translates an $a$ one position to the left. Similarly, $q_2$, $q_4$, $q_5$, $q_2$ shift a $b$ to the left. These cycles are repeated until the entire second argument has been translated one position to the left, producing the configuration $q_fBuvB$. □

Turing machines that compute functions can also be used to accept languages. The **characteristic function** of a language L is a function $\chi_L : \Sigma^* \to \{0, 1\}$ defined by

$$\chi_L(u) = \begin{cases} 1 & \text{if } u \in L \\ 0 & \text{if } u \notin L. \end{cases}$$

A language L is recursive if there is a Turing machine M that computes the characteristic function $\chi_L$. The results of the computations of M indicate the acceptability of strings. A machine that computes the partial characteristic function

$$\hat{\chi}_L(u) = \begin{cases} 1 & \text{if } u \in L \\ 0 \text{ or } \uparrow & \text{if } u \notin L \end{cases}$$

shows that L is recursively enumerable. Exercises 2, 3, and 4 establish the equivalence between acceptance of a language by a Turing machine and the computability of its characteristic function.

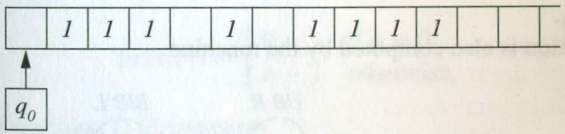


## 9.2 Numeric Computation

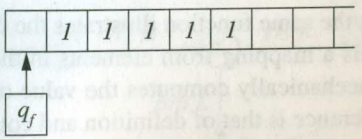
We have seen that Turing machines can be used to compute the values of functions whose domain and range consist of strings over the input alphabet. In this section we turn our attention to numeric computation, in particular the computation of number-theoretic functions. A **number-theoretic function** is a function of the form $f : \mathbf{N} \times \mathbf{N} \times \cdots \times \mathbf{N} \to \mathbf{N}$. The domain consists of natural numbers or $n$-tuples of natural numbers. The function $sq : \mathbf{N} \to \mathbf{N}$ defined by $sq(n) = n^2$ is a unary number-theoretic function. The standard operations of addition and multiplication are binary number-theoretic functions.

The transition from symbolic to numeric computation requires only a change of perspective since numbers are represented by strings of symbols. The input alphabet of the Turing machine is determined by the representation of the natural numbers used in the computation. We will represent the natural number $n$ by the string $1^{n+1}$. The number zero is represented by the string *1*, the number one by *11*, and so on. This notational scheme is known as the *unary representation* of the natural numbers. The unary representation of a natural number $n$ is denoted $\overline{n}$. When numbers are encoded using the unary representation, the input alphabet for a machine that computes a number-theoretic function is the singleton set $\{1\}$.

The computation of $f(2, 0, 3)$ in a Turing machine that computes a ternary number-theoretic function $f$ begins with the machine configuration




If $f(2, 0, 3) = 4$, the computation terminates with the configuration



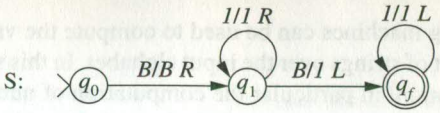

A $k$-variable total number-theoretic function $r : \mathbf{N} \times \mathbf{N} \times \cdots \times \mathbf{N} \to \{0, 1\}$ defines a $k$-ary relation R on the domain of the function. The relation is defined by

$$[n_1, n_2, \ldots, n_k] \in R \text{ if } r(n_1, n_2, \ldots, n_k) = 1$$
$$[n_1, n_2, \ldots, n_k] \notin R \text{ if } r(n_1, n_2, \ldots, n_k) = 0.$$

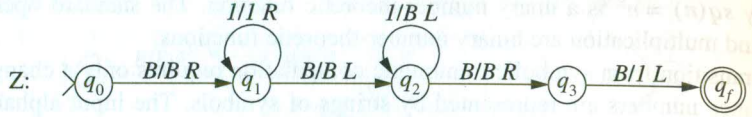
The function $r$ is called the **characteristic function** of the relation R. A relation is Turing computable if its characteristic function is Turing computable.

We will now construct Turing machines that compute several simple, but important, number-theoretic functions. The functions are denoted by lowercase letters and the corresponding machines by capital letters.

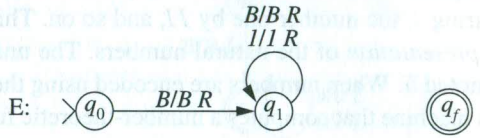
The successor function: $s(n) = n + 1$




The zero function: $z(n) = 0$

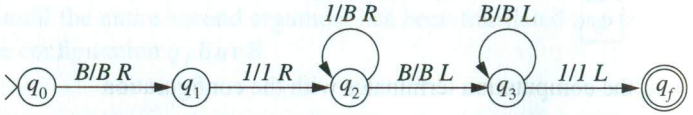



The empty function: $e(n) \uparrow$



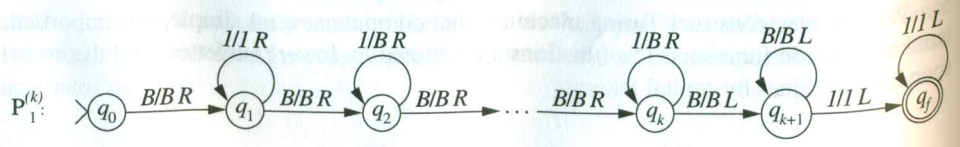

The machine that computes the successor simply adds a *1* to the right end of the input string. The zero function is computed by erasing the input and writing *1* in tape position one. The empty function is undefined for all arguments; the machine moves indefinitely to the right in state $q_1$.

The zero function is also computed by the machine




That two machines compute the same function illustrates the difference between functions and algorithms. A function is a mapping from elements in the domain to elements in the range. A Turing machine mechanically computes the value of the function whenever the function is defined. The difference is that of definition and computation. In Section 9.5 we will see that there are number-theoretic functions that cannot be computed by any Turing machine.

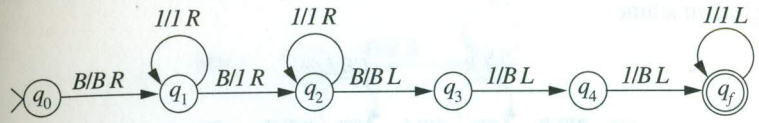
The value of the $k$-variable projection function $p_i^{(k)}$ is defined as the $i$th argument of the input, $p_i^{(k)}(n_1, n_2, \ldots, n_i, \ldots, n_k) = n_i$. The superscript $k$ specifies the number of arguments and the subscript designates the argument that defines the result of the projection. The superscript is placed in parentheses so that it is not mistaken for an exponent. The machine that computes $p_1^{(k)}$ leaves the first argument unchanged and erases the remaining arguments.




The function $p_1^{(1)}$ maps a single input to itself. This function is also called the *identity function* and is denoted $id$. Machines $P_i^{(k)}$ that compute $p_i^{(k)}$ will be designed in Example 9.3.1.

**Example 9.2.1**

The Turing machine A computes the binary function defined by the addition of natural numbers.



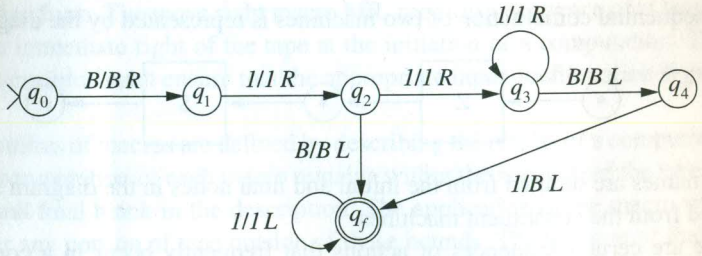

The unary representations of natural numbers $n$ and $m$ are $1^{n+1}$ and $1^{m+1}$. The sum of these numbers is represented by $1^{n+m+1}$. This string is generated by replacing the blank between the arguments with a *1* and erasing two *1*'s from the right end of the second argument. □

**Example 9.2.2**

The predecessor function

$$pred(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

is computed by the machine D (decrement):



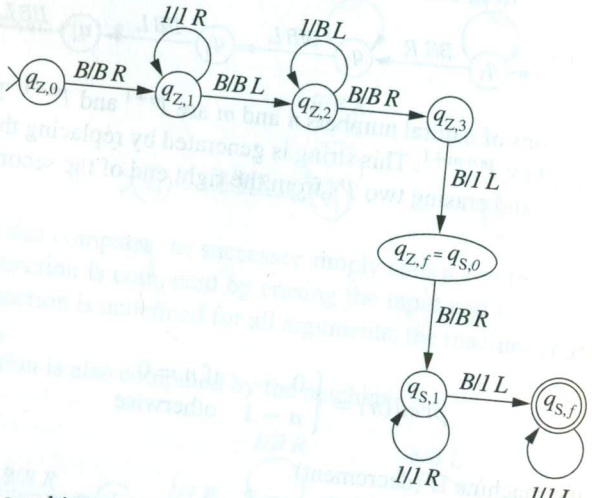

For input greater than zero, the computation erases the rightmost *1* on the tape. □

## 9.3 Sequential Operation of Turing Machines

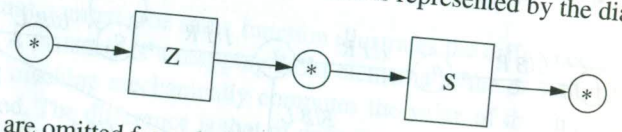
Turing machines designed to accomplish a single task can be combined to construct machines that perform complex computations. Intuitively, the combination is obtained by running the machines sequentially. The result of one computation becomes the input for the succeeding machine. A machine that computes the constant function $c(n) = 1$ can be

onstructed by combining the machines that compute the zero and the successor functions. Regardless of the input, a computation of the machine Z terminates with the value zero on the tape. Running the machine S on this tape configuration produces the number one.

The computation of Z terminates with the tape head in position zero scanning a blank. These are precisely the input conditions for the machine S. The initiation and termination conditions of Definition 9.1.1 were introduced to facilitate this coupling of machines. The handoff between machines is accomplished by identifying the final state of Z with the initial state of S. Except for this handoff, the states of the two machines are assumed to be distinct. This can be ensured by subscripting each state of the composite machine with the name of the original machine.




The sequential combination of two machines is represented by the diagram



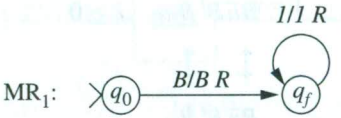

The state names are omitted from the initial and final nodes in the diagram since they may be inferred from the constituent machines.

There are certain sequences of actions that frequently occur in a computation of a Turing machine. Machines can be constructed to perform these recurring tasks. These machines are designed in a manner that allows them to be used as components in more complicated machines. Borrowing terminology from assembly language programming, we call a machine constructed to perform a single simple task a **macro**.

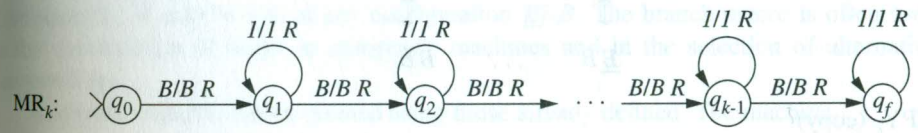
The computations of a macro adhere to several of the restrictions introduced in Definition 9.1.1. The initial state $q_0$ is used strictly to initiate the computation. Since these machines are combined to construct more complex machines, we do not assume that a computation must begin with the tape head at position zero. We do assume, however, that each computation begins with the machine scanning a blank. Depending upon the operation, the

nent of the tape to the immediate right or left of the tape head will be examined by the putation. A macro may contain several states in which a computation may terminate. with machines that compute functions, a macro is not permitted to contain a transition the form $\delta(q_f, B)$ from any halting state $q_f$.

A family of macros is often described by a schema. The macro $\mathrm{MR}_i$ moves the tape ad to the right through $i$ consecutive natural members (sequences of $1$'s) on the tape. $\mathrm{MR}_1$ defined by the machine




$\mathrm{MR}_k$ is constructed by adding states to move the tape head through the sequence of $k$ natural numbers.




The move macros do not affect the tape to the left of the initial position of the tape head. A computation of $\mathrm{MR}_2$ that begins with the configuration $B\overline{n}_1q_0B\overline{n}_2B\overline{n}_3B\overline{n}_4B$ terminates in the configuration $B\overline{n}_1B\overline{n}_2B\overline{n}_3q_fB\overline{n}_4B$.

Macros, like Turing machines that compute functions, expect to be run with the input having a specified form. The move right macro $\mathrm{MR}_i$ requires a sequence of at least $i$ natural numbers to the immediate right of the tape at the initiation of a computation. The design of a composite machine must ensure that the appropriate input configuration is provided to each macro.

Several families of macros are defined by describing the results of a computation of the machine. The computation of each macro remains within the segment of the tape indicated by the initial and final blank in the description. The application of the macro will neither access nor alter any portion of tape outside of these bounds. The location of the tape head is indicated by the underscore. The double arrows indicate identical tape positions in the before and after configurations.

$\mathrm{ML}_k$ (move left):

$$\begin{array}{ll} B\overline{n}_1B\overline{n}_2B\ldots B\underline{\overline{n}_kB} & k \geq 0 \\ \updownarrow \qquad\qquad\quad \updownarrow & \\ \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB & \end{array}$$

FR (find right):

$$\begin{array}{ll} \underline{B}B^i\overline{n}B & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ B^i\underline{B}\overline{n}B & \end{array}$$

FL (find left):

$$\begin{array}{ll} B\overline{n}B^i\underline{B} & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ \underline{B}\overline{n}B^iB & \end{array}$$

$\mathrm{E}_k$ (erase):

$$\begin{array}{lll} \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB & & k \geq 1 \\ \updownarrow \qquad\qquad\qquad \updownarrow & & \\ \underline{B}B \qquad\ldots\qquad BB & & \end{array}$$

$\mathrm{CPY}_k$ (copy):

$$\begin{array}{lll} \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kBBB \quad\ldots\quad BB & & k \geq 1 \\ \updownarrow \qquad\qquad\qquad \updownarrow \qquad\qquad \updownarrow & & \\ \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB & & \end{array}$$

$\mathrm{CPY}_{k,i}$ (copy through $i$ numbers):

$$\begin{array}{lll} \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB\overline{n}_{k+1}\ldots B\overline{n}_{k+i}BB \quad\ldots\quad BB & & k \geq 1 \\ \updownarrow \qquad\updownarrow \qquad\qquad\updownarrow \qquad\qquad\updownarrow & & \\ \underline{B}\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB\overline{n}_{k+1}\ldots B\overline{n}_{k+i}B\overline{n}_1B\overline{n}_2B\ldots B\overline{n}_kB & & \end{array}$$

T (translate):

$$\begin{array}{ll} \underline{B}B^i\overline{n}B & i \geq 0 \\ \updownarrow \quad \updownarrow & \\ \underline{B}\overline{n}B^iB & \end{array}$$

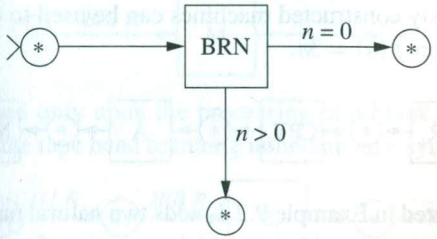
The find macros move the tape head into a position to process the first natural number to the right or left of the current position. $\mathrm{E}_k$ erases a sequence of $k$ natural numbers and halts with the tape head in its original position.

The copy machines produce a copy of the designated number of integers. The segment of the tape on which the copy is produced is assumed to be blank. $\mathrm{CPY}_{k,i}$ expects a sequence



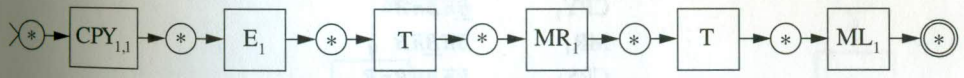
of $k+i$ numbers followed by a blank segment large enough to hold a copy of the first $k$ numbers. The translate macro changes the location of the first natural number to the right of the tape head. A computation terminates with the head in the position it occupied at the beginning of the computation with the translated string to its immediate right.

The BRN (branch on zero) macro has two possible terminating states. The input to the macro BRN, a single natural number, is used to select the halting state of the macro. The branch macro is depicted




The computation of BRN does not alter the tape nor change the position of the tape head. Consequently, it may be run in any configuration $\underline{B}\overline{n}B$. The branch macro is often used in the construction of loops in composite machines and in the selection of alternative computations.

Additional macros can be created using those already defined. The machine




interchanges the order of two numbers. The tape configurations for this macro are

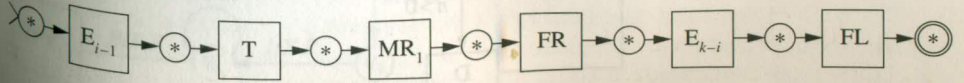
INT (interchange):

$$\begin{array}{ll} \underline{B}\overline{n}B\overline{m}BB^{n+1}B & \\ \updownarrow \qquad\qquad \updownarrow & \\ \underline{B}\overline{m}B\overline{n}BB^{n+1}B & \end{array}$$

In Exercise 6, you are asked to construct a Turing machine for the macro INT that does not leave the tape segment $B\overline{n}B\overline{m}B$.

**Example 9.3.1**

The computation of a machine that evaluates the projection function $p_i^{(k)}$ consists of three distinct actions: erasing the initial $i-1$ arguments, translating the $i$th argument to tape position one, and erasing the remainder of the input. A machine to compute $p_i^{(k)}$ can be designed using the macros FR, FL, $\mathrm{E}_i$, $\mathrm{MR}_1$, and T.

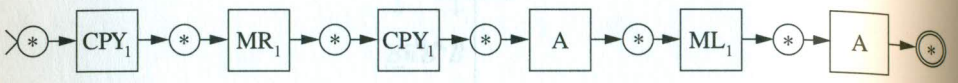



□

Turing machines defined to compute functions can be used like macros in the design of composite machines. Unlike the computations of the macros, there is no a priori bound on the amount of tape required by a computation of such a machine. Consequently, these machines should be run only when the input is followed by a completely blank tape.

**Example 9.3.2**

The macros and previously constructed machines can be used to design a Turing machine that computes the function $f(n) = 3n$.



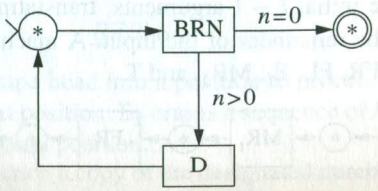

The machine A, constructed in Example 9.2.1, adds two natural numbers. The computation of $f(n)$ combines the copy macro with A to add three copies of $n$. A computation with input $\overline{n}$ generates the following sequence of tape configurations.

| Machine | Configuration |
|---|---|
| | $\underline{B}\overline{n}B$ |
| $CPY_1$ | $\underline{B}\overline{n}B\overline{n}B$ |
| $MR_1$ | $B\overline{n}\underline{B}\overline{n}B$ |
| $CPY_1$ | $B\overline{n}\underline{B}\overline{n}B\overline{n}B$ |
| A | $B\overline{n}\underline{B}\overline{n+n}B$ |
| $ML_1$ | $\underline{B}\overline{n}B\overline{n+n}B$ |
| A | $\underline{B}\overline{n+n+n}B$ |

Note that the addition machine A is run only when its arguments are the two rightmost encoded numbers on the tape. □

**Example 9.3.3**

The one-variable constant function zero defined by $z(n) = 0$, for all $n \in \mathbf{N}$, can be built from the BRN macro and the machine D that computes the predecessor function.



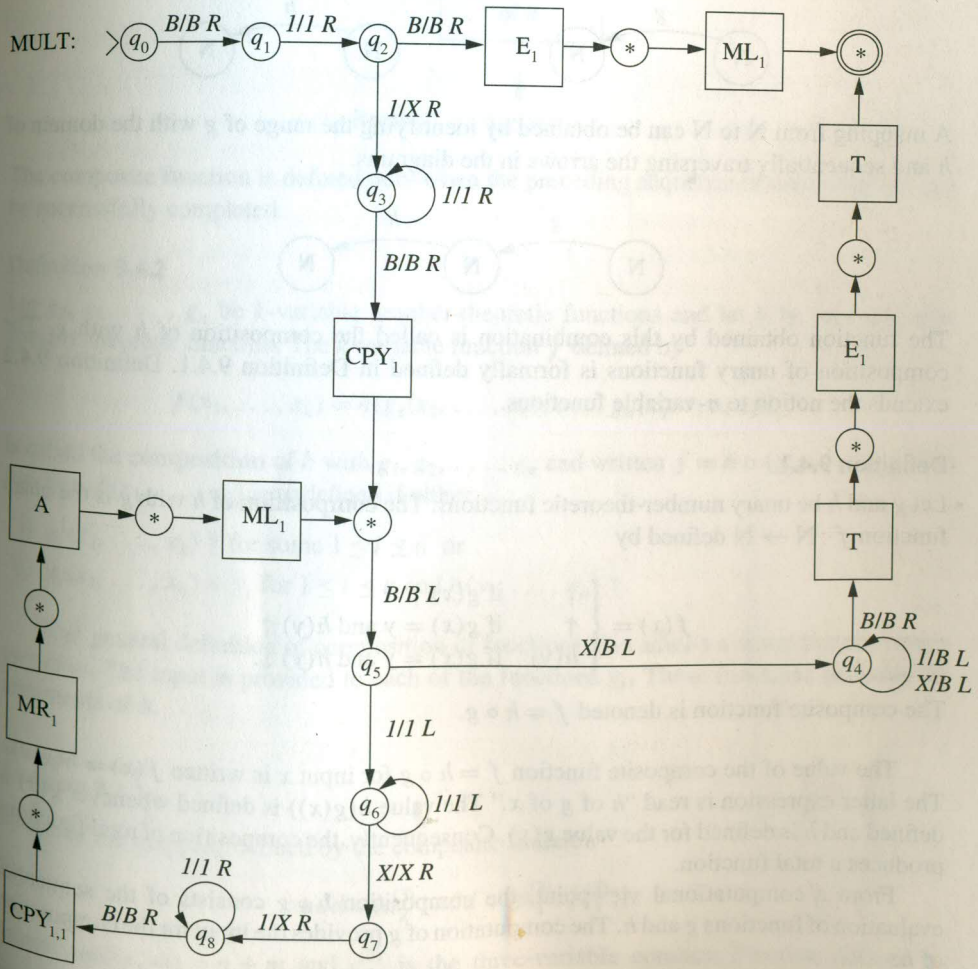

□



**Example 9.3.4**

A Turing machine MULT is constructed to compute the multiplication of natural numbers. Macros can be mixed with standard Turing machine transitions when designing a composite machine. The conditions on the initial state of a macro permit the submachine to be entered upon the processing of a blank from any state. The identification of the start state of a macro with a state $q_i$ is depicted


Since the macro is entered only upon the processing of a blank, transitions may also be defined for state $q_i$ with the tape head scanning nonblank tape symbols.

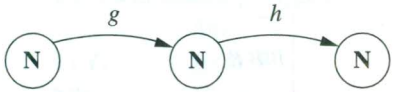


If the first argument is zero, the computation erases the second argument, returns to the initial position, and halts. Otherwise, a computation of MULT adds $m$ to itself $n$ times. The addition is performed by copying $\overline{m}$ and then adding the copy to the previous total. The number of iterations is recorded by replacing a *1* in the first argument with an *X* when a copy is made. □

## 9.4 Composition of Functions

Using the interpretation of a function as a mapping from its domain to its range, we can represent the unary number-theoretic functions $g$ and $h$ by the diagrams


A mapping from **N** to **N** can be obtained by identifying the range of $g$ with the domain of $h$ and sequentially traversing the arrows in the diagrams.




The function obtained by this combination is called the composition of $h$ with $g$. The composition of unary functions is formally defined in Definition 9.4.1. Definition 9.4.2 extends the notion to $n$-variable functions.

**Definition 9.4.1**

Let $g$ and $h$ be unary number-theoretic functions. The **composition** of $h$ with $g$ is the unary function $f : \mathbf{N} \to \mathbf{N}$ defined by

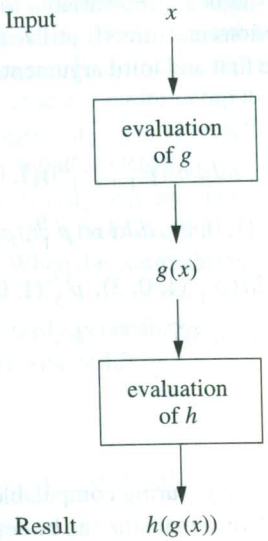
$$f(x) = \begin{cases} \uparrow & \text{if } g(x) \uparrow \\ \uparrow & \text{if } g(x) = y \text{ and } h(y) \uparrow \\ h(y) & \text{if } g(x) = y \text{ and } h(y) \downarrow. \end{cases}$$

The composite function is denoted $f = h \circ g$.

The value of the composite function $f = h \circ g$ for input $x$ is written $f(x) = h(g(x))$. The latter expression is read "$h$ of $g$ of $x$." The value $h(g(x))$ is defined whenever $g(x)$ is defined and $h$ is defined for the value $g(x)$. Consequently, the composition of total functions produces a total function.

From a computational viewpoint, the composition $h \circ g$ consists of the sequential evaluation of functions $g$ and $h$. The computation of $g$ provides the input for the computation of $h$:






The composite function is defined only when the preceding sequence of computations can be successfully completed.

**Definition 9.4.2**

Let $g_1, g_2, \ldots, g_n$ be $k$-variable number-theoretic functions and let $h$ be an $n$-variable number-theoretic function. The $k$-variable function $f$ defined by

$$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

is called the **composition** of $h$ with $g_1, g_2, \ldots, g_n$ and written $f = h \circ (g_1, \ldots, g_n)$. The function $f(x_1, \ldots, x_k)$ is undefined if either

i) $g_i(x_1, \ldots, x_k) \uparrow$ for some $1 \le i \le n$, or

ii) $g_i(x_1, \ldots, x_k) = y_i$ for $1 \le i \le n$ and $h(y_1, \ldots, y_n) \uparrow$.

The general definition of composition of functions also admits a computational interpretation. The input is provided to each of the functions $g_i$. These functions generate the arguments of $h$.

**Example 9.4.1**

Consider the mapping defined by the composite function

$$add \circ (c_2^{(3)}, add \circ (p_1^{(3)}, p_3^{(3)})),$$

where $add(n, m) = n + m$ and $c_2^{(3)}$ is the three-variable constant function defined by

$c_2^{(3)}(n_1, n_2, n_3) = 2$. The composite is a three-variable function since the innermost functions of the composition, the functions that directly utilize the input, require three arguments. The function adds the sum of the first and third arguments to the constant 2. The result for input 1, 0, 3 is

$$\begin{aligned}
&add \circ (c_2^{(3)}, add \circ (p_1^{(3)}, p_3^{(3)}))(1, 0, 3) \\
&= add \circ (c_2^{(3)}(1, 0, 3), add \circ (p_1^{(3)}, p_3^{(3)})(1, 0, 3)) \\
&= add(2,\ add(p_1^{(3)}(1, 0, 3), p_3^{(3)}(1, 0, 3))) \\
&= add(2,\ add(1, 3)) \\
&= add(2, 4) \\
&= 6. \qquad \square
\end{aligned}$$

A function obtained by composing Turing computable functions is itself Turing computable. The argument is constructive; a machine can be designed to compute the composite function by combining the machines that compute the constituent functions and the macros developed in the previous section.

Let $g_1$ and $g_2$ be three-variable Turing computable functions and let $h$ be a Turing computable two-variable function. Since $g_1$, $g_2$, and $h$ are computable, there are machines $G_1$, $G_2$, and H that compute them. The actions of a machine that computes the composite function $h \circ (g_1, g_2)$ are traced for input $n_1$, $n_2$, and $n_3$.

| Machine | Configuration |
|---|---|
| | $\underline{B}\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B$ |
| $CPY_3$ | $\underline{B}\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B \overline{n}_1 B \overline{n}_2 B \overline{n}_3 B$ |
| $MR_3$ | $B\overline{n}_1 B \overline{n}_2 B \overline{n}_3 \underline{B}\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B$ |
| $G_1$ | $B\overline{n}_1 B \overline{n}_2 B \overline{n}_3 \underline{B}\overline{g_1(n_1, n_2, n_3)} B$ |
| $ML_3$ | $\underline{B}\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B \overline{g_1(n_1, n_2, n_3)} B$ |
| $CPY_{3,1}$ | $\underline{B}\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B \overline{g_1(n_1, n_2, n_3)} B \overline{n}_1 B \overline{n}_2 B \overline{n}_3 B$ |
| $MR_4$ | $B\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B \overline{g_1(n_1, n_2, n_3)} \underline{B}\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B$ |
| $G_2$ | $B\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B \overline{g_1(n_1, n_2, n_3)} \underline{B}\overline{g_2(n_1, n_2, n_3)} B$ |
| $ML_1$ | $B\overline{n}_1 B \overline{n}_2 B \overline{n}_3 \underline{B}\overline{g_1(n_1, n_2, n_3)} B \overline{g_2(n_1, n_2, n_3)} B$ |
| H | $B\overline{n}_1 B \overline{n}_2 B \overline{n}_3 \underline{B}\overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))} B$ |
| $ML_3$ | $\underline{B}\overline{n}_1 B \overline{n}_2 B \overline{n}_3 B \overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))} B$ |
| $E_3$ | $\underline{B}B \quad \ldots \quad B\overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))} B$ |
| T | $\underline{B}\overline{h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))} B$ |



The computation copies the input and computes the value of $g_1$ using the newly created copy as the arguments. Since the machine $G_1$ does not move to the left of its starting position, the original input remains unchanged. If $g_1(n_1, n_2, n_3)$ is undefined, the computation of $G_1$ continues indefinitely. In this case the entire computation fails to terminate, correctly indicating that $h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))$ is undefined. Upon the termination of $G_1$, the input is copied and $G_2$ is run on the new copy.

If both $g_1(n_1, n_2, n_3)$ and $g_2(n_1, n_2, n_3)$ are defined, $G_2$ terminates with the input for H on the tape preceded by the original input. The machine H is run computing $h(g_1(n_1, n_2, n_3), g_2(n_1, n_2, n_3))$. When the computation of H terminates, the result is translated to the correct position.

The preceding construction easily generalizes to the composition of functions of any number of variables, yielding Theorem 9.4.3.

**Theorem 9.4.3**

The Turing computable functions are closed under the operation of composition.

Theorem 9.4.3 can be used to show that a function $f$ is Turing computable without explicitly constructing a machine that computes it. If $f$ can be defined as the composition of Turing computable functions then, by Theorem 9.4.3, $f$ is also Turing computable.

**Example 9.4.2**

The $k$-variable constant functions $c_i^{(k)}$ whose values are given by $c_i^{(k)}(n_1, \ldots, n_k) = i$ are Turing computable. The function $c_i^{(k)}$ can be defined by

$$c_i^{(k)} = \underbrace{s \circ s \circ \cdots \circ s}_{i \text{ times}} \circ z \circ p_1^{(k)}.$$

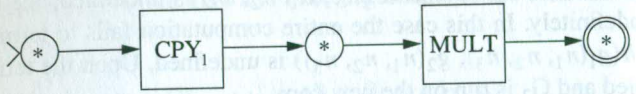
The projection function accepts the $k$-variable input and passes the first value to the zero function. The composition of $i$ successor functions produces the desired value. Since each of the functions in the composition is Turing computable, the function $c_i^{(k)}$ is Turing computable by Theorem 9.4.3. $\square$

**Example 9.4.3**

The binary function $smsq(n, m) = n^2 + m^2$ is Turing computable. The sum-of-squares function can be written as the composition of functions

$$smsq = add \circ (sq \circ p_1^{(2)},\ sq \circ p_2^{(2)}),$$

where $sq$ is defined by $sq(n) = n^2$. The function $add$ is computed by the machine constructed in Example 9.2.1 and $sq$ by

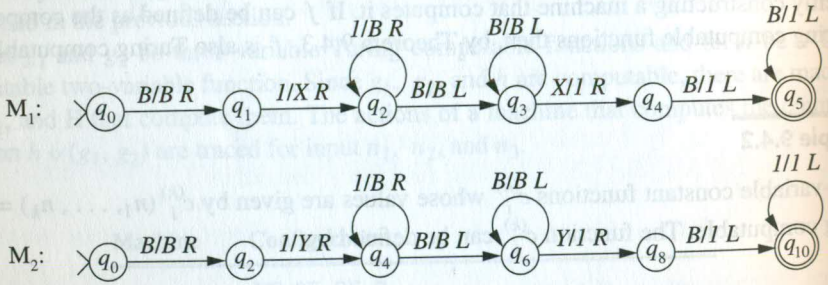



□

## 9.5 Uncomputable Functions

A function is Turing computable only if there is a Turing machine that computes it. The existence of number-theoretic functions that are not Turing computable can be demonstrated by a simple counting argument. We begin by showing that the set of computable functions is countably infinite.

A Turing machine is completely defined by its transition function. The states and tape alphabet used in computations of the machine can be extracted from the transitions. Consider the machines $M_1$ and $M_2$ defined by




Both $M_1$ and $M_2$ compute the unary constant function $c_1^{(1)}$. The two machines differ only in the names given to the states and the markers used during the computation. These symbols have no effect on the result of a computation and hence the function computed by the machine.

Since the names of the states and tape symbols other than $B$ and $1$ are immaterial, we adopt the following conventions concerning the naming of the components of a Turing machine:

i) The set of states is a finite subset of $Q_0 = \{q_i \mid i \geq 0\}$.

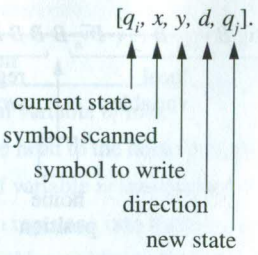
ii) The input alphabet is $\{1\}$.

iii) The tape alphabet is a finite subset of the set $\Gamma_0 = \{B, 1, X_i \mid i \geq 0\}$.

iv) The initial state is $q_0$.

The transitions of a Turing machine have been specified using functional notation; the transition defined for state $q_i$ and tape symbol $x$ is represented by $\delta(q_i, x) = [q_j, y, d]$. This information can also be represented by the quintuple






With the preceding naming conventions, a transition of a Turing machine is an element of the set $T = Q_0 \times \Gamma_0 \times \Gamma_0 \times \{L, R\} \times Q_0$. The set T is countable since it is the Cartesian product of countable sets.

The transitions of a deterministic Turing machine form a finite subset of T in which the first two components of every element are distinct. There are only a countable number of such subsets. It follows that the number of Turing computable functions is at most countably infinite. On the other hand, the number of Turing computable functions is at least countably infinite since there are countably many constant functions, all of which are Turing computable by Example 9.4.2. These observations yield

**Theorem 9.5.1**

The set of Turing computable number-theoretic functions is countably infinite.

In Section 1.4, the diagonalization technique was used to prove that there are uncountably many total unary number-theoretic functions. Combining this with Theorem 9.5.1, we obtain Corollary 9.5.2.

**Corollary 9.5.2**

There is a total unary number-theoretic function that is not Turing computable.

Corollary 9.5.2 vastly understates the relationship between computable and uncomputable functions. The former constitute a countable set and the latter an uncountable set.

## 9.6 Toward a Programming Language

High-level programming languages are the most commonly employed type of computational system. A program defines a mechanistic and deterministic process, the hallmark of algorithmic computation. The intuitive argument that the computation of a program written in a programming language and executed on a computer can be simulated by a Turing machine rests in the fact that a machine (computer) instruction simply changes the bits in some location of memory. This is precisely the type of action performed by a Turing machine, writing $0$'s and $1$'s in memory. Although it may take a large number of Turing machine transitions to accomplish the task, it is not at all difficult to envision a sequence of transitions that will access the correct position and rewrite the memory.

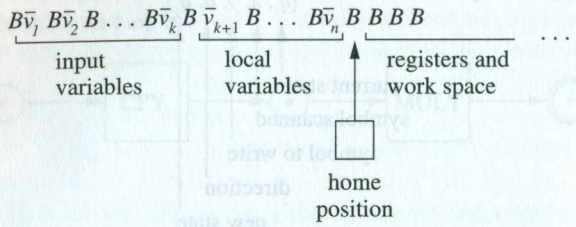



**FIGURE 9.1** Turing machine architecture for high-level computation.

In this section we will explore the possibility of using the Turing machine architecture as the underlying framework for high-level programming. The development of a programming language based on the Turing machine architecture further demonstrates the power of the Turing machine model. In describing our assembly language, we use Turing machines and macros to define the operations. The objective of this section is not to create a functional assembly language, but rather to demonstrate further the universality of the Turing machine architecture.

The standard Turing machine provides the computational framework used throughout this section. We will design an assembly language TM to bridge the gap between the Turing machine architecture and programming languages. The first objective of the assembly language is to provide a sequential description of the actions of the Turing machine. The "program flow" of a Turing machine is determined by the arcs in the state diagram of the machine. The flow of an assembly language program consists of the sequential execution of the instructions unless this pattern is specifically altered by an instruction that redirects the flow. In assembly language, branch and goto instructions are used to alter sequential program flow. The second objective of the assembly language is to provide instructions that simplify memory management.

The underlying architecture of the Turing machine used to evaluate an assembly language program is pictured in Figure 9.1. The input values are assigned to variables $v_1, \ldots, v_k$, and $v_{k+1}, \ldots, v_n$ are the local variables used in the program. The values of the variables are stored sequentially and separated by blanks. The input variables are in the standard input position for a Turing machine evaluating a function. A TM program begins by declaring the local variables used in the program. Each local variable is initialized to 0 at the start of a computation.

When the initialization is complete, the tape head is stationed at the blank separating the variables from the remainder of the tape. This will be referred to as the *home position*. Between the evaluation of instructions, the tape head returns to the home position. To the right of the home position is the Turing machine version of registers. The first value to the right is considered to be in register 1, the second value in register 2, and so on. The registers must be assigned sequentially; that is, register $i$ may be written to or read from



**TABLE 9.1** TM Instructions

| TM Instruction | Interpretation |
|---|---|
| INIT $v_i$ | Initialize local variable $v_i$ to 0. |
| HOME $t$ | Move the tape head to the home position when $t$ variables are allocated. |
| LOAD $v_i$, $t$ | Load value of variable $v_i$ into register $t$. |
| STOR $v_i$, $t$ | Store value in register $t$ into location of $v_i$. |
| RETURN $v_i$ | Erase the variables and leave the value of $v_i$ in the output position. |
| CLEAR $t$ | Erase value in register $t$. |
| BRN L, $t$ | Branch to instruction labeled L if value in register $t$ is 0. |
| GOTO L | Execute instruction labeled L. |
| NOP | No operation (used in conjunction with GOTO commands). |
| INC $t$ | Increment the value of register $t$. |
| DEC $t$ | Decrement the value of register $t$. |
| ZERO $t$ | Replace value in register $t$ with 0. |

only if registers 1, 2, . . . , $i - 1$ are assigned values. The instructions of the language TM are given in Table 9.1.

The tape initialization is accomplished using the INIT and HOME commands. INIT $v_i$ reserves the location for local variable $v_i$ and initializes the value to 0. Since variables are stored sequentially on the tape, local variables must be initialized in order at the beginning of a TM program. Upon completion of the initialization of the local variables, the HOME instruction moves the tape head to the home position. These instructions are defined by

| Instruction | Definition |
|---|---|
| INIT $v_i$ | $MR_{i-1}$<br>ZR<br>$ML_{i-1}$ |
| HOME $t$ | $MR_t$ |

where ZR is the macro that writes the value 0 to the immediate right of the tape head position (Exercise 6). The initialization phase of a program with one input and two local variables would produce the following sequence of Turing machine configurations:

| Instruction | Configuration |
|---|---|
| | $\underline{B\bar{i}}B$ |
| INIT 2 | $\underline{B\bar{i}}B\bar{0}B$ |
| INIT 3 | $\underline{B\bar{i}}B\bar{0}B\bar{0}B$ |
| HOME 3 | $B\bar{i}B\bar{0}B\bar{0}\underline{B}$ |

where $i$ is the value of the input to the computation. The position of the tape head is indicated by the underscore.

In TM, the LOAD and STOR instructions are used to access and store the values of the variables. The objective of these instructions is to make the details of memory management transparent to the user. In Turing machines there is no upper bound to the amount of tape that may be required to store the value of a variable. The lack of a preassigned limit to the amount of tape allotted to each variable complicates the memory management of a Turing machine. This omission, however, is intentional, allowing maximum flexibility in Turing machine computations. Assigning a fixed amount of memory to a variable, the standard approach used by conventional compilers, causes an overflow error when the memory required to store a value exceeds the preassigned allocation.

The STOR command takes the value from register $t$ and stores it in the specified variable location. The command may be used only when $t$ is the largest register that has an assigned value. In storing the value of register $t$ in a variable $v_i$, the proper spacing is maintained for all the variables. The Turing machine implementation of the store command utilizes the macro INT to move the value in the register to the proper position. The macro INT is assumed to stay within the tape segment $\underline{B\bar{x}}B\bar{y}B$ (Exercise 6).

The STOR command is defined by

| Instruction | Definition |
|---|---|
| STOR $v_i$, 1 | $\begin{pmatrix}\text{ML}_1\\ \text{INT}\end{pmatrix}^{n-i+1}$ |
| | $\begin{pmatrix}\text{MR}_1\\ \text{INT}\end{pmatrix}^{n-i}$ |
| | $\text{MR}_1$ |
| | $\text{ER}_1$ |

| Instruction | Definition |
|---|---|
| STOR $v_i$, $t$ | $\text{MR}_{t-2}$ |
| | INT |
| | $\begin{pmatrix}\text{ML}_1\\ \text{INT}\end{pmatrix}^{t+n-i-1}$ |
| | $\begin{pmatrix}\text{MR}_1\\ \text{INT}\end{pmatrix}^{t+n-i-1}$ |
| | $\text{MR}_1$ |
| | $\text{ER}_1$ |
| | $\text{ML}_{t-1}$ |



where $t > 1$ and $n$ is the total number of input and local variables. The exponents $n - i + 1$ and $n - i$ indicate repetition of the sequence of macros. After the value of register $t$ is stored, the register is erased.

The configurations of a Turing machine obtained by the execution of the instruction STOR $v_2$, 1 are traced to show the role of the macros in TM memory management. Prior to the execution of the instruction, the tape head is at the home position.

| Machine | Configuration |
|---|---|
| | $B\bar{v}_1B\bar{v}_2B\bar{v}_3\underline{B\bar{r}}B$ |
| $\text{ML}_1$ | $B\bar{v}_1B\bar{v}_2\underline{B\bar{v}_3}B\bar{r}B$ |
| INT | $B\bar{v}_1B\bar{v}_2\underline{B\bar{r}}B\bar{v}_3B$ |
| $\text{ML}_1$ | $B\bar{v}_1\underline{B\bar{v}_2}B\bar{r}B\bar{v}_3B$ |
| INT | $B\bar{v}_1\underline{B\bar{r}}B\bar{v}_2B\bar{v}_3B$ |
| $\text{MR}_1$ | $B\bar{v}_1B\bar{r}\underline{B\bar{v}_2}B\bar{v}_3B$ |
| INT | $B\bar{v}_1B\bar{r}\underline{B\bar{v}_3}B\bar{v}_2B$ |
| $\text{MR}_1$ | $B\bar{v}_1B\bar{r}B\bar{v}_3\underline{B\bar{v}_2}B$ |
| $\text{E}_1$ | $B\bar{v}_1B\bar{r}B\bar{v}_3\underline{B}B$ |

The Turing machine implementation of the LOAD instruction simply copies the value of variable $v_i$ to the specified register.

| Instruction | Definition |
|---|---|
| LOAD $v_i$, $t$ | $\text{ML}_{n-i+1}$ |
| | $\text{CPY}_{1,n-i+1+t}$ |
| | $\text{MR}_{n-i+1}$ |

As previously mentioned, to load a value into register $t$ requires registers 1, 2, . . . , $t - 1$ to be filled. Thus the Turing machine must be in configuration

$$B\bar{v}_1\,B\bar{v}_2\,B\ldots B\bar{v}_k\,B\bar{v}_{k+1}B\ldots B\bar{v}_n\,B\bar{r}_1\,B\bar{r}_2\,B\ldots B\bar{r}_{t-1}\,B$$

for the instruction LOAD $v_i$, $t$ to be executed.

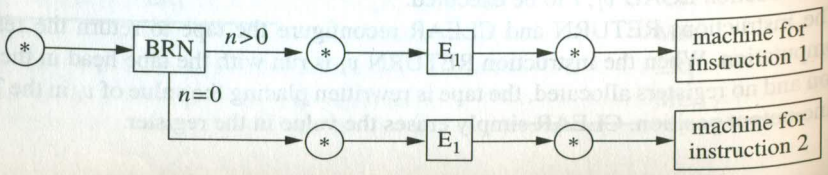
The instructions RETURN and CLEAR reconfigure the tape to return the result of the computation. When the instruction RETURN $v_i$ is run with the tape head in the home position and no registers allocated, the tape is rewritten placing the value of $v_i$ in the Turing machine output position. CLEAR simply erases the value in the register.

| Instruction | Definition |
|---|---|
| RETURN $v_i$ | $ML_n$ |
| | $E_{i-1}$ |
| | T |
| | $MR_1$ |
| | FR |
| | $E_{n-i+1}$ |
| | FL |
| CLEAR $t$ | $MR_{t-1}$ |
| | $E_1$ |
| | $ML_{t-1}$ |

Arithmetic operations alter the values in the registers. INC, DEC, and ZERO are defined by the machines computing the successor, predecessor, and zero functions. Additional arithmetic operations may be defined for our assembly language by creating a Turing machine that computes the operation. For example, an assembly language instruction ADD could be defined using the Turing machine implementation of addition given by the machine A in Example 9.2.1. The resulting instruction ADD would add the values in registers 1 and 2 and store the result in register 1. While we could greatly increase the number of assembly language instructions by adding additional arithmetic operations, INC, DEC, and ZERO will be sufficient for purposes of developing our language.

The execution of assembly language instructions consists of the sequential operation of the Turing machines and macros that define each of the instructions. The BRN and GOTO instructions interrupt the sequential evaluation by explicitly specifying the next instruction to be executed. GOTO L indicates that the instruction labeled L is the next to be executed. BRN L,$t$ tests register $t$ before indicating the subsequent instruction. If the register is nonzero, the instruction immediately following the branch is executed. Otherwise, the statement labeled by L is executed. The Turing machine implementation of the branch is illustrated by

```
        BRN L,1
        "instruction 1"
        ⋮
L       "instruction 2"
```






The value is tested, the register erased, and the machines that define the appropriate instruction are then executed.

**Example 9.6.1**

The TM program with one input variable and two local variables defined below computes the function $f(n) = 2n + 1$. The input variable is $v_1$ and the computation uses local variables $v_2$ and $v_3$.

```
        INIT v2
        INIT v3
        HOME 3
        LOAD v1,1
        STOR v2, 1
L1      LOAD v2,1
        BRN L2,1
        LOAD v1,1
        INC
        STOR v1, 1
        LOAD v2, 1
        DEC
        STOR v2, 1
        GOTO L1
L2      LOAD v1, 1
        INC
        STOR v1, 1
        RETURN v1
```

The variable $v_2$ is used as a counter, which is decremented each time through the loop defined by the label L1 and the GOTO instruction. In each iteration, the value of $v_1$ is incremented. The loop is exited after $n$ iterations, where $n$ is the input. Upon exiting the loop, the value is incremented again and the result $2v_1 + 1$ is left on the tape. □

The objective of constructing the TM assembly language is to show that instructions of Turing machines, like those of conventional machines, can be formulated as commands in a higher-level language. Utilizing the standard approach to programming language definition and compilation, the commands of a high-level language may be defined by a sequence of the assembly language instructions. This would bring Turing machine computations even closer in form to the algorithmic systems most familiar to many of us.

## Exercises

1. Construct Turing machines with input alphabet $\{a, b\}$ that compute the specified functions. The symbols $u$ and $v$ represent arbitrary strings over $\{a, b\}^*$.
   a) $f(u) = aaa$
   b) $f(u) = \begin{cases} a & \text{if } length(u) \text{ is even} \\ b & \text{otherwise} \end{cases}$
   c) $f(u) = u^R$
   d) $f(u, v) = \begin{cases} u & \text{if } length(u) > length(v) \\ v & \text{otherwise} \end{cases}$

2. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_f)$ be a Turing machine that computes the partial characteristic function of the language L. Use M to build a standard Turing machine that accepts L.

3. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a standard Turing machine that accepts a language L. Construct a machine $M'$ that computes the partial characteristic function of L. Recall that the tape of $M'$ must have the form $q_f B0B$ or $q_f B1B$ upon the completion of a computation of $\hat{\chi}_L$.

4. Let L be a language over $\Sigma$ and let

$$\chi_L(w) = \begin{cases} 1 & \text{if } w \in L \\ 0 & \text{otherwise} \end{cases}$$

be the characteristic function of L.
   a) If $\chi_L$ is Turing computable, prove that L is recursive.
   b) If L is recursive, prove that there is a Turing machine that computes $\chi_L$.

5. Construct Turing machines that compute the following number-theoretic functions and relations. Do not use macros in the design of these machines.
   a) $f(n) = 2n + 3$
   b) $half(n) = \lfloor n/2 \rfloor$ where $\lfloor x \rfloor$ is the greatest integer less than or equal to $x$
   c) $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$
   d) $even(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{otherwise} \end{cases}$
   e) $eq(n, m) = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{otherwise} \end{cases}$
   f) $lt(n, m) = \begin{cases} 1 & \text{if } n < m \\ 0 & \text{otherwise} \end{cases}$



   g) $n \dot{-} m = \begin{cases} n - m & \text{if } n \geq m \\ 0 & \text{otherwise} \end{cases}$

6. Construct Turing machines that perform the actions specified by the following macros. The computation should not leave the segment of the tape specified in the input configuration.
   a) ZR; input $\underline{B}BB$, output $\underline{B}\bar{0}B$
   b) FL; input $B\bar{n}B^i\underline{B}$, output $\underline{B}\bar{n}B^iB$
   c) $E_2$; input $\underline{B}\bar{n}B\bar{m}B$, output $\underline{B}B^{n+m+3}B$
   d) T; input $\underline{B}B^i\bar{n}B$, output $\underline{B}\bar{n}B^iB$
   e) BRN; input $\underline{B}\bar{n}B$, output $\underline{B}\bar{n}B$
   f) INT; input $\underline{B}\bar{n}B\bar{m}B$, output $\underline{B}\bar{m}B\bar{n}B$

7. Use the macros and machines constructed in Sections 9.2 through 9.4 to design machines that compute the following functions:
   a) $f(n) = 2n + 3$
   b) $f(n) = n^2 + 2n + 2$
   c) $f(n_1, n_2, n_3) = n_1 + n_2 + n_3$
   d) $f(n, m) = m^3$
   e) $f(n_1, n_2, n_3) = n_2 + 2n_3$

8. Design machines that compute the following relations. You may use the macros and machines constructed in Sections 9.2 through 9.4 and the machines constructed in Exercise 5.
   a) $gt(n, m) = \begin{cases} 1 & \text{if } n > m \\ 0 & \text{otherwise} \end{cases}$
   b) $persq(n) = \begin{cases} 1 & \text{if } n \text{ is a perfect square} \\ 0 & \text{otherwise} \end{cases}$
   c) $divides(n, m) = \begin{cases} 1 & \text{if } n > 0,\ m > 0, \text{ and } m \text{ divides } n \\ 0 & \text{otherwise} \end{cases}$

9. Trace the actions of the machine MULT for computations with input
   a) $n = 0, m = 4$
   b) $n = 1, m = 0$
   c) $n = 2, m = 2$.

10. Describe the mapping defined by each of the following composite functions:
   a) $add \circ (mult \circ (id, id), add \circ (id, id))$
   b) $p_1^{(2)} \circ (s \circ p_1^{(2)}, e \circ p_2^{(2)})$
   c) $mult \circ (c_2^{(3)}, add \circ (p_1^{(3)}, s \circ p_2^{(3)}))$
   d) $mult \circ (mult \circ (p_1^{(1)}, p_1^{(1)}), p_1^{(1)})$.

11. Give examples of total unary number-theoretic functions that satisfy the following conditions:
    a) $g$ is not $id$ and $h$ is not $id$ but $g \circ h = id$.
    b) $g$ is not a constant function and $h$ is not a constant function but $g \circ h$ is a constant function.
12. Give examples of unary number-theoretic functions that satisfy the following conditions:
    a) $g$ is not one-to-one, $h$ is not total, $h \circ g$ is total.
    b) $g \neq e$, $h \neq e$, $h \circ g = e$, where $e$ is the empty function.
    c) $g \neq id$, $h \neq id$, $h \circ g = id$, where $id$ is the identity function.
    d) $g$ is total, $h$ is not one-to-one, $h \circ g = id$.

* 13. Let F be a Turing machine that computes a total unary number-theoretic function $f$. Design a machine that returns the first natural number $n$ such that $f(n) = 0$. A computation should continue indefinitely if no such $n$ exists. What will happen if the function computed by F is not total?

14. Let F be a Turing machine that computes a total unary number-theoretic function $f$. Design a machine that computes the function

$$g(n) = \sum_{i=0}^{n} f(i).$$

15. Let F and G be Turing machines that compute total unary number-theoretic functions $f$ and $g$, respectively. Design a Turing machine that computes the function

$$h(n) = \sum_{i=0}^{n} eq(f(i), g(i)).$$

That is, $h(n)$ is the number of values in the range 0 to $n$ for which the functions $f$ and $g$ assume the same value.

16. A unary relation R over **N** is Turing computable if its characteristic function is computable. Prove that every computable unary relation over **N** defines a recursive language. *Hint:* Construct a machine that accepts R from the machine that computes its characteristic function.

* 17. Let $R \subseteq \{I\}^+$ be a recursive language. Prove that R defines a computable unary relation over **N**.

18. Prove that there are unary relations over **N** that are not Turing computable.

19. Let F be the set consisting of all total unary number-theoretic functions that satisfy $f(i) = i$ for every even natural number $i$. Prove that there are functions in F that are not Turing computable.



20. Let $v_1$, $v_2$, $v_3$, $v_4$ be a listing of the variables used in a TM program and assume register 1 contains a value. Trace the action of the instruction STOR $v_2$,1. To trace the actions, use the technique in Example 9.3.2.

21. Give a TM program that computes the function $f(v_1, v_2) = v_1 \dot{-} v_2$.

## Bibliographic Notes

The Turing machine assembly language provides an architecture that resembles another family of abstract computing devices known as random access machines [Cook and Reckhow, 1973]. Random access machines consist of an infinite number of memory locations and a finite number of registers, each of which is capable of storing a single integer. The instructions of a random access machine manipulate the registers and memory and perform arithmetic operations. These machines provide an abstraction of the standard von Neumann computer architecture. An introduction to random access machines and their equivalence to Turing machines can be found in Aho, Hopcroft, and Ullman [1974].