Phil 320
<u>Chapter 3: Turing computability</u>

1. Effective computability

Informal definition: A function f is *effectively computable* if there are definite, explicit and 'mechanical' instructions for computing each value of f. We ignore physical limitations on time, speed, and storage.

Q. 1: How do we make the notion of computability precise? [Turing-computability]
Q. 2: What are some examples of computable functions? [several in this chapter]
Q. 3: Are there non-computable functions, and can examples be provided? [ch. 4]

Turing's thesis: any effectively computable function is Turing-computable.

2. Numbering systems

Hindu-Arabic numerals: 0,1,2,3, …
Roman numerals: I, II, III, IV, V, …, X, …, L, etc.
Monadic or block numbering: 1, 11, 111, 1111, …

The choice of numbering system (for arguments and values of f) makes no difference to whether f is computable, provided there is an effective [mechanical] procedure for converting from one numbering system to another. We use mainly monadic numbering for Turing computability.

3. Turing Machines

a) Tape: marked into squares; endless in both directions; all but finitely many squares are blank at any stage. The Turing machine is located in one square at each step.

b) Symbols:

- a finite set S_0, S_1, …, S_n. [BBJ: just S_0 and S_1]
- use S_0 (or B or 0) for blank squares and 1 for S_1
- exactly one symbol is printed on each square

c) Actions at each step:

- machine *scans* the current square and reads the printed symbol
- machine is in one of finitely many *internal states* q_1, …, q_m
- conditional on the *current symbol scanned* and the *current internal state*, the machine performs one *overt action* and one *covert action*

  a) *Overt actions*: (1) Halt the computation; (2) L: move one square left; (3) R: move one square right; (4) S_0: write S_0 in place of what is there; …; (n+4) S_n: write S_n in place of what is there. [BBJ: only S_0 and S_1.]

  b) *Covert action*: Assign a new internal state.

Unless stated otherwise, machines always start in the lowest state, q_1.

4. Representation and examples

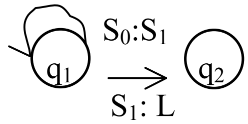
To represent a Turing machine: i) machine table; ii) flow graph; iii) set of quadruples

Example 1: Write S_1 in the current square, move left, and halt. (Tape is initially blank.)

i) Machine table

Curr. State	Scanned symbol S_0	S_1
q_1	S_1q_1	Lq_2

ii) Flow graph

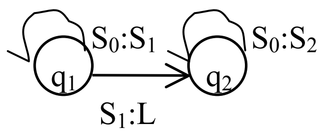


iii) Quadruples

$q_1S_0S_1q_1$ $q_1S_1Lq_2$

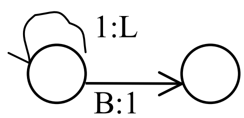
Example 2: Initially blank type. Write S_1, move left, write S_2, and then halt.

	S_0	S_1	S_2
q_1	S_1q_1	Lq_2	-*
q_2	S_2q_2	-*	-



$q_1S_0S_1q_1$ $q_1S_1Lq_2$
$q_2S_0S_2q_2$

Example 3: Tape has a continuous string of 1's, 0's everywhere else. Starting at the rightmost 1, write one additional 1 to the left, and then halt.



Convention: omit q_i; work L-R

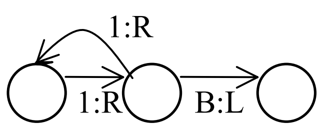
Configurations:

We can trace the progress of a Turing machine computation by writing down a sequence of configurations – snapshots of the computation in progress. A *configuration* lists what is on the tape (the rest will be blank) and indicates both the square currently scanned and the current internal state.

Example: $\underset{2}{0}110$ [in state 2; tape must be otherwise blank]

Example 4: Start at leftmost of a string of 1's on otherwise blank tape. Halt on a "1" if odd number of 1's, "blank" if even. (Use B for S_0, 1 for S_1.)

	B	1
q_1	-	R:q_2
q_2	L:q_3	R:q_1
q_3	-	-



5. Turing-computability

i) Monadic Notation. To represent one number a on a tape, we use a block of a 1's, with B everywhere else.
To represent several numbers $a_1, a_2, \ldots, a_n$, we use blocks separated by a single blank: a_1 1's, B, a_2 1's, B, …, B, a_n 1's. So to represent the triple (3, 5, 7), our tape would be:

…BBB111B11111B1111111BBB…

ii) Definition of a (Turing-)computable function

All such functions are defined using Turing machines that read and write only S_0/B/0 and S_1/1.

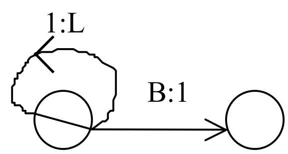
a) Functions with one argument

A Turing machine in internal state 1, scanning the leftmost 1 in a block of 1's on an otherwise blank tape, is said to be in *standard starting position* (s.s.p.); the same set-up (omitting internal state 1) is also *standard final position* (s.f.p.).

A Turing machine M *defines* (or *computes*) a function f:P → P. For each positive integer a:

- f(a) = b if, when M starts in s.s.p. scanning leftmost of a 1's, M halts in s.f.p. scanning the leftmost of b 1's.
- f(a) is *undefined* if, when M starts in s.s.p. scanning leftmost of a 1's, M either does not halt in s.f.p. or it does not halt at all.

Example: Let M be the machine



M defines the function
f(a) = a + 1

Definition: f:P → P is *Turing-computable* if there is some Turing machine (that reads only B and 1) that defines f.

b) Functions with more than one argument

A Turing machine is in s.s.p. if scanning the *leftmost* 1 in the *leftmost* block of 1's on a tape that has a finite number of continuous blocks of 1's, each separated from the next by a single blank. The machine is in standard final position (s.f.p.) if scanning the leftmost of a single block of 1's on an othewise blank tape.

A Turing machine M *defines* (or *computes*) an n-place function f:P^n → P:

- f($a_1, a_2, \ldots, a_n$) = b if, when M starts in s.s.p. scanning leftmost 1 on a tape containing blocks of a_1 1's, a_2 1's, …, a_n 1's in that order (separated by B), M halts in s.f.p. scanning leftmost of b 1's.
- f($a_1, a_2, \ldots, a_n$) is *undefined* if, starting this way, M either does not halt in s.f.p. at leftmost of a single block of 1's, or M does not halt at all.

Example: The above machine M defines the functions:
f(x) = x + 1; f(x, y) = undefined; f(x, y, z) = undefined; etc.

The monadic adder (discussed in class) defines the function
f(x) = x; f(x, y) = $x + y$; f(x, y, z) = undefined, etc.

Note: Every Turing machine M defines one n-place function for each n.