CSCI 340: Computational Models

Computers

Chapter 25

Department of Computer Science

Defining a Computer

- Finite automata are *language acceptors*
- FAs with Output (Mealy, Moore machines) are *tranducers*
- Pushdown automata — *language acceptors*
- Turing machines have a natural output — what is on the TAPE
  - Sometimes the TAPE is only a scratchpad
  - But we can use TMs for a completely different purpose

Idea: use a TM as a "calculator" of sorts

The code for $0 = \lambda$

The code for $1 = a$

The code for $2 = aa$

The code for $3 = aaa$

This is called **unary encoding**. Numbers can be separated with b's

Decoding and "Calculators"

- Every word in $(\mathbf{a} + \mathbf{b})^*$ can be interpreted as a sequence:

  Example: *bbabbaa*

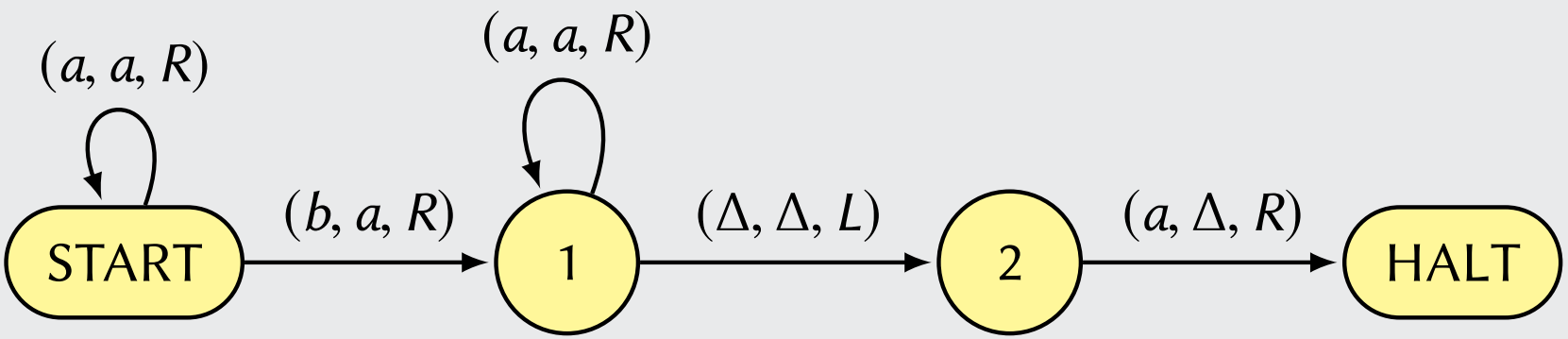
  (no *a*'s) *b* (no *a*'s) *b* (one *a*) *b* (no *a*'s) *b* (two *a*'s)

  0, 0, 1, 0, 2

- Whatever the TM leaves on the TAPE is deemed *output*.

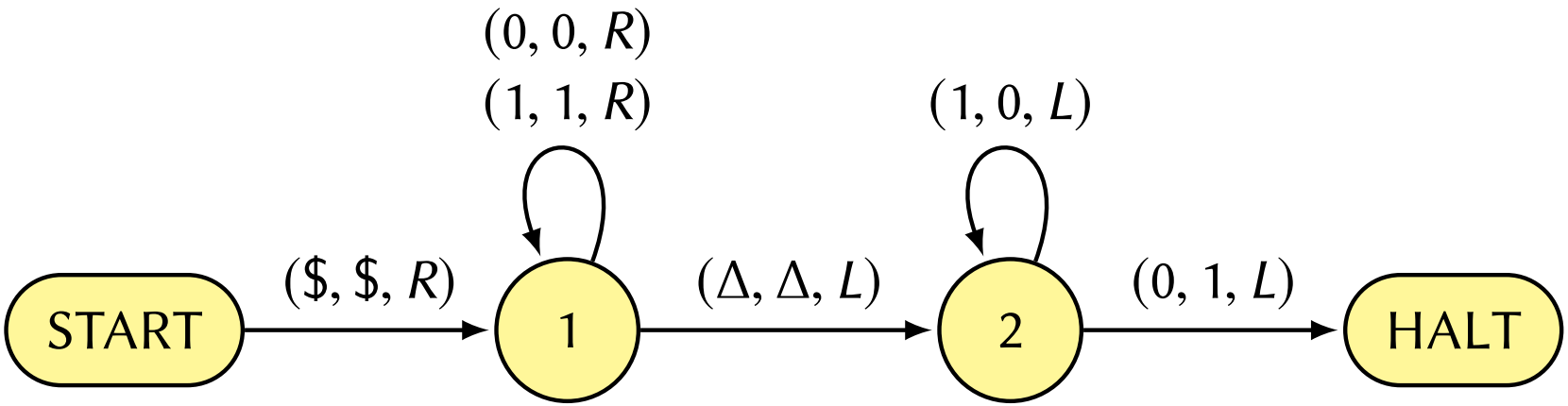
Example

Consider an ADDER:



A Base-2 Adder — Part 1

A simple "incrementer" shall be known as T_1:

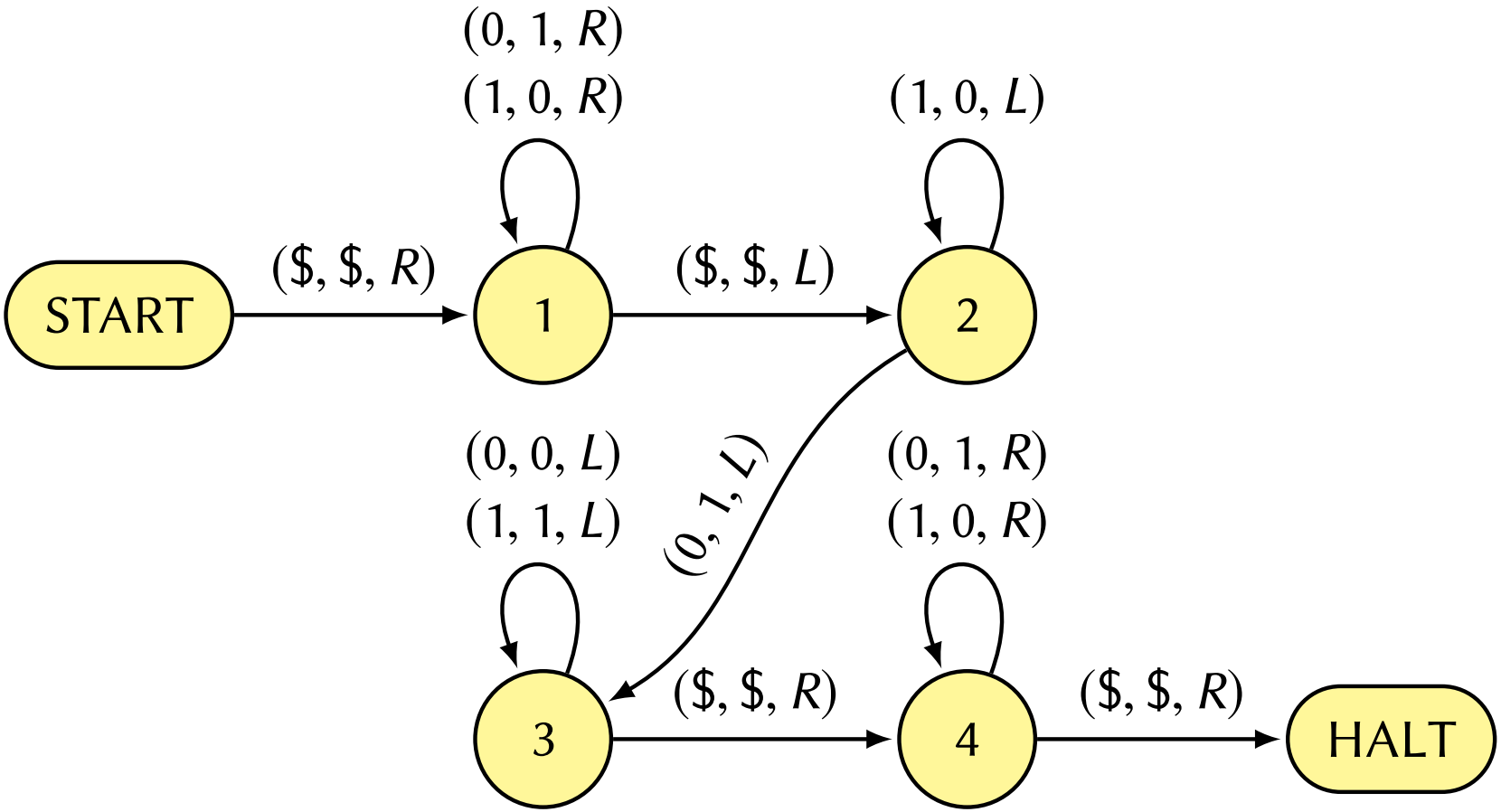


- Find the last bit of the binary number
- Reverses the last bit of the number
- If the last bit was 1, it backs up to the left and changes the whole clump of 1's to 0's... and the first 0 to the left gets turned into a 1
- If the input only contained 1, the machine crashes

A Base-2 Adder — Part 2

Let us consider another TM — T_2 that subtracts 1 from it

1. Reverse all 0's and 1's between \$ (one's complement)
2. Use T_1 to add one to the number between \$
3. Reverse all 0's and 1's between \$ (one's complement)



A Base-2 Adder — Part 3

Assuming we have input of the form \$ *x-part* \$ *y-part*
we want to calculate $x + y$. We assume $x + y$ does not overflow

- If y is the larger number and starts with 0, we guarantee there will **not** be overflow. If not, we can **insert** 0 in front of *y-part*.
- The algorithm is as follows:
  1. Check the *x-part* to see if it a 0. If yes, HALT
  2. Subtract 1 from the *x-part* using T_2
  3. Add 1 to the *y-part* using T_1
  4. Go to Step 1

The full machine is omitted but lives on page 598 in the textbook

Example: `$10$0110`

So What Is A Computer?

Definition

If a TM has a property that for every word it accepts, at the time it halts, it leaves one solid string of a's and b's on its TAPE starting at the beginning, we call it a **computer**.

The input string we call the **input** (or **string of input numbers**), and we always identify it as a sequence of nonnegative integers.

The string left on the TAPE we call the **output** and identify it also as a sequence of nonnegative integers.

Note:
a's and b's could be 0's and 1's or use some other form of encoding

Computable Functions

Definition

If a TM takes a sequence of numbers as input and leaves only one number as output, we say that the computer has acted like a mathematical **function**.

Any operation that is defined on all sequences of K numbers (for some number $K \geq 1$) and that can be performed by a TM is called **Turing-computable** or just **computable**.

Theorem

Addition and simple subtraction are computable

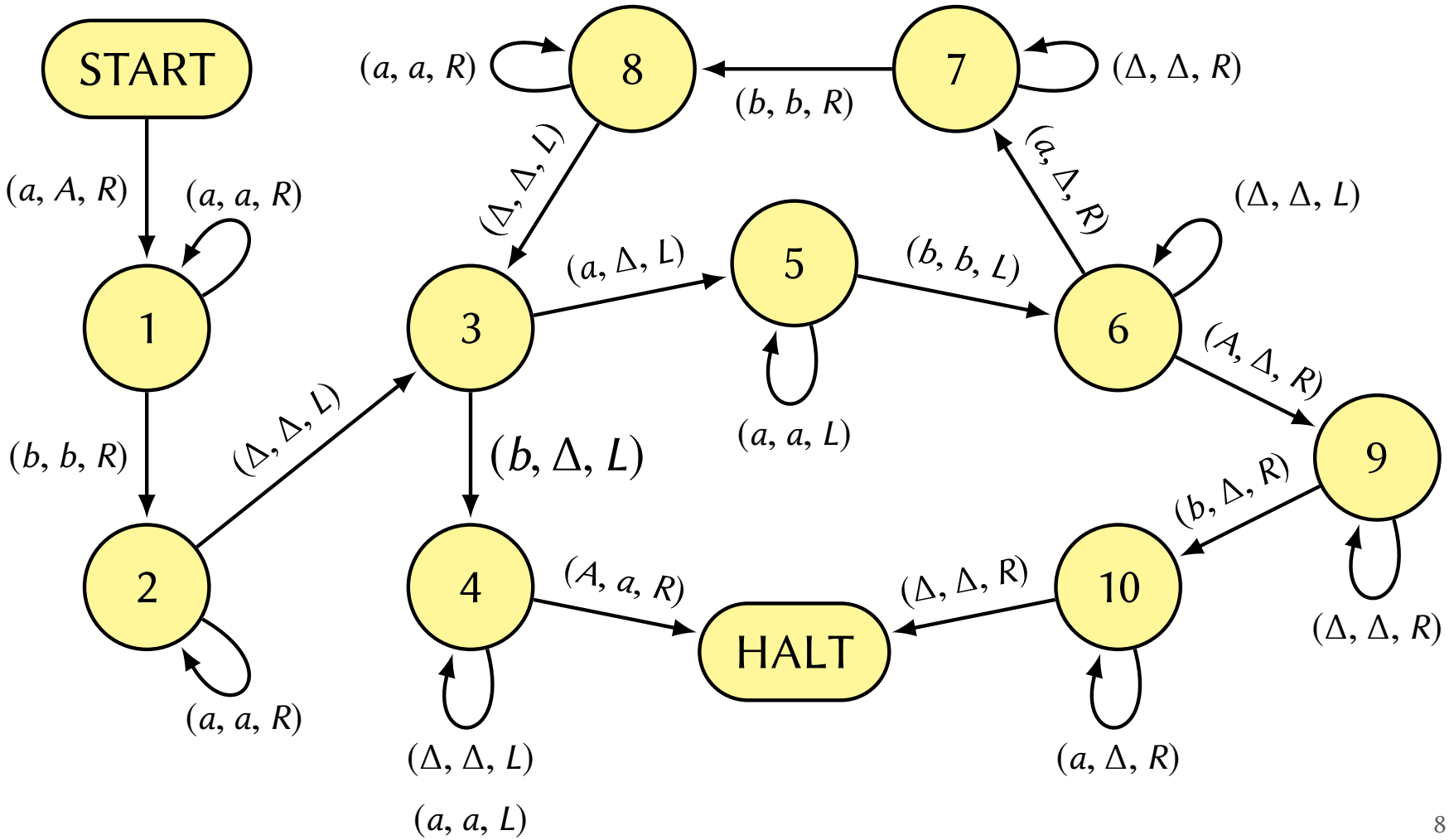
In both of these examples, $K = 2$. addition and simple subtraction are defined on sequences of two numbers and both leave one-number answers.

Theorem

MAX(x, y) which is equal to the larger of the two nonnegative integers x and y is computable.

Addition and Simple Subtraction

- Addition was shown to be computable
- Simple subtraction (**monus**) performs $x - y$ but ensures the result is at least 0 (we have no notation for negative numbers)



MAX

- On the previous slide, state 4 represented the first input group had more *a*'s than the second input group. State 9 represented the second input group had more *a*'s than the first input group.
- Instead of *erasing*, we should use *x* and *y* to remember our "counts" for each.
- The only modifications necessary is the path toward HALT
  - In the first case, every *x* in the first group should become an *a* and everything to the right of (and including) the *b* should be erased
  - In the second case, the first group and *b* should be erased and every *y* in the second group should become an *a*

Identity and Successor

Theorem

The IDENTITY *function*

$$\text{IDENTITY}(n) = n \qquad \text{for all } n \geq 0$$

and the SUCCESSOR *function*

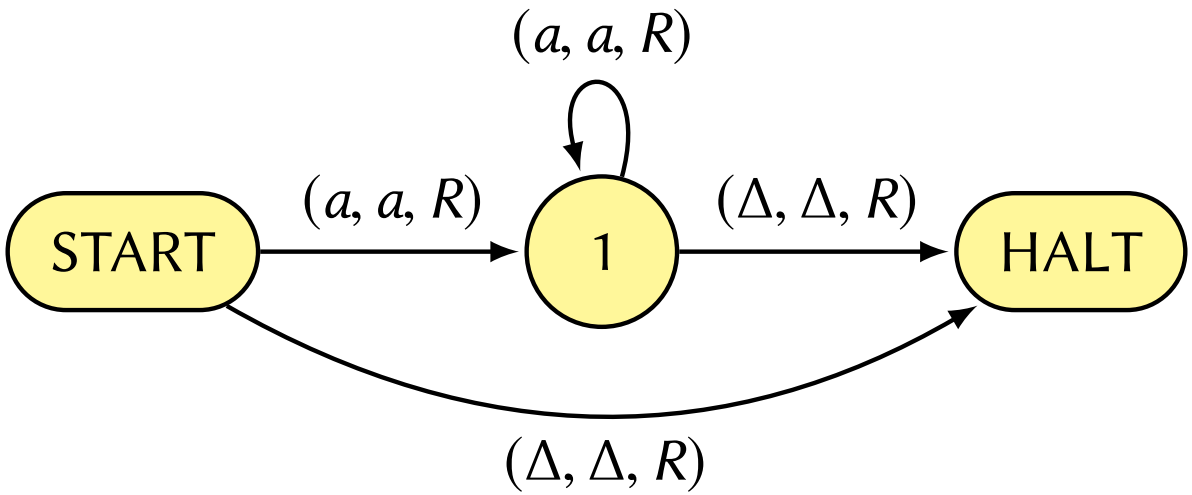
$$\text{SUCCESSOR}(n) = n + 1 \qquad \text{for all } n \geq 0$$

are computable.

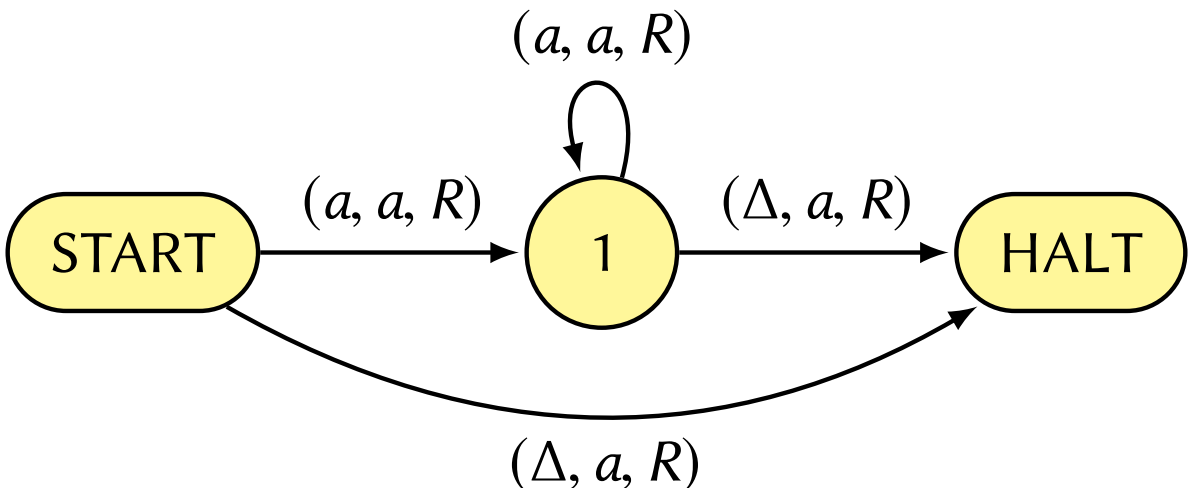
Note: these functions are defined on only one number $K = 1$ *so we expect input of the form* $\mathbf{a}^*$

Identity and Successor

Identity



Successor



Multiplication

Theorem

Multiplication is computable

Proof (part of...)

- imagine our tape contains $a^n\ b\ a^m$
- introduce a special symbol # on the far right
- for each *a* to the left of the *b*, *mark* it as consumed and **copy** *n* *a*'s to the right of the #
- erase everything up to the # symbol

Example:

```
aaa b aaaa #
Aaa b aaaa # aaaa
AAa b aaaa # aaaaaaaa
AAA b aaaa # aaaaaaaaaaaa
```

Church's Thesis

Which functions cannot be computed by a Turing Machine?

"It is believed that there are *no* functions that can be defined by humans whose calculation can be described by *any* well-defined mathematical algorithm that people can be taught to perform, that *cannot* be computed by Turing Machines. The Turing Machine is believed to be the ultimate calculating mechanism"

Church actually combined the works of recursive functions and computable functions and stated that turing machines can algorithmically represent recursive functions and model computable functions. This is called **Lambda Calculus**

Lambda Calculus

Natural Numbers (Church Numerals)

Number	Expression	JavaScript
0	$\lambda f.\lambda x.x$	`f => x => x`
1	$\lambda f.\lambda x.fx$	`f => x => f(x)`
2	$\lambda f.\lambda x.f(fx)$	`f => x => f(f(x))`
3	$\lambda f.\lambda x.f(f(fx))$	`f => x => f(f(f(x)))`

SUCC (n is a church numeral fn, f is a function, x is the "value")

$\lambda n.\lambda f.\lambda x.f((n\,f)x)$	`n => f => x => f(n(f)(x))`

PLUS (n and m are church numerals, f is a function, x is the "value")

$\lambda m.\lambda n.\lambda f.\lambda x.(m\,f)((n\,f)x))$	`m => n => f => x => m(f)(n(f)(x))`
$\lambda m.\lambda n.(m\,\mathrm{SUCC})n$	`m => n => m(SUCC)(n)`

MULT (n and m are church numerals, f is a function, x is the "value")

$\lambda m.\lambda n.\lambda f.\lambda x.(m(n\,f))x$	`m => n => f => x => m(n(f))(x)`

Homework 12b

4. [10pts] Construct a Turing Machine that accepts a number in unary and converts it to binary
5. [5pts] Describe how you would construct a Turing Machine that applies unary number exponentiation. For example, input of the form aaabaa should yield 9 *a*'s and aabaaaaa should yield 32 *a*'s on the tape.
6. [5pts] Trace the function application of `MULT(N2)(N3)(SUCC)(0)` until a single value is produced. *N2 and N3 are Church numerals representing the values of 2 and 3.* The first few substitutions are made below:

```
m => n => f => x => m(n(f))(x)
n => f => x => N2(n(f))(x)
f => x => N2(N3(f))(x)
x => N2(N3(SUCC))(x)
N2(N3(SUCC))(0)
N3(SUCC)(N3(SUCC)(0))
```

```
m -> N2
n -> N3
f -> SUCC
x -> 0
N2(y)(z) -> y(y(z))
```