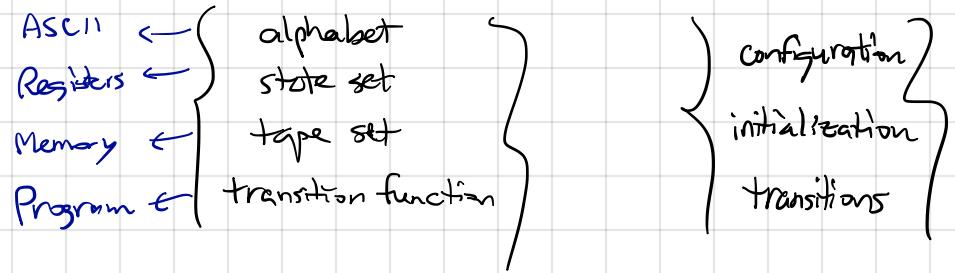


12 April 2019

Turing machines vs SJAVA.

Announcement: no homework this week.

Last time: defined Turing machine & what it computes.



This time: ① Small example of something interesting (?) that a TM can compute.

② Introduce computationally equivalent model, "Simplified Java" aka. SJAVA, that looks like ordinary computer programming.

The substring problem: given two strings $x_1, x_2 \in \{0, 1\}^*$ is x_1 a (contiguous) substring of x_2 ?

```
for i = 0, ..., |x2| - |x1|
    match = true
    for j = 0, ..., |x1| - 1
        if x1[j] ≠ x2(i+j) { match = false }
    endfor
    if (match=true) { return true }
endfor
return false
```

Plan: Represent i by modifying location i on the x_2 input tape.

Represent j by position of read/write head on the x_1 input tape.

Plan: Represent i by modifying location i on the X_2 input tape.
Represent j by position of read/write head on the X_1 input tape.

bookmark for keeping track
of "i", the pos'n where we started
looking for a match in X_2 .

My original plan:

What T.M. do:

Alphabet

$\{-, 0, 1, \bar{0}, \bar{1}\}$

Alphabet

$\{-, 0, 1\}$

Tapes

{input 1, input 2, output}

States

$\{s, t, c, r\}$

return
(analogous to
match == false)
(analogous
to match == true)

Ingredients of SFTA:

Data types

boolean

{true, false}

char

ASCII

int

$\mathbb{Z} \cup \{\perp\}$

string

ASCII*

array

int*

ϵ denotes empty string

Functions

Program is a seq of functions.

First one is the "base function"

(like main in a C or Java program)

Statements

Body of a function

• Assignment

$x = \text{expr}$, or $\boxed{x[n] = \text{expr}}$

pad x with
 \emptyset or —
as necessary

- if (expr) { ... }
- while (expr) { ... }
- return expr

Expressions :

- variable
 - constant
 - binary operation applied to 2 expressions
 - built-in function
 - e.g. `length(x)` * elements in a string or array
 - accessing element of string or array
 - e.g. `x[n]`
 - function application (calling another func in the same prog)

Semantics?

- ① Configurations
 - ② Initialization
 - ③ Transitions,

2 terms: function state & execution state
(local state of one function call) (global state of program)

Def. A function state is an ordered pair (c, ϕ) where c is a program counter (natural number indicating position of next statement to be executed, as an index in the string encoding the entire program).

ϕ is a dictionary: a function mapping variable names to values.

An **execution state** is a ^{finite} sequence of 0 or more function states called the stack.

$(c_0, \phi_0), (c_1, \phi_1), \dots, (c_n, \phi_n)$ if program is running
 The execution state of a program that terminated is
 a string representing its output.

The exec state is initialized with initial state of base function. (program counter points to 1st statement in function body, function arguments initialized with input values. Every other variable initialized with default value.)

Defaults	
boolean	false
int	#
char	-
string	" "
array	[]

Transitions:

- ① Ordinary statement that doesn't call a function: only (C_h, ϕ_h) , changes.
(According to ordinary rules of variable assignment & control flow.)

- ② Statement includes an expression that calls another function: no change in (C_h, ϕ_h) . Instead push (C_{h+1}, ϕ_{h+1}) onto stack.
(Namely, the initial state of the func that was called.)

- ③ Return statement.
in the return statement, Evaluate Expr
calling a function, push onto stack as in ②.
But if the expr can be evaluated immediately,
pop stack. New top of stack is pause on a
line of code containing an expression that
called the function that just finished.
Substitute function's return value for that
expression and execute the statement. If
that statement is a "return" statement, pop
stack again, recursively.

If stack becomes empty, return value
is program output. Put it into the exec state
and halt execution.

(Closing curly brace of function body
is interpreted as implicit "return default value"
command.)