Turing Machines

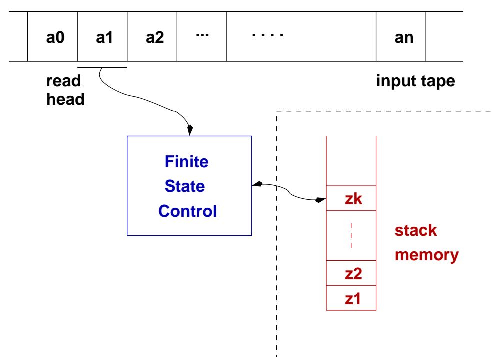
COMP2600 — Formal Methods for Software Engineering

Jeremy Dawson

Australian National University

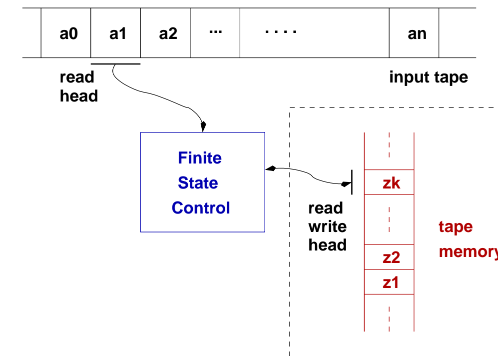
Semester 2, 2011

Push Down Automata, reloaded



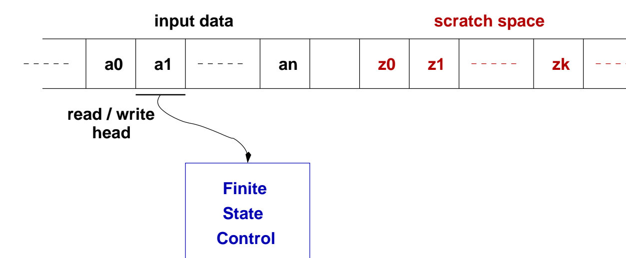
- A PDA with its auxiliary store is *almost* a whole computer, except we can only directly access the symbol on the top of the stack.

Turing Machines



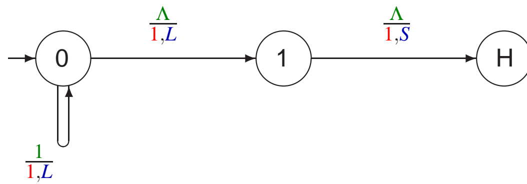
- We generalise the PDA into a Turing Machine by using a *tape memory* for the store instead of a *stack memory*.
- We can access an arbitrary symbol by moving the tape head.

Single tape Turing Machines



- We can simplify the two tape TM by using a single tape for both input data and auxiliary storage. This is the standard form of Turing Machines.
- The tape is assumed to be *infinite* in both directions, so we will never run out of space.

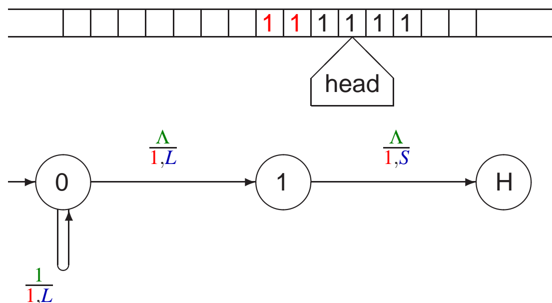
Representing the Finite State Control



Similar to FSA, annotate transition edges with commands for accessing tape.

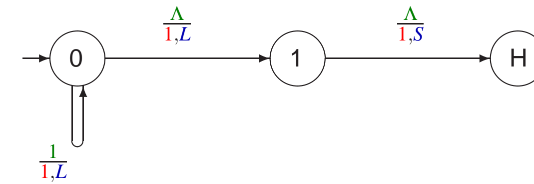
- Numerator: symbol read from tape.
  - Λ means the tape is blank at that position.
- Denominator: symbol written / direction of *head movement.*
  - direction one of L, R, S for Left, Right, Stay.

What does it do?



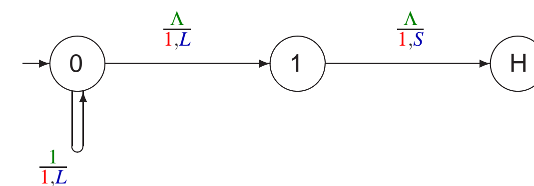
- Adds two to a unary number!
- Assume the head starts over the input data.
- First phase scans left.
- Second phase writes two extra 1's.

The Convention for Errors



- If there is no arc from a node corresponding to one of the vocabulary symbols then it is assumed that the machine halts in an error state.
- This error state is not shown on the diagram.
- For example, in this TM, the detection of a one in state 1 is treated as an error.

Turing Machine Programs

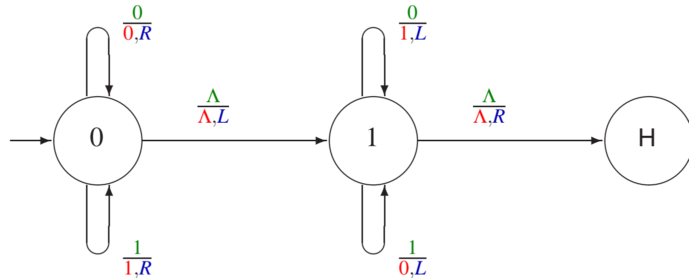


- A notation for instructions: $< S,\ I,\ O,\ D,\ S' >$

  If the machine is in state S and reads symbol I, then write symbol O back to the tape, move the head in direction D and go to state S'.

- This machine would then be represented as:
  $< 0,\ 1,\ 1,\ L,\ 0 >$
  $< 0,\ \Lambda,\ 1,\ L,\ 1 >$
  $< 1,\ \Lambda,\ 1,\ S,\ Halt >$ (though humans prefer pictures!)

What does this one do?



- Don't you see two phases?
- What does each phase accomplish?

Solution

- This is a Turing machine that complements a binary number.
- The head is assumed to be initially over the number.
  It scans right & then traverses back complementing bits as it goes.
- Written as a program (for machine consumption, perhaps):

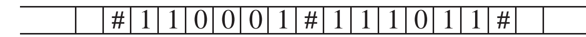
$< 0, 0, 0, R, 0 >$	Scan right, over 0's
$< 0, 1, 1, R, 0 >$	Scan right, over 1's
$< 0, \Lambda, \Lambda, L, 1 >$	End scan
$< 1, 0, 1, L, 1 >$	Invert a bit
$< 1, 1, 0, L, 1 >$	Invert a bit
$< 1, \Lambda, \Lambda, R, halt >$	At left end, halt.

Harder Problems?

- Incrementing a binary number

  You should try this!

- Adding numbers - need terminators



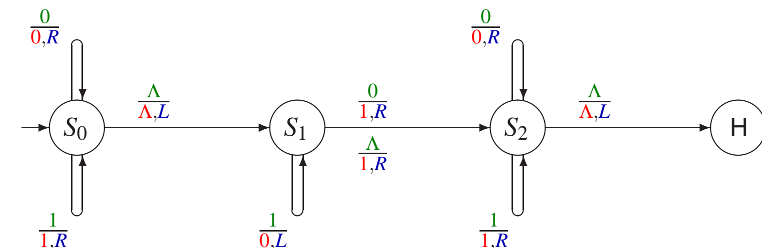
  Convenient to write the result before the data.

- Multiplication - and so on!

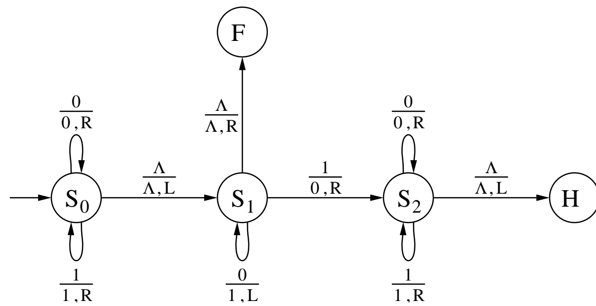
Problem Solution

- Incrementing a binary number

- Detailed Solution:



Decrementing a number is similar



If given number is 0 it fails (goes to failure state F, having changed the number, which was all zeroes, to ones).

How to add two numbers?

Input eg $\overbrace{10100101}^{n}\#\overbrace{100101010}^{m}\#$

Go back and forth between m and n, decrementing one (until this fails) and incrementing the other.

We decrement m, and increment n, because n will expand leftwards.

m gets changed to $11\ldots1$, n is replaced by the sum.

How to multiply two numbers?

Input eg $\underbrace{\quad}_{p}\#\underbrace{10100101}_{n}\#\underbrace{100101010}_{m}\#$

Repeatedly decrement m (until this fails) and add n to p (p is initially blank)

We must modify the addition routine to not erase the number n being added.

Easy! Two new tape symbols, $0'$ and $1'$.

Before each addition stage, change all the 1s in n to $1'$.

When decrementing n, change 0 to 1 and vv as usual, but keep the primes.

When finished adding n to p, go back and use the primes to restore n.

This sort of trick is typical in showing how to do *any* computation in a TM

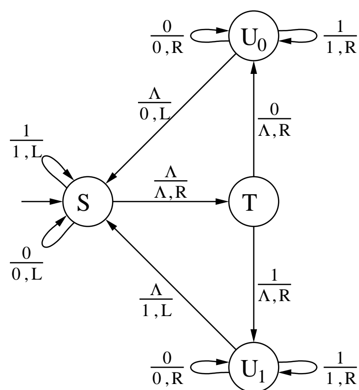
Programming Issues – Data

- Number representation: Usually use unary or binary for integers.
- Vocabulary
  Can be arbitrary. Just need to assume tape of bits.
  Characters are represented as strings of bits.
- Variables, Arrays, Files
  Use markers on the tape to separate values.
- Common techniques include:
  Insert a symbol in the tape (requires moving everything to the right (left) of the insertion square one square further to the right (left)) Exercise!
  Delete a symbol and closing up the gap Exercise!
  These two require “remembering” a symbol, to write it elsewhere

Programming Issues – Control

- Importance of Idiom
  We've seen scanning to blank; scanning is common.
- Use control states to remember information
  In particular, we often need to "remember" a symbol, to write it elsewhere: this typically requires a set of states, one per symbol
- Composition
  If you have a TM to multiply by 3 and one to multiply by 5, put them together to multiply by 15.
- Decisions (conditional computation)
  As we have seen, we can branch on 0 or 1 (or T or F).
- Loops — of course.

Using States to Remember a Tape Symbol



Given a string of 0 or 1 surrounded by blanks, this machine repeatedly forever erases the leftmost bit, and writes it on the right hand end. (Not so useful, but illustrates the point)

We use the choice of states U_0 or U_1 to remember which symbol has been erased and is to be written

Universal Turing Machines and Turing Completeness

- We can construct a TM to simulate any conventional computer. (Cost effectiveness is not brilliant.)
- From this point, just think of a TM as like a computer program (with a machine that will run it)
- We can construct a TM that first reads a description of some other TM and then simulates it. This is a *universal* TM. (Think of a Haskell program whose purpose is to read any other Haskell program and run that)
- Turing machines can simulate themselves.
- Turing machines can *compute properties of themselves*.
- Any computing device which can simulate a universal Turing Machine is also called *universal* or ***Turing Complete***.

Extra features do not increase strength.

- Multi-tape machines: Adding an extra tape for scratch space would make the TM easier to program, but provide no extra power.
- Multi-head machines: Adding multiple heads to access multiple symbols at once would make the TM easier to program, but provide no extra power.
- Similarly to Church Encoding, we can translate a TM with extra features into the single tape, single head machine.
- Efficiency: An efficient TM? You're joking!
  The main problem is the lack of random access memory. Adding it would make the TM faster, but provide no extra power. Translate into multi-tape, multi-head machine....

Church-Turing Thesis

Every effectively calculable function ...

...is a computable function.
...can be computed with a Turing Machine.
...can be computed with the Lambda Calculus.
...can be computed with a Turing Complete computing device.

- We can build a Turing machine which interprets the lambda calculus.
- We can write a lambda calculus term which simulates a Turing machine.

..mumble.. but Turing Machines can't exist!

- A Turing Machine has an *infinite tape*.
- My laptop computer has 17,179,869,184 bits of RAM (2 Gigabytes), but that's no where near infinite.
- Physical computing devices must have finite memory, so they can only exist in a finite number of states. They are Finite State Automata.
- .. but the number of states can be very, very large.
  How big is $2^{17179869184}$? *– pretty big!*
- Physical computers can only *approximate* Turing Machines :-(

So what's the point?

- If a Turing Machine has a finite tape we can simulate it with an FSA.
- Just take the finite state control and replicate it for every possible string that could exist on the tape.
- Combinatorial explosion!
- For a tape containing 16 binary symbols, we need *at least* 2^{16} states in our FSA. I'm not going to wait for you to draw one.

Random Access Memory saves the day.

- An FSA has no auxiliary store.
  This makes it hard to program.
- A push-down automaton has an auxiliary store...
  ... but we can only directly access the symbol on the top of the stack.
- A Turing Machine is infinite...
  ... but we have to access the tape sequentially.
- My laptop is finite...
  ... but accessing a store location requires only a small, constant time.
  ... that's usually enough.