Lecture 28:

- TMs (cont)
- λ-calculus

Announcements:

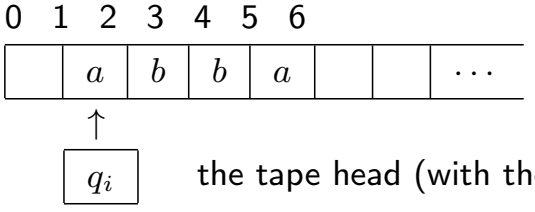
- HW-6 out
- Extra Credit proposal due Fri
- Exam 2 Mon

Turing Machines

Basic Idea: ... even lower level than RAM Machines!

1. ***Infinite tape*** of memory cells, each cell holds one symbol
2. ***Read/write head*** that can move left/right (L/R) one cell at a time
3. ***State register*** that stores the current state of the machine
4. ***Transition table***: Curr State & Symbol $\rightarrow$ New State, Symbol, & Direction

Example: replace a's with b's



the infinite tape

the tape head (with the machine in state q_i)

Special blank ($\sqcup$) symbol, and halting states q_{halt}, q_{yes}, q_{no}

Many variants (e.g., multitape machines, non-deterministic)

Turing Machines

Example transition table: (where q_1 is start symbol)

Current State	Current Symbol	New Symbol	New State	Direction
q_1	a	b	q_1	R
q_1	b	b	q_1	R
q_1	⊔	⊔	q_{halt}	L

A TM to check if a string of 0's and 1's has an even number of 1's ...

Current State	Current Symbol	New Symbol	New State	Direction
q_1	0	0	q_1	R
q_1	1	1	q_2	R
q_1	⊔	⊔	q_{yes}	R
q_2	0	0	q_2	R
q_2	1	1	q_1	R
q_2	⊔	⊔	q_{no}	R

Turing Machines

More examples:

- Add one to a binary number (go to end, move left adding 1)
- Check for palindrome (zig-zag, mark as read, states as symbols)

Turing Machines are an ***imperative*** model of computation ...

- specify how computation should be carried out (very low level)
- inspiration for RAM machines

It is possible to write a TM that simulates all other TMs

- the input is an encoding of the transition table of the TM to run
- called a "**Universal**" TM

Turing Machines

A PL is "**Turing Complete**" if it can simulate any TM (i.e., is universal)

- All computable functions are computable by a TM (Church-Turing thesis)
- If a PL is turing complete, it can express *all possible computations*
- No other model of computation found that can perform tasks a TM cannot

Examples of languages that are *not* Turing Complete:

- Markup languages: HTML, XML, JSON, YAML, ...
- Many "domain-specific" languages: (basic) SQL, regular expressions

Turing Completeness not necessarily tied to specific constructs

- imperative languages with conditional branching (if-goto, while loops) and arbitrary mem access (# of variables)
- whereas functional and logic-based languages have other constructs such as pattern matching and recursion (no goto, no loop constructs)

From λ-Calculus to Functional Programming

TMs are (roughly) the MoC for imperative languages
... λ-calculus is (roughly) the MoC for functional languages

Basic idea of λ-calculus

(1) Unnamed, single-variable functions ... λ *functions* aka "abstractions"

- $\lambda x.x$ takes an x and returns an x
- $\lambda x.(\lambda y.x)$ takes x and returns a function that takes y and returns x
- ... shorthand for multi-argument functions: $\lambda xy.x$

(2) Function application

- $(\lambda x.x)0$ applies the identity function to 0 (resulting in 0)
- $(\lambda x.(\lambda y.x))ab$ reduces to a ... $(\lambda x.(\lambda y.x))ab \Rightarrow (\lambda y.a)b \Rightarrow a$
- ... where $\Rightarrow$ denotes a one-step *application*

The λ-Calculus

(3) Expressions

- Either a function, an application, a variable, or a constant
- General form of a function: $\lambda x.e$ where x is a variable and e an expression
- An application has the form: $e_1 e_2$ where both e's are expressions

Computation in λ-calculus is via function application

- Given an expression (function application) such as:

$$(\lambda x.x)y$$

- An application is evaluated by substituting x's in the function body with y:

$$(\lambda x.x)y = [y/x]x = y$$