29: Limits and the real world

A few loose ends
Difficulty of problems
Limits on computation
Recursive Diagrams: Why use them?
Modules: Why use them?

• Clarity: gather together related pieces of code
• Generosity: avoid name conflicts
• Data hiding: using module *types* lets you limit what users of your module can see about the implementation
  • Implementation neutrality
• Our next big module: GAME.
What is the domain of parse?

- *Any* concrete program is allowed
- *Some* are parseable, and produce abstract programs; others generate errors.
- *Error-generation* is part of the required functionality
  - Precise form of error is not specified
  - You can choose your form, and require it via check-expects
  - (slightly) violates our rules about check-expects being derived *only* from the specification
- Same thing applies to *eval*
Are there any limits on computation?

- Proved that comparison based sorting of a list of $n$ items had to take time at least proportional to $n \log n$
- *Subset-sum* for a set of $n$ weights appeared to need at least $2^n$ time
- *Are there computational problems that are inherently hard? Harder even than exponential?*
Decision problems

- Consume an input; produce a boolean output
- Example:
  - Subset-sum: is there a subset of these weights that adds up to that total?
    - Input: weights, total
  - Subset-sum-check: are these items a subset of weights that adds up to the desired total?
    - Input: a list of weights; the original list of weights; the total
Is subset-sum-check hard?

• Not really: you just add up the purported solution and see whether the result is the desired target.

• *Not so fast!*
  • You also have to check that the items in the purported subset are in fact in the original list!

• If original list has length \( n \), then the first part is in \( O(n \rightarrow n) \), and the subset-check is in \( O(n \rightarrow n^2) \), so total work is in \( O(n \rightarrow n^2) \).

• We say that it's "solvable in polynomial time"

• Measurement of "size" of the input is tricky (see CS1010).
Classes of problems

• Some decision problems are known to be solvable (by algorithms like the ones we've been studying) in polynomial time. The class of all such problems is called P. "subset-sum" does not appear to be in class P.

• Some decision problems can be verified in polynomial time (as we saw for subset-sum-check); such problems are said to be in class NP.
  • So subset-sum is in NP, because subset-sum-check is in P

• Big fat open question: are there any problems in NP that are not in P?
  • Everyone thinks the answer is 'yes' – there are problems that are "easy" to check, but hard to solve.
  • No one has a proof...but it's worth a million bucks if you find one.
How hard can a problem be?

• Is there a computational problem that we cannot solve?
• Hilbert did not think so
Problems about programs

• Does this program use lots of memory?
  • Depends on computer...
• Does this program correctly compute the desired value?

• Does this program *terminate* (i.e., halt)? (on an idealized computer)

```
(define (b-len alod)
  (cond
    [(empty? alod)  0]
    [(cons? alod) (+ 1 (b-len alod))]))
```
Easier questions

• Does this program halt before it has performed 10,000 operations?
• Does this program halt within 3 minutes?
Slight generalization: handle some data as well

Can we write a program with the following spec?

; input: proc, a procedure, represented as a string
; data, a datum that the procedure can consume,
; represented as a string
; output: true if (proc data) terminates, false otherwise.
(halt? proc data)
Suppose we had written "halt?"
Then what?

Some programs, like Rackette, consume programs as input. If we wrote Rackette in Racket rather than Ocaml, we could feed it the source-code for Rackette as input!

(define rackette-as-string "(define (evaluate exp) ... ")
We could then do this:
(halt? rackette-as-string rackette-as-string)
Going further

(define (halts-on-self s)  
  (halt? s s))

(define (loop) (loop)) ;; never halts!

(define (loop-if-halts-on-self s)  
  (if (halts-on-self s) (loop) true))

(define qs "(define (loop-if-halts-on-self s) ...")  
  ;; that same program, as a string."
(define (loop-if-halts-on-self s)
    (if (halts-on-self s) (loop) true))

(define qs "(define (loop-if-halts-on-self s) ...") be that program, as a string.
Now evaluate this:
(loop-if-halts-on-self qs)
(loop-if-halts-on-self qs)
• Apply qs to qs and see if it halts
  • If so, then this red program does not halt.
  • if not, then this red program does halt.
• But applying qs to qs is exactly...
  (loop-if-halts-on-self qs).
  • So if this green program halts, then the red program does not
    • If the green program does not halt, then the red program does
• In other words, the red program halts only if it does not halt, and vice versa
• Contradiction.
• Where was our "assumption"?
  The existence of a "halt?" program
• The "halting problem" cannot be solved by any program!
• Computation has limits
Connecting with the real world
CS17 approach

• First learn to write some programs, *then* deal with the dirty mess that is real data
• Let's look at a bit of real-world data: the filesystem
OCaml "Sys" module: access to the filesystem!

- **val** file_exists : string -> bool Test if a file with the given name exists.
- **val** is_directory : string -> bool Returns **true** if the given name refers to a directory, **false** if it refers to another kind of file. Raise Sys_error if no file exists with the given name.
- **val** chdir : string -> unit Change the current working directory of the process.
- **val** getcwd : unit -> string Return the current working directory of the process.
- **val** readdir : string -> string array Return the names of all files present in the given directory. Names denoting the current directory and the parent directory ("." and "..") in Unix) are not returned. Each string in the result is a file name rather than a complete path. There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.
readdir returns an `array`

• For us:
  • `val` `length` : `'a array` `->` `int`  
    Return the length (number of elements) of the given array.
  • `val` `get` : `'a array` `->` `int` `->` `'a`  
    `Array.get a n` returns the element number `n` of array `a`. The first element has number 0. The last element has number `Array.length a - 1`. You can also write `a.(n)` instead of `Array.get a n`.
  • `val` `iter` : `(('a` `->` `unit`)` `->` `'a array` `->` `unit`  
    `Array.iter f a` applies function `f` in turn to all the elements of `a`. It is equivalent to `f a.(0); f a.(1); ...; f a.(Array.length a - 1); ()`.
  • `val` `iteri` : `(int` `->` `'a` `->` `unit`)` `->` `'a array` `->` `unit`  
    Same as `Array.iter`, but the function is applied with the index of the element as first argument, and the element itself as second argument.
  • `val` `map` : `('a` `->` `'b`)` `->` `'a array` `->` `'b array`  
    `Array.map f a` applies function `f` to all the elements of `a`, and builds an array with the results returned by `f`: `[| f a.(0); f a.(1); ...; f a.(Array.length a - 1) |]`.
  • `val` `mapi` : `(int` `->` `'a` `->` `'b`)` `->` `'a array` `->` `'b array`  
    Same as `Array.map`, but the function is applied to the index of the element as first argument, and the element itself as second argument.
  • `val` `to_list` : `'a array` `->` `'a list`  
    `Array.to_list a` returns the list of all the elements of `a`.
  • `val` `of_list` : `'a list` `->` `'a array`  
    `Array.of_list l` returns a fresh array containing the elements of `l`. 
Find and print the current directory name

```ocaml
open Sys;
print_string (getcwd ());
```

- Designed to be run with
  `% ocaml dir1.ml`

which reads each line of `dir1.ml`, but prints out only error messages or "printed" output. Without the "print_string", there'd be no output at all.
Print directory contents nicely

open Sys ;;
open Array ;;
let s = (readdir ".");; (* get an array of names *)
let q = to_list s;; (* convert to a list *)
let r = List.sort compare q;; (* sort it *)
let t = of_list r;; (* and convert back *)
iter (fun x -> print_string x; print_string "\n") t;;
(* "iter" is like "map" in Ocaml; Array.iter applies
 * to arrays
 *)