Administrative Announcements

Exam
- Due Monday

Next Homework
- Available Monday

Review

New template for *generative recursion*

(define (gen-recur-func problem-data)
  (cond
    [(trivial-to-solve? arg_1 arg_2 … arg_n) (solve arg_1 arg_2 … arg_n)]
    [else
      (combine-solutions
        … (gen-recur-func (generate-problem1 problem-data)) …
        … (gen-recur-func (generate-problem2 problem-data)) …
          …
        … (gen-recur-func (generate-problemk problem-data)) …)]
  ))

This one offers less guidance than the structural templates did!
⇒ Need to ask some questions before we fill it in !

Review

List of questions to help fill in the template

1. What is the trivial case?
   - → How do we identify it?
   - → How do we solve it?
2. How do we generate subproblems?
   - → How many should we generate?
   - → How do we generate them?
3. Does the subproblem solution solve the original problem?
4. Must we combine solutions from multiple subproblems?
   - → How do we combine them?

Back to Generative Recursion

Last class we were looking at hi-lo

```
;; hi-lo:  integer  integer -> integer
;; Purpose: consumes an interval and returns the number
;;           hidden by guess (lo <= guess <= hi)
(define (hi-lo  lo hi)
  (local [(define midpoint  (/ (+ lo hi) 2))
          (define answer    (guess midpoint))]
         (cond
            [(symbol=?  answer  ‘lower)   (hi-lo midpoint hi)]
            [(symbol=?  answer  ‘equal)   midpoint]
            [(symbol=?  answer  ‘higher)  (hi-lo lo midpoint)]
   ))
```

An Aside

Debugging hi-lo

- To understand what happens with hi-lo, we need to see the values of lo, midpoint, and hi
- Two new Scheme expressions, begin and printf

  (begin $expr_1$ $expr_2$ … $expr_n$)

  → Evaluates $expr_1$, then $expr_2$, then …, then $expr_n$

  (printf string arg_1 arg_2 … arg_n)

  → Prints string, with the arguments substituted in place of tilde expressions (see DrScheme's Help Desk for details)
- We use them to print out the values on each recursive call

Try hi-lo on [0 12]
Try hi-lo on [0-15]
} *To DrScheme*

Back to Generative Recursion

What happened?

- The recursive calls passed midpoint to hi-lo
- Midpoint took on non-integer values (/ (+ 0 15)) = 15/2
  - → Violates the contract
  - → Non-integer endpoints cannot match hidden value!

```
;; hi-lo2:  integer  integer -> integer
;; Purpose: consumes an interval and returns the number
;;           hidden by guess (lo <= guess <= hi)
(define (hi-lo2  lo hi)
  (local [(define midpoint  (truncate (/ (+ lo hi) 2)))
           (define answer    (guess midpoint))]
          (cond
             [(symbol=?  answer  'lower)   (hi-lo2 midpoint hi)]
             [(symbol=?  answer  'equal)   midpoint]
             [(symbol=?  answer  'higher)  (hi-lo2 lo midpoint)]
  ))
```

Returns next lower integer

Try it on [0 - 15]

Hi-lo, again

Termination argument

At each step, hi-lo2 splits the interval into two sub-intervals [lo - midpoint] and [midpoint - hi], where midpoint is computed as the integer below (/ (+ lo hi)). It uses guess to determine if the hidden number lies in [lo - midpoint], in [midpoint - hi], or is equal to midpoint. At each recursive call, the interval becomes smaller.

Eventually, the interval converges to a trivial range, where the midpoint is equal to the hidden number.

Try it on [0 - 3]

More Test Cases

(hi-lo2 0 3)

- Oops. It never terminates (on its own)
- Why?

When the interval reaches the point where (- hi lo) is 1, it can only test lo, not hi. Anytime the hidden number becomes the upper end of an interval, the hi-lo2 will stop making progress.

We need to test the upper bound explicitly, as in hi-lo3

- Obvious way to fix it is to check hi explicitly

Hi-lo3

Checking hi explicitly

```
;; hi-lo3:  integer  integer -> integer
;; Purpose: consumes an interval and returns the number
;;            hidden by guess (lo <= guess <= hi)
(define (hi-lo3  lo hi)
   (cond [(symbol=? (guess hi) ‘equal) hi]
            [else   (local [ (define midpoint  (truncate (/ (+ lo hi) 2)))
                                (define answer    (guess midpoint)) ]
                            (cond
                                [(symbol=?  answer  ‘lower)   (hi-lo3 midpoint hi)]
                                [(symbol=?  answer  ‘equal)   midpoint]
                                [(symbol=?  answer  ‘higher)  (hi-lo3 lo midpoint)]
                             ))] ))
```

Try it on [0 - 3]
Try it on [0 - 6]

Hi-lo3

Termination argument

At each call to hi-lo3, it checks the value of hi against the hidden number. If that test fails, it computes the midpoint of the interval and tests it. If the midpoint equals the hidden value, it returns the hidden value. Otherwise, it takes the appropriate subinterval and recurs.

In the worst case, this continues until the interval contains exactly two integers, lo and hi. It explicitly checks hi. The computation of midpoint produces lo (due to the truncate). Thus, it checks both endpoints, one of which must be the hidden number.

Whew. That seems pretty complex.

Hi-lo3

But wait

- Hi-lo3 checks hi on every call
- After first call, every hi has already been checked
  - → Many extra calls to guess
- Can use local to elide this redundant check

```
;; hi-lo3a: integer integer -> integer
(define (hi-lo3a lo hi)
  (cond
    [(symbol=? (guess hi) 'equal) hi]  ;; check it once
    [else (local [(define (helper lo hi)  ;; recur without checking hi
                    (local [(define midpoint (truncate (/ (+ lo hi) 2)))
                            (define answer   (guess midpoint))]
                      (cond [(symbol=? answer 'higher)(helper lo midpoint)]
                            [(symbol=? answer 'equal)   midpoint]
                            [(symbol=? answer 'lower) (helper midpoint hi)])))]
            (helper lo hi))]))
```


HI-lo3a

Termination argument

The call to hi-lo3a checks the value of hi. If it equals the hidden number, hi-lo3a returns.

Otherwise, it invokes helper. It picks a midpoint and checks it for equality to the hidden number. If the midpoint is not equal to the hidden number, it recurs on the appropriate interval [lo - midpoint] or [midpoint - hi]. Note that both hi and midpoint have all been checked against the hidden number.

In the worst case, this continues until the interval contains exactly two integers, lo and hi. Hi has already been checked. Midpoint becomes equal to lo and helper checks it.

This version works…

Hi-lo4

Another approach

```
;; hi-lo4:  integer  integer -> integer
;; Purpose: consumes an interval and returns the number
;;           hidden by guess (lo <= guess <= hi)
(define (hi-lo4  lo hi)
  (local [(define midpoint  (truncate (/ (+ lo hi) 2)))
          (define answer    (guess midpoint))]
         (cond
            [(symbol=?  answer  'lower)   (hi-lo4 (add1 midpoint) hi)]
            [(symbol=?  answer  'equal)   midpoint]
            [(symbol=?  answer  'higher)  (hi-lo4 lo (sub1 midpoint))]
  ))
```

Try it on [0 - 3]
Try it on [0 - 6]

Hi-lo4

Termination Condition

The range between lo and hi gets strictly smaller on each recursive call. It narrows the interval by excluding midpoint, which it has already tested and rejected.

In the extreme case, the interval becomes a single number (hi = lo = midpoint). At that point, the algorithm terminates because guess must return 'equal.

This is much simpler.

In fact, the simpler termination argument suggests that this might be the better solution for the problem.

Hi-lo

Conclusions

- Thinking about the termination condition led us to a number of different solutions
- Simplifying the termination condition led to a simpler implementation
- The simpler implementation may actually do less work, since it shrinks the interval more quickly (& avoids duplicate tests)

In some sense, the extent to which you do this kind of structured reasoning about termination and correctness determines whether you are a recreational programmer—hacking together something and checking it on a few simple examples—or a professional developer who writes robust, reliable applications.