Generative Recursion Illustrated

Corky Cartwright
Department of Computer Science
Rice University

Big Picture

- Functional program design in Scheme
  - Data-directed (functional) program design
    2-10
  - Algorithm design (generative recursion, accumulators)
    11-15
  - Applied functional programming
    16-18
- Object-oriented (OO) program design in Java 19-40
  - ...

Plan for Today

- Template for Generative Recursion
- Looks at a simple example of generative recursion (algorithms) in detail: (very) simple parsing
- Book: focuses on more challenging numerical algorithms but the challenge is the underlying mathematics not the coding

Generative Recursion

- Structural recursion
  - Template derived directly from data definition
  - Termination for all programs is guaranteed
  - Conceptually includes *complete* structural recursion such as naive Fibonacci
    $f(n) = f(n-1) + f(n-2)$
    but complete structural recursion does not fit our structural recursion template.
- Generative recursion
  - Data definition does not strictly guide design of function
  - Must address termination in each such function
  - Degenerate cases: complete/pseudo structural recursion

Impact on Design Recipe

- Only effects:
  - choice of template; and
  - inclusion of termination argument
- Impact on template:
  - "Divide and Conquer" decomposition of the problem requires some creativity
    - Determine solution for trivial problems
    - Determine how to break big problems into smaller ones
    - Determine how to combine solutions of smaller problems to solve the bigger problem

Generative Template

```
(define (gr-fun problem)
  (cond
    [(trivially-solvable? Problem)
     ;; computer trivial solution
      ... ]
    [else
     ;; combine-solutions
       ... problem ...
       (gr-fun (gen-subproblem-1 problem))
       ...
       (gr-fun (gen-subproblem-n problem))
       ... ]))
```

Numerical Algorithms; Stream Algorithms

Algorithms that process *real numbers* are not structural

Examples:

- Bi-section for finding roots
- Newton's algorithm for finding root of a function f (square root is best known application)
- Formulas for constructing fractals
- Series approximations

Explanation: real numbers are not a structural type (Dedekind cuts, Cauchy sequences)

Algorithms that process *(infinite) streams* are not structural

- Parsing
- Arithmetic operations on radix representations of real numbers (exact real arithmetic)

Explanation: (infinite) streams are not a structural type

Examples of stream-processing algorithm

- Parsing console input
- Parsing according to a context-free grammar (deferred to Comp 311/314/412)

Parsing Console Input

- Used every time a program reads a text file
- Basic idea: a file is a sequence of proper chars separated by newline (improper) chars. A read operation returns the sequence of chars starting at the cursor position ending with the next newline and advances the cursor. In a functional setting, a stream of chars is converted to a stream of lines

  `(parse '(a b newline c d e f newline g h i …`

  produces

  `'((a b) (c d e f) (g h i) ... )`
- True functional characterization requires potentially infinite streams (constructed using non-strict cons).

Parsing cont.

Is there a generative decomposition?

Consider writing the following function

```
; parse : (list-of symbol) -> (list-of (list-of symbol))
; symbol is a convenient substitute for char
```

We will start with structural template but we will revise it as we fill in code.

```
; parse : (list-of symbol) -> (list-of (list-of symbol))
(define (parse input)
  (cond [(empty? input) ...]
        [(cons? input)
         ... (first input) ... (parse (rest input))]))
```

Parsing cont.

Collective in class exercise

The primitives first and rest are clearly wrong.

What should we use instead?

```
 parse : (list-of symbol) -> (list-of (list-of symbol))
(define (parse input)
  (cond [(empty? input) ...]
        [(cons? input)
         ... (first?? input) ... (parse (rest?? input))]))
```

For Next Class

- Homework due on Friday
- Reading:
  - Study chs. 25-28: many generative (non-structural) algorithms
- Lab
  - Practice with generative recursion