UMass Boston Computer Science
CS450 High Level Languages (section 2)
ASTs and Interpreters

Wednesday, November 8, 2023
Logistics

• HW 6 out
  • due: Sun 11/13 11:59 pm EST
Design Recipe For Accumulator Functions

When a function needs to “remember” extra information:

1. **Specify accumulator:**
   - Name
   - Signature
   - Invariant

2. **Define internal “helper” fn with extra accumulator arg**
   - Helper fn does **not** need to repeat description, statement, or examples, (if they are the same) ...

3. **Call “helper” fn, with initial accumulator value, from original fn**
Design Recipe For Accumulators

`; valid-bst? : Tree<X> -> Bool
`; Returns true if t is a BST

(define (valid-bst? t)
  ;; accumulator p? : (X -> Bool)
  ;; invariant: if t = (node l data r), p? remembers valid vals
  ;; for node-data such that (p? (node-data t)) is always true

  (define (valid-bst/p? p? t)
    (or (empty? t)
      (and (p? (node-data t))
        (valid-bst/p? (conjoin p? (curry > (node-data t)))
          (node-left t))
        (valid-bst/p? (conjoin p? (curry <= (node-data t)))
          (node-right t))))

  (valid-bst/p? (lambda (x) true) t))

1. Specify accumulator: name, signature, invariant

2. Define internal “helper” fn with accumulator arg

3. Call “helper” fn, with initial accumulator

Function needs to “remember” extra information ... ...
range of allowed values for node-data
In-class Coding: Reverse, with accumulator

(define (rev lst0)
  ;; accumulator rev-lst-so-far: List<X>
  ;; invariant: reversed elements of “list so far”,
  ;; i.e., lst0 “minus” remaining-lst

  (cond
    [(empty? remaining-lst) rev-lst-so-far]
    [else (rev/a (cons (first remaining-lst) rev-lst-so-far)
                  (rest remaining-lst))]]
  (rev/a empty lst0))
In-class Coding: Tree Max

;;; tree-max : TreeNode<Int> -> Int
;;; Returns the maximum value in a given (non-empty) (non-BST) tree

(define (tree-max t0)
    (cond
        [(empty? t) root-val]
        [else (max (node-data t)
            (tree-max/a (node-left t) root-val)
            (tree-max/a (node-right t) root-val))]]))

1. Specify accumulator: name, signature, invariant

;;; tree-max/a : Tree<Int> -> Int
;;; accumulator root-val: Int
;;; invariant: node-data of t0 root node (max of empty tree)

(define (tree-max/a t root-val)
    (cond
        [(empty? t) root-val]
        [else (max (node-data t)
            (tree-max/a (node-left t) root-val)
            (tree-max/a (node-right t) root-val))]]))

2. Define “helper” fn with accumulator (and other args)

accumulator used for “remembering” info, but doesn’t always “accumulate”

This is not the only possible accumulator choice

3. Call “helper” fn, with initial accumulator
In-class Coding: Tree Max #2

;; tree-max : TreeNode<Int> -> Int
;; Returns the maximum value in a given (non-empty) (non-BST) tree
(define (tree-max t0)
  ;; tree-max/a : Tree<Int> -> Int
  ;; accumulator root-val: Int
  ;; invariant: node-data of root parent node (max of empty tree)
  (define (tree-max/a t root-val parent-val)
    (cond
      [(empty? t) root-val parent-val]
      [else (max (node-data t) parent-val
                (tree-max/a (node-left t) root-val parent-val
                (tree-max/a (node-right t) root-val
                (node-data t))))]))
  (tree-max/a t0 (node-data t0)))
Previously

Intertwined Data Definitions

• Come up with a Data Definition for ...

• ... valid Racket Programs
Basic Valid Racket Programs

- 1
- “one”
- (+ 1 2)

;; A RacketProg is a:
;; - Number
;; - String
;; - ???
Valid Racket Programs

• 1
• “one”
• (+ 1 2)

;; A RacketProg is a:
;; - Atom

;; - ???

;; An Atom is a:
;; - Number
;; - String

12
Valid Racket Programs

• (+ 1 2)

List of ... atoms?

“symbol”

;; A RacketProg is a:
;; - Atom
;; - List<Atom> ???

;; An Atom is a:
;; - Number
;; - String
;; - Symbol
Valid Racket Programs

• $((+ 1 2) (- 4 3))$
  - Tree?

• $((+ 1 2) (- 4 3) (/ 10 5))$
  - Each tree “node” is a list, of ...
  - RacketProgs ???
  - But: how many values does each node have??
    - Unknown!

;; A RacketProg is a:
;;   - Atom
;;   - Tree<???>

;; An Atom is a:
;;   - Number
;;   - String
;;   - Symbol
Valid Racket Programs

- (* (+ 1 2)
  (- 4 3))

- (* (+ 1 2)
  (- 4 3)
  (/ 10 5))

- A RacketProg is a:
  - Atom
  - ProgTree

- A ProgTree is one of:
  - empty
  - (cons RacketProg ProgTree)

- Each tree “node” is a list, of ...
- RacketProgs ??

- But: how many values does each node have??

- An Atom is a:
  - Number
  - String
  - Symbol

- Recursive Data Def!
Valid Racket Programs

Also, **Intertwined** DataDefs!

```
;; A RacketProg is a:
;; - Atom
;; - ProgTree

;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```
Intertwined Data

- A set of Data Definitions that reference each other
- Templates should be defined together ...

;;; A RacketProg is a:
;;; - Atom
;;; - ProgTree

;;; A ProgTree is one of:
;;; - empty
;;; - (cons RacketProg ProgTree)

;;; An Atom is one of:
;;; - Number
;;; - String
;;; - Symbol
Intertwined Data

• A set of Data Definitions that reference each other

• Templates should be defined together ...
  • ... and should reference each other’s templates (when needed)

```scheme
;; A RacketProg is one of:
;; - Atom
;; - ProgTree
(define (prog-fn p) "...")

;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
(define (ptree-fn t) "...")

;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
(define (atom-fn a) "...")
```
"Racket Prog" = S-expression!

An S-expression is a Racket program's **syntax** ...

What about its **semantics** (meaning)?

;; A **RacketProg** is one of:
;; - Atom
;; - ProgTree

;; A **ProgTree** is one of:
;; - empty
;; - (cons RacketProg ProgTree)

;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
Syntax vs Semantics (Spoken Language)

Syntax
• Specifies: valid language structures
  • E.g., sentence = noun (subject) + verb + noun (object)
  • “the ball threw the child”
    • Syntactically: valid!
    • Semantically: ???

Semantics
• Specifies: meaning of language structures
Syntax vs Semantics (Programming Language)

**Syntax**
- Specifies: valid language structures
  - E.g., ???

**Semantics**
- Specifies: meaning of language structures
Syntax vs Semantics (Programming Language)

**Syntax**
- Specifies: valid language structures
  - E.g., valid Racket program = s-expressions
  - E.g., valid Python program = ...

**Semantics**
- Specifies: meaning of language structures

Q: What is the “meaning” of a program?
A: The result from “running” it

How does a program “run”?
Running Programs: eval

;; eval : Sexpr -> Result
;; “runs” a given Racket program, producing a “result”

An “eval” function turns a “program” into a “result”

An “eval” function is more generally called an interpreter

(Programs are usually not directly interpreted)

More commonly, a high-level program is first compiled to a lower-level language (and then interpreted)

Q: What is the “meaning” of a program?
A: The result from “running” it

How does a program “run”?
More commonly, a high-level program is first **compiled** to a lower-level language (and then interpreted).

From Lecture 1

“high” level (easier for humans to understand)

![Diagram with a hierarchy of programming languages]

<table>
<thead>
<tr>
<th>English</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification langs</td>
<td>Types? pre/post cond?</td>
</tr>
<tr>
<td>Markup (html, markdown)</td>
<td>tags</td>
</tr>
<tr>
<td>Database (SQL)</td>
<td>queries</td>
</tr>
<tr>
<td>Logic Program (Prolog)</td>
<td>relations</td>
</tr>
<tr>
<td>Lazy lang (Haskell, R)</td>
<td>Delayed computation</td>
</tr>
<tr>
<td>Functional lang (Racket)</td>
<td>Expressions (no stmts)</td>
</tr>
<tr>
<td>JavaScript, Python</td>
<td>“eval”</td>
</tr>
<tr>
<td>C# / Java</td>
<td>GC (no alloc, ptrs)</td>
</tr>
<tr>
<td>C++</td>
<td>Classes, objects</td>
</tr>
<tr>
<td>C</td>
<td>Scoped vars, fns</td>
</tr>
<tr>
<td>Assembly Language</td>
<td>Named instructions</td>
</tr>
<tr>
<td>Machine code</td>
<td>Binary</td>
</tr>
</tbody>
</table>

NOTE: This hierarchy is **approximate**
“High” level (easier for humans to understand)

**Surface language**

“Declarative”

Compiler

**Target language**

More commonly, a high-level program is first compiled to a lower-level language (and then interpreted)

**Specification langs**
- Markup (html, markdown)
- Database (SQL)
- Logic Program (Prolog)
- Lazy lang (Haskell, R)
- Functional lang (Racket)

**JavaScript, Python**
- C# / Java
- C++
- C
- Assembly Language
- Machine code

Common **target languages**:
- bytecode (e.g., JS, Java)
- assembly
- machine code

A virtual machine is just a bytecode interpreter

A (hardware) CPU is just a machine code interpreter!
Semantics

- Specifies: meaning of language **structures**
- So: to “run” a program, we need to see the structure first
Compiler, step 1 = parser

input

surface language

(a compiler actually has many steps ... take a compilers course!)

abstract syntax tree (AST)

output

while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a

e.g., string of chars (less structure)

e.g., tree (more structure)
This itself is a program!

surface language

These must have representations (data definitions!)

input

parser

abstract syntax tree (AST)

output

These must have representations (data definitions!)

SExpr

???
These must have representations (data definitions!)

input

surface language

parser

abstract syntax tree (AST)

output

These must have representations (data definitions!)

???
surface language

These must have representations (data definitions!)

input  SExpr

;; A SimpleSexpr (Ssexpr) is one of:
;;   - Number
;;   - (list '+ Ssexpr Ssexpr)
;;   - (list '-' Ssexpr Ssexpr)
Data Definition Template

When a **Data Definition** is an itemization of compound data ...

**Template** =
- cond to distinguish cases
- Getters to extract pieces
- recursive calls

```scheme
(define (ss-fn s)
  (cond
    [(number? s) ...]
    [(and (list? s) (equal? ‘+ (first s)))
      ... (ss-fn (second s)) ... (ss-fn (third s)) ...]
    [(and (list? s) (equal? ‘- (first s)))
      ... (ss-fn (second s)) ... (ss-fn (third s)) ...]))
```

**Cond guards** must distinguish the different cases

**Cond clause** has getters and recursive calls

;;; A SimpleSexpr (Ssexpr) is one of:
;;; - Number
;;; - (list ‘+ Ssexpr Ssexpr)
;;; - (list ‘- Ssexpr Ssexpr)
Interlude: pattern matching (again)

When a **Data Definition** is an **itemization** of compound data ...

- **Template** =
  - cond to distinguish cases
  - match = cond + getters
  - recursive calls

```
;;; A SimpleSexpr (Sexpr) is one of:
;;;  - Number
;;;  - (list ‘+ Sexpr Sexpr)
;;;  - (list ‘- Sexpr Sexpr)
```

```
(define (ss-fn s)
  (match s
    [(? number?) ... ]
    [`(+ ,x ,y) “Quasiquote” pattern
      ... (ss-fn x) ... (ss-fn y) ... ]
    [`(- ,x ,y) Symbols match exactly
      ... (ss-fn x) ... (ss-fn y) ... ])))
```

“Unquote” defines **new variable name** (for value at that position)
Interlude: pattern matching (again)

• See Racket docs for the full pattern language
Interlude: pattern matching (again)

When a Data Definition is an itemization of compound data ...

• Template =
  • cond to distinguish cases
  • match = cond + getters
  • recursive calls

```
(define (ss-fn s)
  (match s
    [(? number?) ... ]
    [`(+ ,x ,y)
      ... (ss-fn x) ... (ss-fn y) ... ]
    [`(− ,x ,y)
      ... (ss-fn x) ... (ss-fn y) ... ]))
```

• match can be more concise and readable

```
(define (ss-fn s)
  (cond
    [(number? s) ... ]
    [(and (list? s) (equal? ‘+ (first s)))
      ... (ss-fn (second s)) ... ]
    [(and (list? s) (equal? ‘− (first s)))
      ... (ss-fn (second s)) ... ]))
```
A SimpleSexpr (Sexpr) is one of:

- Number
- (list ‘+ Sexpr Sexpr)
- (list ‘- Sexpr Sexpr)
A surface language is a program! This itself is a program!

An **AST** is one of:

- (num Number)
- (plus AST AST)
- (minus AST AST)

**Interp:** Tree structure for Ssexpr prog

(struct num [val])
(struct plus [left right])
(struct minus [left right])

These must have representations (data definitions!)
• Template =

```
(define (ast-fn p)
  (cond
    [(num? p) ... ]
    [(plus? p) ... (ast-fn (plus-left p))
     ... (ast-fn (plus-right p)) ... ]
    [(minus? p) ... (ast-fn (minus-left p))
     ... (ast-fn (minus-right p)) ... ])
```

;; An AST is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)

;; Interp: Tree structure for Sexpr prog
(struct num [val])
(struct plus [left right])
(struct minus [left right])
• **Template** (with match) =

```
(define (ast-fn p)
  (cond match p
        [(num n) ... ]
        [(plus x y) ... (ast-fn x) ...
          ... (ast-fn y) ... ]
        [(minus x y) ... (ast-fn x) ...
          (ast-fn y) ... ]))
```

---

;; An **AST** is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)
;; Interp: **Tree** structure for Ssexpr prog

```
(struct num [val])
(struct plus [left right])
(struct minus [left right])
```
In-class Coding 11/8 #1: parser

```racket
;; parse: SimpleSexpr -> AST
;; Converts a (simple) S-expression to language AST

;; A SimpleSexpr (Sexpr) is a:
;; - Number
;; - (list '+ Sexpr Sexpr)
;; - (list '-' Sexpr Sexpr)

;; An AST is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)
;; Interp: Tree structure for Sexpr
(struct num [val])
(struct plus [left right])
(struct minus [left right])
```
In-class Coding 11/8 #2: eval

```rkt
;; eval-ast: AST -> Result
;; computes the result of given program AST

;; An AST is one of:
;; - (num Number)
;; - (plus AST AST)
;; - (minus AST AST)
;; Interp: Tree structure for Sexpr
(struct num [val])
(struct plus [left right])
(struct minus [left right])
```
No More Quizzes!

but push your in-class work to:
Repo: cs450f23/lecture18-inclass