Programming Languages

Lecture 2
Functions, Pairs, Lists

Adapted from Dan Grossman's
PL class, Univ. of Washington

Review

- Building up Scheme one construct at a time via precise definitions
  - Constructs have *syntax*, *type-checking rules*, *evaluation rules*
    - And reasons they're in the language
  - Evaluation converts an *expression* to a *value*
- So far:
  - Variable bindings
  - Several expression forms: addition, conditionals, …
  - Several types: **`integer, rational, real, boolean`**
- Today:
  - Brief discussion on aspects of learning a PL
  - Functions, pairs, and lists

Five different things

1. Syntax: How do you write language constructs?
2. Semantics: What do programs mean? (Evaluation rules)
3. Idioms: What are typical patterns for using language features to express your computation?
4. Libraries: What facilities does the language (or a well-known project) provide "standard"? (E.g., file access, data structures)
5. Tools: What do language implementations provide to make your job easier? (E.g., REPL, debugger, code formatter, …)

These are 5 separate issues

- In practice, all are essential for good programmers
- Many people confuse them, but shouldn't

Our Focus

This course focuses on semantics and idioms

- Syntax is usually uninteresting
  - A fact to learn, like “The American Civil War ended in 1865”
  - People obsess over subjective preferences [yawn]

- Libraries and tools crucial, but often learn new ones on the job
  - We’re learning language semantics and how to use that knowledge to do great things

Function definitions

Functions: the most important building block in the whole course

- Like Python/C++, have arguments and result
- But no classes, **`this`**, **`return`**, etc.

Example *function binding*:

```
; Note: correct only if y>=0

(define (pow x y)
  (if (= y 0)
      1
      (* x (pow x (- y 1)))))
```

Note: The body includes a (recursive) *function call*: **`(pow x (- y 1))`**

Quick Scheme/Racket review

Simple values: **`34, #f, #t, x, 2/3`**

```
(define v e)  ; evaluates e, becomes the value for variable v.
(+ e1 e2 e3 …)    ; all math is in prefix form.
(if test e1 e2)  ; if-else statement: if test evaluates to #t,
                 ; evaluates and returns e1, else evaluates and
                 ; returns e2.
```

One new one:

```
(cond (test1 e1)   ; if/else if/else statement:
      (test2 e2)   ; if test1 evaluates to #t, returns
      (test3 e3)   ;    whatever e1 evaluates to.
      …)           ; otherwise, if test2 evaluates to #t, returns
                   ;    whatever e2 evaluates to.
                   ; continues with other tests—usually last
                   ; test is #t, which serves as an "else"
```

Function bindings: 3 questions

- Syntax: `(define (f x1 x2 . . . xn) e)`
  - (Will generalize in later lecture)
- Evaluation: ***A function is a value!*** (Don't know how to evaluate it yet.)
  - Adds **f** to environment so *later* expressions can *call* it
  - (Function-call semantics will also allow recursion)
- Type-checking:
  - Again, none done at compile-time.
  - User-defined functions do not allow for any built-in type checking.
    - Similar to Python -- the onus is on the programmer to not call any functions with arguments of the wrong type.
    - Not like C++, where every function you write declares what types the arguments must be.

Function Calls

A new kind of expression!

Syntax: `(e0 e1 e2 . . . en)`

Evaluation:

1. (Under current environment,) evaluate **`e0`** to a function **`f`** that takes arguments **`x1`** through **`xn`** and has **`e`** as the body.
2. (Under current environment,) evaluate arguments **`e1`** through **`en`** resulting in values **`v1`** through **`vn`.**
3. Result is evaluation of **`e`** in an environment extended to map **`x1`** to **`v1`**, …, **`xn`** to **`vn`**
   - ("An environment" is actually the environment where the function was defined, and includes **`f`** for recursion)

Example, extended

```
; only correct for y >= 0
(define (pow x y)
  (if (= y 0)
      1
      (* x (pow x (- y 1)))))

(define (cube x)
  (pow x 3))

(define sixtyfour (cube 4))

(define fortytwo (+ (pow 2 4) (pow 4 2) (cube 2) 2))
```

Some gotchas

- Can't add extra parentheses like in Python/C++.
  - (+ 1 2) is fine… (+ (1 2)) is not fine, and neither is ((+ 1 2)).
  - Parentheses have a very particular meaning in Scheme; they are not just used for changing precedence or grouping.
    - Using prefix notation for everything pretty much eliminates having to use parens for precedence.
- No "return" statement.
  - Functions only have a single expression as the body anyway.
  - Evaluating that statement becomes the return value.

Recursion

- If you're not yet comfortable with recursion, you will be soon ☺
  - Will use for most functions taking or returning lists

- "Makes sense" because calls to same function solve "simpler" problems

- Recursion more powerful than loops
  - We won't use a single loop in Scheme
  - Loops often (not always) obscure simple, elegant solutions

Pairs and lists

So far: numbers, booleans (#t and #f), conditionals, variables, functions

- Now ways to build up data with multiple parts
- This is essential
- C++ examples: classes with fields, arrays

Rest of lecture:

- Pairs and lists
- These are our basic data structures that we use to create all other data structures.

Later: Other more general ways to create compound data

Pairs

We need a way to *build* pairs and a way to *access* the pieces

Build:

- Syntax: `(cons e1 e2)`
- Evaluation: Evaluate **`e1`** to **`v1`** and **`e2`** to **`v2`**; result is **`(v1 . v2)`**
  - A pair of values is a value.

Pairs

We need a way to *build* pairs and a way to *access* the pieces

Build:

- Alternate syntax: `'(v1 . v2)`
- Evaluation: No evaluation!
  - This is how to make a “literal” pair, where v1 and v2 are not evaluated.
  - Similar to using double quotes to make a string literal in C++.
  - E.g.: (cons (+ 1 2) (+ 3 4)) makes the pair (3 . 7).
  - E.g.: ‘(3 . 7) also makes the pair (3 . 7).
  - E.g.: However, ‘((+ 1 2) . (+ 3 4)) makes the pair ((+ 1 2) . (+ 3 4))

Pairs

We need a way to *build* pairs and a way to *access* the pieces

Access:

- Syntax: `(car e)` and `(cdr e)`

- Evaluation: Evaluate `e` to a pair of values and return first or second piece
  - Example: If `e` is a variable `x`, then look up `x` in environment

Examples

Functions can take and return pairs

```
(define (swap pair)
  (cons (cdr pair) (car pair)))

(define (sum-two-pairs p1 p2)
  (+ (car p1) (cdr p1) (car p1) (cdr p2)))

(define (div-mod n1 n2)
  (cons (quotient n1 n2) (remainder n1 n2)))
  ; returning more than one value is a pain in C++
```

Lists

- No triples or longer “tuples.” (where the # of elements is fixed)
- However, we do have lists that can hold any number of elements.

Need ways to *build* lists and *access* the pieces…

Building Lists

- The empty list is a value:

```
'()
```

- In general, a list of values is a value; elements separated by spaces:

```
'(v1 v2 ...vn)
```

- If **`e1`** evaluates to **`v`** and **`e2`** evaluates to a list **`(v1 … vn)`**, then **`(cons e1 e2)`** evaluates to **`(v v1 … vn)`**

Accessing Lists

- **`(null? e)`** evaluates to **`#t`** if and only if **`e`** evaluates to **`'().`**

- If **`e`** evaluates to **`'(v1 v2 … vn)`** then **`(car e)`** evaluates to **`v1`**
  - (raise exception if **`e`** evaluates to **`'()`**)

- If **`e`** evaluates to **`(v1 v2 … vn)`** then **`(cdr e)`** evaluates to **`(v2 … vn)`**
  - (raise exception if **`e`** evaluates to **`'()`**)
  - Notice result is a list

Example list functions

```
(define (sum-list lst)
  (if (null? lst)
      0
      (+ (car lst) (sum-list (cdr lst)))))


(define (countdown num)
  (if (= num 0)
      '()
      (cons num (countdown (- num 1)))))
```

Recursion again

Functions over lists are usually recursive

- Only way to “get to all the elements”

- What should the answer be for the empty list?
  - Usually, this is your base case.
- What should the answer be for a non-empty list?
  - Typically in terms of the answer for the cdr of the list!

Similarly, functions that produce lists of potentially any size will be recursive

- You create a list is out of smaller lists.

Two other ways to build lists

- List function
  - Makes a list out of all arguments.
  - Arguments can be of any data type.
  - **`(list e1 e2 … en)`** evaluates **`e1`** through **`en`** to values **`v1`** through **`vn`**; returns the list **`'(v1 v2 … vn)`**.
- Append function
  - Concatenates values inside lists given as arguments.
  - Arguments *must* be lists.
  - **`(append e1 e2 … en)`** evaluates **`e1`** through **`en`** to values **`v1`** through **`vn`**;
  - If **`v1`** = **`(v11 v12 … )`** and **`v2`** = **`(v21 v22 … )`** etc, then return value is **`(v11 v12 … v21 v22 … v31 v32 …)`**.

Lists of pairs

Processing lists of pairs requires no new features. Examples:

```
(define (sum-pair-list lst)
   (if (null? lst)
    0
    (+ (car (car lst)) (cdr (car lst)) (sum-pair-list (cdr lst)))))

(define (firsts lst)
   (if (null? lst)
       '()
       (cons (car (car lst)) (firsts (cdr lst)))))

(define (seconds lst)
   (if (null? lst)
       '()
       (cons (cdr (car lst)) (seconds (cdr lst)))))

(define (sum-pair-list2 lst)
   (+ (sum-list (firsts lst)) (sum-list (seconds lst))))
```