CSC 347 - Concepts of Programming Languages

Scheme

Instructor: Stefan Mitsch

Learning Objectives

- Transition to functional programming
- Understand expressions in Scheme
- Understand cons cells and lists in Scheme
- Revisit recursion

Lisp and Scheme

- Lisp (LISt Processor)
- Influential programming language from the 1950s
- Originally motivated by logic / AI applications
- Pioneered many PL concepts:
  - automatic garbage collection
  - first-class, higher-order, nested functions
  - read-eval-print loop including runtime compilation with "eval"
  - sophisticated macro system
  - multiple dispatch / multi-methods

Lisp and Scheme

- Dialects: Common Lisp, Scheme, Clojure, Racket
- We will use Scheme
- Sample Scheme function to find the length of a list

```
; This (recursive) function calculates the length of a linked list.
(define (length l)
  (if (equal? l ())
    0
    (+ 1 (length (cdr l)))))
```

- Lots of Infuriating and Silly Parentheses

Running Scheme

- Use repl.it
- On MacOS, use homebrew: `brew search scheme` and `brew search chicken`

Literals

- Number literal: `5`
- String literal: `"hello world"`
- Symbol `'helloworld`

Arithmetic

- Arithmetic expressions use prefix notation

```
; (1 + 2) * 3 would be written in Scheme as follows
(* (+ 1 2) 3)
```

- Parentheses are required for each operator
- Benefit: operator precedence not necessary
- But careful with operator associativity; try out

```
(+ 10 5 2)
(- 10 5 2)
```

Operator Terminology

- Prefix notation: operator *before* arguments: `+ 1 2`
- Infix notation: operator *between* arguments: `1 + 2`
- Postfix notation: operator *after* arguments: `1 2 +`

Functions

- Define a function `square` with parameter `n`

```
(define (square n) (* n n))
```

- Invoke the `square` function: `(square 5)`
- Invoke the `square` function twice: `(square (square 5))`

Defining Functions

- General form is

```
(define (f param_1 param_2 ... param_m)
  e_1 e_2 ... e_n)
```

- Takes `m` arguments
- Body of function is a sequence of expressions
- `e_1`, `e_2`, ..., `e_n-1` evaluated for side effect
- `e_n` is evaluated and its result is returned
- No `return` keyword, no statements, just expressions
- Optional keyword `begin`

```
(define (f param_1 param_2 ... param_m)
  (begin e_1 e_2 ... e_n))
```

Invoking Functions

- Invoke function `f` with `m` arguments: `(f e_1 e_2 ... e_m)`
- Parentheses are required: `(square 5)`
- Try in Scheme REPL: `square 5`

Evaluation Order

- Expression `(f M N)` is evaluated by
    1. Evaluating expression `M` to value `U`
    2. Evaluating expression `N` to value `V`
    3. Invoking function `f` with values `U` and `V`
- `define` is a *special form*, not a *function*, so it does not obey this convention

Booleans and Conditionals

- Operator `=` tests number equality

```
(define (zero n) (= n 0))
```

- Boolean values are `#t` and `#f`
- Conditional `if` is a non-strict *special form*

```
(define (safe-divide m n)
  (if (= n 0)
      "divide by zero"
      (/ m n)))
```

Recursive Functions

- Recursive functions are common in Scheme
- Factorial using conditional *expressions*

```
(define (fact n)
  (if (<= n 1)
      1
      (* n (fact (- n 1)))))
```

- Recall C factorial using conditional *expressions*

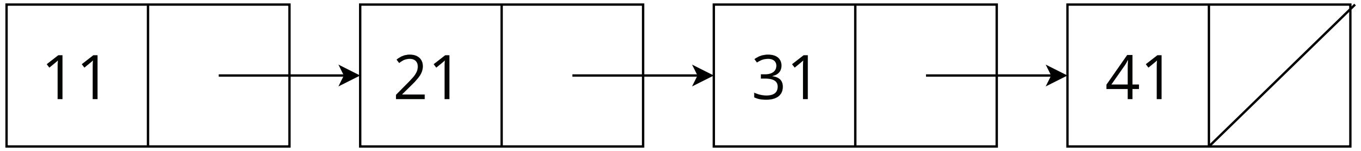
```
int fact (int n) {
  return (n <= 1) ? 1 : n * fact (n - 1);
}
```

Cons Cells

- A *cons cell* is a pair of two pieces of data
- Pair of numbers: `(cons 1 2)`
- Pair of strings: `(cons "hello" "world")`
- Pair of a number and a string: `(cons 1 "world")`
- Functions `car` and `cdr` extract components

```
(car (cons 1 "world"))
(cdr (cons 1 "world"))
```

Cons Cells for Linked Lists



- Cons cells (pairs) are used to represent linked lists

```
(let ((mylist (cons 11 (cons 21 (cons 31 (cons 41 ()))))))))
```

- `car` position for elements: `(car mylist)` is `11`
- `cdr` position for next cons cell: `(cdr mylist)` is `(cons 21 (cons 31 (cons 41 ())))`

Cons Cells for Linked Lists

- Linked lists built up using `()` and `cons`
- Empty list: `()`
- Singleton list containing 41 only: `(cons 41 ())`
- List containing 11, 21, 31, 41: `(cons 11 (cons 21 (cons 31 (cons 41 ()))))`
- Lists can be heterogeneous: `(cons 11 (cons "hello" ()))`
- Cons cells can create more complex data structures:

```
(cons
  (cons 11 (cons 12 ()))
  (cons 21 ())
)
```

Syntactic Sugar for Lists

- Quotation `quote` special form prevents evaluation

```
(quote (3))
(quote (1 2 3))
```

- Operator `'` is shorthand for `quote`

```
'(3)
'(1 2 3)
```

- Function `list` evaluates args, puts results in a list

```
(list 3)
(list 1 2 3)
(list 1 2 (+ 1 2))
```

Equality Testing for Lists

❓ Different ways of comparing for equality?

- Pointer equality compares two pointers
- Structural equality traverses two structures
- `eq?` for pointer equality: `(eq? (cons 1 (cons 2 (cons 3 ()))) '(1 2 3))`
- `equal?` for structural equality: `(equal? (cons 1 (cons 2 (cons 3 ()))) '(1 2 3))`

Recursive Functions on Lists

❓ How do we manipulate complex data structures one element at a time?

- Compute length of linked list recursively

```
(define (length l)
  (if (equal? l ())
    0
    (+ 1 (length (cdr l)))))
```

- Call: `(length '(5 6 7 8 9))`

Evaluate a Recursive Function

Evaluate `(length '(5 6 7))`

- > `(if (equal? '(5 6 7) '()) 0 (+ 1 (length (cdr '(5 6 7)))))`
- > `(+ 1 (length (cdr '(5 6 7))))`
- > `(+ 1 (length '(6 7)))`
- > `(+ 1 (+ 1 (length '(7))))`
- > `(+ 1 (+ 1 (+ 1 (length '()))))`
- > `(+ 1 (+ 1 (+ 1 0)))`
- > `(+ 1 (+ 1 1))`
- > `(+ 1 2)`
- > `3`

Dynamic Types

```scheme
(symbol? 'x)
(number? 1)
(boolean? #t)
(string? "x")
(procedure? (lambda (x) (+ x 1)))
```

```scheme
(pair? '(1 . 2))
(pair? '(1))
; (pair? '())
; (list? '(1 . 2))
(list? '(1))
(list? '())
```

- List only has one structure type: the pair.
- A non-empty list is just a special type of pair, with a terminal

S-Expressions

- Pairs are a kind of *Symbolic-Expression* (*S-Exp*)
- S-Exps also include non-structured values, including numbers, booleans, strings, symbols and `'()`
- Parsing a scheme program results in an S-Exp, which is then sent to `eval` for evaluation
- `(quote exp)` causes `exp` to be parsed without evaluation, resulting in an S-Exp

Read-Eval-Print Loop (REPL)

- Quoting delays evaluation

```
(+ 1 2)
'(+ 1 2)
(cons '+ '(1 2))
(car '(+ 1 2))
```

- Function `eval` evaluates an expression

```
(eval (cons '+ '(1 2)))
(define (add-all l) (eval (append '(+) l)))
(add-all '(1 2 3))
```

- Function `read` reads an expression

```
(read)
(eval (read))
(eval (append '(+) (read)))
```

Summary

- See Worksheet 1 for how to install the SISC Scheme interpreter
- Books
  - R. Kent Dybvig: The Scheme Programming Language, 4th Edition.
  - Scheme Programming Wikibook
- Revised Reports on the Algorithmic Language Scheme
  - Revised Report on the Algorithmic Language Scheme 1978
  - R5RS-Revised, 1998
  - R6RS-Revised, 2007
  - R7RS-Revised, 2013