Project 4 Overview

- Scheme programming
  - Write some functions in Scheme
- Scheme interpreter
  - Given Scheme AST
  - Evaluate AST
- Scheme parser
  - Write recursive descent parser
  - Build Scheme AST

Scheme

- Functional programming language
  - Steele & Sussman @ MIT, 1975
  - Based on LISP
    - Lots of idiotic Stupid Parentheses
    - But uses lexical scoping
    - Resembles lambda calculus

- Features
  - Higher order functions – lambda (x) (…)
  - Builds lists using cons cells - cons, car, cdr

Scheme Examples

- Function evaluation
  - (+ 1 2) evaluates to 3
  - (+ 1 2 3 4 5) evaluates to 15
  - (- 3 4) evaluates to -1
  - (- 3 4 5) evaluates to -6

- Booleans
  - (= 1 2) evaluates to #f
  - (= 1 1) evaluates to #t
  - (bool? #t) evaluates to #t
  - (bool? 6) evaluates to #f

- Global bindings
  - (define three 3)
  - (+ three 4) evaluates to 7

- Creating functions
  - (define add-two (lambda (n) (+ n 2)))
  - (add-two 5) evaluates to 7
Scheme Examples

• Recursive functions
  – (define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))
  – (fact 3) evaluates to 6
  – (fact 5) evaluates to 120

• Higher order functions
  – (define x 52)
  – (define foo (lambda (x) (lambda (y) (+ x y))))
  – (define x 25)
  – ((foo 3) 4) evaluates to 7

Note lexical scoping for x in foo

Scheme Examples

• Lists
  – (define x (cons 3 2))
  – (car x) evaluates to 3
  – (cdr x) evaluates to 2
  – (define y (cons 4 x))
  – (car y) evaluates to 4
  – (cdr y) evaluates to (3 2)

Use cons to build lists
Use car / cdr to deconstruct lists

Starting OCaml Code – scheme.ml

• Type ast
  – Represents Scheme abstract syntax tree
    • type ast =
      | Id of string
      | Num of int
      | Bool of bool
      | String of string
      | List of ast list

• Type value
  – Represents Scheme values
    • type value =
      | Val_Num of int
      | Val_Bool of bool
      | Val_String of string
      | Val_Nil
      | Val_Cons of value * value

  – You will need to add new fields (e.g., closure)
    • But do not modify existing fields!

Note lexical scoping for x in foo
Project 4 – Part 1

- Scheme programming
  - Gain experience with Scheme programs
  - Write simple recursive functions
    - `double x` → two times x
    - `powof2 x` → true (#t) iff x is a power of 2
    - `sum l` → sum of the integer list l
    - `map f l` → list containing elements of l with f applied to them

Project 4 – Part 2

- Scheme interpreter
  - Given Scheme AST
    - Evaluate AST to produce value
      - define, values, lambda, identifiers, function calls, primitives
  - Can test interpreter without parser
    - Using manually constructed Scheme AST

Project 4 – Part 3

- Scheme parser
  - Given scanner
    - Converts input strings into sequence of tokens
  - Write recursive descent parser
    - Convert list of tokens into Scheme AST

Project 4 Notes

- Project files
  - `basic.scm` → your scheme code for part 1
  - `scheme.ml` → your code. Make all your edits here
  - `main.ml` → interpreter using code from scheme.ml
  - `sample.output` → example input / output (public test)

- Testing
  - `ocaml scheme.ml` → test for syntax / type errors
  - `ocaml main.ml` → run scheme interpreter
  - `ocaml public_eval1.ml` → run 1st eval public test
  - Etc…