CSE 341: Programming Languages

Autumn 2005

Lecture 18 — Scheme Intro, Several Binding Forms

Scheme

- Like ML, functional focus with imperative features
  - anonymous functions, function closures, etc.
  - but every binding is mutable
- A really minimalist syntax/semantics
  - In the LISP tradition
  - Current standard is 50 pages
- Dynamically typed, type safe
  - Less “compile-time” checking
  - Accepts more perfectly reasonable programs
- Some “advanced” features for decades
  - Programs as data, hygienic macros, continuations

Which Scheme?

Scheme has a few dialects and many extensions.

We will use "PLT → Pretty Big" for the language and DrScheme as a convenient environment. Available in the ugrad labs, or you can download it for a personal machine.

Most of what we do will be "pure Scheme".

Good documentation available online, including the entire text of *Structure and Interpretation of Computer Programs* (linked from the 341 page)

Scheme syntax

Syntactically, a Scheme term is either an *atom* (identifier, number, symbol, string, ...) or a sequence of terms *(t1 ... tn)*.

Note: Scheme used to get (still gets?) “paren bashed”, which is hilarious in an XML world.

Semantically, identifiers are resolved in an environment and other atoms are values.

The semantics of a sequence depends on *t1*:

- certain character sequences are “special forms”
- otherwise a sequence is a function application. Semantics same as ML — evaluate them, then call function (call-by-value)

Some special forms

- `define`
- `lambda`
- `if`, `cond`, `and`, `or`
- `let`, `let*`, `letrec`

Some predefined values

- `#t`, `#f`
- `()`, `cons`, `car`, `cdr`, `null?`, `list`
- `eq?`, `equal?`
- a "numeric tower" (integer, rational, real, complex, number) with math operations (e.g., +, =, <) defined on all of them
- tons more (strings vs. symbols discussed later)

Note: Prefix and variable-arity help make lots of things functions.

Parens Matter

Every parenthesis you write has meaning – get used to that fast!

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1))))) ; correct
(define (fact n) (if (= n 0) (1) (* n (fact (- n 1)))))
(define (fact n) (if = n 0 (1) (* n (fact (- n 1)))))
(define fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))
(define (fact n) (if (= n 0) 1 (* n fact (- n 1))))
(define (fact n) (if (= n 0) 1 (* n ((fact) (- n 1)))))
```

Local bindings

There are 3 forms of local bindings with different semantics:

- `let`
- `let*`
- `letrec`

Also, in function bodies, a sequence of definitions is equivalent to letrec.

But at top-level redefinition is assignment!

This makes it ghastly hard to encapsulate code, but in practice:

- people assume non-malicious clients
- implementations provide access to "real primitives"

For your homework, assume top-level definitions are immutable.