

# CSE 341: Programming Languages

Hal Perkins

Spring 2011

Lecture 13— Introduction to Scheme

# Scheme

---

- Like ML, functional focus with imperative features
  - anonymous functions, closures, no return statement, etc.
  - but every binding is mutable in standard Scheme (although not always in Racket's Pretty Big version)
- A really minimalist syntax/semantics
  - In the LISP tradition
  - Classic standard is 50 pages
- Dynamically typed
  - 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  $\rightarrow$  Pretty Big” for the language and DrRacket as a convenient environment.

- Installed on Windows machines in the lab, maybe Linux later
- Easy download/install for your own machine (Mac, Windows, Linux)

Most of what we do will be “pure Scheme”.

Exceptions include `define-struct` and the Racket/DrScheme module system.

Historical note: Racket is the new name (as of Summer 2010) for the venerable DrScheme environment. The old name may slip in now and then, even though we’re trying to bring things up to date.

# Scheme syntax

---

Syntactically, a Scheme term is either an *atom* (identifier, number, symbol, string, ...) or a sequence of terms ( $t_1 \dots t_n$ ).

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  $t_1$ :

- certain character sequences are “special forms”
- otherwise a sequence is a function application (semantics same as ML)

## Some special forms

---

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

## Some predefined values

---

- `#t`, `#f`
- `()`, `cons`, `car`, `cdr`, `null?`, `list`
- a “numeric tower” 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!

Correct:

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

Incorrect:

```
(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)))))
```

# Dynamic Typing

---

Many things that the “type-checker” forbids in Java or ML are run-time errors in Scheme (like a Java NullPointerException):

- Calling a function with the wrong number of arguments
- Passing a “cons cell” to +
- Passing a function to car
- ...

Don't need datatype/class definitions, etc. to appease the type-checker

- Just return a boolean or a number; caller can use *predicates* to determine at run-time what it got
- Make a list that can hold numbers or other lists or whatever
- ...

Most of a later lecture will consider pros/cons of static checking

# Local bindings

---

There are 3 forms of local bindings with different semantics:

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

(Also, in function bodies, a sequence of `defines` is the same as `letrec`.)

But at top-level, redefinition is assignment!

- And unlike ML, forward references work fine, as long as they are evaluated “later” (under a function body)

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.