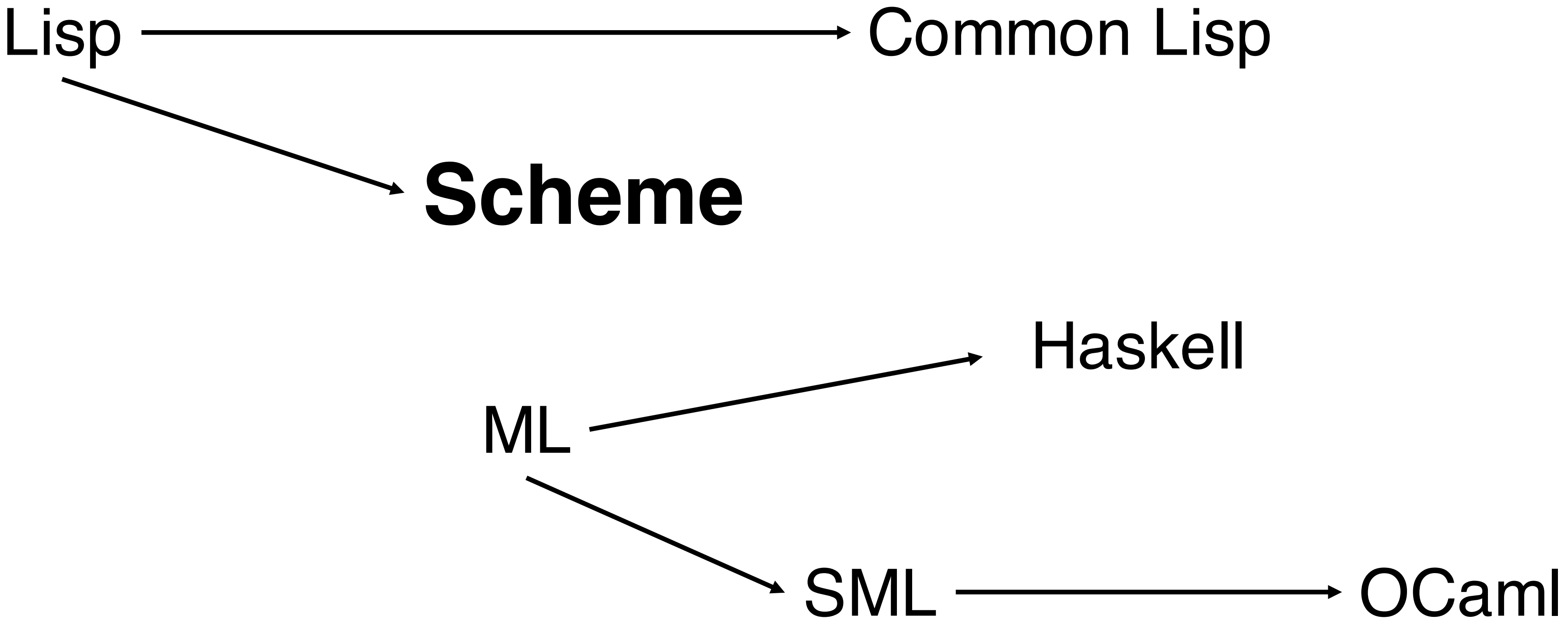
Functional Programming and Scheme

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Functional Languages



Lisp

Common Lisp

Scheme

Haskell

ML

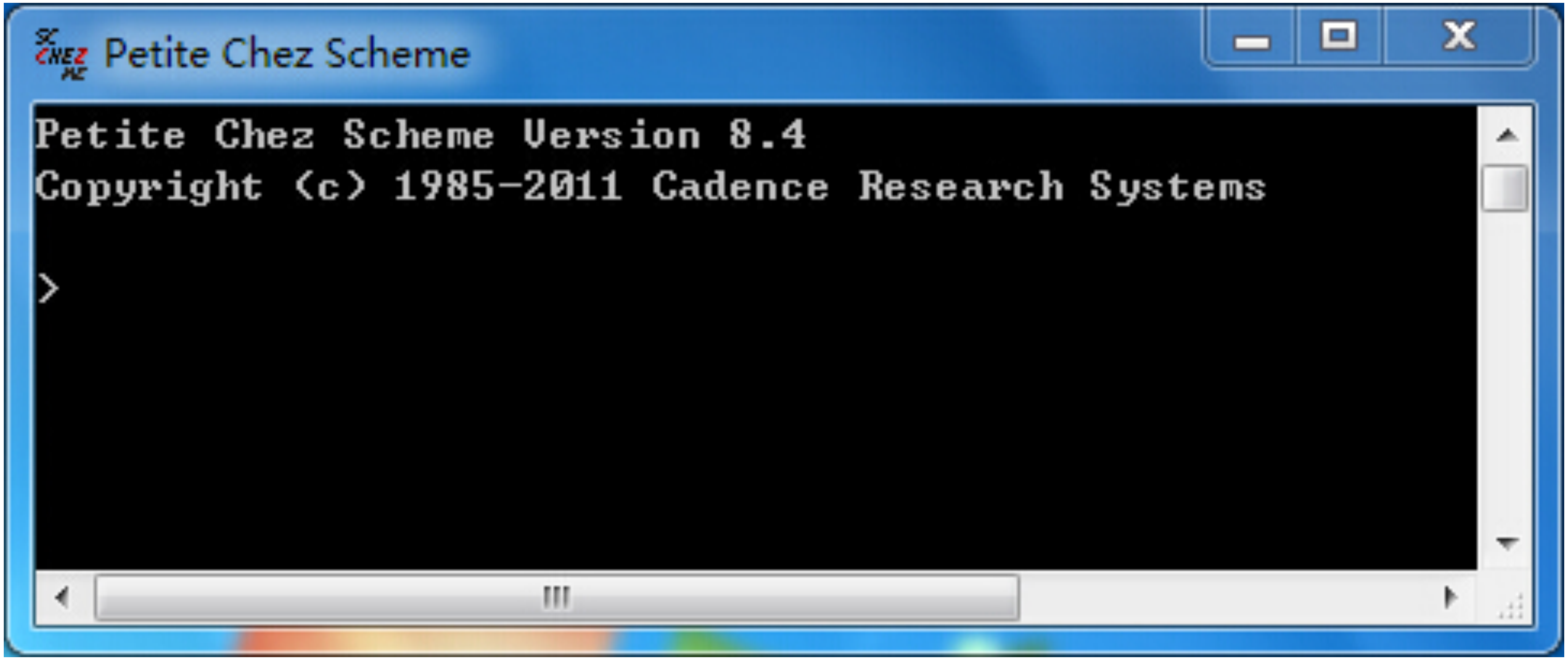
SML

OCaml

Running Scheme

Petite Scheme

http://www.scheme.com/



```
Petite Chez Scheme Version 8.4
Copyright (c) 1985-2011 Cadence Research Systems

>
```

Scheme

We focus on the non-imperative subset of Scheme (no assignments, whiles)

Syntax:

- Atoms: e.g., 3, #t, #f, “abc”
- Lists: (3 #t “abc”)

Prefix form

- Functions: (sqrt 2), (+ 1 2 4)
- Comments: (+ 1 2) *; evaluate to 3*

Read-Eval-Print Interpreter

- (sqrt 2) is the same as (eval (sqrt 2))

Parenthesis in Scheme

In Scheme, () is used for lists, and both code and data are *lists*

- `(+ 2 3)`
- `(2 3 4)`

Function is a *first-class* value

Lists are evaluated as function calls

- `(+ 2 3)` Evaluates to 5
- `((+ 2 3))` Exception: 5 is not a procedure
- `(2 3 4)` Exception: 2 is not a procedure

Lists w/o evaluation: use apostrophe (')

- `'(2 3 4)` Returns `(2 3 4)`

Expressions

Arithmetic:

```
(* 1 2 3 4)
(+ (* 1 2) (* 2 4))
```

Relational:

```
(< (* 1 2) (+ 1 1))
(and (> 2 4) #f)
```

Conditional:

```
(if (< 2 3) 1 (+ 2 3))
(* 2 (if (< 3 5) 3 “abc”))
```

Type test:

```
(number? 2)   (number? #f)
(string? “a”) (string? 1)
(boolean? #t) (boolean? 1)
```

Environment

Environment binds identifiers to values

- `+,*,-,/` are identifier bound to values
- Functions are values in Scheme

Built-in identifiers in initial environment

- `+, sqrt, positive?, max, …`

Add bindings to the environment

- Key word: define

Definition

Built-in identifiers

```
(max 1 2 3)
```

“define” introduce global identifiers

```
 (define pi 3.14159)
 (define radius 2)
 (* pi (* radius radius))
 (define circumference (* 2 pi radius))
circumference
```

Local Definition

Let Expr.:

```
(let ((x 3)
      (y (sqrt 7)))
  (+ x y))
```

General form:

```
(let ((x1 exp1)
      (x2 exp2)
       …
      (xn expn))
  body_exp)
```

x1,x2,…,xn can be used in body_exp

Local Definition

Let* Expr.:

```
(let* ((x 3)
       (y x))
  (+ x y))
```

General form:

```
(let* ((x1 exp1)
       (x2 exp2)
        …
       (xn expn))
  body_exp)
```

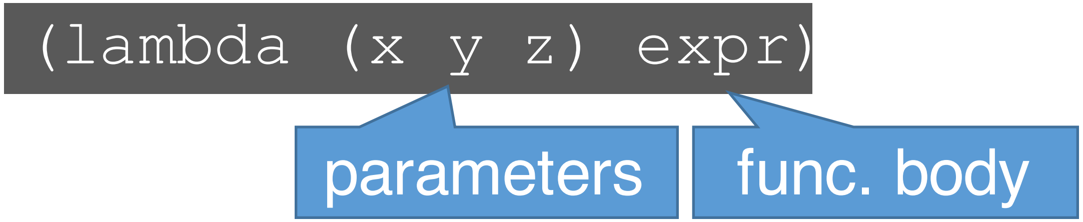
x1 can be used in exp2, exp3, ..., body_exp

x2 can be used in exp3, exp4, ..., body_exp

...

Anonymous Function

Anonymous functions are introduced by `lambda`



```
(lambda (x y z) expr)
```

parameters

func. body

Example

```
((lambda (x) x) 1)
```

```
((lambda (x y z) (+ x y z)) 1 2 3)
```

```
((lambda (f) (f 2)) (lambda (x) (* 2 x)))
```

Named Function

```
(define pi 3.14159)
```

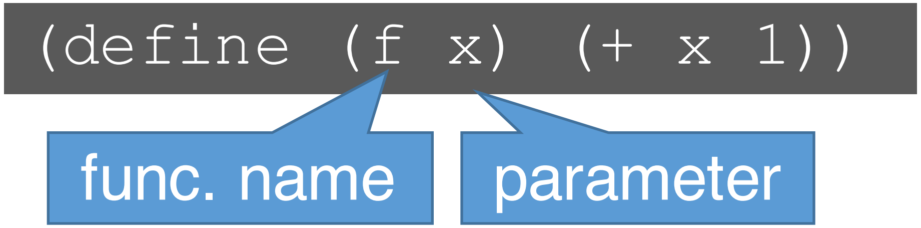
In Scheme, function is a first-class value, so similarly, we can define named functions

```
(define f (lambda (x) (+ x 1)))
(f 1)
```

```
(let ((f (lambda (x) (+ x 1)))
      (g (lambda (y) (* y 2))))
    (+ (f 1) (g 2)))
```

Named Function

In Scheme, named functions can be introduced in a more concise way:

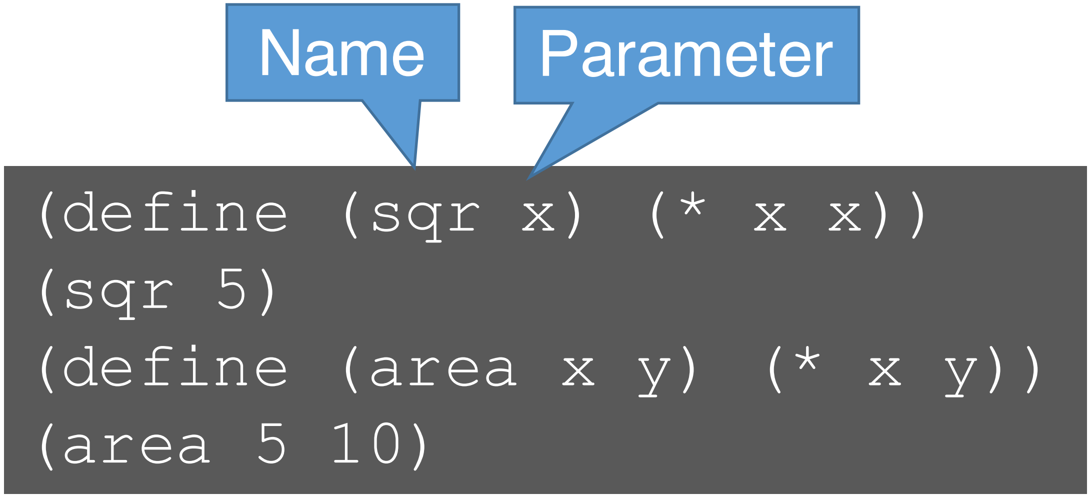


```
(define (f x) (+ x 1))
```

is the same as

```
(define f (lambda (x) (+ x 1)))
```

Named Function



```
(define (sqr x) (* x x))
(sqr 5)
(define (area x y) (* x y))
(area 5 10)
```

Common use:

```
(define (f x y z)
  (let ((x1 exp)
        (x2 exp))
   f's body))
```

Function Examples

```
(define (test x)
        (cond ((number? x) “num”)
             ((string? x) “str”)
             ((list? x) “list”)
           (else “other”))))
```

```
(define (abs x)
         (cond ((< x 0) (- x))
                (else x)))
```

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

Lazy vs. Eager Evaluation

Call by value (eager evaluation)

- Function arguments are fully evaluated before application (default in Scheme)

```
  ((lambda (x y) x) (* 1 2) (* 2 4))
= ((lambda (x y) x) 2 8)
= 2
```

Call by name (lazy evaluation)

- Function arguments are evaluated at use location

```
  ((lambda (x y) x) (* 1 2) (* 2 4))
= (* 1 2)
= 2
```

Lazy vs. Eager Evaluation

Any evaluation order give the same result, ***as long as the evaluation terminates***

```
(define Omega (lambda (x) (x x)))
```

Call by value (eager evaluation)

```
  ((lambda (x) 0) (Omega Omega))
= ((lambda (x) 0) (Omega Omega))
= …
```

Call by name (lazy evaluation)

```
  ((lambda (x) 0) (Omega Omega))
= 0
```

Scheme: Key Points

Evaluation of expressions

```
(e0 e1 e2 e3)
```

Key words

```
define, if, cond, `
```

No static type system

```
(define (f x) (+ x “abc”))
```

```
(* 2 (if (< 3 5) 3 “abc”))
```

Scheme: Key Points

No assignments, no iterations (loops)

All variables are immutable (mathematical symbols)

Need to think computation in recursive way

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