

Scheme

- Sources of Information:
 - Revised⁴ Report on the Algorithmic Language Scheme. (Revised⁵ Report also).
 - IEEE Standard for the Scheme Programming Language (\$\$\$).
 - Books:
 - The Scheme Programming Language.
 - The Schematics of Computation.
 - The Little Schemer, The Seasoned Schemer.
 - MIT Scheme Users and Reference Manuals
 - The WWW.

Scheme Features

- functional programming language
 - program is expressed as function call(s).
- Structured data
 - strings, lists & vectors.
- garbage collection
- procedures are first-class data objects
- lexical scoping

Scheme Features (cont.)

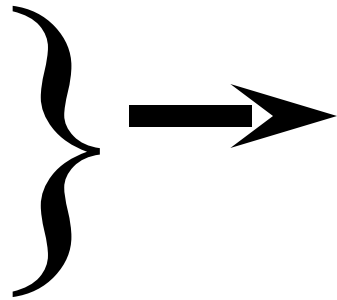
- programs share a common printed format with data.
 - programs can easily be used as data
- tail recursion
 - to support iteration efficiently
- continuations
 - goto !?!?!?
 - backtracking, multithreading

Scheme Features (cont.)

- Few syntactic forms
- syntactic extension is possible (R5)

Scheme Syntax

- keywords
- variables
- symbols
- structured forms
- constant data
- whitespace
- comments



Identifiers:

a-z

A-Z

0-9

? ! . + - * / < = >

: \$ % ^ & _ ~

can't start with
numeric, +, - or .

Identifiers

- Identifiers must be delimited by whitespace, parentheses, double quote or the comment character ‘;’.
- No length limit.
- Case insensitive.

Structured Forms and List Constant Syntax

- *Structured forms* and list constants are enclosed within parentheses:

(a b c)

(* (- x 2) y)

- The empty list is written **()**

Boolean values

- Boolean value for true is written **#t**
- Boolean value for false is written **#f**
- Scheme conditional expressions treat **#f** as false and everything else as true.
- Some Scheme implementations treat **()** as false.

Whitespace and Comments

- Whitespace includes spaces, tabs and newlines.
- Whitespace length is not important (one space is the same as 100).
- Scheme expressions can span several lines.
- Comments appear between a ‘;’ and the end of a line.

Some Naming Conventions

- Predicate names end in a ?
 - examples: **eq?** **zero?** **string=?**
 - exceptions: **=** **<** **>** **<=** **>=**
- Procedures and syntactic forms that cause side effects end with !
 - example: **set!**
 - exceptions: **write** **display** **read** **load**

Scheme Expressions

- An expression can be a constant data object
 - string, number, symbol, list
 - examples: **17.5** **3/5** **"Hello World"**
- Or an expression can be a *procedure application*:

(foo 1 2)

application of the procedure foo.

Procedure Application

- Any procedure application is written in prefix form:

(procedure_name arg1 arg2 ...)

- Arithmetic operations are not special, they are written in prefix form:

`(+ 10 20)`

`(* (* (* 52 7) 24) 60)`

Precedence and Associativity

- Using prefix notation means that there are no rules for operator precedence or operator associativity!
- In a nested expression, innermost expressions are computed first (so the programmer determines the order of evaluation, not a bunch of rules).

Lists

- Lists are written as sequences of objects surrounded by parentheses.

- Examples:

`(1 2 3 4)`

`("Hi" "Dave")`

`("One" 1)`

`(a b c (d e f))`

List vs. Procedure Application

- A procedure application looks just like a list, so how does scheme know the difference ?
- We must tell scheme to treat a list as data rather than as a procedure application.
- The **quote** procedure forces a list to be treated as data: **(quote (+ 3 4))**
- This is so common we also can use the shorthand notation: **`(+ 3 4)**

List manipulation procedures

- **car** returns the first element of the list
- **cdr** returns the remainder of the list

`(car '(a b c)) => a`

`(cdr '(a b c)) => (b c)`

- **cons** constructs lists by adding a new element to the beginning of a list

`(cons 'a '(b c)) => (a b c)`

Proper and Improper Lists

- a *list* is a sequence of *pairs*.
- Each *pair*'s **cdr** is the next *pair*.
- The **cdr** of the last pair in a proper list is the empty list.
- If the **cdr** of the last pair is not `()`, the list is an *improper list*.

Dotted Pair

- Improper lists are printed as a *dotted-pair*.

`(cons 'a 'b) => (a . b)`

`(cdr '(a . b)) => b`

`(cons 'a '(b . c)) => (a b . c)`

The procedure **list**

- **list** takes any number of arguments and builds a proper list:

`(list 'a 'b 'c) => (a b c)`

`(list 'a) => (a)`

`(list) => ()`

`(list '(a . b)) => ((a . b))`

Expression Evaluation

- Procedure application:

(procedure_name arg1 arg2 ...)

find the value of *procedure_name* ←

find the value of *arg1, arg2, ...* ←

apply the value of *procedure_name* to the
values of *arg1, arg2, ...*

In any order!

quote evaluation

- The rules for evaluation don't work for **quote** - why not ?
- **quote** does not evaluate the subexpression at all.
- **quote** is a different *syntactic form!*
- So far we've seen 3 syntactic forms:
 - procedure application, constant objects and quote expressions.

Variables

- during procedure application the value of variables is determined - so how do we set the value of a variable?
- The **let** syntactic form:

```
(let ((x 1))  
  (+ x 3)) => 5
```

```
(let ((x 1) (y 2))  
  (+ x y)) => 3
```

Let Expressions

- Let expressions include a list of variable-value pairs and a sequence of expressions (called the body).
- Let expressions are often used to simplify an expression that contains multiple copies of the same subexpression:

```
(let ((x (+ 2 2))) (* x x)) => 16
```

Let Expressions

- Procedure names are no different than any other list element - so we can do this:

`(let ((f +)) (f 1 2)) => 3`

`(let ((+ *)) (+ 2 3)) => 6`

- The values bound by a **let** are only *visible* within the body of the **let**.

Let can be nested

```
(let ((x 1))  
  (let ((x (+ x 1)))  
    (+ x x))) => 4
```

The scope of each variable can be determined by the placement within the text of the program == lexical scoping.

Quiz

- What is the value of:

```
(let ((x 9))
  (* x
     (let ((x (/ x 3)))
       (+ x x))))
```

Lambda Expressions

- The syntactic form **lambda** creates a new procedure:

(lambda (var ...) exp₁ exp₂ ...)

- The list of variables are the formal parameters and the expressions are the body.
- The variables in the var list are *bound* and all other variable are called *free*.

Lambda Expression Example

```
( (lambda (x) (+ x x))  
  (* 3 4) )
```

The expression `(lambda (x) (+ x x))`
defines an unnamed procedure which is then
applied to the value `(* 3 4)`.

The result of this expression is 24.

Lambda Expression Quiz

```
(let ((foo (lambda (x) (+ x 1))))  
  (let ((y 3))  
    (foo y)))
```

Hint: In this case we are assigning a name to the procedure defined by the lambda expression.

Lambda Examples:

```
(let ((double (lambda (x) (+ x x))))  
  (list (double (* 3 4))  
        (double (/ 99 11))  
        (double (- 2 7)))))
```

=> (24 18 -10)

```
(let ((double-cons)  
      (lambda (x) (cons x x))))  
  (double-cons 'a))
```

=> (a . a)

Fancy Dancy

```
(let ((double-any  
      (lambda (f x) (f x x))))  
  (list (double-any + 13)  
        (double-any cons 'a)))  
=> (26 (a . a))
```

We can pass a procedure as a parameter!

Lexical Scoping

```
(let ((f (let ((x `a))
           (lambda (y) (cons x y))))))
  (f `b))
```

=> (a . b)

```
(let ((f (let ((x `a))
           (lambda (y) (cons x y))))))
  (let ((x `nota))
    (f `b)))
```

=> (a . b)

Lambda formal parameters

- There are 3 possible forms for the formal parameter specification in a lambda expression:
 - proper list of variables: $(\mathbf{x} \ \mathbf{y} \ \mathbf{z})$
 - a single variable: \mathbf{y}
 - an improper list of variables: $(\mathbf{x} \ \mathbf{y} \ . \ \mathbf{z})$

Lambda Formal Parameters

- Proper list - exactly n actual parameters must be supplied.
- Single variable - any number of parameters can be supplied and the variable is bound to a list of the values.
- Improper list - at least n actual parameters must be supplied, any extra are bound to the last formal parameter.

More Examples

```
(let ((f (lambda (x) x))) (f 1 2))  
=> (1 2)
```

```
(let ((g (lambda (x . y)  
            (list x y))))  
      (g 1 2 3 4))  
=> (1 (2 3 4))
```