This week: Survey advanced languages

- Today: Scheme
- Thursday: Scala

Introduction to Programming and Compiling Scheme

Matthew Might
University of Utah
matt.might.net

Why learn Scheme?

A good FORTRAN programmer can write FORTRAN in any language.

Why learn Scheme?

A good FORTRAN programmer can write FORTRAN in any language.

A good Lisp programmer can write any language in Lisp.

Why learn Scheme?

A good C programmer can
write C in any language.

A good Scheme programmer
can write any language in
Scheme.

Origins of Scheme: Lisp

John McCarthy

- Lisp created in 1958
- Based on λ-calculus
- Based on list-processing
- S-Expression syntax
- Code acts as data
- Data acts as code

Core S-Expression syntax

```
<s-exp> ::= <symbol>
          |  (<s-exp> ...)
```

S-Expressions: Benefits

- Human-readable
- Human-writable
- Easy to parse

S-Expression example

```
(ships
 (ship (name Enterprise)
       (x    120)
       (y    150))
 (ship (name Defiant)
       (x    130)
       (y    42)))
```

XML: High-calorie S-Expressions

```
<ships>
 <ship>
  <name>Enterprise</name>
  <x>120</x>
  <y>150</y>
 </ship>
 <ship>
  <name>Defiant</name>
  <x>130</x>
  <y>42</y>
 </ship>
</ships>
```

S-Expressions: Universal syntax

```
class Dog extends Pet {
 void bark() {
  print("woof") ;
}}
```

S-Expressions: Universal syntax

```
class Dog extends Pet {
 void bark() {
  print("woof") ;
}}
```

```
(class Dog extends Pet
 (void (bark)
  (print "woof")))
```

S-Expressions: Universal syntax

```
class Dog extends Pet {
 void bark() {
  print("woof") ;
}}
```

```
(class
 (name    Dog)
 (extends Pet)
 (methods
  (method
   (name bark)
   (args ())
   (type (-> () void))
   (body (print "woof"))))))
```

Scheme

Guy Steele

- Lisp done right, 1978
- Fixed scoping oversight
- Purified formal semantics
- Hygienic macro system

Scheme example: Factorial

```
(define (fact n)
  (cond
    ((<= n 0)  1)
    (else      (* n (fact (- n 1))))))
(fact 10)
```

Scheme programs

<body> ::= <def> ... <exp> ...

Definitions

```
<def> ::= (define (<id> <params>) <body>)
        |  (define <id> <exp>)
```

A subset of Scheme expressions

```
<exp> ::= <var>
        |  <num>
        |  <prim>
        |  <bool>
        |  (<exp> <exp> ...)
        |  (λ (<params>) <body>)
        |  (if <exp> <exp> <exp>)
        |  (cond (<exp> <exp>) ...)
        |  (let ((<var> <exp>)  ...) <body>)
        |  (let* ((<var> <exp>) ...) <body>)
        |  (letrec ((<var> <exp>) ...) <body>)
        |  (quote <sexp>)
        |  (quasiquote <qq-sexp>)
        |  (begin <body>)
        |  (set! <var> <exp>)
```

A subset of Scheme expressions

```
<exp> ::= <var>
        |  <num>
        |  <prim>
        |  <bool>
        |  (<exp> <exp> ...)
        |  (λ (<params>) <body>)
        |  (if <exp> <exp> <exp>)
        |  (begin <body>)
        |  (set! <var> <exp>)
```

A subset of Scheme expressions

```
<exp> ::= <var>

        |  (<exp> <exp> ...)
        |  (λ (<params>) <body>)

        |  (begin <body>)
        |  (set! <var> <exp>)
```

A subset of Scheme expressions

<exp> ::= <var>

| (<exp> <exp> ...)
| (λ (<params>) <body>)

A tour of Scheme

Numbers

- Integers: `1`, `-3`
- Rationals: `(/ 8 10)` => `4/5`
- "Reals": `(sqrt 2)` => `1.4142135623730951`
- Imaginary: `(sqrt -1)` => `0+1i`
- Complex: `(* 1+3i 1-22/7i)` => `73/7-1/7i`
- Precise: `(fact 18)` => `6402373705728000`

Booleans

- Boolean literals `#t`, `#f`
- Only `#f` is false
- Anything else is true

Quoted S-Expressions

- `'<s-exp>  ==  (quote <s-exp>)`
- `'<id> => <id>`
- `'3 => 3`
- `'a => a`
- `'(1 2 3) => (1 2 3)`
- `(cdr '(1 2 3)) => (2 3)`
- `'(f (g x)) => (f (g x))`
- `(car '(f (g x))) => f`
- `'() => ()`

Primitives

- Arithmetic: `+`, `modulo`, `quotient`
- Types: `null?`, `pair?`, `number?`, `integer?`
- Lists: `list`, `cons`, `car`, `cdr`
- Equality: `=`, `eq?`, `eqv?`, `equal?`
- Ex: `(gcd 248 184) => 8`
- Ex: `(cons 3 (cons 4 '())) => (3 4)`

`let`

- **`<let-exp> ::=`**
  `(let ((<var> <exp>) ...)`
  `<body>)`
- **`let`** binds values to variable names

let

- **<let-exp> ::=**
  (let ((<var> <exp>) ...)
  <body>)
- **let** binds values to variable names
- **(let ((π 3.14)**
  (two 2))
  (* two π)) => 6.28

let

- **<let-exp> ::=**
  (let ((<var> <exp>) ...)
  <body>)
- **let** binds values to variable names
- **(let ((π 3.14)**
  (two 2))
  (* two π)) **=> 6.28**
- **(let ((π 3.14)**
  (2*π (* 2 π)))
  2*π) **=/=> 6.28**

let*

- **<let*-exp> ::=**
  (let* ((<var> <exp>) ...)
  <body>)
- **let*** binds values to variable names sequentially

let*

- **<let*-exp> ::=**
  (let* ((<var> <exp>) ...)
  <body>)
- **let*** binds values to variable names sequentially
- **(let* ((π 3.14)**
  (π 3.14159))
  (* 2 π)) **=> 6.28318**

let*

- **<let*-exp> ::=**
  (let* ((<var> <exp>) ...)
  <body>)
- **let*** binds values to variable names sequentially
- **(let* ((π 3.14)**
  (π 3.14159))
  (* 2 π)) **=> 6.28318**
- **(let* ((π 3.14)**
  (2*π (* 2 π)))
  2*π) **=> 6.28**

Variables

- Any symbol allowed
- Roughly: `[^()'` ,]+`
- Ex: **`foo`**
- Ex: **`foo-bar`**
- Ex: π, , ∞, Σ, , ∅,
- Ex: **`pair?`**
- Ex: **`call-with-current-continuation`**
- Ex: **`(let ((let 3)) let) => 3`**

λ

- λ produces anonymous functions
- **<lambda-exp> ::= (λ (<params>) <body>)**
- **<params> ::= <var> ...**
- **((λ (x) (* x x)) 3) => 9**
- **(let ((λ (λ (λ) (λ λ)))) (λ λ))**

Lexical scope

To what does the following evaluate?

```
(let* ((x 3)
       (f (λ () x))
       (x 10))
  (f))
```

Lexical scope

- λ-terms capture bindings in scope
- WYSIWYG for anonymous functions

Obligatory derivative example

```
(define ε 0.001)
(define (D f) (λ (x) (/ (- (f (+ x ε)) (f x))
                        ε)))
(define (g x) (* x x))
((D g) 0) ≈ 0
((D g) 2) ≈ 4
```

Conditionals

- **<if-exp> ::= (if <exp> <exp> <exp>)**
- **<cond-exp> ::=**
  (cond (<exp> <exp>) ...)
- **(if 'a 'b 'c) => b**
- **(if #f 'b 'c) => c**
- **(cond**
  ((< 3 0) 'negative)
  ((= 3 0) 'zero)
  ((> 3 0) 'positive)) => positive

Quasiquotes

- Special syntax for constructing literals
- `<qq-sexp> == (quasiquote <qq-sexp>)
- ,<exp> == (unquote <exp>)
- `(3 + 4 is ,(+ 3 4)) => (3 + 4 is 7)

```
(let ((new-ship (λ (name x y)
                  `(ship (name ,name)
                         (x    ,x)
                         (y    ,y)))))
  (new-ship 'Enterprise 42 1701))
```

Quasiquotes

- Special syntax for constructing literals
- `` `<qq-sexp> == (quasiquote <qq-sexp>) ``
- `,<exp> == (unquote <exp>)`
- `` `(3 + 4 is ,(+ 3 4)) => (3 + 4 is 7) ``

```
(let ((new-ship (λ (name x y)
                  `(ship (name ,name)
                         (x    ,x)
                         (y    ,y)))))
  (new-ship 'Enterprise 42 1701))

      => (ship (name Enteprise)
               (x 42)
               (y 1701))
```

Mutating variables

- **`begin`** sequences expressions, like **`{}`** in C

```
(begin
  1
  2
  3)    => 3
```

- **`set!`** assigns to a variable, like **`=`** in C

```
(let ((π 3.14))
  (begin
    (set! π 3.14159)
    π))                    => 3.14159
```

Compiling Scheme to Scheme

Strategy

Translate Scheme to ever-smaller subsets of Scheme.

Converting **<body>** to **<exp>**

<def> ... <exp> ...

=>

Converting <body> to <exp>

```
<def> ... <exp> ...

=>

(begin <def> ... <exp> ...)
```

Converting **define** to **letrec**

```
(begin
  (define <var> <exp>) ...
  <exp*> ...)

            =>
```

Converting **define** to **letrec**

```
(begin
  (define <var> <exp>) ...
  <exp*> ...)

              =>

(letrec ((<var> <exp>) ...)
  (begin <exp*>))
```

Example

```
(begin
  (define (volume radius height)
    (* (area radius) height))
  (define (area r) (* π r r))
  (define π 3.14)
  (volume 3 5))
```

Example

```
(letrec ((volume
          (lambda (radius height)
            (* (area radius) height)))
         (area (lambda (r) (* π r r)))
         (π 3.14))
  (volume 3 5))
```

Converting **letrec** to **let** + **set!**

```
(letrec ((<var> <exp>) ...)
  <body>)
```

=>

Converting **letrec** to **let** + **set!**

```
(letrec ((<var> <exp>) ...)
  <body>)

                =>

(let ((<var> #f) ...)
  (begin
    (set! <var> <exp>) ...
    <body>))
```

Example

```
(letrec ((volume
          (lambda (radius height)
            (* (area radius) height)))
         (area (lambda (r) (* π r r)))
         (π 3.14))
  (volume 3 5))
```

Example

```
(let ((volume #f) (area #f) (π #f))
  (begin
    (set! volume
      (lambda (radius height)
        (* (area radius) height)))
    (set! area (lambda (r) (* π r r)))
    (set! π 3.14)
    (volume 3 5)))
```

Converting **`let`** to λ

```
(let ((<var> <exp>) ...)
  <body>)
```

=>

Converting **`let`** to λ

```
(let ((<var> <exp>) ...)
  <body>)

                =>

((lambda (<var> ...) <body>) <exp> ...)
```

Example

```
(let ((volume #f) (area #f) (π #f))
  (begin
    (set! volume
      (lambda (radius height)
        (* (area radius) height)))
    (set! area (lambda (r) (* π r r)))
    (set! π 3.14)
    (volume 3 5)))
```

Example

```
((lambda (volume area π)
   (begin
     (set! volume
       (lambda (radius height)
         (* (area radius) height)))
     (set! area (lambda (r) (* π r r)))
     (set! π 3.14)
     (volume 3 5)))
 #f
 #f
 #f)
```

Converting **let*** to **let**

```
(let* ((<var> <exp>) ...)
  <body>)
```

=>

Converting **let*** to **let**

```
(let* ((<var> <exp>) ...)
  <body>)

                =>

(let ((<var> <exp>))
  (let* (...)
  <body>))
```

Example

```
(begin
  (let* ((π 3.14)
         (2*π (* 2 π)))
    2*π))
```

Example

```
(begin
  (let ((π 3.14))
    (let ((2*π (* 2 π)))
      2*π)))
```

Converting **quasiquote** to **cons**

```
`<qq> => '<qq>
`,<exp> => <exp>
`(<qq> <qq*> ...) => (cons `<qq> `(<qq*> ...))
```

Example

```
(begin
  (define (new-ship name x y)
    `(ship (name ,name) (x ,x) (y ,y)))
  (new-ship 'Galactica 13 12))
```

Example

```
(begin
  (define (new-ship name x y)
    (cons
     'ship
     (cons
      (cons 'name (cons name '()))
      (cons
       (cons 'x (cons x '()))
       (cons (cons 'y (cons y '())) '())))))
  (new-ship 'Galactica 13 12))
```

Converting **cons** to λ

```
cons =>
(λ (a b) (λ (on-null on-pair) (on-pair a b)))

      '() =>
      (λ (on-null on-pair) (on-null))

   car =>
   (λ (p) (p (λ () 'error) (λ (a b) a)))

   cdr =>
   (λ (p) (p (λ () 'error) (λ (a b) b)))
```

Example

```
(begin
  (define (new-ship name x y)
    (cons
     'ship
     (cons
      (cons 'name (cons name '()))
      (cons
       (cons 'x (cons x '()))
       (cons (cons 'y (cons y '())) '())))))
  (new-ship 'Galactica 13 12))
```

Example

```
(begin
  (define (new-ship name x y)
    ((lambda (a b)
       (lambda (on-pair on-nil) (on-pair a b)))
     'ship
     ((lambda (a b)
        (lambda (on-pair on-nil) (on-pair a b)))
      ((lambda (a b)
         (lambda (on-pair on-nil) (on-pair a b)))
       'name
       ((lambda (a b)
          (lambda (on-pair on-nil) (on-pair a b)))
        name
        (lambda (on-pair on-nil) (on-nil))))
      ((lambda (a b)
         (lambda (on-pair on-nil) (on-pair a b)))
       ((lambda (a b)
          (lambda (on-pair on-nil) (on-pair a b)))
```

Converting **cond** to **if**

```
(cond (<exp> <exp*>) ...)
```

=>

```
(if <exp> <exp*> (cond ...))
```

```
(cond) => (void)
```

Example

```
(begin
  (define (sign n)
    (cond
     ((< n 0) 'negative)
     ((= n 0) 'zero)
     ((> n 0) 'positive)))
  (sign 2.17))
```

Example

```
(begin
  (define (sign n)
    (if (< n 0)
      'negative
      (if (= n 0)
        'zero
        (if (> n 0) 'positive (void)))))
  (sign 2.17))
```

A subset of Scheme expressions

```
<exp> ::= <var>

        |  (<exp> <exp> ...)
        |  (λ (<params>) <body>)

        |  (begin <body>)
        |  (set! <var> <exp>)
```

A subset of Scheme expressions

<exp> ::= <var>

| (<exp> <exp> ...)
| (λ (<params>) <body>)