Computability Theory Of and With `Scheme`

Prof. Albert R. Meyer

MIT Computer Science &
Artificial Intelligence Laboratory

`Scheme`-based Texts

1) Abelson & Sussman. 2nd ed. 1996
2) Friedman & Felleisen. 4th ed. 1995.
3) Friedman, Wand, & Haynes. 1992.
4) Harvey & Wright. 1994.
5) Hailperin, Kaiser & Knight. 1999.
6) Manis & Little. 1995.
7) Springer & Friedman. 1989.

More `Scheme`-based Texts

8. *La programmation: une approche fonctionnelle et récursive avec Scheme*, Arditi & Ducasse, 1996
9. *Initiation à la programmation avec Scheme*, Bloch, 2001
10. *Programmer avec Scheme: De la pratique à la théorie*, Chazarain, 1996
11. *The Scheme Programming Language* (2/e), Dybvig, 1996
12. *How to Design Programs*, Felleisen, Findler, Flatt & Krishnamurthi
13. *The Schemers Guide* (2/e), Ferguson w/Martin & Kaufman, 1995
14. *The Seasoned Schemer*, Friedman & Felleisen, 1995
15. *Exploring Computer Science with Scheme*, Grillmeyer, 1998
16. *Programmation Fonctionnelle en Scheme*, Hufflen, 1996
17. *Vom Problem zum Programm: Architektur und Bedeutung von Computerprogrammen* (3/e), Klaeren & Sperber, 2001
18. *Recueil de Petits Problèmes en Scheme*, Moreau, Queinnec, Ribbens & Serrano, 1999
19. *Programming and Meta Programming in Scheme*, Pearce, 1998
20. *Débuter la programmation avec Scheme* Routier& Wegrzynowski, 1997
21. *Teach Yourself Scheme in Fixnum Days*, Sitaram, Web document, 2003
22. *Principles of Programming Languages*, Krishnamurthi & Felleisen, 1997
23. *Les Langages Lisp*, Queinnec, 1994

Schools using Scheme

- 154 colleges/universities in USA
  (50 for intro courses)
- 278 colleges/universities worldwide
- 51 secondary schools in USA
  Rice/NE U. TeachScheme!
- Commercial education: Schemers inc.

(as of Nov. 2002, from `http://www.schemers.com/schools.html`)

The `Scheme` Underground

… an effort to develop useful software packages in Scheme for use by research projects and for distribution on the net.

We want to take over the world. The internet badly needs a public domain software environment that allows the rapid construction of software tools using a modern programming language. Our goal is to build such a system using `Scheme`….

Today's Lecture

- *Kernel* `Scheme`: overview
- A rigorous, intuitive
  `Scheme` "Substitution" Model
- Theorems about `Scheme`
- Computability theory w/ `Scheme`

Scheme Expressions

Scheme has

lots of parentheses :-)

- variables **x**, **myage**, **factorial**
- procedures **+**, **cons**,
  procedure?
  (lambda(x) <expr>**)**

Scheme Expressions

- if's **(if** <test> <expr> <expr>**)**
- apply's **(**<procedure> <args>**)**

```
(factorial 3)

((lambda(func)(func 2 3))
  +)
```


Scheme Expressions

- letrec's

```
(letrec ((x 1)
         (func +))
     (func x 5))
>> 6
```


Substitution

```
(letrec ((x 1)
         (func +))
      ( +   1 5))
```

6

Substitution

```
(letrec ((x 1)
         (func +))
     (if      #t
          "done"
          x)))
```

NO!

only substitute where necessary

```
 >> "done"
```


Bindings & Lambda

```
((lambda (y) (+ y 5)) 2)

        →

(letrec ((y 2)) (+ y 5))
```

Applying **lambda** creates a *binding* -- of **y** to 2 in this example

Bindings & `set!`

`(letrec ((y 3 ))`

`(number? 3))`

`>> #t`

`Scheme` Values

basic: `3 "done" #t `aaa`

procedure: `+ cons`

`(lambda …)`

list: `nil`

`(list` <val> …<val>`)`

Scheme Evaluation

Evaluation rules change an expression into another expression.

Expressions *evaluate* until a unique value is reached.

`(if #f` Texp Fexp`)` → Fexp

`(if` <non-`#f` val> Texp Fexp`)`

→ Texp

Redexes

A *redex* is an expression matching lefthand side of an evaluation rule.

Redexes of `if`-rules:

`(if` <value> Texp Fexp`)`

`lambda` Evaluation Rule

`((lambda (x)` *E*`)` <val>`)`

→

`(letrec ((x` <val>`))` *E*`)`

new binding

Scheme Evaluation

```
(if (= (+ 1 12)
       (- 22
         ((lambda(x) (* x x))
           3))
    <#t case to do>
    <#f case to do>)
```

Scheme Evaluation

```
(if (= (+ 1 12)  (apply rule here?)
       (- 22
          ((lambda(x) (* x x))
            3))
     <#t case to do>
     <#f case to do>)
```


Scheme Evaluation

```
(if (= (+ 1 12)
       (- 22
          ((lambda(x) (* x x))
            3))  (or here?)
     <#t case to do>
     <#f case to do>)
```


Scheme Evaluation

```
(if (= (+ 1 12)
       (- 22
          ((lambda(x) (* x x))
            3))
     <#t case to do>   (or here?)
     <#f case to do>)
```


Scheme Evaluation

```
(if (= (+ 1 12)
       (- 22
          ((lambda(x) (* x x))
            3))
     <#t case to do>
     <#f case to do>)   (or here?)
```


Which expression to rewrite?

Expression to rewrite given by a

Control context, *R*[ ]

Control Parsing Lemma. Every *non-value* Scheme expression parses uniquely as

R[<redex>]

Control Context Syntax

R[ ] is

- [ ]
- **(if** *R*[ ] <expr> <expr>**)**
- **(set! x** *R*[ ]**)**
- **(**<val> …<val> *R*[ ] <expr>…**)**

Substitution Rules

x → <what **x** is bound to>

(set! x <newval>**)**→ **'set!-done**

(letrec (…(x <oldval>**)…) …)** →

(letrec (…(x <newval>**)…) …)**

Substitution Rule

(letrec (…(x <val>**)…)**

R[**x**]**)** →

(letrec (…(x <val>**)…)**

R[<val>]**)**

letrec Rule

(letrec (…(x *<oldval>***)…)**

R[**(set! x** <newval>**)**]**)**

→

(letrec (…(x <newval>**)…)**

R[**'set!-done**]**)**

The Variable Convention

Rename bound variables so

- no variable appears in two bindings

```
(letrec((x 1))
  (+ x
      (lambda(y)(* y y))..)))
```


The Variable Convention

Rename bound variables so

- no variable is both bound & free

```
(letrec((z 1))
  (+ x
      (lambda(y)(* y y))..)))
```

Teaching Scheme : `call/cc`

`call/cc` – a general “escape” construct

-- normally in advanced course.

SubModel explains with two simple rules (after Felleisen & Heib).

Teaching Scheme : `call/cc`

1) R[`(call/cc` V`)`]

$\rightarrow$

R[`(`V `(lambda(x)(abort` R[`x`]`))`]

2) R[`(abort` V`)`] $\rightarrow$ V

It runs! http://theory.lcs.mit.edu/classes/6.844/spring03-6844/

```
>> (set! dec (lambda (n) (- n 1)))
>> (dec (dec 3))
==(0, instantiate-global)==>
((lambda (n) (- n 1)) (dec 3))
==(1, instantiate-global)==>
((lambda (n) (- n 1)) ((lambda (n) (- n 1)) 3))
==(2, lambda)==>
(letrec ((n 3))
  ((lambda (n) (- n 1))  ((lambda () (- n 1)))))
==(3, lambda)==>
(letrec ((n 3))          ((lambda (n) (- n 1)) (- n 1)))
==(4, instantiate)==>
(letrec ((n 3))          ((lambda (n) (- n 1)) (- 3 1)))
==(5, builtin)==>
(letrec ((n 3))          ((lambda (n) (- n 1)) 2))
==(6, lambda)==>
(letrec ((n 3) (n_0 2))  ((lambda () (- n_0 1))))
==(7, lambda)==>
(letrec ((n 3) (n_0 2))  (- n_0 1))
==(8, instantiate)==>
(letrec ((n 3) (n_0 2))  (- 2 1))
==(9, builtin)==>
(letrec ((n 3) (n_0 2))  1)
Final value after 10 steps:     1
```


Scheme Theory

With **Scheme** mathematically defined by simple rules, can *prove theorems* about it.

Scheme Theory

Equivalence Lemma.

If $E \rightarrow \cdots \rightarrow F$, then E and F are *observably equivalent*:

$$E \equiv F.$$

(ignoring a compication from `call/cc`)

Scheme Theory

“Black Box” Procedures Lemma.

If P, Q procedure val’s and $P \not\equiv Q$, then there must be val’s, V_1, V_2, s.t.

$$(V_1 (P\ V_2)) \rightarrow \cdots \rightarrow \mathbf{1}$$

$$(V_1 (Q\ V_2)) \not\rightarrow \cdots \not\rightarrow \mathbf{1}$$

(or vice-versa).

Computability Theory

With **Scheme** mathematically defined by simple rules, it can replace Turing Machines.

Computability Theory w/Scheme

Example: "self-reproducing" expression.
Define **double** so that

(double '<expr>**)**

→···→ <list-value>

which prints out as:

(<expr> '<expr>**)**

Computability Theory

```
(set! double
 (lambda (expr)
   (list expr
       (list 'quote expr)))
```


Computability Theory

So

(double 'double)

prints out as:

(double 'double)

"Self-reproducing"!

Computability Theory: complications

Scheme easier to program than Turing Machines, but makes some basic results harder to show (or false):

If a set is recognizable, then it is enumerable.

Computability Theory: complications

Symbols are recognizable – by builtin **symbol?** procedure.

How to enumerate them? say with

(nth-symbol 0)
(nth-symbol 1)
(nth-symbol 2) ...

Computability Theory: complications

`nth-symbol`
definable using *builtin*
`string->symbol`
procedure
– not in good taste.

Computability Theory: complications

Procedures are recognizable – by builtin `procedure?`

How to enumerate them? say with

```
(nth-procedure 0)
(nth-procedure 1)
(nth-procedure 2)...
```


Computability Theory: complications

`nth-procedure`
definable (up to ≡) by constructing an *interpreter* `eval`:

`(eval '`<expr>`)` ≡ <expr>

– interesting CS project, but tricky, not in good taste for computability theory.

Computability Theory: complications

More generally:
insight of Computability theory is that it is *machine independent*.
Using `Scheme` – or any "real" language – can shift focus to programming instead of machine independent results.

Submodel in intro programming?

Why isn't the Submodel of interest to my programming colleagues?
My guess: can shift focus to details of progr. *language* instead of progr. abstractions & techniques.

Computability Theory of & with…?

- Found a place in introductory grad course.
- Enjoyed by a few students who had taken grad programming
  but not computability theory.
- Remains a boutique course.

Computability Theory of & with…?

- But I believe in material.
- See it developing into genuine methodology for reasoning about Scheme programs.
- But not for computability.