CS 4450: Principles of Programming Languages

Introduction to Scheme

Dr William Harrison

Today

- An introduction to the Scheme language
- This is a prelude to μScheme
  - the language in the Ramsey/Kamin text; Chapter 3
- **Almost** the same language
  - there are slight syntactic differences
- **Important**: these slides introduce Scheme and not μScheme
  - we'll go over the syntax and operational semantics for μScheme later this week

DrScheme: a reference interpreter

- DrScheme is an interactive interpreter for Scheme
    - http://download.plt-scheme.org/drscheme/
- Whenever you need to know "what does this Scheme expression mean?", ask DrScheme

Scheme

- Scheme expressions (S-exps) are simple; a **S-exp** is either:
  - an "atom" or
  - a list of S-exps
- What are atoms?
- What are lists?
  - car, cdr, cons, null?,...
- Definitions with "define"
  - lambda expressions
- Practical considerations
  - loading files, etc.

Atoms in Scheme

- An atom is any
  - primitive value: 0,1,… #t, #f
  - string of symbols beginning with a letter or special symbol
    - a1, abcde, *ab$6
    - think of these symbol atoms as being variables

Lists in Scheme

- A list is any sequence of S-exps enclosed by “(“ and “)”
  - (1 2 3) (#t 3 *ab$) are lists of atoms
- An **S-exp** is either
  - an atom, or
  - a list of S-exp’s
    - (() 5 ((abc)))
    - #t
    - …

Using the DrScheme interpreter

- Scheme is, just like Haskell, an interactive interpreted language
- (load “*filename*”)
  - reads in *filename* from the current directory

“car”, “cdr”, and “cons”

- “car” of a list is just like “head” in Haskell
  - (car ‘(1 2 3)) = 1
    - what’s the single quote for?
- “cdr” is like “tail” in Haskell
  - (cdr ‘(1 2 3)) = (2 3)
- “cons” is like “:” in Haskell*
  - (cons 1 ‘(2 3)) = (1 2 3)

* there is a difference we’ll see shortly

Lists vs. Function calls

We've already seen some Scheme expressions

```
(+ 1 2)
```

...this means "apply function + to the sequence 1 2"

If we want to consider the above **as a list** we must quote it

```
'(+ 1 2)
```

to quote or not to quote

- When you type an unquoted list at the Scheme prompt, it will always assume the first element of the list is a function
  - ...and will try to apply it.

```
> (1 2 3)
 Error! Tried to apply non-procedure "1"
```

  - so, if you want a constant list, you must quote it

```
> (car `(1 2 3))
 1
```

Other list functions

- The empty list is **`()`**
- **`null?`** tests whether an S-exp is **`()`**
- **`list`**
  - **`(list 1 2 3 x)`** returns **`(list 1 2 3 x)`**

Lambda expressions in Scheme

Haskell

```
\ x -> x + 1
```

Scheme

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

Definitions with "define"

- Scheme has the "define" directive to define functions, values, etc.
- Functions are defined as lambda expressions; E.g.,
  - (define inc (lambda (x) (+ x 1)))
    - N.b., in μScheme, this would be: (define inc (x) (+ x 1))

Writing list functions in Scheme: length

First, recall its definition in Haskell

```
length []     = 0
length (x:xs) = 1 + length xs
```

* Scheme doesn't have pattern matching, so have to write this slightly differently.

Compare Scheme & Haskell

```
(define length
      (lambda (l)
           (if (null? l)
                0
                (+ 1 (length (cdr l))))))
```

```
length []     = 0
length (x:xs) = 1 + length xs
```

An abstract syntax for Scheme

```
data Term =
{- arithmetic language -}
    Litreal Float |
    Litint Int    |
    Plus          |
    Minus         |
   Times          |
    Div           |
{- functions -}
    App [Term]    |
    Lambda [String] Term |
{- variable references -}
         Var String
…
```

Scheme expression `(+ x 1)`

…as a **`Term`**

`App [Plus, Var “x”, litint 1]`

Read-eval-print loop

- Many interpreters are interactive
  - type in a term, and it tries to evaluate it
  - e.g., the Hugs interpreter we use
- Interactive interpreters do the following
  - **read**: scan/parse/type-check what you type
  - **eval**: interpret the result AST
  - **print**: give you the answer (including errors)
- …our interpreter has a REP loop:

```
Hugs> repl
TigerScheme> (+ x 1)
    val it = 2
TigerScheme>
```

Read Eval Print loop

```
repl = top initEnv

top :: Env -> IO ()
top env = do
            putStr "TigerScheme> "
            iline <- getLine
            process env iline

process :: Env -> String -> IO ()
process env "quit" = return ()
process env iline  =
  case (parse iline) of
       Just term ->
         putStr (" val = " ++ show v ++ "\n") >> top env
                  where v = eval term env
       Nothing   -> putStr "Syntax Error!\n" >> top env
```

provided to you

what you write

Scheme values in Haskell

```
data Value = I Int | R Float
```

will be extended with values representing functions, cons cells, etc.

The form of the evaluation function

These are some cases we've seen already

```
eval :: Term -> Env -> Value
eval (Litint i) rho  = I i
eval (Litreal r) rho = R r
eval (Var x) rho     = applyenv rho x
```

...more to come...

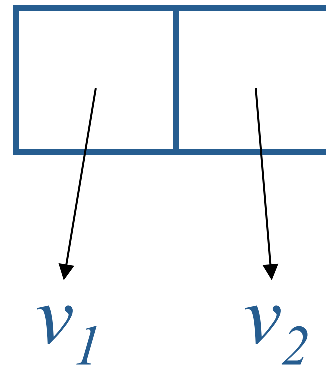
What is “`cons`” really?

`cons` can take any two Scheme values and create a new “**cons cell**”

Pictorially

(`cons` v_1 v_2)

creates a “cons cell”



Scheme values in Haskell, redux

```
data Value = ConsCell Value Value |
             I Int | R Float

car (ConsCell sv1 sv2) = sv1

cdr (ConsCell sv1 sv2) = sv2

cons x y = ConsCell x y
```

Adding "cons" to the interpreter

- What do we have to add to the interpreter to handle "cons"?
- The syntax & front-end
  - already handled for you

```
datatype Term = … | Cons | …
```

- The interpreter
  - Values: need to add a "ConsCell"
  - eval function: Need to add a case for "`Cons`"

Adding Cons

```
eval :: Term -> Env -> Value
eval (Litint i) rho  = I i
eval (Litreal r) rho = R r
eval (Var x) rho     = applyenv rho x
eval Cons rho        = ???
```

CS4450: Principles of Programming Languages

Interpreting Scheme lists and functions

William Harrison

“More Scheme!”, they cried.

- One more bit of Scheme
  - Why “cons” **is** like “:” (in Haskell)
  - Why “cons” **is not** like “:”
- Scheme Values & how to represent them
- Scheme Procedures
  - Procedure calls (e_1 e_2…e_n)
  - Procedures (lambda (x_1…x_n) body)

`cons` in Scheme vs. ":" in Haskell

- Review: lists in Scheme are constructed with "**`cons`**"
  - `(cons 1 ())`, `(cons 1 (cons 2 ()))` ,…
  - pretty-printed: `(1) (1 2)` …
  - `(car (1 2)) = 1`
  - `(cdr (1 2)) = (2)`
  - **Notation:** `(cons x y)` can be written `(x . y)`
- In Haskell, use ":"
  - `(1 : []) (1 : (2 : []))`
  - pretty-printed: `[1] [1,2]`

…but "**cons**" is different

```
> (cons 1 2)
(1 . 2)
> (cdr (cons 1 2))
2
```

N.b., **cons** doesn't only apply to lists – it applies to **any** Scheme value

* note also the alternative notation for `(cons x y)` is `(x . y)`

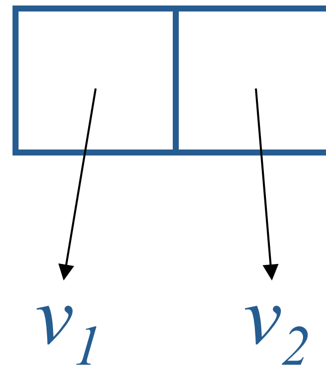
What is "**`cons`**" really?

`cons` can take any two Scheme values and create a new "**cons cell**"

Pictorially

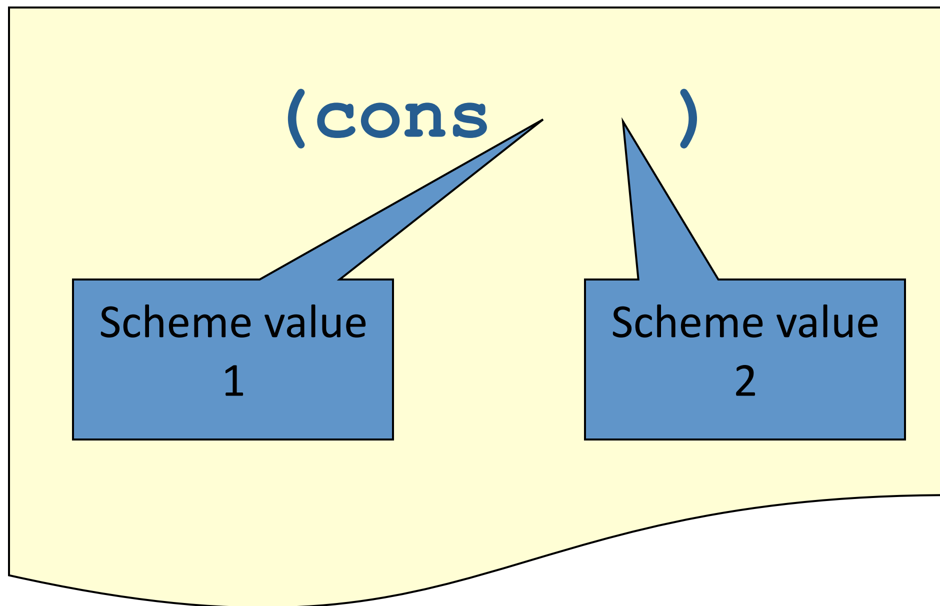
`(cons` v_1 v_2**`)`**

creates a "cons cell"

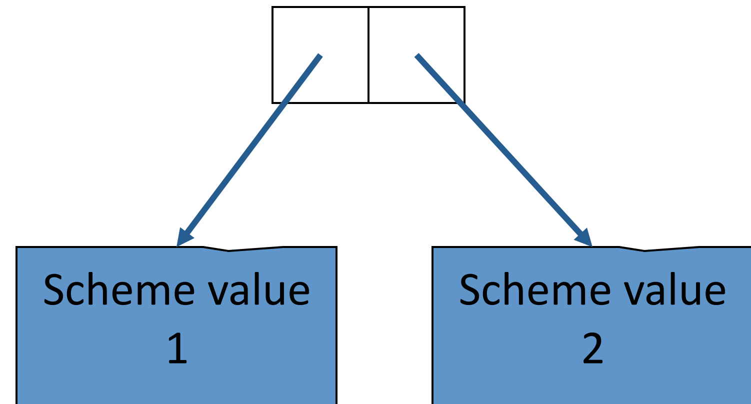


What's going on here?

language syntax



language implementation



...uses pointers

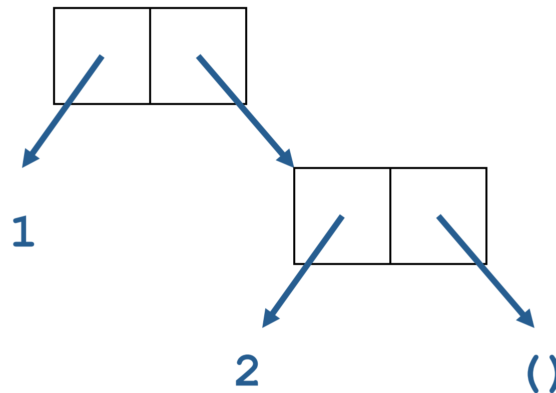
If used correctly, cons creates lists

language syntax

```
(cons 1 (cons 2 ()))
```

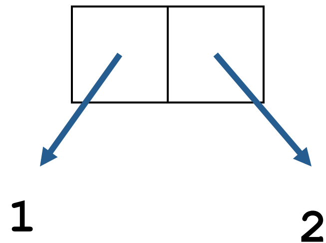
i.e., 2nd arg is a list

language implementation



Cons can also be used to create pairs

`(cons 1 2)`



is like `(1, 2)` *in Haskell*

How might we represent Scheme cons/car/cdr in Haskell?

- First off, what are "Scheme Values"
  - Recall: `data Value = I Int | R float`
  - What kind of data do these correspond to in Scheme?
- Secondly, what are the types of car, cdr, cons?

What are "Scheme Values"

- They are "atomic" data values
  - integers, reals, strings, …
  - `data Value = I Int | R float`
- …or constructed data
  - …as in "constructed with cons"
- …and procedures
  - but we will ignore this variety until later

Adding "cons" to the interpreter

- What do we have to add to the interpreter to handle "cons"?
- The syntax & front-end
  - already handled for you

```
datatype Term = … | Cons | …
```

- The interpreter
  - Values: need to add a "ConsCell"
  - eval function: Need to add a case for "`Cons`"

Scheme values in Haskell, redux

```
data Value = ConsCell Value Value |
             NilVal |
             I Int | R Float

car (ConsCell sv1 sv2) = sv1

cdr (ConsCell sv1 sv2) = sv2

cons x y = ConsCell x y
```

Adding Cons

```
eval :: Term -> Env -> Value
eval (Litint i) rho  = I i
eval (Litreal r) rho = R r
eval (Var x) rho     = applyenv rho x
eval Cons rho        = ???
```

Adding Cons

```
eval :: Term -> Env -> Value
eval (Litint i) rho  = I i
eval (Litreal r) rho = R r
eval (Var x) rho     = applyenv rho x
eval Cons rho        = ???
```

The problem is that the function cons is itself a value, but we have no representation of function values in Value

```
Welcome to DrScheme, version 360.
> cons
#<primitive:cons>
>
```

Overview: Functions as Values

- procedures as values
- syntax for procedures
- What do we add to **`Value`** to represent procedures?

lambda expressions **are** procedures

C

```
void strcopy(char *s, char *t)
{
   int i = 0;

   while ((s[i]=t[i]) != '\0') i++;
}
```

Scheme

```
(define strcopy
     (lambda (s t)
          (let
               ((i 0))
               (while …)
          )
     )
)
```

* the red part we don't know about yet

Procedures as Values

Consider the following Haskell definition

```
add :: Int→Int→Int
add x y = x + y
```

What is: `add 1` ?

Procedures as Values

Consider the following Haskell definition

```
add :: Int→Int→Int
add x y = x + y
```

Q: What is: `add 1` ?

A: It is a function from `Int→Int` that adds one to it's argument

Procedures as Values

Consider the following Haskell definition

```
add :: Int→Int→Int
add x y = x + y
```

Q: What is: `add 1` ?

A: It is a function value from `Int→Int`

Procedures as values in Scheme

```
add :: Int→Int→Int
add x y = x + y
```

We can write the same thing in Scheme:

```
(define add
    (lambda (x)
        (lambda (y) (+ x y))))
```

`(add 1)` is a value

```
Hugs> :t add 1
     add 1 :: Int -> Int
```

```
> (define add (lambda (x) (lambda (y) (+ x y))))
> (add 1)
#<procedure>
>
```

`add` itself is a value

```
Hugs.Base> :set -u
Hugs.Base> (+)
primPlusInteger
```

turns off “show”

```
> add
#<procedure>
```

* ***Upshot***: *Both Haskell and Scheme can return procedures as values just as they might return a number or a list*

…and both Haskell & Scheme can pass procedures as values as well

Haskell

```
Hugs> map (add 1) [1,2,3]
     [2,3,4] :: [Int]
```

Scheme

```
> (map (add 1) '(1 2 3))
(2 3 4)
```

*** **Upshot**: having procedures both as arguments and values is why Haskell/Scheme are called "higher-order"*

Scheme values as **`Value`**s

```
data Value = ConsCell Value Value |
             I Int | R Float
```

Q: how would we write a Haskell function from `Value→Value` representing `(add 1)` ?

`addOne =` ?

Scheme values (so far)

```
data Value = ConsCell Value Value |
             I Int | R Float
```

Q: how would we write a Haskell function from `Value→Value` representing `(add 1)` ?

A: use the Haskell lambda notation "`\ y ->` …"

```
addOne :: Value→Value
addOne = \ (I i) -> I (i+1)
```

Structure of Interpreter

Concrete Syntax

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

```
parse :: String -> Maybe Term
```

Abstract Syntax

```
data Term = Litint Int | …
```

```
eval :: Term -> Env -> Value
```

Values

```
data Value = I Int | …
```

Extending the Interpreter

Concrete Syntax

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

```
parse :: String -> Maybe Term
```

Abstract Syntax

```
data Term = Litint Int | … | Nil
```

```
eval :: Term -> Env -> Value
eval Nil rho = NilVal
```

Values

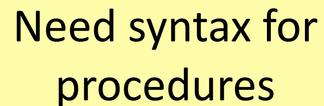
```
data Value = I Int | … | NilVal
```

Procedures

- What is a procedure?
  - It's a kind of **function**
    - has input parameters
      - called "formal parameters"
    - and a "body" (i.e., the code of the procedure)
- What is a procedure call?
  - what steps are taken to evaluate a procedure call?

How would **`eval`** look for procedures?

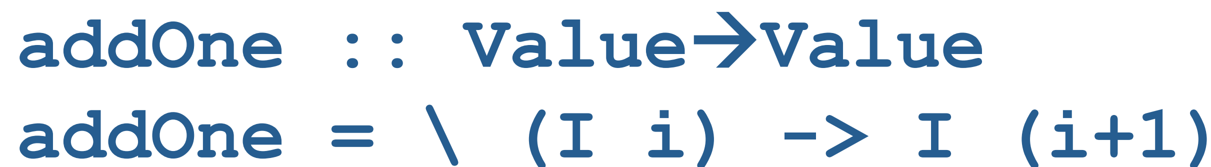
```
eval : Term → Env → Value
eval "(lambda (x) (+ x 1))" env = \ (I x) -> I (x+1)
```



Scheme values for functions?

```
data Value = ConsCell Value Value |
             I Int | R Float
```

A: use the Haskell lambda notation “`\ y ->` …”



How would the constructor
for this kind of Value be declared?

Scheme values for procedures

```
data Value = I Int | R Float
           | ConsCell Value Value
           | FunVal (Value → Value)
```

we're going to modify this slightly to handle multiple arguments*

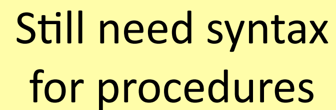
addOne as a **Value**

```
addOne :: Value -- was Value->Value
addOne = FunVal (\ (I i) -> I (i+1))
```

* we'll also explore a related representation next time.

How would **eval** look for procedures?

```
eval :: Term → Env → Value
eval "(lambda (x) (+ x 1))" env
                 = FunVal (\ (I x) -> I (x+1))
```



lambda expressions **are** procedures*

C

```
int incr(int i)
{
   return (i+1);
}
```

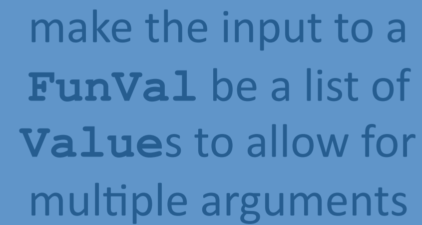
Scheme

```
(define incr
     (lambda (i)
          (+ i 1)
      )
)
```

* and vice versa

Scheme values for procedures

```
data Value = …
      | FunVal ([Value] → Value)
```



What is **eval Cons env**?

- Must be a function value
  - i.e., a Value of the form (FunVal f)
- What's the behavior of (cons …)?
  - For this, we check with DrScheme

Which FunVal is cons?

Because cons must have two and only two arguments whenever
it is called, we use pattern-matching to discount other possibilities

```
eval Cons rho = FunVal consVal
   where
      consVal [v1,v2] = ConsCell v1 v2
      consVal _       = error "…"
```

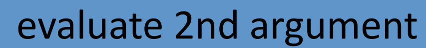
Evaluating an App

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

     = (cons 3 (* 3 4))

     = (cons 3 12)

     = (3 . 12)
```



Evaluating an App (Under the Interpreter's Hood)

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

     = (cons 3 (* 3 4))

     = (cons 3 12)

     = (3 . 12)
```

eval "(+ 1 2)" rho0 = I 3

eval "(* 3 4)" rho0 = I 12

ConsCell (I 3) (I 12)

Evaluating `(App` $(\mathit{rator} : \mathit{rands})$`)`

Given an environment **`env`**

- evaluate *rator*
  - producing either
    - a function value f (i.e., a **`FunVal`**)
    - …or some other value (e.g., **`(I 1)`**) which can't be applied
      - …& should cause an error
- evaluate *rands*
  - producing a list of values $[v_1, \ldots, v_n]$
- …then applies f to $[v_1, \ldots, v_n]$

Evaluating `(App` $(rator : rands)$`)`

It is simply:

```
eval (App (rator:rands)) env
    = apply (eval rator env)
            (map (\ t -> eval t env) rands)
```

Defined as:

```
apply (FunVal f) vs = f vs
```

Evaluating **(App** *(rator* **:** *rands*))

```
eval (App (rator:rands)) env
    = apply (eval rator env)
            (map (\ t -> eval t env) rands)

apply (FunVal f) vs = f vs
```

Questions:

- Does this have the same failure behavior as Scheme functions?
- Are Scheme operators like "+", "-", etc., functions in the same sense as "(lambda (x y) ...)?

Syntax for Scheme procedures

Procedures can have zero or more *formal arguments*

```
(lambda () body)
(lambda (x) body)
(lambda (x y) body)
(lambda (x y z) body)
        ...
```

For example

```
(define addboth (lambda (x y) (+ x y)))
(define add     (lambda (x)
                   (lambda (y)
                          (+ x y))))
```

Syntax for Scheme procedures, cont'd

How do `addboth` & `add` differ?

```
(define addboth (lambda (x y) (+ x y)))
(define add     (lambda (x)
                   (lambda (y)
                        (+ x y))))
```

…they differ in how they're applied

```
(addboth 1 2)
((add 1) 2)
```

return 3

```
((addboth 1) 2)
(add 1 2)
```

cause errors

Abstract Syntax for Scheme procedures in the interpreter

```
data Term =
        …
   | Lambda [String] Term
```

…allows for multiple arguments as in:

```
(lambda () body)
(lambda (x) body)
(lambda (x y) body)
(lambda (x y z) body)
          …
```

What is `(Lambda [`x_1`,…,`x_n`]` *body*`))`?

It is the **`FunVal`** such that

- Given an environment **`env`**
  - …and input values `[`v_1`,…,`v_n`]`,
- …evaluates *body* in the environment **`env′`**
  - where `env′ = extendenv [`x_1`,…,`x_n`] [`v_1`,…,`v_n`] env`

What is `(Lambda [`$x_1, \ldots, x_n$`]` *body*`)`?

It is the **`FunVal`** such that

- Given an environment **`env`**
  - …and input values `[`$v_1, \ldots, v_n$`]`,
- …evaluates *body* in the environment **`env'`**
  - where `env' = extendenv [`$x_1, \ldots, x_n$`] [`$v_1, \ldots, v_n$`] env`

We write a helper function to do this:

```
closure xs body env =
         \ vs -> eval body (extendEnv xs vs env)
```

What is

`eval (Lambda xs body) rho`?

It is simply:

```
eval (Lambda xs body) rho
        = FunVal (closure xs body rho)
```

```
{- closure :: [String]→
                        Term →
                             Env →
                                 [Value] → Value -}
closure xs body rho =
        \ vs -> eval body (extendEnv xs vs rho)
```

Next time

- Finish procedures
  - procedure application
  - Closures as a representation for procedures
- Start recursion
  - Once we have recursion, then we're in a position to really exercise the interpreter.