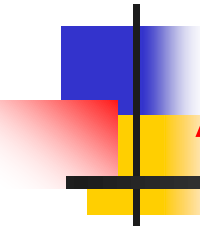
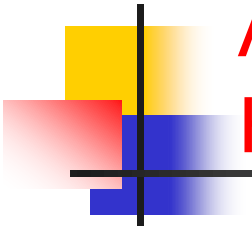


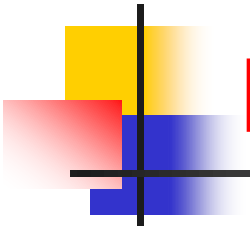
A Glimpse at Imperative Racket and Memoization

Comp 311
Rice University
Corky Cartwright



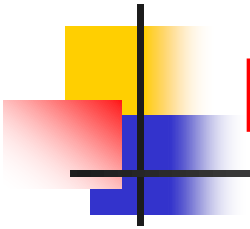
All Real "Functional" Languages Except Haskell Support Imperative Operations

- Why do nearly all real "functional" languages include imperative operations?
  - The real world is imperative (changes state). The real world and many conceptual models evolve (change state) over time. In computations simulating these models, it is often convenient (and conceptually economical to let *execution recapitulate evolution*. During the simulation of the model, changes in the state of the model are represented by changes in the current program state. Imperativity may be simpler in some cases.
  - Our computer hardware is imperative. At some point, even pure functional code must be executed on hardware where every computation step (execution of a machine instruction) involves mutation. To produce efficient machine code to solve a problem, we often need to describe the computation in imperative terms. Most fast algorithms perform incremental operations that are imperative.



Functional Programming Culture

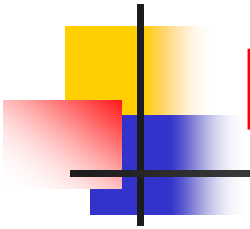
- Imperative computation must be clearly identified as such. When imperativity is used "internally" to improve performance, it should be encapsulated when possible behind APIs that are functional.
  Examples:
  - Memoization
  - Fast imperative algorithms for solving problems which may have slower functional equivalents
  - Simulation of physical systems: the change in state over short time intervals is typically small and the successor state in discrete is often a simple update to the current state. In some cases, it is possible to preserve the old state and construct the new state by sharing pieces of the previous state but many data structures (like arrays) must be completely copied if the old state is to be preserved, compromising the efficiency of the update operation. Essentially all physical simulations rely on destructive updates. On the other hand, it may be convenient to build a complete new representation particularly in the context of parallelism. Parallel execution often required copying for the sake of data locality.



FP Culture Continued

In Racket/Scheme

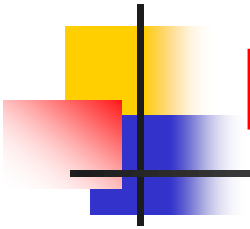
- Most mutation operations (at least those in libraries) end with a ! (read “bang”) character. Matthias Felleisien loved to title the lecture introducing the imperative extension of Scheme “The Big Bang!”
- Simulation of physical systems: the change in state over short time intervals is typically small and the successor state in discrete is often a simple update to the current state. In some cases, it is possible to preserve the old state and construct the new state by sharing pieces of the previous state but many data structures (like arrays) must be completely copied if the old state is to be preserved, compromising the efficiency of the update operation. Essentially all physical simulations rely on destructive updates. On the other hand, it may be convenient to build a complete new representation particularly in the context of parallelism. Parallel execution often required copying for the sake of data locality.



Mutation Example

Naïve Fibonacci with Memoization

```
(define fib
  (local
    [(define results (make-hash))  ;; results is empty hash table
     (define (fibHelp n)
       (cond [(< n 2) 1]
             ;; if n is in the memo table, return the cached value
             [(hash-has-key? results n) (hash-ref results n)]
             [else  ;; bind sum to fib(n)
               (let [(sum (+ (fibHelp (- n 1)) (fibHelp (- n 2))))]
                 (begin
                   (hash-set! results n sum) ;; add <n,sum> to table
                   sum))]))]
    fibHelp))
```



Bottom Up Improvement

Aside: Introducing local vars

Racket/Scheme supports *three* different forms of "let" (a common name in functional langugaes for an expression that introduces new local variables) that only differ on the text that is in the scope of the new bindings

- **`(let`** **`[(`*`x1 E1`*`) … (`*`xn En`*`)]`** ***`E`*****`)`**
- **`(let*`** **`[(`*`x1 E1`*`) … (`*`xn En`*`)]`** ***`E`*****`)`**
- **`(letrec`** **`[(`*`x1 E1`*`) … (`*`xn En`*`)]`** ***`E`*****`)`**

In all three constructs, the new local variables ***`x1`***, …, ***`xn`*** are "visible" in the body ***`E`***. In **`let`**, the new local variables are "invisible" (not in scope) in the right-hand-sides ***`E1`***, …, ***`En`*** of the new local bindings. In **`let*`**, each local variable ***`xi`*** is visible in *subsequent* right-hand-sides ***`Ei+1`***, …, ***`En`***. In **`letrec`**, all local variable are visible (but not *necessarily defined*) in all right-hand-sides ***`E1`***, …, ***`En`***.

Observations About Various `let` forms

Ordinary **`let`** appears in most functional languages because it simply abbreviates a lambda application:

```
(let [(x1 E1) … (xn En)] E) ≡ ((lambda (x1 … xn) E) E1 … En)
```

The **`let*`** operation has a nearly trivial definition:

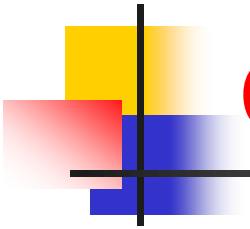
```
(let* [(x1 E1) … (xn En)] E) ≡
   (let [(x1 E1)]
       (let …
           (let [(xn En)] E) … E) … ))
```

which is how Java defines the meaning of a block of local bindings/

The **`letrec`** operation is alternate notation for **`local`**:

```
(letrec [(x1 E1) … (xn En)] E) ≡
   (local [(define x1 E1) … (define xn En)] E)
```

which is how Algol-like languages define the meaning of a block of local bindings. **`letrec`** is the most expressive of the three because it supports recursive definitions. If you use fresh names for local variables, it subsumes the others.



Core Imperative Operations

- Assignment

  `(set! v E)` rebinds variable *v* to the value of *E*

- Struct field mutation (except cons)

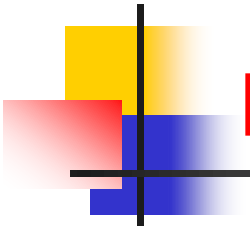
  `(<struct-name>-<field-name>! E1 E2)`

  changes the specified field in the struct determined by *E1* to the value of *E2*; if the value of *E1* is not an instance of the specified struct, an error is thrown

- Imperative sequencing

  `(begin E1 … En)`

  evaluates *E1*, … , *En* and returns the value of *E*



Example: a Racket Macro for Lazy cons

- The workaround: define lazy constructors as macros that expand lazy constructor applications to application of the corresponding strict constructor composed with thunk-wrapping each argument.
- What is a macro? A rule performed by the compiler that expands a macro invocation (which typically looks just like a function application) into standard language code that actually implements the macro operation.
- Macros are under-utilized in modern languages because surface (concrete) program syntax is so ugly and messy to manipulate. Strings separated by varying amounts of whitespace. Ugh!
- Programs conceptually have an intelligible tree-based (abstract) syntax that programmers never see. At this level macros are easy to express and understand. The Racket/Scheme/Lisp family of languages is ideal for macros because concrete syntax ≈ abstract syntax.

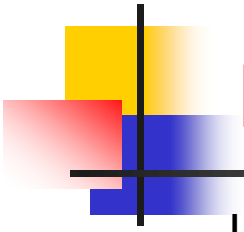
[The following slide as corrected now appears as slide 6 in Lecture 12.]

Corrected LazyRacket Macros

- Racket incorporates a very sophisticated macro system
- Simply macros are defined using the construct **`define-syntax-rule`**.
- To learn more about Racket macros, read the Chapter 16 of the Racket Guide (bundled as part of your DrRacket installation) entitled *Macros.*
- Using **`define-syntax-rule`**, we can easily define lazy-cons, lazy-first, and lazy-rest as follows:

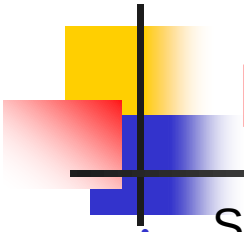
```
#lang racket
(define-syntax-rule (lazy-cons f r)
  (cons (lambda () f) (lambda () r)))
(define-syntax-rule (lazy-first lc) ((car lc)))
(define-syntax-rule (lazy-rest lc) ((cdr lc)))
```

Note: The standard dialect of Racket supports macros, while the HTDP languages do not. Standard Racket requires the use of **`car`** and **`cdr`** above instead of **`first`** and **`rest`** for technical reasons.



Practical Critique of LazyRacket Macros

- In Scheme, a **`cons`** struct (which is built-in to support lists) is mutable; in the official Racket languages, it is not. I don't think it is in the HTDP languages either even though the documentation states that it is permitted in the Advanced Student Language. I searched for documentation on the associated mutation operators and could not find any. In official Racket, there is a mutable form of **`cons`** called **`mcons`** the HTDP languages predate **`mcons`**. In the days of DrScheme (when the first edition of HTDP was written), **`cons`** `was mutable at the` **`Advanced Student`** `level and beyond (like` **`R5RS, PrettyBig`**`).` Today, mutable **`cons`** still exists in R5RS, but the implementation is now done using the Racket **`mcons`** package with **`mcons`**, **`set-mcons-car!`**, and **`set-mcons-cdr!`** renamed as **`mcons`**, **`set-mcons-car!`**, and **`set-mcons-cdr!`** to meet the R5RS standard. I don't know if **`mcons`** package precisely complies the old R5RS **`cons`** standard but I would not be surprised if it did. Why introduce all of this complication to support immutable **`cons`**? Optimization! **`mcons`** is pathological.



Practical Critique cont.

- Since **`cons`** is immutable, we cannot directly change the contents of the **`car`** and **`cdr`** fields. We have to embed boxes (**`box`** is a unary *mutable* constructor built-in to Racket and Scheme) inside the **`cons`** struct adding a level of indirection in the machine representation. At this point, it is probably better to use **`delay`**, a built-in lazy unary constructor in Racket/Scheme (it is not part of Core Racket) which performs our optimization. Built-in primitives typically are often optimized beyond the that what be achieved by equivalent source code. Since **`delay`** is lazy, it eliminates the need for thunks (it already includes similar machinery). Nevertheless, I am not pleased with this implementation of laziness because of the extra level of indirection, it will be significantly slower that the macro I wanted to write but could not because **`cons`** is now immutable.
- I am no more optimistic about the eventual fate or Racket than I am about the fate of Scala. Both are byzantine platforms created primarily for use by insiders (wizards who have spent years learning, implementing, and extending the platform).