Metacircular Interpreters

Kristopher Micinski
CIS 700 —Program Analysis: Foundations and Applications
Fall ’19, Syracuse University

Recap: Pattern Matching

Necessary idea for today: *quasipatterns*

Allow breaking down **list-like** data

```
(match e
  [`(,x z) x]
  [`(,x ,y) x]
  [`(,(? number? x) (,y ,z)) x])
```

Recap: Pattern Matching

Necessary idea for today: *quasipatterns*

Allow breaking down **list-like** data

```
(match e
  [`(,x z) x]
  [`(,x ,y) x]
  [`(,(? number? x) (,y ,z)) x])
```

Matches a list containing anything followed by the symbol z

‘(2 z) but not ‘(2 3) or ‘(2 y)

Recap: Pattern Matching

Necessary idea for today: *quasipatterns*

Allow breaking down **list-like** data

```
(match e
  [`(,x z) x]
  [`(,x ,y) x]
  [`(,(? number? x) (,y ,z)) x])
```

Matches a list of length 2, binds x & y

Note: clauses match **in order**

Recap: Pattern Matching

Necessary idea for today: *quasipatterns*

Allow breaking down **list-like** data

```
(match e
  [`(,x z) x]
  [`(,x ,y) x]
  [`(,(? number? x) (,y ,z)) x])
```

Matches list of length two:

- First element is a number?
- Second element is a list of length two
- Binds x, y, and z

Recap: Closures

Representing a function at runtime requires the code for the function **plus** pointers to free variables that that function relies upon.

To understand why, consider the following lambda.

```
(let ([x 1])
  (lambda (y) x))
```

Racket represents this at runtime as something like this:

Code	**Data (environment)**
`(lambda (y) x)`	`{x |-> 1}`

So when the code runs the value of **x** can be looked up from the environment.

Recap: Closures

Representing a function at runtime requires the code for the function **plus** pointers to free variables that that function relies upon.

To understand why, consider the following lambda.

```
(let ([x 1])
  (lambda (y) x))
```

Racket represents this at runtime as something like this:

Code	**Data (environment)**
`(lambda (y) x)`	`{x |-> 1}`

(To get technical about it: x points to a heap location that points to 1)

So when the code runs the value of **x** can be looked up from the environment.

Recap: Closures

Representing a function at runtime requires the code for the function **plus** pointers to free variables that that function relies upon.

To understand why, consider the following lambda.

```
(let ([x 1])
  (lambda (y) x))
```

Racket represents this at runtime as something like this:

Code	**Data (environment)**
`(lambda (y) x)`	`{x |-> 1}`

(To get technical about it: x points to a heap location that points to 1)

So when the code runs the value of **x** can be looked up from the environment.

A closure is **code plus environment**

Today’s insight

If you want to build an interpreter for a language, you can leverage features of the metalanguage.

(These interpreters are called **metacircular** (or **definitional**) interpreters.)

Metalanguage (defining language)	Language you use to define the implementation of the language (e.g., Racket, C, etc…)
Source Language	Language being implemented

Core Scheme

This is “core scheme”, a minimal language we will use for a part of CIS 700

```
(define (expr? e)
  (match e
    [(? symbol? var) #t]
    [(? lit?) #t]
    [`(lambda (,x) ,body) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse)) #t]
    [`(let* ([,(? xs) ,(? expr?)] ...)
        ,(? expr? body))]
    [`(letrec ([,(? symbol?) ,(? expr?)])
        ,(? expr? body)) #t]))

(define (lit? e)
  [(? number?) #t]
  [(? string?) #t]
  [(? boolean?) #t]
  [else #f])
```

Core Scheme

This is "core scheme", a minimal language we will use for a part of CIS 700

It includes…

- Variables (e.g., x)
- Literals (e.g., 5, "hello")
- Lambda expressions
- Function application
- If statements
- Let* (sequenced let)
- Letrec (recursive definitions)

```
(define (expr? e)
  (match e
    [(? symbol? var) #t]
    [(? lit?) #t]
    [`(lambda (,x) ,body) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse)) #t]
    [`(let* ([,(? xs) ,(? expr?)] ...)
        ,(? expr? body))]
    [`(letrec ([,(? symbol?) ,(? expr?)])
        ,(? expr? body)) #t]))

(define (lit? e)
  [(? number?) #t]
  [(? string?) #t]
  [(? boolean?) #t]
  [else #f])
```

Core Scheme

In this lecture, we will cover the implementation of **these**

It includes…

- **Variables** (e.g., x)
- **Literals** (e.g., 5, “hello”)
- **Lambda expressions**
- **Function application**
- **If statements**
- Let* (sequenced let)
- **Letrec** (recursive definitions)

```
(define (expr? e)
  (match e
    [(? symbol? var) #t]
    [(? lit?) #t]
    [`(lambda (,x) ,body) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse)) #t]
    [`(let* ([,(? xs) ,(? expr?)] ...)
        ,(? expr? body))]
    [`(letrec ([,(? symbol?) ,(? expr?)])
        ,(? expr? body)) #t]))

(define (lit? e)
  [(? number?) #t]
  [(? string?) #t]
  [(? boolean?) #t]
  [else #f])
```

It’s worth pointing out that—especially once we add some builtin functions (like car, +, etc…)—core Scheme is really expressive enough for very general programming…

```
(letrec
    ([foldl
      (lambda (fun default lst)
        (if (empty? lst)
            default
            (fun (car lst)
                 (foldl fun default (cdr lst)))))])
  (foldl (lambda (x y) (+ x y)) 0 (list 1 2 3)))
```

It’s worth pointing out that—especially once we add some builtin functions (like car, +, etc…)—core Scheme is really expressive enough for very general programming…

```
(letrec
    ([foldl
      (lambda (fun default lst)
        (if (empty? lst)
            default
            (fun (car lst)
                 (foldl fun default (cdr lst)))))])
  (foldl (lambda (x y) (+ x y)) 0 (list 1 2 3)))
```

Missing set! and continuations, but can already write real code!

First Coding Session: Figuring out what expr? actually means…

Why build Core Scheme in Racket?

Even though it seems kind of pointless to build an interpreter for a language in the same language, it’s going to help us build intuition for interpreter design.

This will build up to allowing us to build abstract versions of these interpreters, which are exactly program analyses.

Definitional Interpreters

Use as much of the metalanguage (defining language) as possible when building an interpreter.

We will currently define Core Scheme in Racket (easy, as Core Scheme is a subset of Racket).

So, how do we do it?

Idea: write a function, **interp(e)** that interprets a program and returns a resulting value (as a Scheme value)

Everything just returns ‘undefined…

```
;; Attempt 0
(define (interp e)
  (match e
    [(? symbol? var) 'undefined]
    [(? lit?) 'undefined]
    [`(lambda (,var) ,body) 'undefined]
    [`(,(? expr? e0) ,(? expr? e1)) 'undefined]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse)) 'undefined]
    [`(let* ([,(? symbol?) ,(? expr?)] ...)
        ,(? expr? body))
     'undefined]
    [`(letrec ([,(? symbol?) ,(? expr?)])
        ,(? expr? body))
     'undefined]))
```

Next attempt…

```
;; Attempt 1
(define (interp e env)
  (match e
    [(? symbol? var) (hash-ref env var)]
    [(? lit?) e]
    [`(lambda (,var) ,body)
     (lambda (x) (interp body (hash-set env var x)))]
    [`(,(? expr? e0) ,(? expr? e1))
     ((interp e0 env) (interp e1 env))]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse))
     (if (interp guard env)
         (interp etrue env)
         (interp efalse env))]
   … ;; Undefined other cases))
```

First idea: to interpret variables, use a hash table from Racket.

```
;; Attempt 1
(define (interp e env)
  (match e
    [(? symbol? var) (hash-ref env var)]
    [(? lit?) e]
    [`(lambda (,var) ,body)
     (lambda (x) (interp body (hash-set env var x)))]
    [`(,(? expr? e0) ,(? expr? e1))
     ((interp e0 env) (interp e1 env))]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse))
     (if (interp guard env)
         (interp etrue env)
         (interp efalse env))]
   … ;; Undefined other cases))
```

If we’re really being **true** to the spirit of definitional interpreters, we might be **tempted** to reuse variables from the metalanguage.

However, this is **hard**!

If we're really being **true** to the spirit of definitional interpreters, we might be **tempted** to reuse variables from the metalanguage.

However, this is **hard**!

There's no way to make this work, because var is not in the **lexical scope** here.

```
;; Attempt 1
(define (interp e env)
  (match e
    ;; Say we did this
    [(? symbol? var) var]
  … ;; Undefined other cases))
```

If we're really being **true** to the spirit of definitional interpreters, we might be **tempted** to reuse variables from the metalanguage.

However, this is **hard**!

There's no way to make this work, because var is not in the **lexical scope** here.

Instead, it is in an *(effectively)* dynamic scope implemented via the hash table **env**

```
;; Attempt 1
(define (interp e env)
  (match e
    ;; Say we did this
    [(? symbol? var) var]
  … ;; Undefined other cases))
```

However, our interpreter uses env in such a way as to recover static scope!

This is one way in which our interpreter has to compromise.

Variables are hard to implement via the defining language, but we can use a hash table (also in the defining language) to implement them.

This is important because we will do this with other language features

In fact, we **must** do it for features that are not default in the metalanguage

(For example, say we wanted lazy evaluation, as in Haskell!)

Next: literals are just themselves..

```
;; Attempt 1
(define (interp e env)
  (match e
    [(? symbol? var) (hash-ref env var)]
    [(? lit?) e]
    [`(lambda (,var) ,body)
     (lambda (x) (interp body (hash-set env var x)))]
    [`(,(? expr? e0) ,(? expr? e1))
     ((interp e0 env) (interp e1 env))]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse))
     (if (interp guard env)
         (interp etrue env)
         (interp efalse env))]
   … ;; Undefined other cases))
```

Next: interpret the lambda expression as a **Racket lambda**

```
;; Attempt 1
(define (interp e env)
  (match e
    [(? symbol? var) (hash-ref env var)]
    [(? lit?) e]
    [`(lambda (,var) ,body)
     (lambda (x) (interp body (hash-set env var x)))]
    [`(,(? expr? e0) ,(? expr? e1))
     ((interp e0 env) (interp e1 env))]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse))
     (if (interp guard env)
         (interp etrue env)
         (interp efalse env))]
   … ;; Undefined other cases))
```

Next: interpret the lambda expression as a **Racket lambda**

```
;; Attempt 1
(define (interp e env)
  (match e
    [(? symbol? var) (hash-ref env var)]
    [(? lit?) e]
    [`(lambda (,var) ,body)
     (lambda (x) (interp body (hash-set env var x)))]
    [`(,(? expr? e0) ,(? expr? e1))
     ((interp e0 env) (interp e1 env))]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse))
     (if (interp guard env)
         (interp etrue env)
         (interp efalse env))]
   … ;; Undefined other cases))
```

IMPORTANT:
Make sure that you understand that we are matching a **source** lambda and turning it into a lambda in the **metalanguage**!

Recall our recap of closures! Technically the "Racket lambda" is actually a #<procedure> (aka: *closure!*)

Assuming lambdas evaluate to Racket lambdas, we can then simply **apply** them to perform function application!

```
;; Attempt 1
(define (interp e env)
  (match e
    [(? symbol? var) (hash-ref env var)]
    [(? lit?) e]
    [`(lambda (,var) ,body)
     (lambda (x) (interp body (hash-set env var x)))]
    [`(,(? expr? e0) ,(? expr? e1))
     ((interp e0 env) (interp e1 env))]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse))
     (if (interp guard env)
         (interp etrue env)
         (interp efalse env))]
   … ;; Undefined other cases))
```

Note:
Think about why this is—*at runtime* the left hand side of he application (e0 e1) must evaluate to a lambda (or else an error arises).

Similarly, **if** can be interpreted by simply by evaluating the guard and then the appropriate expression.

```
;; Attempt 1
(define (interp e env)
  (match e
    [(? symbol? var) (hash-ref env var)]
    [(? lit?) e]
    [`(lambda (,var) ,body)
     (lambda (x) (interp body (hash-set env var x)))]
    [`(,(? expr? e0) ,(? expr? e1))
     ((interp e0 env) (interp e1 env))]
    [`(if ,(? expr? guard)
          ,(? expr? etrue)
          ,(? expr? efalse))
     (if (interp guard env)
         (interp etrue env)
         (interp efalse env))]
   … ;; Undefined other cases))
```

Two small examples for Core Scheme

```
(interp '((lambda (x) x) "Hello, World!") (hash))

(interp '(((lambda (x) (x x)) (lambda (x) x)) 1) (hash))
```

Adding builtins…

(Because our language is boring without them.)

Idea: add them to a hash map to give their implementation

Notice: if I had multi-argument application, this would be **unnecessary**…
(As I could just make builtins the initial environment!)

```
(define (builtin? x)
  (set-member? (set '+ '- '* '/ 'cons 'car 'cdr 'list '=) x))

(define builtins
  (hash '+ + '- - '* * '/ / 'cons cons 'car cdr 'list list '= equal?))

 (define (interp e env)
   (match e
     ;; Builtins!
     [`(,(? builtin? op) ,(? expr? builtin-args) ...)
      ;; Evaluate each argument and then apply the builtin denotation
      ;; of the function.
      (let ([evaluated-args (map
                             (lambda (arg) (interp arg env))
                             builtin-args)])
        ;; Take care to undersand this one!
        (apply (hash-ref builtins op) evaluated-args))]))
```

Adding builtins…

(Because our language is boring without them.)

Idea: add them to a hash map to give their implementation

Notice: if I had multi-argument application, this would be **unnecessary**…
(As I could just make builtins the initial environment!)

(Could also do this via currying..)

```
(define (builtin? x)
  (set-member? (set '+ '- '* '/ 'cons 'car 'cdr 'list '=) x))

(define builtins
  (hash '+ + '- - '* * '/ / 'cons cons 'car cdr 'list list '= equal?))

 (define (interp e env)
   (match e
     ;; Builtins!
     [`(,(? builtin? op) ,(? expr? builtin-args) ...)
      ;; Evaluate each argument and then apply the builtin denotation
      ;; of the function.
      (let ([evaluated-args (map
                             (lambda (arg) (interp arg env))
                             builtin-args)])
        ;; Take care to undersand this one!
        (apply (hash-ref builtins op) evaluated-args))]))
```

Currying

Let’s say my language only has single-argument lambdas

I can get multi-argument lambdas by using a trick

```
(lambda (x y z) (+ x y z))
```

Currying

Let's say my language only has single-argument lambdas

I can get multi-argument lambdas by using a trick

```
(lambda (x y z) (+ x y z))
```

Curry!

```
(lambda (x) (lambda (y) (lambda (z) (+ x y z))))
```

Replace multi-argument lambdas by **sequences** of lambdas!

Currying

Let’s say my language only has single-argument lambdas

I can get multi-argument lambdas by using a trick

```
(lambda (x y z) (+ x y z))
```

Curry!

```
(lambda (x) (lambda (y) (lambda (z) (+ x y z))))
```

Replace multi-argument lambdas by **sequences** of lambdas!

```
((lambda (x y z) (+ x y z)) 1 2 3)
```

Uses must be rewritten, too…

```
((((lambda (x) (lambda (y) (lambda (z) (+ x y z)))) 1) 2) 3)
```

Adding builtins…

(Because our language is boring without them.)

Idea: add them to a hash map to give their implementation

Notice: if I had multi-argument application, this would be **unnecessary**…
(As I could just make builtins the initial environment!)

```
(define (builtin? x)
  (set-member? (set '+ '- '* '/ 'cons 'car 'cdr 'list '=) x))

(define builtins
  (hash '+ + '- - '* * '/ / 'cons cons 'car cdr 'list list '= equal?))

 (define (interp e env)
   (match e
     ;; Builtins!
     [`(,(? builtin? op) ,(? expr? builtin-args) ...)
      ;; Evaluate each argument and then apply the builtin denotation
      ;; of the function.
      (let ([evaluated-args (map
                             (lambda (arg) (interp arg env))
                             builtin-args)])
        ;; Take care to undersand this one!
        (apply (hash-ref builtins op) evaluated-args))]))
```

An example using builtins…

```
(interp '((lambda (x) (if (= x 0) 1
                          (+ ((lambda (y) (if (= y 3) 1 2)) x) x)))
          (* 1 (if #f 1 4)))
        (hash))
```

This leaves only a few cases left undone:
- Multi-argument lambdas
- Let* (sequenced let)
- Letrec

You will do the first two for your project, along with an extension to core scheme where you add a minimal form of pattern matching.

However, letrec is generally quite interesting and presents some challenging questions about how to implement it.

Implementing Let

Intuition from last week (and Peter Landin):
let is just lambda by another name

```
(let ([x x-defn]) body)
```

```
                    ((lambda (x) body) x-defn)
```

This pattern (immediately apply a lambda) is sometimes called the “left-left-lambda” pattern

Implementing Let

Intuition from last week (and Peter Landin):
let is just lambda by another name

```
(let ([x x-defn]) body)

                          ((lambda (x) body) x-defn)
```

This pattern (immediately apply a lambda) is sometimes called the “left-left-lambda” pattern

An optimizing compiler will (probably) inline this.

Implementing Let*

Let transformation **easily generalizes**

```
(let* ([x 1]
       [y (+ x 1)])
  body)
```

$\longrightarrow$

```
((lambda (x)
   ((lambda (y) body)
    (+ x 1)))
 1)
```

Similar idea to currying
Chain of left-left-lambdas

Letrec

Letrec is the fundamental way that our language forms **loops**!

```
(letrec ([f
          (lambda (x) (if (= x 0)
                          1
                          (* x (f (- x 1)))))])
  (f 10))
```

This allows us to go from the finite to the infinite!
Thus this is (imo) most important form!

Here’s the *simple* way to implement letrec

```
(define f (lambda (x) x))
(set! f (lambda (x) (if (= x 0)
                        1
                        (* x (f (- x 1))))))
(f 10)
```

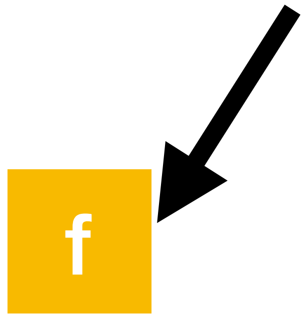
This is a huge hack.
It relies on set!

However, it works and is a simple way to understand how to implement recursion.

Why does it work?

```
(define f (lambda (x) x))
(set! f (lambda (x) (if (= x 0)
                        1
                        (* x (f (- x 1))))))
(f 10)
```

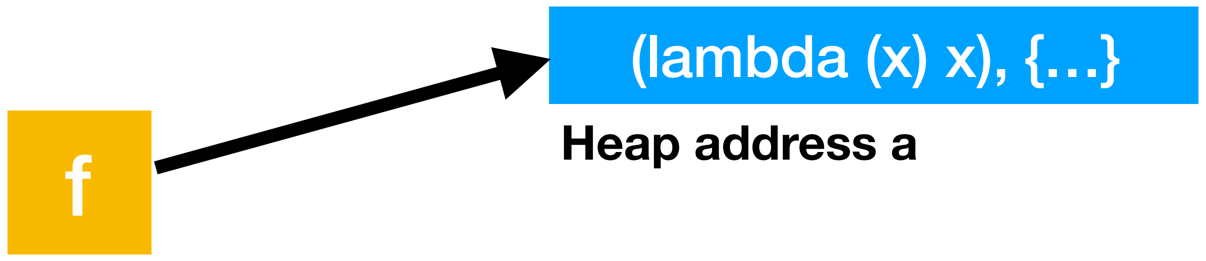
At this point, **f** becomes a box on the heap, initialized to the closure **(lambda (x) x)**



Why does it work?

```
(define f (lambda (x) x))
(set! f (lambda (x) (if (= x 0)
                        1
                        (* x (f (- x 1))))))
(f 10)
```

At this point, **f** becomes a box on the heap, initialized to the closure **(lambda (x) x)**

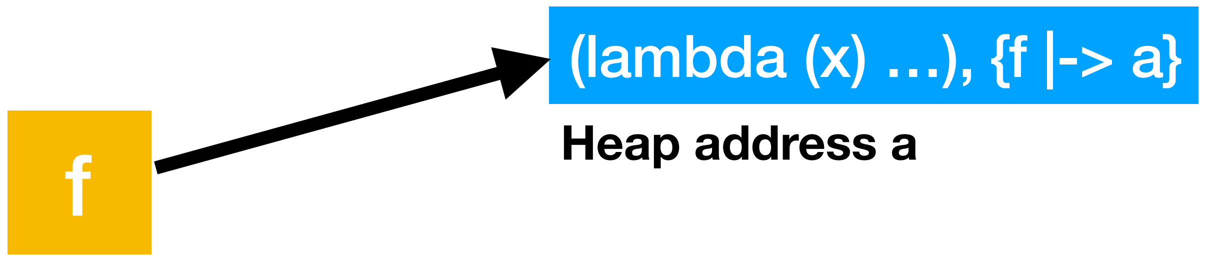


Which is really a pointer to some underlying address a

Why does it work?

```
(define f (lambda (x) x))
(set! f (lambda (x) (if (= x 0)
                        1
                        (* x (f (- x 1))))))
(f 10)
```

Then we change it to a lambda that mentions f, but in an environment that includes f!



We have leveraged the linked structure of the **metalanguage's** heap to implement the looping structure inherent to recursion.

This is the insight of definitional interpreters: use as much of the metalanguage as you can.

How can we implement it in **our interpreter**?

```
(define (interp-letrec e env)
  (match e
    [`(letrec ([,(? symbol? f) (lambda (,f-arg) ,f-body)])
        ,(? expr? body))
     ;; Make a mutable copy of the hash table
     (let ([env-copy (apply make-hash
                            (hash-map env
                                      (lambda (x y) (cons x y))))])
       ;; Mutably set f to be a lambda that points at itself
       (hash-set! env-copy
                  f
                  (lambda (x) (interp
                               f-body
                               (begin
                                 (hash-set! env-copy f-arg x)
                                 env-copy))))
       ;; Now interpret the body with this updated (mutable) env
       (interp body env-copy))]))
```

There are other ways we could have gotten this result

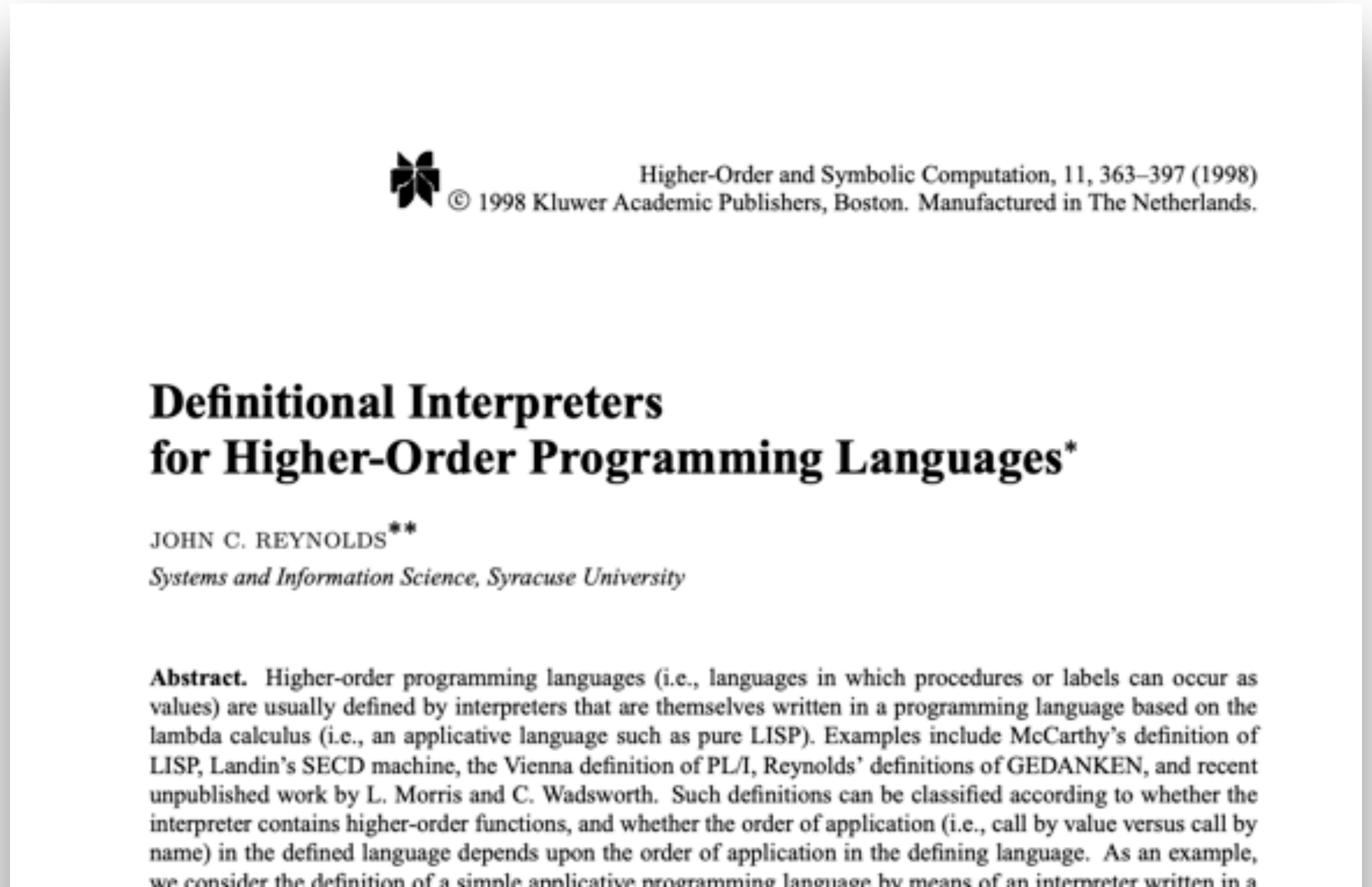
The effectual map is **not intrinsic to the semantics**
(which is important, since we would like a purely applicative semantics!)

We will see how to do this by using the Y combinator (next week)

I will also say that Reynolds takes a slightly different approach in his paper…

He uses a purely functional environment, which allows us to exploit the naturally linked structure of closures!

This is the reading for **next week**



Higher-Order and Symbolic Computation, 11, 363–397 (1998)
© 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Definitional Interpreters
for Higher-Order Programming Languages*

JOHN C. REYNOLDS**
Systems and Information Science, Syracuse University

Abstract. Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters that are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a

Homework 1

- Extend the interpreter from class w/
  - **Multi-argument lambdas, applications, and letrecs**
  - **Let* (sequencing let)**
  - **Simple pattern matching**
- **Bonus points** will be given for more complex pattern matching

Multi-argument lambdas

First task: make lambdas accept multiple arguments.

```
(lambda (x y) (+ x y))
```

Two possible ways to do this:

- Interpret directly using “apply”
  - Create (Racket) lambda that accepts list as argument
    - `(lambda l …)` <— l is treated as a list inside …
    - `((lambda l l) 1 2 3) = ‘(1 2 3)`
  - Allows you to apply a function to a list
- Generate new nested lambda forms and use currying
  - (lambda (x y …) …) -> (lambda (x) (lambda (y) …))..)
  - (f x y z) -> (((f x) y) z)
- **Your choice** as to which one you use!

Multi-argument letrec

Also need to do this with letrec.

Follows the same trick as multi-argument lambdas,
When you figure that out, letrec will immediately follow

Let*

You have two options for how to do this:

- Either implement as desugaring (recall let* slide)
- Or implement directly, make sure you allow sequencing

Pattern Matching

This one should be fun / creative.

Three types of patterns:

- Flat patterns (i.e., match a predicate)
- Cons patterns (i.e., match car / cdr)
- List patterns (match lists of finite length k, each w/ predicates applied)

Earlier...

Today's insight

If you want to build an interpreter for a language, you can leverage features of the metalanguage.

(These interpreters are called **metacircular** (or **definitional**) interpreters.)

Ways to Build Languages

In this lecture, we talked about two main "ways" to build languages:

- Write a definitional interpreter
- Desugar more complex forms into simpler ones
    - Then, use the definitional interpreter on **those**

We will cover **many other ways** as we continue on through lectures

Which will **set the stage** for being able to **easily** step to analyses!

We want the metalanguage to be small and have a mathematically-simple structure.

Because if we don’t understand the metalanguage, how can we possibly understand our implementations using it?

https://www.youtube.com/watch?v=Ow9AtuIuMLw

- Today—metacircular interpreters: interpreters that use features of metalanguage to implement source language.
- This is **good**: quickly build interpreter that is easy to write
- This is **bad**: if we want to be **fully-precise**, we want the metalanguage to be **as close to math** as possible.
- If we use a definitional interpreter, the semantics of the metalanguage is implicitly tied up in the language’s model
    - Mutability, order-of-eval, etc…! (This is Reynolds’ point!)
- We will **next** ask: how can we make the metalanguage *even simpler*? Goal: want smaller metalanguage.