Interpreting IfArith

CIS352 — Fall 2022

Kris Micinski

Today, we’re going to start building our **own** languages

We’re going to do this by writing **interpreters**

To build a programming language, we need two things:

A **syntax** for the language (and the ability to **parse** it)

A **semantics** for the language. Typically either an **interpreter** or a **compiler**

For this class, all of our programs are going to be written as Racket datums

We specify syntax via a predicate that uses pattern matching

This means we can just write programs in our language just by building data in Racket

Here is the first language we will define:

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

"Any integer is a program in our language."

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

"If e0 is an expression in our language, and e1 is an expression in our language, `(plus ,e0 ,e1) is, too."

```
(define (expr? e)
  (match e
    [(? integer? n) #t]
    [`(plus ,(? expr? e0) ,(? expr? e1)) #t]
    [`(div ,(? expr? e0) ,(? expr? e1)) #t]
    [`(not ,(? expr? e-guard)) #t]
    [`(if ,(? expr? e0) ,(? expr? e1) ,(? expr? e2)) #t]
    [_ #f]))
```

Here are some example expressions:

```
‘(plus 1 (div 2 3))
'(if 0 (plus 1 2) (div 2 2))
'(if 0 (plus 1 (div 2 3)) (if 1 (plus 2 3) 0))
```

IMPORTANT NOTE

We are defining a **new language** by **using** Racket. But our language is **not** Racket. In Racket, booleans are #t and #f. In **our** language, we will use 0 to represent false and non-0 to represent true (as in C).

Again, because this is confusing

When writing interpreters, always be careful to mentally separate the **language you are defining** and the language you are using to build the interpreter (Racket).

This can become confusing as the languages we build will “look like” Racket. Try to be mindful.

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

The “result” of programs will be a Racket integer:

```
(define value? integer?)
```

Key idea: write an **interp** function that takes in expressions as an argument, and returns **Racket** values

The “result” of programs will be a Racket integer:

```
(define value? integer?)

(define/contract (evaluate e)
  (-> expr? value?)
  ‘todo)
```

What should the following return…?
Remember, this is our own **new language we are defining, not necessarily Racket**

```
(evaluate '(plus 1 2))
=> 3
(evaluate '(if 0 (plus 1 2) (div 2 2)))
=> ‘todo
(evaluate '(if 1 (div 4 3) (plus 1 -1)))
=> ‘todo
```

What should the following return…?
Remember, this is our own **new language we are defining, not necessarily Racket**

```
(evaluate '(plus 1 2))
=> 3
(evaluate '(if 0 (plus 1 2) (div 2 2)))
=> 1
(evaluate '(if 1 (div 4 3) (plus 1 -1)))
=> 4/3
```

Now, let’s build **evaluate** ourselves

In this lecture, we built a **metacircular** interpreter

Important Definition

A metacircular interpreter is an interpreter which uses features of a "host" language to define the semantics of a "target" language

Which features of Racket did we use to define our language…?

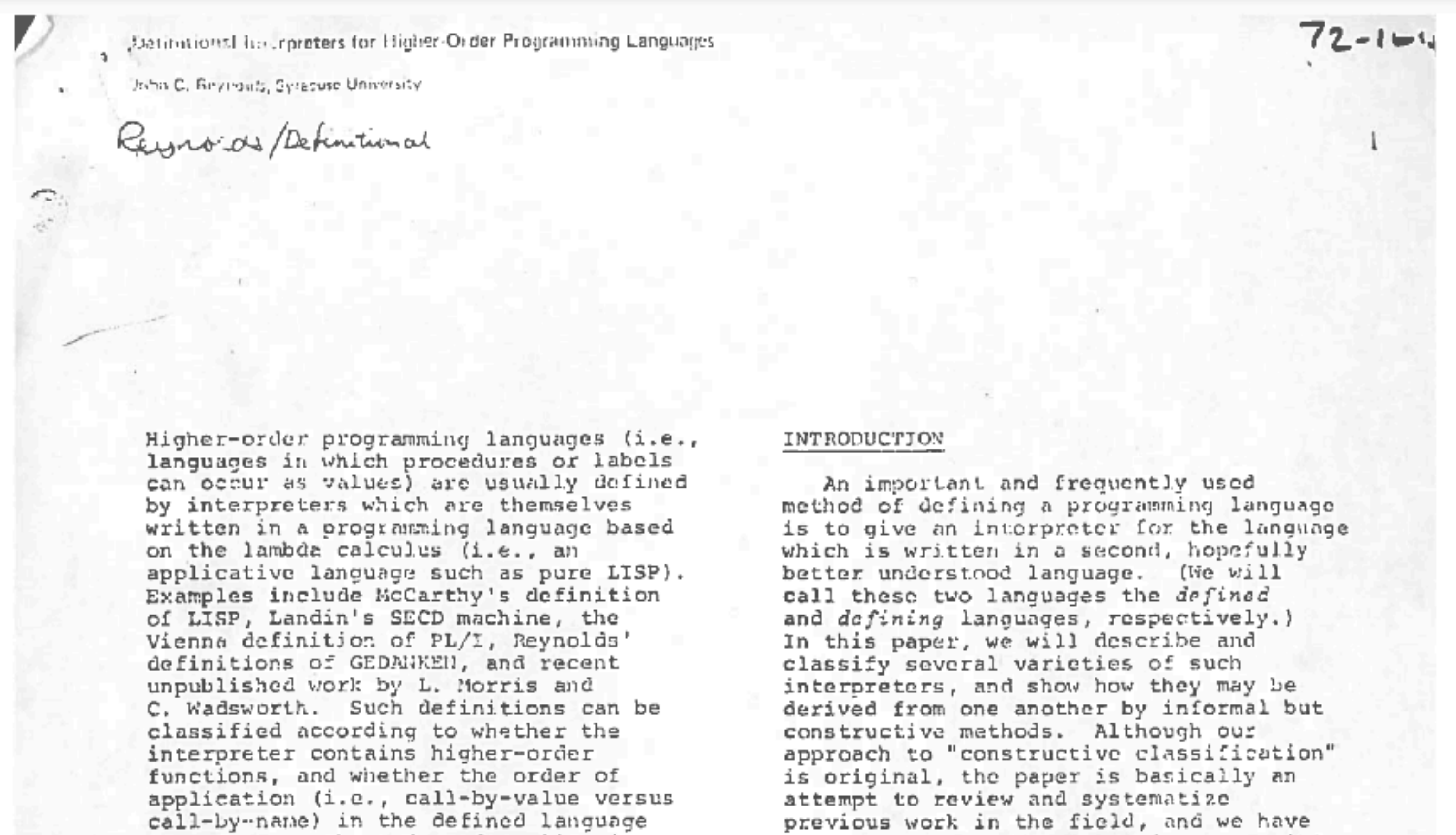
Important Definition

A metacircular interpreter is an interpreter which uses features of a "host" language to define the semantics of a "target" language

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    …
```

Notice how we **inherit** the definition of + from Racket

John Reynolds introduced metacircular interpreters in 1978. One key idea: metacircular interpreters inherit properties of their host language!



Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Reynolds/Definitional

Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters which 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

INTRODUCTION

An important and frequently used method of defining a programming language is to give an interpreter for the language which is written in a second, hopefully better understood language. (We will call these two languages the *defined* and *defining* languages, respectively.) In this paper, we will describe and classify several varieties of such interpreters, and show how they may be derived from one another by informal but constructive methods. Although our approach to "constructive classification" is original, the paper is basically an attempt to review and systematize previous work in the field, and we have

Note: our interpreter is **direct-style**, it is **not** tail recursive

```
(define (evaluate e)
  (match e
    [(? integer? n) n]
    [`(plus ,(? expr? e0) ,(? expr? e1))
     (+ (evaluate e0) (evaluate e1))]
    …
```

This means we are relying on Racket's **stack** as well

We will later see how to eliminate the need for this