Concepts of Programming Languages

Recursion and Types of Interpreters

Dr. Sebastian Erdweg
Dr. Guido Salvaneschi
Prof. Dr. Mira Mezini

Software Technology Group
TU Darmstadt | FB Informatik

Outline

- **Recursion**
  - Understanding recursion
  - Implementing recursion

- **Reflecting on representational choices**
  - Data structures for environments, numbers, functions
  - Types of interpreters

Summary of Languages So Far

- We started with **first-order functions (F1WAE)**.
- Function definitions are not expressions in the language - they are passed as a parameter to the interpreter.
- Functions are not values.

```
<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {let {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}
```

```
interp(
    App('f, 10),
    Map(
      'f -> FunDef('x, App('g, Add('x, 3))),
      'g -> FunDef('y, Sub('y, 1))))
```

Is special treatment for recursion needed in F1WAE?

Summary of Languages So Far

- Next, we studied/implemented first-class functions (FWAE).
- Function definitions are expressions that evaluate to values.
- We looked at both dynamic and static scoping.

```
<FWAE> ::= <num>
         | {+ <FWAE> <FWAE>}
         | {- <FWAE> <FWAE>}
         | {let {<id> <FWAE>} <FWAE>}
         | <id>
         | {fun {<id>} <FWAE>}
         | {<FWAE> <FWAE>}
```

Starting Point for This Lecture: CFWAE

- Suppose we have extended FWAE with **multiplication** and and **if0-conditional.**

- The result is called CFWAE

Is special treatment for recursion needed?

```
<CFWAE> ::= <num>
          | {+ <CFWAE> <CFWAE>}
          | {- <CFWAE> <CFWAE>}
          | {let {<id> <CFWAE>} <CFWAE>}
          | <id>
          | {fun {<id>} <CFWAE>}
          | {<CFWAE> <CFWAE>}

          | {* <CFWAE> <CFWAE>}
          | {if0 <CFWAE> <CFAE> <CFWAE>}
```

Recursive Factorial in CFAWE

```
{let fact {fun {n} {if0 n, 1, {* n {fact {- n 1}}}}}
  {fact 5}}
```

What does this expression evaluate to?

- If we have implemented static scoping, the first recursive call will fail.
- `fact` is bound in the body of `let`, but not in the named expression, i.e., not in the definition of `fact`.

What if we had implemented dynamic scoping?

RCFAE – A Language with Recursion

- We add a new binding construct letrec
- **Makes the new binding available in both its body and in its named expression**.

- The resulting language is called RCFAE

```
<RCFAE> ::= <num>
          | {+ <RCFAE> <RCFAE>}
          | {- <RCFAE> <RCFAE>}
          | <id>
          | {let {<id> <RCFAE>} <RCFAE>}
          | {fun {<id>} <RCFAE>}
          | {<RCFAE> <RCFAE>}
          | {* <RCFAE> <RCFAE>}
          | {if0 <RCFAE> <RCFAE> <RCFAE>}
          | {letrec {<id> <RCFAE>} <RCFAE>}
```

RCFAE – A Language with Recursion

Now we can write

```
{letrec fact {fun {n} {if0 n, 1, {* n {fact {- n 1}}}}}
   {fact 5}}
```

But what is the meaning of `letrec`?

Bindings and Environments

- Binding constructs such as `let` transform environments
- Given the current environment **e,** when it is evaluated, `let` produces two new environments, one for each of its sub-expressions:

for named expression

$$\rho_{\texttt{let,named}}(\texttt{e}) = \texttt{e}$$

for body

$$\rho_{\texttt{let,body}}(\texttt{e}) = \texttt{e + (boundId -> boundValue)}$$

`letrec`'s Environment for Body

$\rho_{\texttt{letrec,body}}$ of `letrec` remains probably the same …

Created by the evaluation of the named expression part

```
ρletrec,body(e) =
e + ('fact ->
      Closure('n, If0( /*…fact…*/ ), ?))
```

What should we put at the question mark?

`let`'s Environment for Body

```
ρlet,body(e) = e +
     ('fact -> Closure('n, If0( /*…fact…*/ ), e))
```

`let` closes the closure over the environment in which we interpret `let` – the ambient environment.

- Obviously, not OK for fact - first recursive call fails because `e` doesn't have a binding for `fact.`
- We need to have a binding for `fact` in the environment of the closure.

`letrec`'s Environment for the Named Expression

The evaluation of the named expression must ensure that the right environment is computed for the `fact` closure.

$$\texttt{\{letrec \{fact } \underbrace{\texttt{\{fun \{n\} \{if0 ... fact ...\} \}}} \texttt{\} \{fact 5\}\}}$$

$\rho_{\texttt{letrec,named}}(\texttt{e})$ = environment for evaluating this expression

So far, we know that with

$\rho_{\texttt{letrec,named}}(\texttt{e}) = \texttt{e}$

no recursive calls are possible.

Environments for Recursion

If we consider:

$\rho_{\texttt{letrec,named}}$`(e) =`

```
e +
('fact -> Closure('n, If0( /*…fact…*/ ), e))
```

$\texttt{interp(FunDef('n, If0(/*…fact…*/)),}\ \rho_{\texttt{letrec,named}}\texttt{(e)) =}$

```
Closure('n,
        If0( /*…fact…*/ ),
        e +
        ('fact -> Closure('n, If0( /*…fact…*/ ), e))
       )
```

Environments for Recursion

Which further means:

$\rho_{\texttt{letrec,body}}$ `(e) =`

```
e + ('fact -> Closure(
                'n,
                 If0(/*…fact…*/),
                e + ('fact -> Closure(
                                'n,
                                 If0(/*…fact…*/),
                                 e ))))
```

For initial call (`fact 3`) the body of `fact` will be evaluated in this env. extended with `[n → 3]`

→One recursive call (`fact 2`) **OK.**

Environment after the **first** recursive call, i.e., for evaluating (`fact 1`).

→ Second recursive call fails

Environments for Recursion

To perform the **second** recursive call we need

$\rho_{\texttt{letrec,body}}(\textit{e})$ =

```
e + ('fact ->
      Closure(
       'n,
       If0( /*…fact…*/ ),
       e + ('fact ->
             Closure(
             'n,
              If0( /*…fact…*/ ),
              e + ('fact ->
                    Closure(
                    'n,
                     If0( /*…fact…*/ ),
                     e ))))))
```

Environments for Recursion

- We recognize a reoccurring pattern ...
- We need environments that never “run out” of factorial definitions.

- The environment over which the closure closes should somehow be the environment that binds the closure to a name ...

Environments for Recursion

Let us assume a helper function, `ρ′` , that consumes the ambient environment *e*, and the environment to be put in the closure, let's call it *E*

```
ρ′(e) = λE.e + ('fact ->
                        Closure('n,
            If0( /*…fact…*/ ),
                           E))
```

Environments for Recursion

- For some $\boldsymbol{e_0}$ let's set $\boxed{F e_0 = \rho'(e_0)}$

- $\mathtt{F}e_0$ consumes an E and returns an environment that extends the ambient environment such that recursive calls in the body of `rec` can unfold "forever".

What E_0 will enable this?

Environments for Recursion

$E_0 = F_{e0}(E_0)$

We have to supply the answer that we want to produce!

An input value, for which the function result is the value itself, is called a **fixed-point** of the function.

It is not trivial to define a fixed point mathematically.

Is the problem easier to solve in programming?

Cyclic Environments for Recursion

Recall: We need environments that never “run out” of factorial definitions. The environment over which the closure closes should be the one that binds the closure....

$\rho_{\texttt{letrec,body}}$`(e) = e + ('fact ->`

Closure('n,

If0(/*…fact…*/),

)

The environment must be a cyclic data structure.

Recursiveness and Cyclicality

A recursive object contains references to instances of objects of the same kind as itself.

A cyclic object contains references to itself.

- Example of a recursive structure: A family tree, where each person refers to its parents. A family tree is recursive, but acyclic.

- Example of cyclic structure: The Web, a page can refer to a page, which can refer back to the first one.
- Naïve recursion over cyclic data may not terminate.

Outline

- **Recursion**
  - Understanding recursion
  - Implementing recursion

- **Reflecting on representational choices**
  - Data Structures for environments, numbers, functions
  - Types of Interpreters

Implementing Recursion

Recall the interpreter for `let`:

```
def interp(expr: Expr, env: Env = Map()): Val = expr match {
  ...
 case Let(boundId, namedExpr, boundBody) =>
        interp(boundBody, env + (boundId -> interp(namedExpr, env)))
 ...
}
```

Implementing Recursion

An analogous interpreter for `letrec` ..

```
def interp(expr: Expr, env: Env = Map()): Val = expr match {
  ...
 case LetRec(boundId, namedExpr, boundBody) =>
        interp(boundBody, env + (boundId -> interp(namedExpr, env)))
 ...
}
```

The closure will be closed over an environment that does not contain the recursive definition!

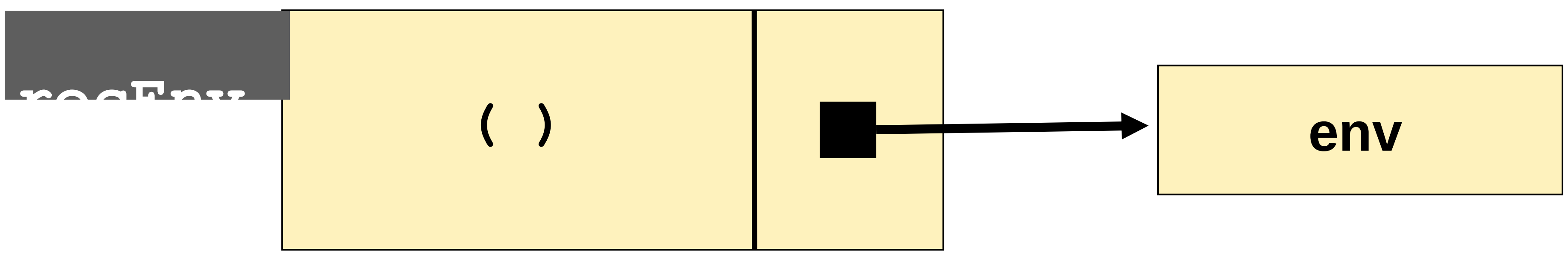
Implementing Recursion

Need to create an environment **`e`** that binds **`boundId`** to a closure that closes the function definition in **`namedExpr`** over **`e`**.

`e` is cyclic, since it refers to a value that, in turn, refers to **`e`**.

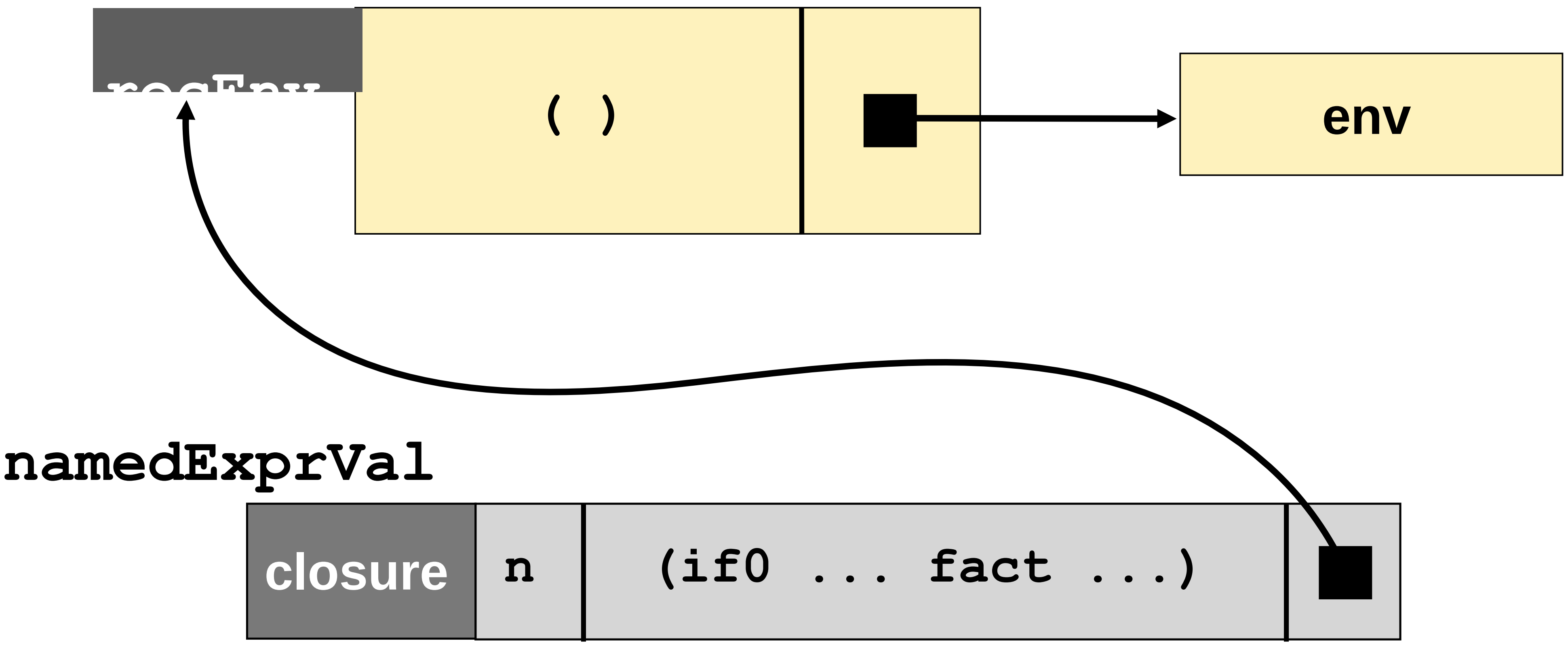
```
def interp(expr: Expr, env: Env = Map()): Val = expr match {
 ...
 case LetRec(boundId, namedExpr, boundBody) =>

   interp(boundBody,
           recBind(boundId, namedExpr, env) )

  ...

}
```

Cyclic Environments: The Idea

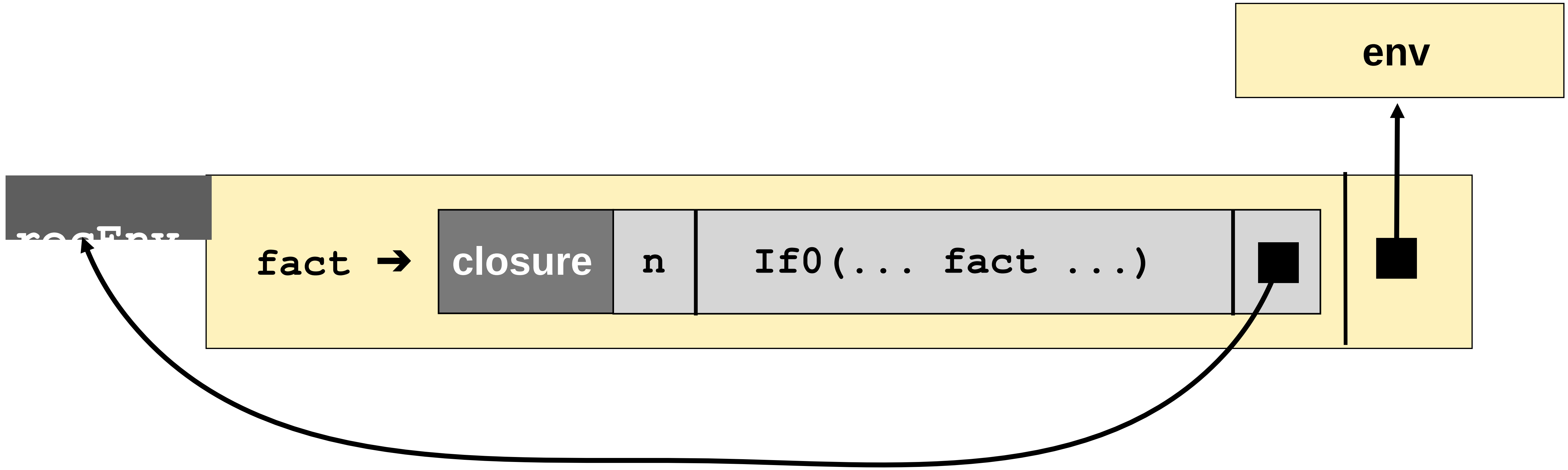


- First, create an empty mutable map and append **`env`** to it.
- Now we can interpret **`namedExpr`** in the new environment.

Cyclic Environments: The Idea



Cyclic Environments: The Idea



Add a binding of **`fact`** to the result of interpreting the named expression (**`namedExprVa`**l) in the **`recEnv`**

Done

<<Show and run the interpreter>>

Quiz

How would you extend a CFWAE interpreter that employs substitution to support recursive functions?

<<fill in the holes in the CFWAE interpreter with substitution>>

Outline

- **Recursion**
  - Understanding recursion
  - Implementing recursion
- **Reflecting on representational choices**
  - Data structures for environments, numbers, functions
  - Types of interpreters

Representation Choices

We made several choices about the representation of certain language concepts in our interpreter ...

- **Numbers** are represented by Scala numbers.
- **Environments** are represented by hash maps.
- **Functions** are represented by a custom data structure (`Closure`).

Alternative Representation Choices

Next ...

- we‘ll consider what other alternatives we had ...
- we’ll reflect on the choices we made or could have made…

Alternative Representation Choices

- **Numbers** are represented by Scala numbers.

Numbers are not interesting; we will stay with Scala numbers.

- **Environments** are represented by hash maps.
- **Functions** are represented by a custom data structure (`Closure`).

Alternative Representation of Environments

What could the alternative be?

- An environment maps identifiers to values ... it's just a (partial) function.
- Hence, Scala functions can represent environments …

```
type Env = (Symbol => Val)

def createEnv(name: Symbol, value: Val, oldEnv: Env): Env = ...
```

Environments as Scala Functions

```
def createEnv(name: Symbol, value: Val, oldEnv: Env): Env =
  id => if (id == name) value
        else oldEnv(id)
```

What's lookup now?

```
def emptyEnv(name: Symbol): Val =
  sys.error("No binding for " + name)
```

Show the interpreter CFWAE-fun-env

Environments as Scala Functions

How do we implement recursion with environments represented by Scala functions?

```
case LetRec(boundId, namedExpr, boundBody) =>
  interp(boundBody(recBind(boundId, namedExpr, env))}
```

Environments as Scala Functions

How would **`recBind`** look like?

```
def recBind(boundId: Symbol, namedExpr: Expr, env: Env): Env = {
    ...

    (id: Symbol) => { ... }


    ...
}
```

We know, it will return a function ...
But, what's in the holes?

Environments as Scala Functions

```
def recBind(boundId: Symbol, namedExpr: Expr, env: Env): Env = {
  def recEnv: Env = { id =>
      if (id == boundId) interp(namedExpr, recEnv)
      else env(id)
  }
  recEnv
}
```

Any problems with this?

Show the interpreter RCFWAE-fun-env

Outline

- **Recursion**
  - Understanding recursion
  - Implementing recursion

- **Reflecting on representational choices**
  - Data structures for environments, numbers, functions
  - Types of interpreters

Types of Interpreters

- **Syntactic interpreter**
  - uses the interpreting language only for the purpose of representing terms of the interpreted language,
  - implements all the corresponding behavior explicitly
- **Meta interpreter**
  - uses features of the interpreting language to directly implement behavior of the interpreted language
- **Meta-circular interpreter**
  - a meta interpreter in which the interpreting and interpreted language are the same

How do the interpreters we have seen so far fit into these definitions?

Environments as Scala Functions

- The new interpreter for RCFWAE using functional environments is a meta-interpreter because it uses Scala's recursion to directly implement recursion.
- The original RCFWAE interpreter is a syntactic interpreter because it does not assume recursion in the underlying language.

How do you like/dislike the solution that uses Scala functions to model environments?

Alternative Representation of Functions

Can you think of an alternative to the custom data structure for representing functions?

We could use a Scala function...

```
sealed abstract class Val
case class Num(n: Int) extends Val
case class Closure(f: Val => Val) extends Val
```

How would the interpreter change?

Representing Functions as Scala Functions

```
def interp(expr: Expr, env: Env = emptyEnv): Val = expr match {
  . . .

 case Fun(arg, body) =>
   Closure( argValue => interp(body, createEnv(arg, argValue, env)) )
 . . .

  case App(funExpr, argExpr) =>
    val funV = interp(funExpr, env)
    val argV = interp(argExpr, env)
    funV match {
      case Closure(f) => f(argV)
      case _ => sys.error("can only apply functions, but got: " + funV)
    }
}
```

Representing Functions as Scala Functions

Questions that are not obvious when we rely on Scala's functions …

Representing Functions as Scala Functions

Is the body of a function evaluated at the definition site, i.e., in the `Fun` branch of the interpreter?

No, because it is “under a lambda”

Representing Functions as Scala Functions

Which environment is used for function application? Does the interpreter implement static scoping or dynamic scoping?

Static scoping because Scala is statically scoped

Pros and Cons of Meta Interpreters

- If the interpreted and the interpreter language match closely, a meta interpreter can be very easy to write

- If they do not match it can be hard
  - Static scoping vs dynamic scoping
  - Lazy vs strict evaluation
  - …

- A meta interpreter does not help in understanding the features we are implementing
  - Started implementing recursion based on a cyclic environments, which are arguably easier to understand than recursion and than ended up implementing cyclic environments in terms of Scala's recursion!

When to Use Meta Interpretation

- Use a meta interpreter approach for features we understand (e.g., numbers).
- But write a syntactic interpreter for features under examination.
- Once we understand features, we can replace them with a meta interpreter to move on to more complex features.