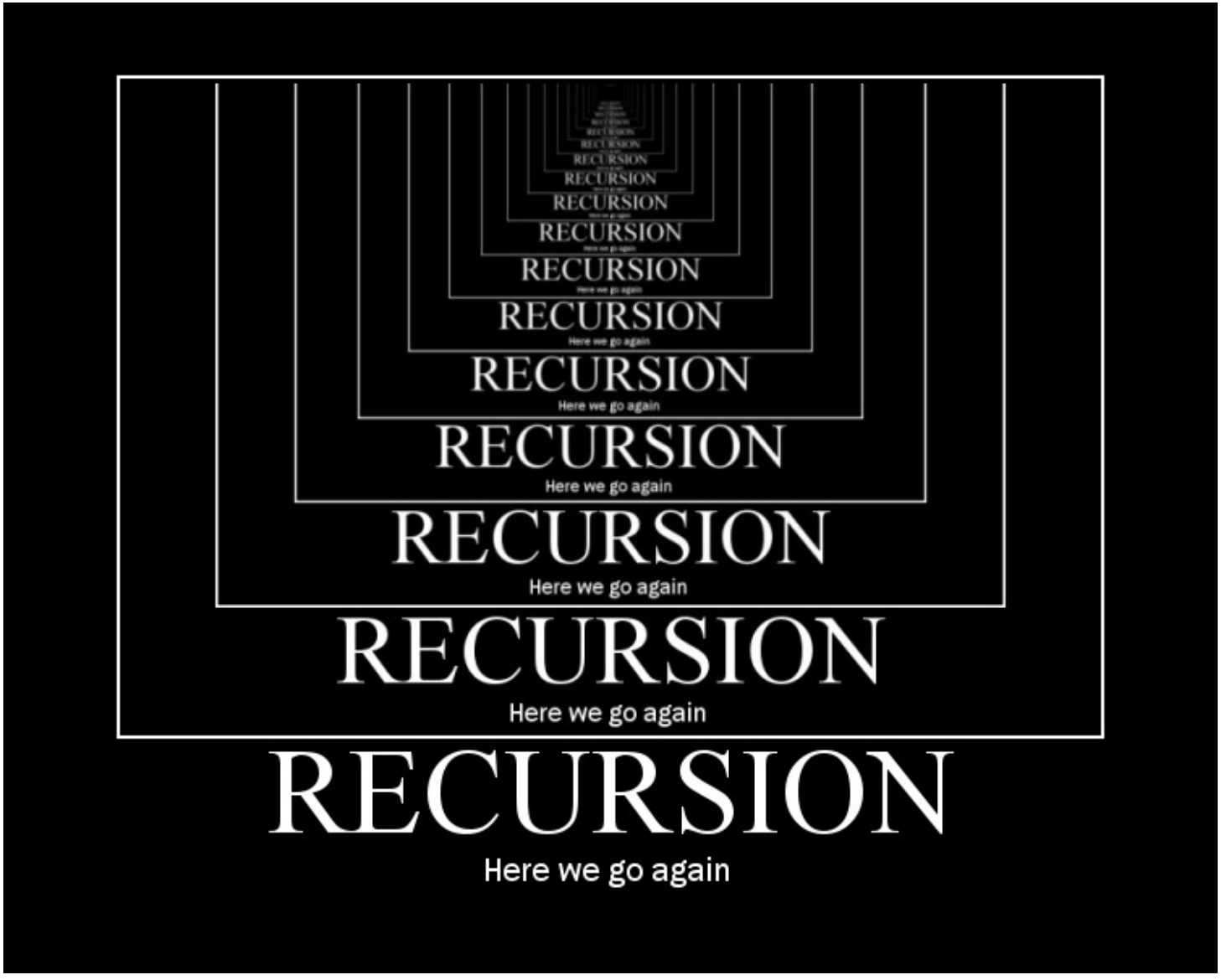
CS4450/7450
AoPL, Chapter 5: Recursion
Principles of Programming Languages

Dr. William Harrison

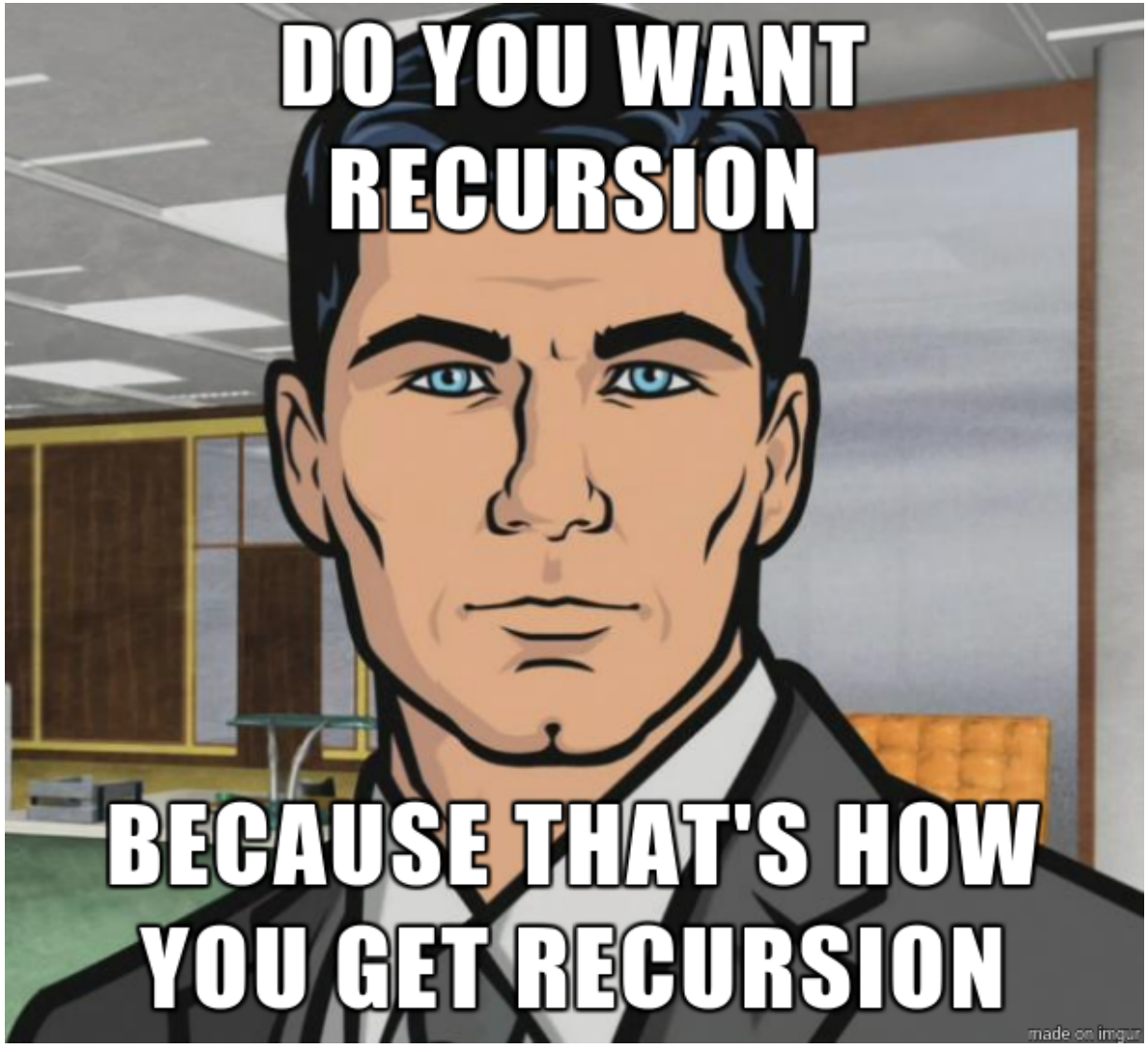
University of Missouri

October 17, 2018

What is Recursion?

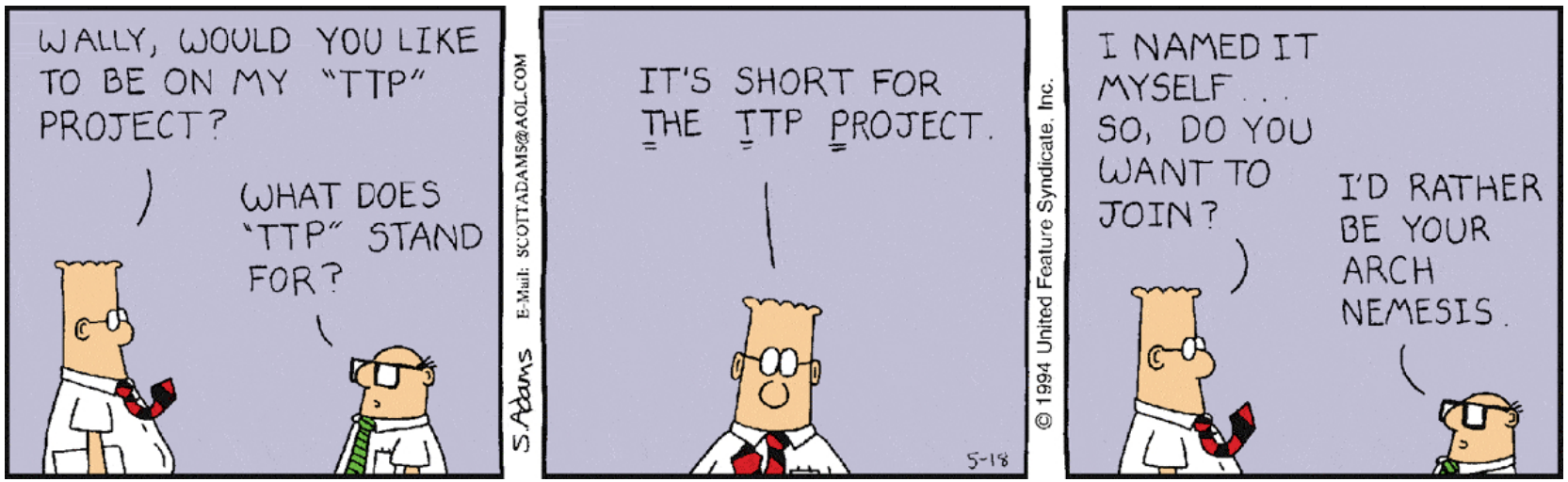


What is Recursion?



What is Recursion?

What is Recursion?



Announcements

- We're continuing with William Cook's online textbook, *Anatomy of Programming Languages*. It is available here. We're in Chapter 5.
- All programming languages have some notion of recursion—even if you don't think of it as recursion:

```
while b { c } = if b then c ; while b { c } else halt
```

- This chapter answers the question: what is recursion?

Outline for section 1

Recursive Functions

- Recursive Functions are functions that call themselves:

```
let
    fac = \ n -> if n == 0 then 1 else n * fac(n-1)
in
    fac(5)
```

- In the concrete syntax of the FirstClassFunctions.hs interpreter, this is written:

```
var fac = function(n) { if (n==0) 1 else n * fac(n-1) };
fac(5)
```

Recursive Functions

- Recursive Functions are functions that call themselves:

```
let
    fac = \ n -> if n == 0 then 1 else n * fac(n-1)
in
    fac(5)
```

- In the concrete syntax of the FirstClassFunctions.hs interpreter, this is written:

```
var fac = function(n) { if (n==0) 1 else n * fac(n-1) };
fac(5)
```

- Let's test this out using First Class Functions.

Review from AoPL 2: Scope

Scope of a Variable Declaration

is the portion of the code text where that declaration holds.

```
let y = 7 in
  let x = 3 in
     5 + (let x = 2 in x + y) * x
```

Scope of y

```
let y = 7 in
  let x = 3 in
     5 + (let x = 2 in x + y) * x
```

Scope of first x

```
let y = 7 in
  let x = 3 in
     5 + (let x = 2 in x + y) * x
```

Scope of second x

What's the Problem?

- The scope of the red declaration is the red code, not the blue code:

```
var fac = function(n) { if (n==0) 1 else n * fac(n-1) };
fac(5)
```

  so there's no binding for **`fac`**

Outline for section 2

Implementing Recursion using Haskell's Recursion

- Here's the way we defined local declarations

```
eval (Declare x exp body) env = eval body newEnv
  where newEnv = (x, eval exp env) : env
```

Implementing Recursion using Haskell's Recursion

- Here's the way we defined local declarations

```
eval (Declare x exp body) env = eval body newEnv
  where newEnv = (x, eval exp env) : env
```

- The problem here is that the bound expression `exp` is evaluated in the parent environment `env`.
  - To allow the bound variable `x` to be used within the expression `exp`, the expression must be evaluated in the new environment.

Implementing Recursion using Haskell's Recursion

- Here's the way we defined local declarations

```
eval (Declare x exp body) env = eval body newEnv
  where newEnv = (x, eval exp env) : env
```

- The problem here is that the bound expression `exp` is evaluated in the parent environment `env`.
  - To allow the bound variable `x` to be used within the expression `exp`, the expression must be evaluated in the new environment.
- Fortunately this is easy to implement in Haskell:

```
eval (Declare x exp body) env = eval body newEnv
  where newEnv = (x, eval exp newEnv) : env
```

- We still need non-recursive declarations; e.g., don't intend for the following to be recursive:

```
let x = x + 1 in x
```

- We still need non-recursive declarations; e.g., don't intend for the following to be recursive:

```
let x = x + 1 in x
```

- Changes to Abstract Syntax for Recursive Functions:

```
data Exp = ...
         | Declare    String Exp Exp
         | RecDeclare String Exp Exp
```

- We still need non-recursive declarations; e.g., don't intend for the following to be recursive:

```
let x = x + 1 in x
```

- Changes to Abstract Syntax for Recursive Functions:

```
data Exp = ...
         | Declare    String Exp Exp
         | RecDeclare String Exp Exp
```

- Additional `eval` clause:

```
eval (RecDeclare x exp body) env = eval body newEnv
  where newEnv = (x, eval exp newEnv) : env
```

- We still need non-recursive declarations; e.g., don't intend for the following to be recursive:

```
let x = x + 1 in x
```

- Changes to Abstract Syntax for Recursive Functions:

```
data Exp = ...
         | Declare    String Exp Exp
         | RecDeclare String Exp Exp
```

- Additional `eval` clause:

```
eval (RecDeclare x exp body) env = eval body newEnv
  where newEnv = (x, eval exp newEnv) : env
```

- In the concrete syntax of the RecursiveFunctions.hs interpreter, this is written:

```
rec fac = function(n) { if (n==0) 1 else n * fac(n-1) };
fac(5)
```