The Lambda Calculus

An lambda calculus expression is one of:

- a name such as x, y, z, …
- a function (\x → M) where M is an expression
- an application (M N) when M, N are expressions

Simplification rule — any instance of a redex in a term can be simplified:

(\x → M) N ⇒ M{N/x}

where M{N/x} represents M in which every (free) occurrence of x is replaced by N

The Lambda Calculus

The lambda calculus can be used as a basis for a programming language.

That's roughly how you get Lisp, ML, Haskell, …

Python and Javascript are obtained by adding state and mutability.

We saw that we can encode Booleans, integers, pairs, lists, recursion.

We take a different path today: we extend the lambda calculus and turn it into an interpreter for a small programming language.

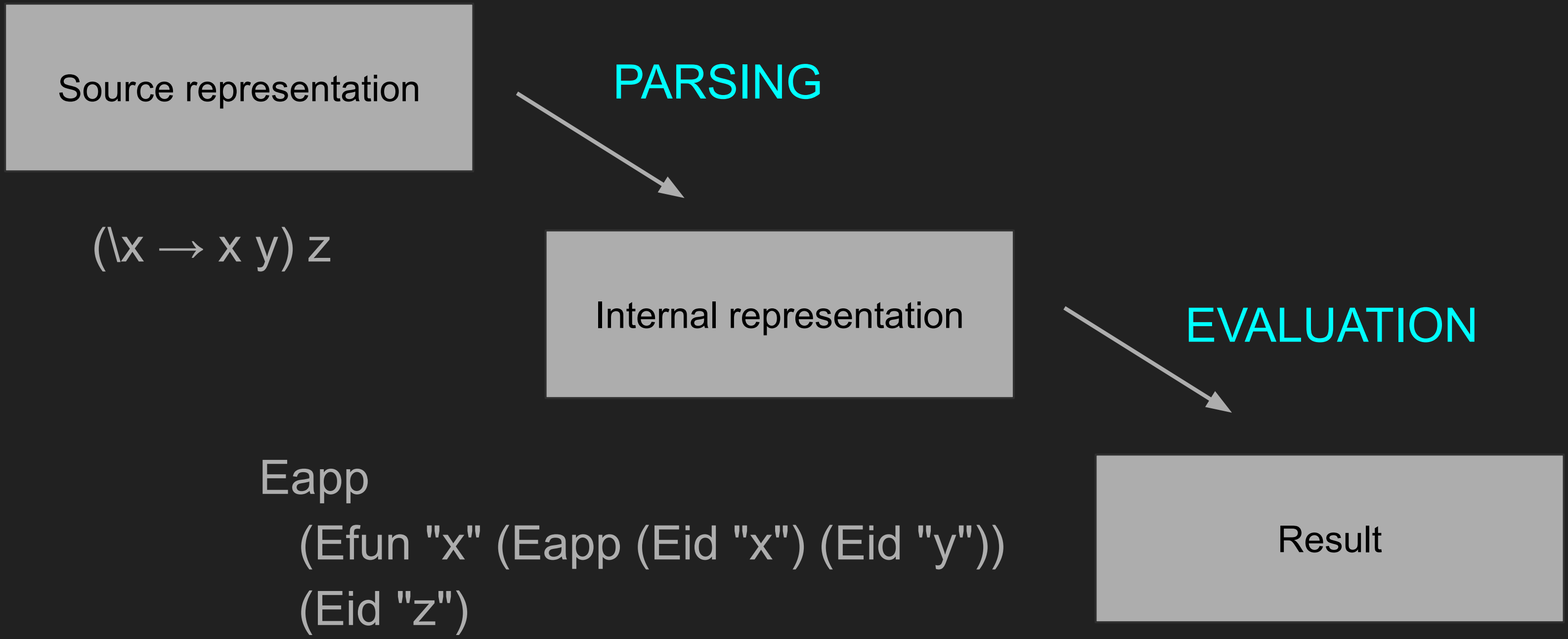
1 — Call-by-value reduction

Representing lambda calculus expression:

```
data Exp =
  Eid String |
  Efun String Exp |
  Eapp Exp Exp
```

(\x → x y) z   can be written   Eapp (Efun "x" (Eapp (Eid "x") (Eid "y")) (Eid "z")

1 — Call-by-value reduction



1 — Call-by-value reduction

Leftmost innermost redex, without simplifying within body of functions

```
eval :: Exp → Exp

eval (Eid s) = Eid s
eval (Efun s e) = Efun s e
eval (Eapp e1 e2) =
   case eval e1 of
     Efun s e → eval (substitute e s (eval e2))
     e → Eapp e (eval e2)
```

This is the core evaluation process of all eager languages

2 — Integer values

The "applied" lambda calculus: expressions evaluate to *values*.

```
data Value =
  Vint Int |
  Vfun String Exp
```

Integer literals, operations:

```
data Exp = … |
  Eint Int |
  Eadd Exp Exp
```

2 — Integer values

```
eval :: Exp → Value

eval (Eid s) = error "unsubstituted name"
eval (Efun s e) = Vfun s e
eval (Eapp e1 e2) =
   let VFun s e = eval e1 in eval (substitute e s (eval e2))
eval (Eint i) = Vint i
eval (Eadd e1 e2) =
  let (Vint i1, Vint i2) = (eval e1, eval e2) in Vint (i1 + i2)
```

3 — Conditionals

```
data Value = … |
    Vbool Bool
```

Booleans literals, operations, conditionals:

```
  data Exp = … |
    Ebool Bool |
    Eiszero Exp |
    Eif Exp Exp Exp
```

3 — Conditionals

```
eval :: Exp → Value
…
eval (Ebool b) = Vbool b
eval (Eiszero e) =
  let Vint i = eval e  in Vbool (i == 0)
eval (Eif e1 e2 e3) =
  let Vbool b = eval e in   if b then eval e2 else eval e3
```

4 — Environments

Let's improve evaluation by using a lookup table instead of explicit substitution:

```
data Env = Env [(String, Value)]

addEnv :: Env → String → Value → Env
lookupEnv :: Env → String → Value
```

Pass the environment to the evaluation function

- add a binding to the environment when applying a function

4 — Environments

Subtlety: to replicate substitution, functions need to "remember" the environment that existed when they were created

(\x → (\y → x)) 3 ⇒ (\y → 3)

So the function (\y → x) needs to remember that x is 3 when created

- A function / environment pair is called a closure

```
data Value = … |
  Vfun String Exp Env
```

4 — Environments

```
eval :: Env → Exp → Value

eval _ (Eid s) = error "unbound identifier"
eval env (Efun s e) = Vfun s e env
eval env (Eapp e1 e2) =
  let VFun s e env1 = eval env e1
      v2 = eval env e2
  in eval (addEnv env1 s v2) e
…
```

4 — Environments

```
…
eval env (Eint i) = Vint i
eval env (Eadd e1 e2) =
  let (Vint i1, Vint i2) = (eval env e1, eval env e2)  in Vint (i1 + i2)
eval env (Ebool b) = Vbool b
eval env (Eiszero e) =
  let Vint i = eval env e  in Vbool (i == 0)
eval env (Eif e1 e2 e3) =
  let Vbool b = eval env e
  in if b then eval env e2 else eval env e3
```

5 — Primitive operations

Once we have environments, we can fold primitive operations (add, iszero, …) into the *initial environment* as a special kind of value

```
data Value = … |
  Voper (Value -> Value)
```

And we can now remove Eadd and Eiszero from the Exp type

- essentially removing them from the syntax
- and making them library functions

5 — Primitive operations

```
eval :: Env → Exp → Value

eval _ (Eid s) = error "unbound identifier"
eval env (Efun s e) = Vfun s e env
eval env (Eapp e1 e2) =
  let v2 = eval env e2
  in case eval env e1 of
      VFun s e env1 → eval (addEnv env1 s v2) e
      Voper op → op v2
…
```

5 — Primitive operations

And now we can make add and iszero built-in functions in the initial environment:

```
init = Env [
  ("iszero", Voper (\v → let Vint i = v  in Vbool (i == 0))),
  ("add", Voper (\v1 → Voper (\v2 →
              let (Vint i1, Vint i2) = (v1, v2) in Vint (i1 + i2))))
]
```

6 — Definitions

We can add definitions such as

```
let x = exp in exp
```

in a natural way:

```
data Exp = … |
  Elet String Exp Exp

…
eval env (Elet s e1 e2) =
  eval (addEnv env s (eval env e1)) e2
```

6 — Definitions

Multi-definitions are more easily implemented in the parser!

```
let x1 = exp1
    x2 = exp2
    …
in exp
```

returns the same internal representation as

```
let x1 = exp1 in let x2 = exp2 in … in exp
```

Challenge: recursive functions!