Part I

Functions

```
fun double(x):
  x + x

fun quadruple(x):
  double(double(x))

quadruple(2)
```

→ 8

Functions

```
1 + (fun double(x): x + x)
```

?

No: a function ***definition*** is not an expression

Functions

```
1 + double(4)
```

?

Yes: a function ***call*** is an expression

We'll use ***call*** and ***application*** interchangably

Function Definitions

```
fun triple(x):
  x + x + x
```

A function has

- a name
- an argument name
- a *body*

```
type BodyExp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: BodyExp, r :: BodyExp)
| multE(l :: BodyExp, r :: BodyExp)
```

?

Function Definitions

```
fun triple(x):
  x + x + x
```

A function has

- a name
- an argument name
- a *body*

Allow `x` to be an expression, and then

```
x + x + x
```

is also an expression

Functions and Function Calls

- numbers
- identifiers
- addition expressions
  - first and second arguments are expressions
- multiplication expressions
  - first and second arguments are expressions
- function-call expressions
  - a function name and an argument expression

- a function definition
  - a function name, argument name, and body expression

Functions and Function Calls

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
         r :: Exp)
| multE(l :: Exp,
         r :: Exp)
| appE(s :: Symbol,
        arg :: Exp)

type FunDef
| fd(name :: Symbol,
      arg :: Symbol,
      body :: Exp)
```

Representing Programs

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
         r :: Exp)
| multE(l :: Exp,
         r :: Exp)
| appE(s :: Symbol,
        arg :: Exp)

type FunDef
| fd(name :: Symbol,
      arg :: Symbol,
      body :: Exp)
```

```
1 + 2
```

```
plusE(intE(1),
      intE(2))
```

Representing Programs

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
         r :: Exp)
| multE(l :: Exp,
         r :: Exp)
| appE(s :: Symbol,
        arg :: Exp)

type FunDef
| fd(name :: Symbol,
      arg :: Symbol,
      body :: Exp)
```

```
x + 2
```

```
plusE(idE(#'x),
      intE(2))
```

Representing Programs

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
         r :: Exp)
| multE(l :: Exp,
         r :: Exp)
| appE(s :: Symbol,
        arg :: Exp)

type FunDef
| fd(name :: Symbol,
      arg :: Symbol,
      body :: Exp)
```

```
fun plus_two(x):
  x + 2
```

```
fd(#'plus_two,
   #'x,
   plusE(idE(#'x),
         intE(2)))
```

Representing Programs

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
         r :: Exp)
| multE(l :: Exp,
         r :: Exp)
| appE(s :: Symbol,
        arg :: Exp)

type FunDef
| fd(name :: Symbol,
      arg :: Symbol,
      body :: Exp)
```

```
plus_two(9)
```

```
appE(#'plus_two,
     intE(9))
```

Representing Programs

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
         r :: Exp)
| multE(l :: Exp,
         r :: Exp)
| appE(s :: Symbol,
        arg :: Exp)

type FunDef
| fd(name :: Symbol,
      arg :: Symbol,
      body :: Exp)
```

```
fun double(x):
  x + x

fun quadruple(x):
  double(double(x))

quadruple(2)
```

```
[fd(#'double, #'x,
    plusE(idE(#'x),
          idE(#'x))),
 fd(#'quadruple, #'x,
    appE(#'double,
         appE(#'double,
              idE(#'x))))]

appE(#'quadruple, intE(2))
```

Part 2

Evaluating Function Calls

```
fun double(x):
  x + x

double(3)
```

↠ `3 + 3`

↠ `6`

```
interp :: (Exp, Listof(FunDef)) -> Int

get_fundef :: (Symbol, Listof(FunDef)) -> FunDef

subst :: (Exp, Symbol, Exp) -> Exp
```