Lenguajes de Programación

- First-class functions

Federico Olmedo
Ismael Figueroa

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f            w/ def.
10}}
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {{fun {x} x} 10}}
```

2

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr Env -> ...
```

2

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f            w/ def.
10}}
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined “before” the program is executed.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr Env -> ...
```

2

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

2

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

2

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
(list (fundef 'f 'x (id 'x)))
```

w/ def.

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

```
interp :: Expr Env -> ...
```

A taxonomy of functions

First-order functions

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

2

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr Env -> ...
```

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
(list (fundef 'f 'x (id 'x)))
```

w/ def.

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

```
interp :: Expr Env -> ...
```

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f        w/ def.
10}}
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f        w/ def.
10}}
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

```
interp :: Expr Env -> ...
```

2

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

```
interp :: Expr Env -> ...
```

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

2

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr Env -> ...
```

2

A taxonomy of functions

First-order functions

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are not values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

2

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f          w/ def.
10}}
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

```
interp :: Expr Env -> ...
```

A taxonomy of functions

First-order functions

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined “before” the program is executed.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {{fun {x} x} 10}}
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

```
interp :: Expr Env -> ...
```

A taxonomy of functions

First-order functions

First-class functions

2 A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

2

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

First-class functions

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

```
{+ 5 {f
10}}
(list (fundef 'f 'x (id 'x)))
```

w/ def.

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.

```
{+ 5 {f 10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable <u>**in a designated portion of the code**</u>, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are <u>not</u> values.
- <u>Cannot be anonymous.</u>
- Only definable **<u>in a designated portion of the code</u>**, where they must be given names.
- They are defined "before" the program is executed.

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr listof[FunDef] Env -> number
```

A taxonomy of functions

First-order functions

- Functions are not values.
- Cannot be anonymous.
- Only definable **in a designated portion of the code**, where they must be given names.
- They are defined “before” the program is executed.

```
{+ 5 {f
10}}
```

w/ def.

```
(list (fundef 'f 'x (id 'x)))
```

```
interp :: Expr listof[FunDef] Env -> number
```

First-class functions

- Functions are values, with all the right of other values.
- Can be defined anonymously.
- Can be stored in any data structure.
- Can be returned as the result of a function.
- Can be passed as parameters to other functions.

```
{+ 5 {{fun {x} x} 10}}
```

```
interp :: Expr Env -> ...
```

Enriching our language with first-class functions

In the extended language, functions can be defined anywhere (within an expression), e.g.

Enriching our language with first-class functions

In the extended language, functions can be defined anywhere (within an expression), e.g.

```
{+ 5 {{fun {x} x} 10}}
```

— function position of an application

Enriching our language with first-class functions

In the extended language, functions can be defined anywhere (within an expression), e.g.

```
{+ 5 {{fun {x} x} 10}}
```

— function position of an application

```
{with {z {fun {x} x}} {+ 5 {z 10}}}
```

— named expression position in a local definition

Enriching our language with first-class functions

In the extended language, functions can be defined anywhere (within an expression), e.g.

```
{+ 5 {{fun {x} x} 10}}
```

— function position of an application

```
{with {z {fun {x} x}} {+ 5 {z 10}}}
```

— named expression position in a local definition

```
{with {y 3} {fun {x} {+ x y}}}
```

— body of a local definition

Enriching our language with first-class functions

In the extended language, functions can be defined anywhere (within an expression), e.g.

```
{+ 5 {{fun {x} x} 10}}
```

— function position of an application

```
{with {z {fun {x} x}} {+ 5 {z 10}}}
```

— named expression position in a local definition

```
{with {y 3} {fun {x} {+ x y}}}
```

— body of a local definition

```
{{ {fun {f} {fun {z} {f {f z}}}}  {fun {x} {+ x 1}} } 10}
```

(apply-twice) (add1)

Enriching our language with first-class functions

In the extended language, functions can be defined anywhere (within an expression), e.g.

```
{+ 5 {{fun {x} x} 10}}
```

— function position of an application

```
{with {z {fun {x} x}} {+ 5 {z 10}}}
```

— named expression position in a local definition

```
{with {y 3} {fun {x} {+ x y}}}
```

— body of a local definition

```
{{ {fun {f} {fun {z} {f {f z}}}}  {fun {x} {+ x 1}} } 10}
```

(apply-twice) (add1)

— actual parameter position of an application

Enriching our language with first-class functions

In the extended language, functions can be defined anywhere (within an expression), e.g.

```
{+ 5 {{fun {x} x} 10}}
```

— function position of an application

```
{with {z {fun {x} x}} {+ 5 {z 10}}}
```

— named expression position in a local definition

```
{with {y 3} {fun {x} {+ x y}}}
```

— body of a local definition

```
{{ {fun {f} {fun {z} {f {f z}}}}  {fun {x} {+ x 1}} } 10}
```

(apply-twice) (add1)

— actual parameter position of an application

— body of a function definition

Enriching our language with first-class functions — Syntax —

Under this extension, **local definitions are syntactic sugar**:

Enriching our language with first-class functions — Syntax —

Under this extension, **local definitions are syntactic sugar**:

```
{with {x 5} {+ x 8}}
```

Enriching our language with first-class functions — Syntax —

Under this extension, **local definitions are syntactic sugar**:

```
{with {x 5} {+ x 8}}
```

can always be re-written as

```
{{fun {x} {+ x 8}} 5}
```

Enriching our language with first-class functions — Syntax —

Under this extension, **local definitions are syntactic sugar**:

```
{with {x 5} {+ x 8}}
```

can always be re-written as

```
{{fun {x} {+ x 8}} 5}
```

We can then:

1. Remove local definitions from the abstract syntax, and
2. Accept them at the concrete syntax level by parsing them as a function application

Enriching our language with first-class functions — Abstract syntax —

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
         | (sub <expr> <expr>)
         | (if0 <expr> <expr> <expr>)
         | (id <sym>)
         | (fun <sym> <expr>)
         | (app <expr> <expr>)
|#
;; Inductive definition of arithmetic
;; expressions with first-class functions.
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Concrete syntax —

```
#|
<s-expr> ::= <num>
           | <sym>
           | (list '+ <s-expr> <s-expr>)
           | (list '- <s-expr> <s-expr>)
           | (list 'if0 <s-expr> <s-expr> <s-expr>)
           | (list 'with (list <sym> <s-expr>) <s-expr>)
           | (list 'fun (list <sym>) <s-expr>)
           | (list <s-expr> <s-expr>)
|#
;; parse :: s-expr -> Expr
;; Converts s-expressions into Exprs.
(define (parse s-expr)
  (match s-expr
    [(? number? n) (num n)]
    [(? symbol? x) (id x)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'if0 c t f) (if0 (parse c) (parse t) (parse f))]
    [(list 'with (list (? symbol? x) e) b)
     (app (fun x (parse b)) (parse e))]
    [(list 'fun (list x) b) (fun x (parse b))]
    [(list f a) (app (parse f) (parse a))]))
```

Enriching our language with first-class functions — Concrete syntax —

```
#|
<s-expr> ::= <num>
           | <sym>
           | (list '+ <s-expr> <s-expr>)
           | (list '- <s-expr> <s-expr>)
           | (list 'if0 <s-expr> <s-expr> <s-expr>)
           | (list 'with (list <sym> <s-expr>) <s-expr>)
           | (list 'fun (list <sym>) <s-expr>)
           | (list <s-expr> <s-expr>)
|#
;; parse :: s-expr -> Expr
;; Converts s-expressions into Exprs.
(define (parse s-expr)
  (match s-expr
    [(? number? n) (num n)]
    [(? symbol? x) (id x)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'if0 c t f) (if0 (parse c) (parse t) (parse f))]
    [(list 'with (list (? symbol? x) e) b)
     (app (fun x (parse b)) (parse e))]
    [(list 'fun (list x) b) (fun x (parse b))]
    [(list f a) (app (parse f) (parse a))]))
```

`with` expressions are translated as internal function applications

Enriching our language with first-class functions — Concrete syntax —

```
#|
<s-expr> ::= <num>
           | <sym>
           | (list '+ <s-expr> <s-expr>)
           | (list '- <s-expr> <s-expr>)
           | (list 'if0 <s-expr> <s-expr> <s-expr>)
           | (list 'with (list <sym> <s-expr>) <s-expr>)
           | (list 'fun (list <sym>) <s-expr>)
           | (list <s-expr> <s-expr>)
|#
;; parse :: s-expr -> Expr
;; Converts s-expressions into Exprs.
(define (parse s-expr)
  (match s-expr
    [(? number? n) (num n)]
    [(? symbol? x) (id x)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'if0 c t f) (if0 (parse c) (parse t) (parse f))]
    [(list 'with (list (? symbol? x) e) b)
     (app (fun x (parse b)) (parse e))]
    [(list 'fun (list x) b) (fun x (parse b))]
    [(list f a) (app (parse f) (parse a))]))
```

Function definition and application are parsed as shown here

Enriching our language with first-class functions — Semantics —

1. What are now values?

```
interp :: Expr Env -> ???
```

Enriching our language with first-class functions — Semantics —

1. What are now values?

```
interp :: Expr Env -> ???
```

```
{+ 5 {{fun {x} x} 10}}
```

Enriching our language with first-class functions — Semantics —

1. What are now values?

```
interp :: Expr Env -> ???
```

```
{+ 5 {{fun {x} x} 10}}          ⇝ 15
```

Enriching our language with first-class functions — Semantics —

1. What are now values?

```
interp :: Expr Env -> ???
```

```
{+ 5 {{fun {x} x} 10}}
```

$\rightsquigarrow$ `15` NUMBER

Enriching our language with first-class functions — Semantics —

1. What are now values?

```
interp :: Expr Env -> ???
```

```
{+ 5 {{fun {x} x} 10}}                    ⇝ 15
```

NUMBER

```
{with {y 3} {fun {x} {+ x y}}
```

Enriching our language with first-class functions — Semantics —

1. What are now values?

```
interp :: Expr Env -> ???
```

```
{+ 5 {{fun {x} x} 10}}            ⇝ 15                          NUMBER

{with {y 3} {fun {x} {+ x y}}}    ⇝ "{fun {x} {+ x 3}}"
```

Enriching our language with first-class functions — Semantics —

1. What are now values?

```
interp :: Expr Env -> ???
```

```
{+ 5 {{fun {x} x} 10}}                ⤳ 15                         NUMBER

{with {y 3} {fun {x} {+ x y}}}        ⤳ "{fun {x} {+ x 3}}"        FUNCTION
```

Enriching our language with first-class functions — Semantics —

1. What are now values?

```
interp :: Expr Env -> ???
```

`{+ 5 {{fun {x} x} 10}}` $\rightsquigarrow$ `15` ← NUMBER

`{with {y 3} {fun {x} {+ x y}}}` $\rightsquigarrow$ "`{fun {x} {+ x 3}}`" ← FUNCTION

Expressions now reduce to either numbers or functions…

In fact they reduce to something more complicated than plain "functions"….

Enriching our language with first-class functions — Semantics —

2. How do we now reduce expressions?

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Semantics —

2. How do we now reduce expressions?

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Semantics —

2. How do we now reduce expressions?

— The cases of **num**, **add**, **sub**, **if0** and **id** remain the same as before

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Semantics —

2. How do we now reduce expressions?

— The cases of **num**, **add**, **sub**, **if0** and **id** remain the same as before

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Semantics —

2. How do we now reduce expressions?

— The cases of **num**, **add**, **sub**, **if0** and **id** remain the same as before

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Isn’t it possible to reduce the function’s body?

Enriching our language with first-class functions — Semantics —

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

2. How do we now reduce expressions?

— The cases of `num`, `add`, `sub`, `if0` and `id` remain the same as before

— The interpretation of a function, i.e. `(fun arg body)`, is the same function

Isn’t it possible to reduce the function’s body?

Enriching our language with first-class functions — Semantics —

3. How do we now reduce expressions?

```
{+ 5 {{fun {x} x} 10}}
```

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Semantics —

3. How do we now reduce expressions?

```
{+ 5 {{fun {x} x} 10}}
```

— As seen before, we must substitute the formal parameter in the body for the argument.

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Semantics —

3. How do we now reduce expressions?

```
{+ 5 {{fun {x} x} 10}}
```

— As seen before, we must substitute the formal parameter in the body for the argument.

— Using deferred substitution, this means we have to extend an environment… **but which one?**

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Semantics —

3. How do we now reduce expressions?

```
{+ 5 {{fun {x} x} 10}}
```

— As seen before, we must substitute the formal parameter in the body for the argument.

— Using deferred substitution, this means we have to extend an environment… **but which one?**

— Extending an empty environment, ***we lose static scope***

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

Enriching our language with first-class functions — Semantics —

```
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (id x)
  (fun arg body)
  (app f arg))
```

3. How do we now reduce expressions?

```
{+ 5 {{fun {x} x} 10}}
```

— As seen before, we must substitute the formal parameter in the body for the argument.

— Using deferred substitution, this means we have to extend an environment… **but which one?**

- — Extending an empty environment, ***we lose static scope***
- — Extending the existing environment ***yields dynamic scope***

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

```
(app (fun 'x
         (app (fun 'f
                  (app (fun 'x
                           (add (id 'x) (app (id 'f) (num 4))))
                       (num 5)))
              (fun 'y
                   (add (id 'x) (id 'y)))))
         (num 3))
```

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

```
(app (fun 'x
          (app (fun 'f
                    (app (fun 'x
                              (add (id 'x) (app (id 'f) (num 4))))
                         (num 5)))
               (fun 'y
                    (add (id 'x) (id 'y)))))
          (num 3))
```

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

```
(app (fun 'x
          (app (fun 'f
                    (app (fun 'x
                              (add (id 'x) (app (id 'f) (num 4))))
                         (num 5)))
               (fun 'y
                    (add (id 'x) (id 'y)))))
          (num 3))
```

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

```
(app (fun 'x
          (app (fun 'f
                    (app (fun 'x
                              (add (id 'x) (app (id 'f) (num 4))))
                         (num 5)))
               (fun 'y
                    (add (id 'x) (id 'y)))))
          (num 3))
```

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
(app (fun 'x
          (app (fun 'f
                    (app (fun 'x
                              (add (id 'x) (app (id 'f) (num 4))))
                         (num 5)))
               (fun 'y
                    (add (id 'x) (id 'y)))))
          (num 3))
```

Initial environment is empty.

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
(app (fun 'x
 1      (app (fun 'f
              (app (fun 'x
                     (add (id 'x) (app (id 'f) (num 4))))
                   (num 5)))
          (fun 'y
              (add (id 'x) (id 'y)))))
        (num 3))
```

Initial environment is empty.

1 extend empty env with [x -> 3]

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
(app (fun 'x
 ①        (app (fun 'f
            ②         (app (fun 'x
                            (add (id 'x) (app (id 'f) (num 4))))
                         (num 5)))
               (fun 'y
                    (add (id 'x) (id 'y))))
          (num 3))
```

Initial environment is empty.

1 extend empty env with `[x -> 3]`

2 extend empty env with `[f -> fun 'y …]`

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
(app (fun 'x
 ❶       (app (fun 'f
           ❷       (app (fun 'x
                     ❸       (add (id 'x) (app (id 'f) (num 4))))
                           (num 5)))
              (fun 'y
                   (add (id 'x) (id 'y)))))
         (num 3))
```

Initial environment is empty.

1 extend empty env with `[x -> 3]`

2 extend empty env with `[f -> fun 'y …]`

3 extend empty env with `[x -> 5]`

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
(app (fun 'x
  ①      (app (fun 'f
           ②      (app (fun 'x
                    ③      (add (id 'x) (app (id 'f) (num 4))))
                        (num 5)))           ④
              (fun 'y
                   (add (id 'x) (id 'y))))
         (num 3))
```

Initial environment is empty.

1. extend empty env with `[x -> 3]`
2. extend empty env with `[f -> fun 'y …]`
3. extend empty env with `[x -> 5]`
4. interpret body with environment `[x -> 5]`

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
(app (fun 'x
 1      (app (fun 'f
          2     (app (fun 'x                       5
                  3     (add (id 'x) (app (id 'f) (num 4))))
                      (num 5)))               4
             (fun 'y
                  (add (id 'x) (id 'y))))
        (num 3))
```

Initial environment is empty.

1. extend empty env with `[x -> 3]`
2. extend empty env with `[f -> fun 'y …]`
3. extend empty env with `[x -> 5]`
4. interpret body with environment `[x -> 5]`
5. Lookup for **`f`** fails.

Enriching our language with first-class functions — Semantics —

— Extending the existing environment ***yields dynamic scope***

```
(app (fun 'x
          (app (fun 'f
                    (app (fun 'x
                              (add (id 'x) (app (id 'f) (num 4))))
                         (num 5)))
               (fun 'y
                    (add (id 'x) (id 'y)))))
          (num 3))
```

Initial environment is empty.

Enriching our language with first-class functions — Semantics —

— Extending the existing environment ***yields dynamic scope***

```
(app (fun 'x
 1        (app (fun 'f
                (app (fun 'x
                          (add (id 'x) (app (id 'f) (num 4))))
                     (num 5)))
               (fun 'y
                    (add (id 'x) (id 'y))))
          (num 3))
```

Initial environment is empty.

1 extend empty env with [x -> 3]

Enriching our language with first-class functions — Semantics —

— Extending the existing environment ***yields dynamic scope***

```
(app (fun 'x
  ❶      (app (fun 'f
           ❷      (app (fun 'x
                         (add (id 'x) (app (id 'f) (num 4))))
                       (num 5)))
              (fun 'y
                   (add (id 'x) (id 'y)))))
         (num 3))
```

Initial environment is empty.

1 extend empty env with `[x -> 3]`

2 extend existing env: `[f -> fun 'y …; x -> 3]`

Enriching our language with first-class functions — Semantics —

— Extending the existing environment ***yields dynamic scope***

```
(app (fun 'x
 1        (app (fun 'f
            2        (app (fun 'x
                       3        (add (id 'x) (app (id 'f) (num 4))))
                             (num 5)))
                (fun 'y
                     (add (id 'x) (id 'y))))
          (num 3))
```

Initial environment is empty.

1 extend empty env with `[x -> 3]`

2 extend existing env: `[f -> fun 'y …; x -> 3]`

3 extend existing env: `[x -> 5; f -> fun 'y …; x -> 3]`

Enriching our language with first-class functions — Semantics —

— Extending the existing environment ***yields dynamic scope***

```
(app (fun 'x
   1      (app (fun 'f
             2      (app (fun 'x
                       3        (add (id 'x) (app (id 'f) (num 4))))
                             (num 5)))            4
                (fun 'y
                     (add (id 'x) (id 'y))))
          (num 3))
```

Initial environment is empty.

1. extend empty env with `[x -> 3]`
2. extend existing env: `[f -> fun ‘y …; x -> 3]`
3. extend existing env: `[x -> 5; f -> fun ‘y …; x -> 3]`
4. interpret body with environment `[x -> 5; f -> fun ‘y …; x -> 3]`

Enriching our language with first-class functions — Semantics —

— Extending the existing environment ***yields dynamic scope***

```
(app (fun 'x
 ❶       (app (fun 'f
          ❷       (app (fun 'x                                    ❺
                   ❸       (add (id 'x) (app (id 'f) (num 4))))
                       (num 5)))              ❹
              (fun 'y
                   (add (id 'x) (id 'y))))
         (num 3))
```

Initial environment is empty.

1. extend empty env with `[x -> 3]`
2. extend existing env: `[f -> fun ‘y …; x -> 3]`
3. extend existing env: `[x -> 5; f -> fun ‘y …; x -> 3]`
4. interpret body with environment `[x -> 5; f -> fun ‘y …; x -> 3]`
5. Lookup for `f` succeeds, but evaluates x as 5 — yielding dynamic scope

Enriching our language with first-class functions — Semantics —

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

$\leadsto$ ***Lookup fail***

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

$\rightsquigarrow$ ***Lookup fail***

— Extending the existing environment ***yields dynamic scope***

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

⤳ ***Lookup fail***

— Extending the existing environment ***yields dynamic scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

⤳ `{+ 5 {+ 5 4}}`

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

$\leadsto$ ***Lookup fail***

— Extending the existing environment ***yields dynamic scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

$\leadsto$ `{+ 5 {+ 5 4}}`

— We need to keep static scope

Enriching our language with first-class functions — Semantics —

— Extending an empty environment, ***we lose static scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

⤳ ***Lookup fail***

— Extending the existing environment ***yields dynamic scope***

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

⤳ `{+ 5 {+ 5 4}}`

— We need to keep static scope

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

⤳ `{+ 5 {+ 3 4}}`

Enriching our language with first-class functions — Semantics —

— Key Insight: Static scoping binds an identifier with the closest binding occurrence, in the textual representation of the source code.

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

$\rightsquigarrow$ `{+ 5 {+ 3 4}}`

Enriching our language with first-class functions — Semantics —

— Key Insight: Static scoping binds an identifier with the closest binding occurrence, in the textual representation of the source code.

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

$\leadsto$ `{+ 5 {+ 3 4}}`

Enriching our language with first-class functions — Semantics —

— Key Insight: Static scoping binds an identifier with the closest binding occurrence, in the textual representation of the source code.

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

⇝ `{+ 5 {+ 3 4}}`

— To keep static scope, whenever we find a function definition we must “wrap” the function definition with the environment at that point during interpretation. This correctly records the pending substitutions at that moment.

Enriching our language with first-class functions — Semantics —

— Key Insight: Static scoping binds an identifier with the closest binding occurrence, in the textual representation of the source code.

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {+ x {f 4}}}}}
```

⇝ `{+ 5 {+ 3 4}}`

— To keep static scope, whenever we find a function definition we must “wrap” the function definition with the environment at that point during interpretation. This correctly records the pending substitutions at that moment.

— The structure that binds together a function definition and its environment at definition time is known as a **closure**.

Enriching our language with first-class functions — Semantics —

Enriching our language with first-class functions — Semantics —

function definition + environment at the definition location = *closure* of the function

(pending substitutions when the function is being defined)

Enriching our language with first-class functions — Semantics —

```
;; Values of expressions
;; <value> ::= (numV <number>)
;;           | (closureV <sym> <expr> <env>)
(deftype Value
  (numV n)
  (closureV arg body env))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n)              ]

    [(fun id body)                      ]

    [(id x)                        ]

    [(add l r)                                   ]

    [(sub l r)                                   ]

    [(if0 c t f)



    [(app f e)

                                         )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n)           ]

    [(fun id body)                      ]

    [(id x)                   ]

    [(add l r)                                    ]

    [(sub l r)                                    ]

    [(if0 c t f)


    [(app f e)
                                   )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body)                      ]

    [(id x)                   ]

    [(add l r)                                    ]

    [(sub l r)                                    ]

    [(if0 c t f)



    [(app f e)

                                  )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x)                    ]

    [(add l r)                                    ]

    [(sub l r)                                    ]

    [(if0 c t f)



    [(app f e)

                                  )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x) (env-lookup x env)]

    [(add l r)                                    ]

    [(sub l r)                                    ]

    [(if0 c t f)


    [(app f e)

                                        )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x) (env-lookup x env)]

    [(add l r) (num+ (interp l env) (interp r env))]

    [(sub l r)                                    ]

    [(if0 c t f)



    [(app f e)

                                        )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x) (env-lookup x env)]

    [(add l r) (num+ (interp l env) (interp r env))]

    [(sub l r) (num- (interp l env) (interp r env))]

    [(if0 c t f)


    [(app f e)

                                                )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x) (env-lookup x env)]

    [(add l r) (num+ (interp l env) (interp r env))]

    [(sub l r) (num- (interp l env) (interp r env))]

    [(if0 c t f) (if (num-zero? (interp c env))
                     (interp t env)
                     (interp f env))]

    [(app f e)
                                  )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x) (env-lookup x env)]

    [(add l r) (num+ (interp l env) (interp r env))]

    [(sub l r) (num- (interp l env) (interp r env))]

    [(if0 c t f) (if (num-zero? (interp c env))
                     (interp t env)
                     (interp f env))]

    [(app f e) (def (closureV the-arg the-body closed-env) (interp f env))

                                              )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x) (env-lookup x env)]

    [(add l r) (num+ (interp l env) (interp r env))]

    [(sub l r) (num- (interp l env) (interp r env))]

    [(if0 c t f) (if (num-zero? (interp c env))
                     (interp t env)
                     (interp f env))]

    [(app f e) (def (closureV the-arg the-body closed-env) (interp f env))
               (def new-env (extend-env the-arg (interp e env) closed-env))
                                                 )
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x) (env-lookup x env)]

    [(add l r) (num+ (interp l env) (interp r env))]

    [(sub l r) (num- (interp l env) (interp r env))]

    [(if0 c t f) (if (num-zero? (interp c env))
                     (interp t env)
                     (interp f env))]

    [(app f e) (def (closureV the-arg the-body closed-env) (interp f env))
               (def new-env (extend-env the-arg (interp e env) closed-env))
               (interp the-body new-env)]))
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Enriching our language with first-class functions — Semantics —

```
;; interp :: Expr Env -> Value
;; Evaluates an expression in
;; a given environment, using static scoping.
(define (interp expr env)
  (match expr

    [(num n) (numV n)]

    [(fun id body) (closureV id body env)]

    [(id x) (env-lookup x env)]

    [(add l r) (num+ (interp l env) (interp r env))]

    [(sub l r) (num- (interp l env) (interp r env))]

    [(if0 c t f) (if (num-zero? (interp c env))
                     (interp t env)
                     (interp f env))]

    [(app f e) (def (closureV the-arg the-body closed-env) (interp f env))
               (def new-env (extend-env the-arg (interp e env) closed-env))
               (interp the-body new-env)]))
```

```
(define (num+ n1 n2)
  (def (numV v1) n1)
  (def (numV v2) n2)
  (numV (+ v1 v2)))

(define (num-zero? n)
  (def (numV v) n)
  (zero? v))
```

Changing this to env yields an interpreter with dynamic scoping

Takeaway

Takeaway

Interpreting a language with both first class functions and static scoping requires the use of function closures (which wrap functions up with the deferred substitutions at their definition location)

Takeaway

Bibliography

- Programming Languages: Application and Interpretation (1st Edition)
  Shriram Krishnamurthi [Download]
  Chapter 6

Source code

- Arithmetical language with first class functions [Download]