Functions

Nate Foster

Spring 2019

Far Above Cayuga's Waters (Dixieland Ramblers)

Review

Previously in 3110:

- **Syntax and semantics**
- **Expressions:** if, let
- **Definitions:** let

Today:

- **Functions**

ANONYMOUS FUNCTION EXPRESSIONS & FUNCTION APPLICATION EXPRESSIONS

Dem
o

Anonymous function expression

Syntax: **`fun x1 ... xn -> e`**

`fun` is a keyword :)

Evaluation:

- A function is a value: no further computation to do
- In particular, body **`e`** is not evaluated until function is applied

Lambda

- Anonymous functions a.k.a. *lambda expressions*
- Math notation: λx . e
- The lambda means "what follows is an anonymous function"

Lambda

- Python
- Java 8
- A popular PL blog
- Lambda style

Functions are values

Can use them **anywhere** we use values:

- Functions can **take** functions as arguments
- Functions can **return** functions as results

This is an incredibly powerful language feature!

Function application

Syntax: `e0 e1 ... en`

No parentheses required!

(unless you need to force particular order of evaluation)

Function application

Evaluation of `e0 e1 ... en`:

1. Evaluate **`e0...en`** to values **`v0...vn`**
2. Type checking will ensure that **`v0`** is a function **`fun x1 ... xn -> e`**
3. Substitute **`vi`** for **`xi`** in **`e`** yielding new expression **`e'`**
4. Evaluate **`e'`** to a value **`v`**, which is result

Example

Let vs. function

These two expressions are **syntactically different** but **semantically equivalent**:

```
let x = 2 in x+1
```

```
(fun x -> x+1) 2
```

FUNCTION DEFINITIONS

Dem
o

Two ways to define functions

These definitions are **syntactically different** but **semantically equivalent**:

```
let inc = fun x -> x+1
```

```
let inc x = x + 1
```

Fundamentally no difference from `let` definitions we saw before

Recursive function definition

Must explicitly state that function is recursive:

```
let rec f ...
```

Dem
o

Reverse application

- Instead of **`f e`** can write **`e |> f`**
- Use: pipeline a value through several functions
  `5 |> inc |> square (* ==> 36*)`

assuming

```
let inc x = x + 1
let square x = x * x
```

FUNCTIONS AND TYPES

Function types

Type `t -> u` is the type of a function that takes input of type **`t`** and returns output of type **`u`**

Type `t1 -> t2 -> u` is the type of a function that takes input of type **`t1`** and another input of type **`t2`** and returns output of type **`u`**

etc.

Note dual purpose for **`->`** syntax:

- Function types
- Function values

Function application

Type checking:

```
If   e0 : t1 -> ... -> tn -> u
And  e1 : t1,
     ...,
     en : tn

Then e0 e1 ... en : u
```

Anonymous function expression

Type checking:

If `x1:t1, ..., xn:tn`

And `e:u`

Then `(fun x1 ... xn -> e) :`

`t1 -> ... -> tn -> u`

PARTIAL APPLICATION

Dem

o

More syntactic sugar

Multi-argument functions do not exist

```
fun x y -> e
```

is syntactic sugar for

```
fun x -> (fun y -> e)
```

More syntactic sugar

Multi-argument functions do not exist

```
fun x y z -> e
```

is syntactic sugar for

```
fun x -> (fun y -> (fun z -> e))
```

More syntactic sugar

Multi-argument functions do not exist

```
let add x y = x + y
```

is syntactic sugar for

```
let add = fun x ->
            fun y ->
              x + y
```

Again: Functions are values

Can use them **anywhere** we use values:

- Functions can **take** functions as arguments
- Functions can **return** functions as results

This is an incredibly powerful language feature!

Upcoming events

- [Tomorrow] A0 released by end of day

This is **`fun`***!*

THIS IS 3110