Functions

Prof. Clarkson

Fall 2018

Today's music: *Expression* by Salt-N-Pepa
Rather than repeat *Function* by E-40 (Clean remix)
ACSU
Attendance Question

What does ACSU stand for?

A. Association of Computer Science Undergraduates
B. Add-Compare-Select Unit
C. Advanced Camel Support Usergroup
D. All Cool Students Unify

First question of day worth the most points. Participation counts, not correctness.
let expressions [Corrected]

let \( x = e_1 \) in \( e_2 \)

Type-checking:

If \( e_1 : t_1 \) and \( x : t_1 \) and \( e_2 : t_2 \)
then \((let \ x = e_1 \ in \ e_2) : t_2\)
Review

Previously in 3110:
• Syntax and semantics
• Expressions: if, let
• Definitions: let

Today:
• Functions
ANONYMOUS FUNCTION EXPRESSIONS & FUNCTION APPLICATION EXPRESSIONS
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: \( \lambda x \ . \ e \)
- 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 $e_0 \ e_1 \ldots \ e_n$:

1. Evaluate $e_0\ldots e_n$ to values $v_0\ldots v_n$
2. Type checking will ensure that $v_0$ is a function $\text{fun } x_1 \ldots x_n \rightarrow e$
3. Substitute $v_i$ for $x_i$ in $e$ yielding new expression $e'$
4. Evaluate $e'$ to a value $v$, which is result
Let vs. function

These two expressions are syntactically different but semantically equivalent:

\[
\text{let } x = 2 \text{ in } x + 1 \\
(\text{fun } x \rightarrow x + 1) \ 2
\]
FUNCTION DEFINITIONS
Two syntaxes to define functions

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

\[
\text{let } \text{inc } = \text{ fun } \ x \ -\to \ x+1
\]

\[
\text{let } \text{inc } \ x \ = \ x \ + \ 1
\]

Fundamentally no difference from \textit{let} definitions we saw before
Recursive function definition

Must explicitly state that function is recursive:

\texttt{let rec f ...}
Reverse application

- Instead of $f \ e$ can write $e \ |> \ f$
- Use: pipeline a value through several functions

5 |> inc |> square (* ==> 36*)

assuming

```ml
let inc x = x + 1
let square x = x * x
```
FUNCTIONS AND TYPES
Function types

Type \( t \rightarrow u \) is the type of a function that takes input of type \( t \) and returns output of type \( u \).

Type \( t_1 \rightarrow t_2 \rightarrow u \) is the type of a function that takes input of type \( t_1 \) and another input of type \( t_2 \) and returns output of type \( u \).

etc.

Note dual purpose for \( \rightarrow \) syntax:
• Function types
• Function values
Function application

Type checking:

If $e_0 : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u$
And $e_1 : t_1$
\[\ldots,\]
$e_n : t_n$

Then $e_0 \ e_1 \ \ldots \ \ e_n : u$
Anonymous function expression

Type checking:

If \( x_1 : t_1, \ldots, x_n : t_n \)
And \( e : u \)
Then \((\text{fun} \ x_1 \ \ldots \ x_n \ -> \ e) : \quad t_1 \ -> \ \ldots \ -> \ t_n \ -> \ u\)
PARTIAL APPLICATION
More syntactic sugar

Multi-argument functions do not exist

\[
\text{fun } x \ y \ \rightarrow \ e
\]

is syntactic sugar for

\[
\text{fun } x \ \rightarrow \ (\text{fun } y \ \rightarrow \ e)
\]
More syntactic sugar

Multi-argument functions do not exist

\[
\text{fun } x \ y \ z \rightarrow e
\]

is syntactic sugar for

\[
\text{fun } x \rightarrow (\text{fun } y \rightarrow (\text{fun } z \rightarrow e))
\]
More syntactic sugar

Multi-argument functions do not exist

```ml
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

• [today] A0 released by end of day
• [Mon] Labor Day:
  • No discussion sections Monday
    – Therefore Tuesday sections (but not lecture) also canceled
  • No consulting hours on Monday

This is fun!

This is 3110