Variables and Scope

Prof. Clarkson
Summer 2016

Today's music: "Let It Go" from Frozen
Review

Previously in 3110:
• Aspects of a PL: syntax, semantics, idioms, libraries, tools
• Expressions and values
• if expressions
• Functions: definitions, application, recursive, anonymous, higher-order, operators, labeled and optional arguments

Today:
• let expressions
• scope: expressions, modules
LET EXPRESSIONS
let expressions

Example:

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

• Like a function definition with `let`, first part binds a value to a name

• But second part (`in`) uses that name in a particular `scope`
**let expressions**

Syntax:

\[
\text{let } x = e_1 \text{ in } e_2
\]

- \(x\) is an *identifier*, must begin with lowercase letter
- \(e_1\) is the *binding expression*
- \(e_2\) is the *body expression*

\[
\text{let } x = e_1 \text{ in } e_2 \text{ is itself an expression}
\]

- \(x = e_1\) is a *binding*

E.g.

\[
\text{let } x = 2 \text{ in } x+x
\]
let expressions

e.g. `let inc x = x+1 in inc 10`

• syntactically looks a little different than
  `let x = e1 in e2`

• but recall is equivalent to
  `let inc = fun x -> x+1 in inc 10`


e.g. `let y = "big" in (let z = "red" in y^z)`

• *nested* let expression

• `e2` is allowed to be any expression!


let expressions

\[ \text{let } x = e1 \text{ in } e2 \]

Evaluation:

- Evaluate \( e1 \) to a value \( v1 \)
- Substitute \( v1 \) for \( x \) in \( e2 \), yielding a new expression \( e2' \)
- Evaluate \( e2' \) to \( v \)
- Result of evaluation is \( v \)
let expressions

let x = e1 in e2

Type-checking:

If \( e_1 : t_1 \),
and if \( e_2 : t_2 \)  (assuming that \( x : t_1 \)),
then \( (\text{let } x = e_1 \text{ in } e_2) : t_2 \)
Let expressions

```plaintext
let x = 1+4 in x*3

--> Evaluate e1 to a value v1

let x = 5 in x*3

--> Substitute v1 for x in e2, yielding a new expression e2'

5*3

--> Evaluate e2' to v

15

Result of evaluation is v
Anonymous functions

These two expressions are syntactically different but semantically equivalent:

```plaintext
let x = 7 in x+1
(fun x -> x+1) 7
```

• We really don't need `let` expressions!
• Could do everything with anonymous functions and application
• But `let` is convenient
Let expressions in toplevel

Syntax:

```
let x = e
```

Implicitly, “in rest of what you type”

E.g., you type:  
```
let a="big";;  
let b="red";;  
let c=a^b;;  
```

...toplevel understands as
```
let a="big" in  
let b="red" in  
let c=a^b in...
```
SCOPE: EXPRESSIONS
Scope

Bindings are in effect only in the scope (the “block”) in which they occur.

```ocaml
let x=42 in
  (* y is not in scope here *)
x + (let y="3110" in
    (* y is in scope here *)
    int_of_string y)
```

Exactly what you’re used to from (e.g.) Java
Overlapping scope

Overlapping bindings of the same name is usually bad idiom (and darn confusing)

```
let x = 5 in ((let x = 6 in x) + x)
```
Question

```
let x = 5 in ((let x = 6 in x) + x)
```

To what value does the above expression evaluate?

A. 10 
B. 11 
C. 12 
D. None of the above
How to substitute

```
let x = 5 in ((let x = 6 in x) + x)
```

-->

```???
```

Not a choice: (why? semantics says substitute outer `let` binding first)

```
let x = 5 in (6 + 6)
```

Two choices:

i. `((let x = 6 in x) + 5)`

ii. `((let x = 6 in 5) + 5)`
How to substitute

\[
\text{let } x = 5 \text{ in } ((\text{let } x = 6 \text{ in } x) + x)
\]

\[
\longrightarrow
\]

\[
???
\]

Not a choice:

\[
\text{let } x = 5 \text{ in } (6 + 6)
\]

Two choices:

i. \((\text{let } x = 6 \text{ in } x) + 5)\)

ii. \((\text{let } x = 6 \text{ in } 5) + 5)\)

Why?
Principle of Name Irrelevance

The name of a variable should not matter.

In math, these are the same functions:
\[ f(x) = x^2 \]
\[ f(y) = y^2 \]

So in programming, these should be the same functions:
```plaintext
let f x = x*x
let f y = y*y
```

This principle is also called *alpha equivalence*
Principle of Name Irrelevance

Likewise, these should be the same expressions:

(\texttt{let } x = 6 \texttt{ in } x)
(\texttt{let } y = 6 \texttt{ in } y)

So these should also be the same:

\texttt{let } x = 5 \texttt{ in } ((\texttt{let } x = 6 \texttt{ in } x) + x)
\texttt{let } x = 5 \texttt{ in } ((\texttt{let } y = 6 \texttt{ in } y) + x)

But if we substitute inside inner \texttt{let} expression, they will not be the same:

(\texttt{let } x = 6 \texttt{ in } 5) + 5 \implies 10
(\texttt{let } y = 6 \texttt{ in } y) + 5 \implies 11
Back to substitution

\[
\text{let } x = 5 \text{ in } ((\text{let } x = 6 \text{ in } x) + x)
\]

-->

???

Not a choice:
\[
\text{let } x = 5 \text{ in } (6 + 6)
\]

Two choices:

A. \((\text{let } x = 6 \text{ in } x) + 5\)

B. \(((\text{let } x = 6 \text{ in } 5) + 5\) Name irrelevance is why!
Shadowing

A new binding *shadows* an older binding of the same name

\[\text{let } x = 5 \text{ in } ((\text{let } x = 6 \text{ in } x) + x)\]

Think of the second binding as a binding of an *entirely different variable* that just happens to have the same name as the old variable
Shadowing

A new binding *shadows* an older binding of the same name

```
let x = 5 in ((let x = 6 in x) + x)
```

Think of the second binding as a binding of an *entirely different variable* that just happens to have the same name as the old variable
Shadowing is not assignment

```
let x = 5 in ((let x = 6 in x) + x)
-----> 11
```

```
let x = 5 in (x + (let x = 6 in x))
-----> 11
```
Shadowing is not assignment

Q: So how is this not assignment?!

# let x=42;;
val x : int = 42
# let x=22;;
val x : int = 22

A: The second let binds an entirely different variable that just happens to have the same name
**Shadowing is not assignment**

```ocaml
# let x=42;;
val x : int = 42
# let f y = x+y;;
val f : int -> int = <fun>
# f 0;;
- : int = 42
# let x=22;;
# let x=22;;
val x : int = 22
# f 0;;
- : int = 42  x did not mutate!
```
Shadowing is not assignment

First: recall it's one big, nested let

let x=42 in
let f y = x+y in
let x=22 in
f 0
Shadowing is not assignment

First: recall it's one big, nested let

let x=42 in
  let f y = x+y in
  let x=22 in
    f 0
Shadowing is not assignment

**Second:** recall semantics and substitution

```plaintext
let f y = 42+y in
  let x=22 in
    f 0
```
Shadowing is not assignment

Also: recall name irrelevance

\[
\begin{align*}
\text{let } & f \ y = 42 + y \text{ in} \\
\text{let } & z = 22 \text{ in} \\
& f \ 0
\end{align*}
\]
Shadowing is not assignment

What have we learned?
• Each `let` binding binds an entirely new variable
• If that new variable happens to have the same name as an old variable, the new variable temporarily shadows the old
• But the old variable is still around
• And its value is immutable: never ever changes

Bottom line: `let` expressions look superficially like assignment statements from imperative languages, but are actually quite different
let expressions (summary)

• Syntax:
  
  \texttt{let } x = e_1 \texttt{ in } e_2

• Type-checking:
  
  If \( e_1 : t_1 \), and if \( e_2 : t_2 \) under the assumption that \( x : t_1 \), then \( \texttt{let } x = e_1 \texttt{ in } e_2 : t_2 \)

• Evaluation:
  
  – Evaluate \( e_1 \) to \( v_1 \)
  – Substitute \( v_1 \) for \( x \) in \( e_2 \) yielding new expression \( e_2' \)
  – Evaluate \( e_2' \) to \( v \)
  – Result of evaluation is \( v \)