Variables and Scope

Prof. Clarkson
Fall 2015

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

Previously in 3110:
• Aspects of a PL: syntax, semantics, idioms, libraries, tools
• if expressions
• Functions (recursive, anonymous)
• Lab: higher-order functions, operators, labeled arguments

Today:
• let expressions
• scope: in expressions, in modules
• logistics of course assignments
LET EXPRESSIONS
let expressions

Syntax:

    let x = e1 in e2

x is an identifier

e1 is the binding expression

e2 is the body expression

let x = e1 in e2 is itself an expression

x = e1 is a binding

e.g.

let x = 2 in x+x
let inc x = x+1 in inc 10
let y = "catch" in (let z = "22" in y^z)
**let expressions**

```
let x = e1 in e2
```

**Evaluation:**

- Evaluate \( e_1 \) to a value \( v_1 \)
- Substitute \( v_1 \) for \( x \) in \( e_2 \), yielding a new expression \( e_2' \)
- Evaluate \( e_2' \) to \( v \)
- Result of evaluation is \( v \)
let expressions

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

Type-checking:

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

\[
\text{let } x = 1+4 \text{ in } x*3
\]

\[\rightarrow\] Evaluate e1 to a value v1

\[
\text{let } x = 5 \text{ in } x*3
\]

\[\rightarrow\] Substitute v1 for x in e2, yielding a new expression e2'

\[
5*3
\]

\[\rightarrow\] 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
```
Let expressions in REPL

Syntax:

```
let x = e
```

Implicitly, “in rest of what you type”

E.g., you type:

```
let a="catch";;
let b="22";;
let c=a^b;;
```

OCaml understands as

```
let a="catch" in
let b="22" in
let c=a^b in...
```
SCOPE: EXPRESSIONS
Scope

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

```plaintext
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)

\[
\text{let } x = 5 \text{ in } ((\text{let } x = 6 \text{ in } x) + x)
\]
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 to evaluate binding expr 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

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

-->

```plaintext
???
```

Not a choice:

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

Two choices:

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

ii. ((let x = 6 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:

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

So these should also also be the same:

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

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

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

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

\[
\rightarrow
\]

???

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

```latex
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.
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

\[
\text{let } x = 5 \text{ in } ((\text{let } x = 6 \text{ in } x) + x) \\
\text{-----> 11}
\]

\[
\text{let } x = 5 \text{ in } (x + (\text{let } x = 6 \text{ in } x)) \\
\text{-----> 11}
\]
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
```

(@13 on Piazza)

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;;
val x : int = 22
# f 0;;
- : int = 42  x did not mutate!
```
Shadowing is not assignment

First: recall it's one big 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 \texttt{let}

\begin{verbatim}
let \texttt{x=42} in
  let \texttt{f y = x+y} in
    let \texttt{x=22} in
      \texttt{f 0}
\end{verbatim}
Shadowing is not assignment

Second: recall semantics

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

**Third:** recall substitution and name irrelevance

\[
\text{let } f \ y = 42 + y \ \text{in} \\
\text{let } z = 22 \ \text{in} \\
f \ 0
\]
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:
  `let x = e1 in e2`

• Type-checking:
  If `e1 : t1`, and if `e2 : t2` under the assumption that `x : t1`, then `let x = e1 in e2 : t2`

• Evaluation:
  – Evaluate `e1` to `v1`
  – Substitute `v1` for `x` in `e2` yielding new expression `e2'`
  – Evaluate `e2'` to `v`
  – Result of evaluation is `v`
SCOPE: MODULES
Modules

- A **module** is a namespace with a collection of definitions
- E.g., the **Char module**, the **Random module**
- Provides another kind of scoping: name is bound inside the module but not outside
- Access the names with dot notation, like in many languages, e.g., **Char.lowercase**
- Modules can do much more, but that will have to wait...
ASSIGNMENT LOGISTICS
A1

- Out now on course website, due in about 9 days
- Implement the Enigma encryption machine from WWII
- Needs lists: will see in lecture on Thursday
- Suggestion: in the next two days, write no code; just figure out the Enigma cipher algorithm
A1

• Individual assignment: work on your own
• (Re)read the CS 1110 Explanation of Academic Integrity:
  – Fraudulently represent someone else’s work as your own: punishable under AI code
  – Truthfully document that what you are submitted is mostly not your own work: grade penalty, but not an AI violation
  – Collaborate lightly with other students, getting an idea or two from them, and documenting that, but never actually designing/writing code with them: fine!!!
• Soft deadline and hard deadline:
  – Submit before soft deadline, you're great!
  – Submit between soft deadline and hard deadline, 25% penalty
    • perfectly fine to do this once or twice a semester as a way to get a little extra time to finish assignment
    • doing it for every assignment will probably reduce your final course grade by about 1 letter grade
  – No submissions after hard deadline
• Note: no grace period in CMS. (Otherwise you'd never be able to tell whether 1-minute late submission was before or after the soft deadline)
Upcoming events

• [this week] No Clarkson office hours: at FOSAD (Foundations of Security Analysis and Design)
• [Thursday] guest lecture by Dean of CIS, Greg Morrisett

This is in scope

THIS IS 3110