Let Expressions

Speaker: David Walker
COS 326
Princeton University

What is the single most important mathematical concept ever developed in human history?

What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable

Why is the mathematical variable so important?

The mathematician says:

“Let x be some integer, we define a polynomial over x ...”

Why is the mathematical variable so important?

The mathematician says:

"Let x be some integer, we define a polynomial over x ..."

What is going on here? The mathematician has separated a *definition* (of x) from its *use* (in the polynomial).

This is the most primitive kind of *abstraction* (x is *some* integer)

Why is the mathematical variable so important?

Abstraction is the key to controlling complexity and without it, modern mathematics, science, and computation would not exist.

Abstraction allows for *reuse* of ideas, values, theorems ...

… functions and programs!

OCAML BASICS:
LET DECLARATIONS

Basic abstraction in OCaml

In OCaml, the most basic technique for factoring your code is to use let expressions

Instead of writing this expression:

```
(2 + 3) * (2 + 3)
```

Abstraction & Abbreviation

In OCaml, the most basic technique for factoring your code is to use let expressions

Instead of writing this expression:

```
(2 + 3) * (2 + 3)
```

We write this one:

```
let x = 2 + 3 in
x * x
```

A Few More Let Expressions

```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
```

A Few More Let Expressions

```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
```

```
let a = "a" in
let b = "b" in
let as = a ^ a ^ a in
let bs = b ^ b ^ b in
as ^ bs
```

A Technical Note: The Structure of a .ml File

Foo.ml

```
<declaration>

<declaration>

      …
```

Every .ml file is a sequence of *declarations*

These “declarations” are a little different than “expressions”

A Technical Note: The Structure of a .ml File

Bar.ml

```
let x = 17 + 5

let y = x + 22
```

Bar.ml contains two *let declarations*

Let declarations do not end with “in”

Let declarations have the form:

let *<var> = <expression>*

A Technical Note: The Structure of a .ml File

Baz.ml

```
let x =
  let z = 22 in
  z + z

let y =
  if x < 17 then
    let w = x + 1 in
    2 * w
  else
    26
```

Because let declarations have this form:

let *<var>* = *<expression>*

they contain expressions

... including “let expressions” which have the form:

let *<var>* = *<expression>* in *<expression>*

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for *x*, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*.

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for *x*, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*.

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for *x*, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*.

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x
```

It does not matter what I write next. add_three will always add 3!

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x

let x = 4

let add_four (y:int) : int = y + x
```

a distinct variable that "happens to be spelled the same"

OCaml Variables are Immutable

A use of a variable always refers to it's *closest* (in terms of syntactic distance) enclosing declaration. Hence, we say OCaml is a *statically scoped* (or *lexically scoped*) language

```
let x = 3

let add_three (y:int) : int = y + x

let x = 4

let add_four (y:int) : int = y + x

let add_seven (y:int) : int =
  add_three (add_four y)
```

we can use add_three without worrying about the second definition of x

OCaml Variables are Immutable

Since the two variables (both happened to be named x) are actually different, unconnected things, we can rename them.

This is known as *alpha-conversion*.

```
let x = 3

let add_three (y:int) : int = y + x

let x = 4

let add_four (y:int) : int = y + x

let add_seven (y:int) : int =
  add_three (add_four y)
```

you can rename
x to zzz
by replacing
the definition
and all its uses with
the new name

OCaml Variables are Immutable

Since the two variables (both happened to be named x) are actually different, unconnected things, we can rename them.

This is known as *alpha-conversion*.

```
let x = 3

let add_three (y:int) : int = y + x

let zzz = 4

let add_four (y:int) : int = y + zzz

let add_seven (y:int) : int =
  add_three (add_four y)
```

you can rename
x to zzz
by replacing
the definition
and all its uses with
the new name

How does OCaml execute a let expression?

```
let x = <expression1> in
<expression2>
```

In a nutshell:

- execute <expression1>, until you get a value v1
- substitute that value v1 for x in <expression2>
- execute <expression2>, until you get a value v2
- the result of the whole execution is v2

How does OCaml execute a let expression?

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

How does OCaml execute a let expression?

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

-->

```
let x = 3 in x * x
```

How does OCaml execute a let expression?

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

-->

```
let x = 3 in x * x
```

substitute
3 for x

-->

```
3 * 3
```

How does OCaml execute a let expression?

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

-->

```
let x = 3 in x * x
```

substitute 3 for x

-->

```
        3 * 3
```

-->

```
        9
```

How does OCaml execute a let expression?

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

-->

let x = 3 in x * x

-->

      3 * 3

-->

      9
```

substitute 3 for x

Note: I write e1 --> e2 when e1 evaluates to e2 in one step

Meta-comment

OCaml expression → `let x = 2 in x + 3`

OCaml expression → `2 + 3`

```
let x = 2 in x + 3      -->      2 + 3
```

I defined the language in terms of itself:
By reduction of one OCaml expression to another

I'm trying to train you to think at a high level of abstraction.

I didn't have to mention low-level abstractions like assembly code or registers or memory layout to tell you how OCaml works.

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4     in
y * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4       in
y * 2
```

substitute
4 for y

-->

```
4 * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute 2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4       in
y * 2
```

substitute 4 for y

-->

```
4 * 2
```

-->

```
8
```

Moral: Let operates by *substituting* computed values for variables

Typing Let Expressions

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression takes on the type of e2

Typing Let Expressions

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression takes on the type of e2

x has type int for use inside the let body

```
let x = 3 + 4 in

string_of_int x
```

overall expression has type string

Let Expressions Really Are Expressions

```
2 + 3
```

← an expression

Let Expressions Really Are Expressions

```
2 + 3
```

an expression

```
let x = 2 + 3 in
x + x
```

an expression

Let Expressions Really Are Expressions

```
2 + 3
```

an expression

```
let x = 2 + 3 in
x + x
```

an expression

an expression

```
let x = let y = 2 + 3 in y + 5 in
1 + x
```

let expressions can appear anywhere other expressions can appear. they can be *nested*

Exercise

(a)

```
let x =
  let y = 2 + 3 in y
in
  let x = “1” in
x + x
```

(b)

```
let x =
  let y = “2” ^ “3” in y
in
  let x = 1 in
x + x
```

Which of (a) or (b) type check? Explain why.

On a piece of paper (or in your favorite editor), show the step-by-step evaluation of the example that type checks.

Critique the *programming style* used in these examples.