Programming Languages and Paradigms
COMP 302

Instructor: Jacob Errington
School of Computer Science
McGill University

Lesson 2: expressions, values, scopes, and types

Definitions

Expressions. Things like `2 + 5`.

Values. These are irreducible expressions, e.g. `7`, `"hello"`.

Evaluation. The process of turning expressions into values, e.g. `(2 + 5) * 6` $\longrightarrow$ `7 * 6` $\longrightarrow$ `42`.

Defining and applying functions

- Basic syntax: `let` creates new *definitions*
- Call a function with a space: `f a` is `f` applied to `a`.
- Multiple arguments, use more spaces: `f a b`
- Surround an argument with parens if it is complicated: `f (n - 1)` is `f` applied to `n - 1`
  Otherwise, `f n - 1` is `f n` minus one.

Defining and applying functions

- Basic syntax: `let` creates new *definitions*
- Call a function with a space: `f a` is `f` applied to `a`.
- Multiple arguments, use more spaces: `f a b`
- Surround an argument with parens if it is complicated:
  `f (n - 1)` is `f` applied to `n - 1`
  Otherwise, `f n - 1` is `f n` minus one.
- Define a function the same as other values, but with *parameters* which come after the name of the function.

```
let rec fib n =
  if n = 0 then 0 else
  if n = 1 then 1 else
  fib (n-1) + fib (n-2)
```

The `rec` keyword is necessary to make a definition *recursive*.

Figuring out the types

```
let rec fib n =
  if n = 0 then 0 else
  if n = 1 then 1 else
  fib (n-1) + fib (n-2)
```

- How does OCaml know that `n : int`?
- How does OCaml know that `fib n : int`?
  i.e. how does it know that the return type of `fib` is `int`?
- How do we write the type of the *function* `fib`?

This time...

- More on variables, functions, and scope.
- Performance implications of recursion.

`let`, `let ... in ...`, and `let rec`

demo

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)
  3. The overall result of the `let`-expression is whatever `e2` evaluates to.

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)
  3. The overall result of the `let`-expression is whatever `e2` evaluates to.

  Example:
  1. `let x = 2 + 2 in x * x` $\rightarrow$ `let x = 4 in x * x`

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)
  3. The overall result of the `let`-expression is whatever `e2` evaluates to.

  Example:
  1. `let x = 2 + 2 in x * x` $\rightarrow$ `let x = 4 in x * x`
  2. With `x` bound to `4` evaluate `x * x` $\rightarrow$ `16`

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)
  3. The overall result of the `let`-expression is whatever `e2` evaluates to.

  Example:
  1. `let x = 2 + 2 in x * x` $\rightarrow$ `let x = 4 in x * x`
  2. With `x` bound to `4` evaluate `x * x` $\rightarrow$ `16`
  3. And that's the overall result of the `let`-expression.

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)
  3. The overall result of the `let`-expression is whatever `e2` evaluates to.

  Example:
  1. `let x = 2 + 2 in x * x` $\rightarrow$ `let x = 4 in x * x`
  2. With `x` bound to `4` evaluate `x * x` $\rightarrow$ `16`
  3. And that's the overall result of the `let`-expression.
- What is the type of `let x = e1 in e2` ?

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)
  3. The overall result of the `let`-expression is whatever `e2` evaluates to.

  Example:
  1. `let x = 2 + 2 in x * x` $\rightarrow$ `let x = 4 in x * x`
  2. With `x` bound to `4` evaluate `x * x` $\rightarrow$ `16`
  3. And that's the overall result of the `let`-expression.
- What is the type of `let x = e1 in e2` ?
  Hint: evaluating an expression does not change its type.

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)
  3. The overall result of the `let`-expression is whatever `e2` evaluates to.

  Example:
  1. `let x = 2 + 2 in x * x` → `let x = 4 in x * x`
  2. With `x` bound to `4` evaluate `x * x` → `16`
  3. And that's the overall result of the `let`-expression.
- What is the type of `let x = e1 in e2` ?
  Solution: same as the type of `e2`
  Example:

  ```
  let n = 330 in "class has " ^ string_of_int n ^ " students"
  ```

  has type `string` because the part after `in` has type `string`.

Recap

- `let x = e1 in e2` is an expression (`e1`, `e2` arbitrary)
  1. Evaluate `e1` to a value `v`
  2. Evaluate `e2` with `x` bound to `v`
     (Occurrences of `x` in `e2` become replaced by `v`)
  3. The overall result of the `let`-expression is whatever `e2` evaluates to.

  Example:

  1. `let x = 2 + 2 in x * x` $\rightarrow$ `let x = 4 in x * x`
  2. With `x` bound to `4` evaluate `x * x` $\rightarrow$ `16`
  3. And that's the overall result of the `let`-expression.
- What is the type of `let x = e1 in e2` ?
- Top-level definitions, e.g. `let rec fib n = ...`, lack an explicit `in` part; the rest of the file is the *implied* `in` part.
  Only works for top-level definitions.

Shadowing ≠ variable update!

- We can *shadow* bindings in OCaml.
  `let x = 5 in let x = 2 in x + x` evaluates to `4`
- This doesn't *change* the value associated to the first binding of `x`!

Compare: Python vs OCaml scoping

```python
# python
x = 5
def f(y): return x + y
x = 8
a = f(10)
```

```ocaml
(* ocaml *)
let x = 5
let f y = x + y
let x = 8
let a = f 10
```

Compare: Python vs OCaml scoping

```python
# python
x = 5
def f(y): return x + y
x = 8
a = f(10)
```

```ocaml
(* ocaml *)
let x = 5
let f y = x + y
let x = 8
let a = f 10
```

What is the value of `a` after running the program?
Discuss with the person beside you.

Compare: Python vs OCaml scoping

```
1 # python
2 x = 5
3 def f(y): return x + y
4 x = 8
5 a = f(10)
```

```
1 (* ocaml *)
2 let x = 5
3 let f y = x + y
4 let x = 8
5 let a = f 10
```

Python. The variable `x` is updated (line 4)

Compare: Python vs OCaml scoping

```
1 # python
2 x = 5
3 def f(y): return x + y
4 x = 8
5 a = f(10)
```

```
1 (* ocaml *)
2 let x = 5
3 let f y = x + y
4 let x = 8
5 let a = f 10
```

Python. The variable `x` is updated (line 4)
The function `f` uses the latest value of the variable.
Python evaluates `f(10)` to `18`.

Compare: Python vs OCaml scoping

```
1 # python
2 x = 5
3 def f(y): return x + y
4 x = 8
5 a = f(10)
```

```
1 (* ocaml *)
2 let x = 5
3 let f y = x + y
4 let x = 8
5 let a = f 10
```

Python. The variable `x` is updated (line 4)
The function `f` uses the latest value of the variable.
Python evaluates `f(10)` to `18`.

OCaml. The variable `x` is shadowed.

Compare: Python vs OCaml scoping

```
1 # python
2 x = 5
3 def f(y): return x + y
4 x = 8
5 a = f(10)
```

```
1 (* ocaml *)
2 let x = 5
3 let f y = x + y
4 let x = 8
5 let a = f 10
```

Python. The variable `x` is updated (line 4)
The function `f` uses the latest value of the variable.
Python evaluates `f(10)` to `18`.

OCaml. The variable `x` is shadowed.
The definition of `f` can only "see" the binding for `x` above it. The below binding is *not in scope.*
OCaml evaluates `f 10` to `15`.

Compare: Python vs OCaml scoping

```
# python
x = 5
def f(y): return x + y
x = 8
a = f(10)
```

```
(* ocaml *)
let x = 5
let f y = x + y
let x = 8
let a = f 10
```

We can shadow a definition, e.g. for `x`, by making a new one with the same name. This does not affect any definitions that were using the old definition of `x`.

Short break before moving on.

Rewriting fibonacci

```
let rec fib n =
  if n = 0 then 0 else
  if n = 1 then 1 else
  fib (n-1) + fib (n-2)
```

We can do better than exponential time.
Let's see how!

Exercises

- `let rec sqrt (i : int) (n : int) : int = ...`
  Finds the greatest integer whose square is less than or equal to `n`.
  Called as `sqrt 0 n`, i.e. `i` starts at 0.
- `let rec is_prime (i : int) (n : int) : bool = ...`
  Decides whether `n` is a prime number, i.e. not divisible by any number other than 1 or n.
  Called as `is_prime 2 n`, i.e. `i` starts at 2.
  Write `n mod i` to calculate the remainder of dividing `n` by `i`.

Exercises – solutions

demo