Programming Languages and Paradigms
COMP 302

Instructor: Jacob Errington
School of Computer Science
McGill University

Lists and higher-order functions

Last time...

- Polymorphic types:

Last time...

- Polymorphic types:
  - Any type involving *type variables* (e.g. `'a`, `'b`) is polymorphic, e.g. `int -> 'a -> 'a mylist`

Last time...

- Polymorphic types:
  - Any type involving *type variables* (e.g. `'a`, `'b`) is polymorphic, e.g. `int -> 'a -> 'a mylist`
  - Usually pronounced as greek letters “alpha”, “beta”; but also “tick A”, “tick B”; and also simply “A” and “B”.

Last time...

- Polymorphic types:
  - Any type involving *type variables* (e.g. `'a`, `'b`) is polymorphic, e.g. `int -> 'a -> 'a mylist`
  - Usually pronounced as greek letters “alpha”, “beta”; but also “tick A”, “tick B”; and also simply “A” and “B”.
  - Data types can also be polymorphic, e.g. generic optionals:

    ```
    type 'a option = None | Some of 'a
    ```

This time...

- OCaml lists.
- Higher-order functions involving lists and options.

OCaml lists

A generic list type is already defined in OCaml.

```
type 'a list =
  | []
  | (::) of 'a * 'a list
```

- The constructor `[]` is the empty list.
- The constructor `::` (called “cons”) is an operator for extending a list by adding one element to the front.

In words:

- `[]` (nil) has type `'a list` for any type `'a`.
- `x :: xs` (x cons xs) has type `'a list` provided that `x : 'a` and `xs : 'a list`

OCaml lists

A generic list type is already defined in OCaml.

```
type 'a list =
  | []
  | (::) of 'a * 'a list
```

- The constructor `[]` is the empty list.
- The constructor `::` (called "cons") is an operator for extending a list by adding one element to the front.

In words:

- `[]` (nil) has type `'a list` for any type `'a`.
- `x :: xs` (x cons xs) has type `'a list` provided that `x : 'a` and `xs : 'a list`

Example: `1 :: (2 :: [])`

List syntax

- :: (cons) is right-associative

List syntax

- `::` (cons) is right-associative
    - `a1 :: a2 :: ... :: aN :: [] =`
      `a1 :: (a2 :: (... :: (aN :: [])))`

List syntax

- :: (cons) is right-associative
  - `a1 :: a2 :: ... :: aN :: []` = `a1 :: (a2 :: (... :: (aN :: [])))`
- We can use a square-brackets notation to write list literals instead of using ::.

List syntax

- :: (cons) is right-associative
  - `a1 :: a2 :: ... :: aN :: []` =
    `a1 :: (a2 :: (... :: (aN :: [])))`
- We can use a square-brackets notation to write list literals instead of using ::.
  - Instead of `a1 :: (... :: (aN :: []))` we can write `[a1; ...; aN]`.

List syntax

- `::` (cons) is right-associative
  - `a1 :: a2 :: ... :: aN :: [] =`
    `a1 :: (a2 :: (... :: (aN :: [])))`
- We can use a square-brackets notation to write list literals instead of using `::`.
  - Instead of `a1 :: (... :: (aN :: []))` we can write `[a1; ...; aN]`.

Important: the list element separator is the *semicolon.* Commas are only used for tuples!

Practising using lists: reverse and find

demo

Short Break

Short Break

Higher-order functions

A function is *higher-order* if:

- One of its inputs is itself a function.
- It computes a function as output.

Higher-order functions

A function is *higher-order* if:

- One of its inputs is itself a function.
- It computes a function as output.

Requires *first-class* functions.

“First Class”

Something in a programming language is “first-class” if we can pass it to functions, return it from functions, and generally manipulate it like we might manipulate an `int` or `bool`.

Function syntax, take 2

So far we defined functions like this: `let f x = 2 * x`
That's actually another syntactic sugar.

Function syntax, take 2

So far we defined functions like this: `let f x = 2 * x`
That's actually another syntactic sugar.

Remember that functions are first-class? We can write an *expression* that represents the function, and then bind that expression to `f`.

Function syntax, take 2

So far we defined functions like this: `let f x = 2 * x`
That's actually another syntactic sugar.

Remember that functions are first-class? We can write an *expression* that represents the function, and then bind that expression to `f`.

The following are equivalent:

- `let f x = 2 * x`
- `let f = fun x -> 2 * x`

This second syntax, using `fun` is useful for creating anonymous functions.

Simple higher-order functions

demo

Where do higher-order functions shine?

- In combination with polymorphic data structures, they enable us to express concise generic algorithms.

Where do higher-order functions shine?

- In combination with polymorphic data structures, they enable us to express concise generic algorithms.
- An N-input function can be refactored into a function of 1 input that returns a function of $N - 1$ inputs; before returning this function, some computation could be performed – staged computation / partial evaluation.

Where do higher-order functions shine?

- In combination with polymorphic data structures, they enable us to express concise generic algorithms.
- An N-input function can be refactored into a function of 1 input that returns a function of $N-1$ inputs; before returning this function, some computation could be performed – staged computation / partial evaluation.
- Enables us to rewrite any function tail-recursively, using continuation-passing style.

Generic list algorithms

demo

Conclusion: the trifecta

These are the three features that when combined give rise to an extremely expressive language:

1. Polymorphism.
2. Higher-order functions.
3. Pattern matching.

The remainder of the course will essentially just be diving deeper into these topics!