CSCI 334:
Principles of Programming Languages

Lecture 12-1: ML and F#

Instructor: Dan Barowy

Williams

Outline

- Logical operators
- `unit` datatype
- More about lists
- Basic pattern matching

Logical operators

Logical operators

operation	syntax
and	`&&`
not	`not`
equals	`=`
not equals	`<>`
inequalities	`<, >, <=, >=`

`unit`

`unit` datatype

```
public static void main(String[] args) { … }
```

```
let main args = …
```

`unit` datatype

```
public static void main(String[] args) { … }
```

```
let main(args: string[]) = …
```

Remember: every expression must **return a value**.

A function **can't** return nothing.

`unit` datatype

```
public static void main(String[] args) { … }
```

```
let main(args: string[]) : unit = …
```

Therefore, "nothing" is a thing… called `unit`.

`unit` datatype

```
$ fsharpi

Microsoft (R) F# Interactive version 10.2.3 for F# 4.5
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> unit;;

  unit;;
  ^^^^

stdin(1,1): error FS0039: The value or constructor 'unit' is
not defined.

> ();;
val it : unit = ()

>
```

How does one obtain a value of `unit`? `()`

You can also `ignore`…

```
> let foo() = 2;;
val foo : unit -> int

> foo();;
val it : int = 2

> ignore (foo());;
val it : unit = ()

> foo() |> ignore;;
val it : unit = ()

>
```


"forward pipe" operator

```
<expr> |> <expr>
```

```
foo()  |> ignore
```

By the way…

```
let main(args: string[]) : unit = …
```

By the way…

```
let main(args: string[]) : int = …
```

Lists

Linked List

A **linked list** is a recursive data structure.

A list is either:

- the **empty list**, or
- a **node**, containing an element and a reference to a list.

Linked List

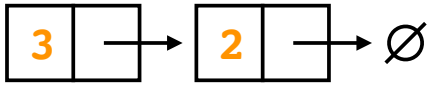
∅

The empty list is defined as **nil** (or `[]`)

Linked List

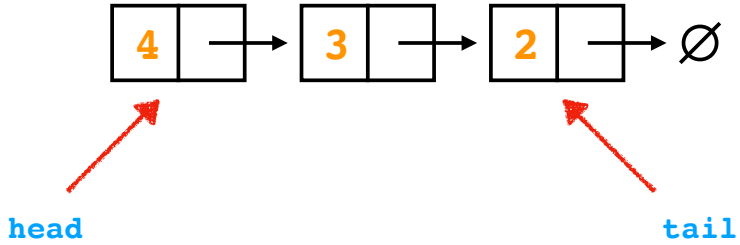
Every other list has at least one list node.

Linked List



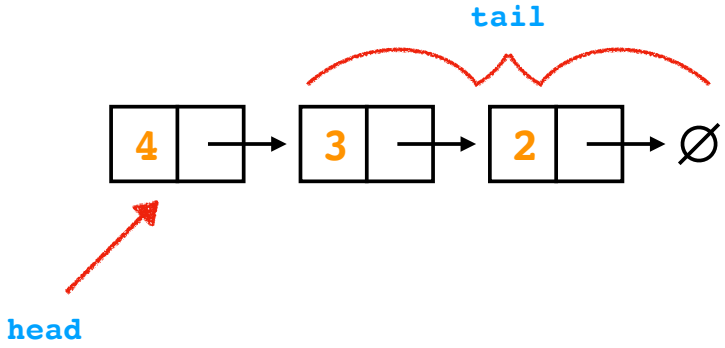
The last node in the list always points to **nil**.

Linked List



A list has parts.

Linked List



A list has parts.

Lists

- Examples
  - `[1; 2; 3; 4]`, `["wombat"; "dingbat"]`
  - `[]` is empty list
  - all elements of list must be same type
- Operations
  - length `length [1;2;3]` ⇒ `3`
  - append `[1;2]@[3;4]` ⇒ `[1; 2; 3; 4]`
  - cons `1::[2;3]` ⇒ `[1; 2; 3]`
  - map `List.map succ [1;2;3]` ⇒ `[2;3;4]`

List types

- `1::2::[] : int list`
  `"wombat"::"numbat"::[] : string list`
- What type of list is `[]`?
  - `[];`
    `val it : 'a list`
- Polymorphic type
  - `'a` is a type variable that represents any type
  - `1::[] : int list`
  - `"a"::[] : string list`

Functions on Lists

Let's define product…

```
> let rec product nums =
    if (nums = []) then
      1
    else
      (List.head nums)
       * product (List.tail nums);;

val product : int list -> int

> product [5; 2; 3];;
val it : int = 30
```

Pattern Matching

Pattern matching

A **pattern** is built from

- **values**,
- (de)**constructors**,
- and **variables**

Tests whether values match "pattern"

If yes, values bound to variables in pattern

Pattern matching

```
let rec product nums =
  if (nums = []) then
    1
  else
    (List.head nums)
    * product (List.tail nums)
```

Using **patterns**…

```
let rec product nums =
  match nums with
  | []     -> 1
  | x::xs -> x * product xs
```

Pattern matching on integers

Write a function `listOfInts` that returns a list of integers from **zero** to `n`.

```
let rec listOfInts n =
  match n with
  | 0 -> [0]
  | i -> i :: listOfInts (i - 1)
```

Oops! This returns the list backward.

Let's flip it around.

Revisiting local declarations

Let's fix our code the lazy way…

```
let listOfInts n =
  let rec li n =
    match n with
    | 0 -> [0]
    | i -> i :: listOfInts (i - 1)
  li n |> List.rev
```

… by defining a function inside our function.

Pattern matching on lists

- Remember, a list is one of two things:
  - `[]`
  - `<first elem> :: <rest of elems>`
  - E.g., `[1; 2; 3] = 1::[2,3] = 1::2::[3]`
    `= 1::2::3::[]`

- Can define function by cases…

```
let rec length xs =
  match xs with
  | []     -> 0
  | x::xs -> 1 + length xs
```

Pattern matching on tuples

Cartesian product…

```
let rec cartesianProduct xs ys =
  match xs,ys with
  | []    ,_   -> []
  |  _    ,[] -> []
  | x::xs',_   ->
    let zs = ys |> List.map (fun y -> (x,y))
    zs @ cartesianProduct xs' ys
```

Patterns in declarations

- Patterns can be used in place of variables
- Most basic pattern form
  - `let <pattern> = <exp>`
- Examples
  - `let x = 3`
  - `let tuple = ("moo", "cow")`
  - `let (x,y) = tuple`
  - `let myList = [1; 2; 3]`
  - `let w::rest = myList`
  - `let v::_ = myList`

Recap & Next Class

Today we covered:

Logical operations

unit datatype

Lists

Pattern matching

Next lecture:

ADTs & advanced F#