CSCI 334:
Principles of Programming Languages

Lecture 4: ML, part 2

Instructor: Dan Barowy
Williams

1

Topics

Combining forms

Pattern matching

2

Your to-dos

1. Read *Advanced F#* **by Thursday**.
2. Lab 2, **due Monday 2/19 by 10pm** (partner lab).

3

Announcements

- Midterm exam dates:
  - Thursday, March 14
  - Thursday, May 2

4

Free your mind

5

Freeing your mind is difficult

6

Remember how I asked you to “be like Neo” and free your mind? Freeing your mind is difficult. If you found the last assignment to be a bit of a challenge, that’s OK. Even Neo tanked it the first time. But keep at it.

Combining forms

7

You can't just make a language out of primitives. How are they combined? We call the operations that "combine" values the "combining forms" of a language.

Functions

```
> let foo a b c = a;;
val foo: a: 'a -> b: 'b -> c: 'c -> 'a
> foo 1 2 3;;
val it: int = 1
```

8

The most obvious and useful combining form is a function. Here I am defining a function and then calling it. Try not to get thrown by the syntax.

Tuples

```
> ("a", 1, 4.4);;
val it: string * int * float = ("a", 1, 4.4)
> let baz (a: string, b: int, c: float) = (a,b,c);;
val baz: a: string * b: int * c: float -> string * int *
float
```

9

Here's another, tuples. If you learned Python, you may have used tuples before and wondered why other languages don't have them. I wonder that too. Tuples are great and F# has them.

Records

```
> type Point = { x: int; y: int; z: int };;
type Point =
  {
    x: int
    y: int
    z: int
  }
> let p = { x = 1; y = 2; z = 3 };;
val p: Point = { x = 1
                 y = 2
                 z = 3 }
> let up pt = { x = pt.x; y = pt.y + 1; z = pt.z };;
val up: pt: Point -> Point
> up p;;
val it: Point = { x = 1
                  y = 3
                  z = 3 }
```

10

We can also make sophisticated objects with named fields as in Java. The customary way to do this in F# is by using something called a Record. They're like classes in Java except that we do not attach methods to them. Or, more accurately, they're like structs in C. If you don't know Java or C don't sweat it. They're just containers for data.

Lists

11

Lists are of fundamental importance in computer science, and the first functional programming language, LISP, was entirely constructed around the idea of list processing. We use lists (specifically, singly linked lists) frequently in F#, and so it has special syntax that make working with them easy.

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.

12

Linked List

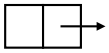
∅

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

13

I draw nil here as a special 0 thing, called “nil”. Note that F# does not use the “null” value for lists. Note that the empty or nil list has no parts.

Linked List



Every other list has at least one list node.

14

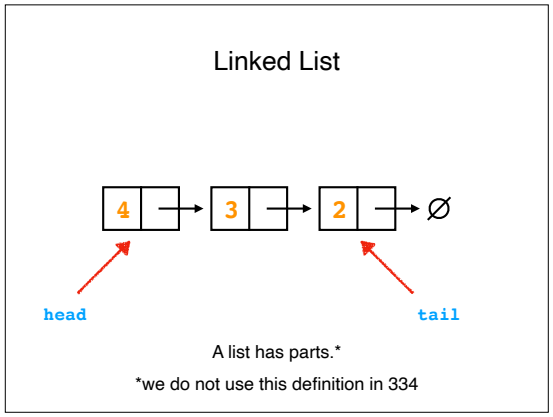
Every other list has parts. The data structure shown here technically is not a list, because the pointer needs to point at something. This is just a list node.

Linked List

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

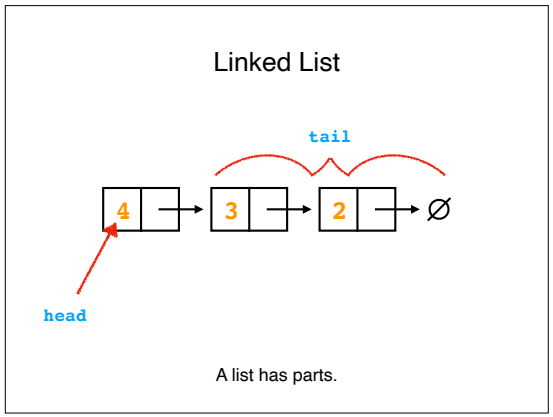
15

This is a well-formed list. A well-formed list must eventually point at nil, because a list is a recursive data structure, and well-formed recursive data structures must exercise their base cases.



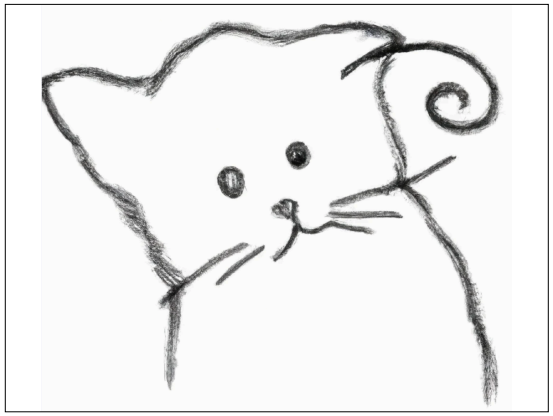
16

You probably learned this definition of a list in CS136. I know, because I teach CS136, and also because I fight with other CS department members about the right definitions. (fighting about data structure terminology is what CS profs do for fun)



17

But you should know that this definition is much more common in functional programming.



18

I asked DALL-E for a picture of a cat's head with a cat's tail coming directly out of it but with no body, but this was the best it could do.

Linked List

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

A list is either:

- the **empty list**, or
- a **node**, containing a head and a tail.

19

Here I restate our definition of a list to use our words head and tail. This has the same meaning as the previous definition slide.

Lists

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

20

Some examples of using lists.

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`

21

Note that lists in F# are statically typed.

Recursive functions

- Note that **recursive** functions must use `rec` keyword.
- Not valid:

```
let fact n =
  if n <= 0 then
    1
  else
    n * fact (n - 1)
```

- Instead:

```
let rec fact n =
  if n <= 0 then
    1
  else
    n * fact (n - 1)
```

22

F# has support for recursive functions but you must use the rec keyword.

Pattern Matching

23

Here's the first feature that is likely VERY different from something you've seen before. Once you get used to this feature, you will miss it in other languages. In fact, some non-functional languages have started to incorporate this feature, like TypeScript.

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
```

24

Suppose somebody asks you to write a program in F# to multiply together all the elements of a list. Since we don't have loops, we will need to use recursion. Remember how recursion works: we need a base case and a recursive case. The base case is to return 1 so that our multiplication problem is grounded. Then we multiple each element one at a time in the recursive case. To do so, we need to remove the head of the list and multiply it by the product of the rest of the list. However, there is a much cleaner way to express this problem using patterns. I'll explain the

difference in a minute, but first, just appreciate how much nicer this looks.

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

25

A pattern is made from values, deconstructors, and variables. A deconstructor is like a constructor, but the inverse. When the value of a variable matches a pattern, we can deconstruct its values and execute a line of code.

Recap & Next Class

Today:

Pattern matching

Next class:

Algebraic data types

Option type

26