Objectives

By the end of this lecture, you will know:

- the runtime of subsets
- what abstract data types are
- what stacks are

By the end of this lecture, you will be able to:

- analyze exponential procedures
- implement stacks as an abstract data type

1 OCaml Suprises

By now, you’ve probably seen some oddities of OCaml. We’re going to discuss a couple things that you may have noticed:
Redefinition

Take a look at the code below. As you can see, we redefine \( a \) after we use \( a \) in \( f \). However, calling \( f \) after the redefinition does not change \( f \). This is because OCaml programs are essentially a whole sequence of let statements, and \( f \) uses the value of \( a \) when \( f \) was defined. Another way of looking at this is to realize that OCaml doesn’t have the same top level environment as Racket does. That being said, this shouldn’t matter much in CS17, as we do not redefine things in this class.

```ocaml
#let a = 2;;
val a : int = 2

#let f x = x + a;;
val f : int -> int = <fun>

#f 4;;
- : int = 6

#let a = 10;;
val a : int = 10

#f 4;;
- : int = 6
```

Here, even after redifining \( a \) to be 10, \((f 4)\) still gives us 6, not 14. This is becuase at the time that \( f \) was defined, \( a \) was bound to 2, and that is the value used in evaluating the closure for \( f \). The new value of \( a \) was added to a local environment that came after the closure, and is therefore not accessible to the closure.

Type Redefinition

Now let’s take a look at this code, which attempts to define the same type multiple times.

```ocaml
# type q = A | B;;
type q = A | B

# let e = A;;
val e : q = A

# type q = A | B;;
type q = A | B

# let f = A;;
val f : q = A

# e = f;;

Error: This expression has type q/1036 but an expression was expected of type q/1032.
```

Some of you may have encountered this issue during Eliza, as you imported files into eliza.ml. Why does this happen? Turns out, the first definition of \( q \) and the second definition refers to completely different types! Type definitions in OCaml are like \texttt{let}. When we define a type, it is like saying use this definition for my type for the rest of the program. However, if we define another type with the same name, the old definition is no longer accessible and the new one is used for the rest of the program. So \( e \) is of the type of the first \( q \), while \( f \) is of type second \( q \). So, defining a type twice hides the first definition.
It is really important to be careful about redefining types especially when importing files. For example, if you have file A with your type definitions, file B with some support code, and file C with your actual program, if B is importing A, you should only import B in C and not also A. Since include basically copies all the code from the file to be imported into your file, your types in A will be defined twice.

This also becomes easy to overlook when using the REPL. If you already have entered some type definitions, and then try to start fresh by copying in all your code again, the REPL does not start a new session automatically and you will end up redefining types. Always exit the REPL before starting afresh!

**Check Error and Failwith**

Have you ever wondered why we wrap our function call in parentheses with `fun()` when we’re trying to run `check_error`?

Think back to the `transpose` procedure you wrote for homework. Say we wrote the following `check_error`:

```ocaml
check_error (transpose []) "A matrix can't be 0-dimensional.";;
```

OCaml would attempt to execute `(transpose [])` which would produce an error and halt execution, so `check_error` would never get a chance to be run and confirm that the error was indeed the expected one. So, instead, we wrap the broken part of the code in a function which we pass to `check_error`. Within `check_error` is a piece that essentially invokes this function (generating an error!), but does it within a “try-catch” block which runs code that may fail, but “catches” any errors, allowing the overall program to continue running. When we “catch” an error, we also get the error message that was generated, and can compare this to the intended error message. And that’s what `check_error` is doing. But to make all that work, we have to delay the execution of the offending code until we’re within a try-catch block, which is why we write this:

```ocaml
check_error (fun() -> (transpose [])) "A matrix can't be 0-dimensional.";;
```

So here, `check_error` takes 2 arguments, the first is a procedure, which when called should fail. The second is the expected contents of the exception produced by `failwith`.

Try-catch blocks are useful when you have code that may fail. These are also used in dealing with uncertain things like user input or availability of network data. For instance, if you are expecting the user to enter a number from 1 to 10 as an argument to your procedure but they enter “hello” instead, your program would halt. What if you wanted it to prompt the user to give the right input, and continue execution? You could then use a try-catch block to catch your error without ungracefully halting the program!

Finally, let’s discuss the type of a `failwith`. As you know, OCaml is very strict about type. Think about the `nth` procedure for example. One of its arguments is of type `'a list` and it outputs type `'a. You may wonder if it is a type signature violation to put a `failwith` statement in such a procedure. Consider the example below:

```ocaml
fun nth (n: int) (alod: 'a list): 'a =
    match (n, alod) with
```

3
What we know is that every expression to the right of -> should have the same type. In the example below, since hd is of type 'a, failwith should also return type 'a. So failwith is of type string ->'a. But failwith is not actually producing something of type 'a. There's a trick here - when OCaml reaches the failwith, the program halts and the expression never returns anything. Thus, by just specifying that failwith is of type string ->'a, our code will compile, and we will not have any problems.

2 Subsets Analysis & Exponential Runtime

Spike did a runtime analysis of subsets from homework 06, proving that the runtime of subsets is exponential. The details of this proof can be found in the uploaded PowerPoint slides for this lecture.

3 Modules as a tool for organization

When software gets big, it gets messy and impossible for one single person to manage, and taming the messiness is one of the big goals of Computer Science. One way to do this is using interfaces. An ATM, for example, is a real life interface; it provides you with useful functionality of a bank, without the user ever needing to understand what is going on behind the scenes to execute your transactions. Furthermore, the way you interact with an ATM are standard across different banks and locations. Essentially, interfaces define the way two entities interact. In OCaml, such interfaces are called modules.

3.1 Examples of Modules

A module we have been using is List, which consists of a collection of functions defined like List.hd, List.tl, List.nth, List.map that all operate on the same kind of data, which are lists.

Another example would be “regular expressions”. You may want to use regular expressions and procedures that operate on them in various programs without having to write these over and over. To achieve this, we can create an “RE” module. A short version of that might have procedures like:

```ocaml
make_regexp : string -> regexp
match : regexp -> string -> bool
```

We would also have to define a data type regexp. There are a few ways in which we can do this. We could just use “string”, or we could use a piece list where,

```ocaml
type piece = Any | One | Few | ...
```
An OCaml module is therefore an assembly of types and procedures (and sometimes values). In code, a module is called a structure, or struct.

To refer to something in the RE module for example, we could write

```ocaml
RE.make_regexp
```

If we are going to use a particular module a lot, we can “open” it and refer to its data and procedures directly without a prefix. In order to build a module, we need to create these datatypes and procedures. As seen before, our regexp data type, for example, could be defined using strings or piece lists, and the users will never see this underlying implementatons. As long as the procedures in our module work as expected, it wouldn’t matter to them.

To expand on that idea, sometimes we promise to provide a service but it doesn’t matter how, just that it gets done. When we asked you to transpose a matrix in the homework, some people took a row wise approach by trasposing the rest of the matrix, and then cons-ing something onto the front of each resulting row. Others took a column wise approach by mapping rest over the matrix to get the last \( n - 1 \) columns, and then transposed that and cons-ed on one last row. Both ways of solving the problem got the job done! If someone decided to reimplement the transpose procedure, the procedure will still work for the user and not change anything.

Similarly, the procedure specification and type definitions in a module constitute an interface. The user does not need to know how things work, just that they do work. Any regular expression module then would have to provide a type called regexp, a way to build a regexp from a string, and a match-testing procedure. In OCaml, such a specification is called a module signature. Your module would also include procedures that actually implement all the procedures the signature requires. A signature is similar to a type signature, but for modules rather than procedures.

Note: In OCaml, there is no RE module. Instead there is Str which includes strings and regular expressions!

### 3.2 Abstract Data Types

We will now introduce a new term: Abstract Data Type or ADT. The kind of specification we have discussed that is implemented though modules in OCaml is called an ADT. To reiterate, An ADT is a specification of a data type and the operations that can be performed on data of that type. As the name suggests, ADTs are abstract; they tell nothing of the underlying representation of the data they contain, or of the concrete implementations of the various operations that manipulate those data. Furthermore, the users (i.e., other programmers) of an ADT cannot access the data directly; they can only access them through an interface. By defining an interface, we can seamlessly switch between different implementations.

### 4 An Example: Stacks

We have seen that OCaml lets us define data types. In addition, OCaml lets us define “operation” types. That is, we can declare a data type and specify the interface through which we access values of that type. This specification is called a module type in OCaml, and its implementation is called a module. We illustrate the syntax for module types and modules by example.
Our first example of an abstract data type is a stack. A stack is based on the principle of LIFO: “last in, first out.” Imagine a face-up pile of cards. If you want to look at the top card, you can do so with very little effort. It’s similarly easy to remove the top card, or to put a new card on the pile, or to check if the pile is empty. But doing anything else—like looking at the middle card, for example—is harder (computationally).

The operations a stack supports are the following:

- **check if the stack is_empty:**
  ```plaintext
  is_empty : 'a stack -> bool
  ```

- **push** a datum onto the top of the stack:
  ```plaintext
  push : 'a -> 'a stack -> 'a stack
  ```

- **pop** the most-recently pushed datum off the stack; and
  ```plaintext
  pop : 'a stack -> 'a stack
  ```

- **look at the datum on the top** of the stack.
  ```plaintext
  top : 'a stack -> 'a
  ```

**Question:** How is this any different than a list?

**Answer:** Lists are a really good way to implement a stack. The empty list `[]` could be used as an empty stack, “cons” could be used to write `push`, “rest” could be used to write `pop`, and “first” could be used to write `top`. But, this is only one implementation of a stack. There are a bunch of others, several of which you will see if you continue on to take CS 18. Though, these implementations do the same thing in the sense that they allow you to store and retrieve data in a sequential way.

### 4.1 Module Types

A **module type**, or a **signature**, is a specification of how you use an ADT. That is, it provides the interface to the ADT—the operations defined on data of that type.

Let’s define a signature for a stack. We start like this:

```ocaml
module type STACK =
  sig
    ...
  end
```

By convention, module type names are written in all capital letters.

In general, the shape of a signature is:

```ocaml
module type (NAME) =
  sig
    (body)
  end
```
A signature contains three things:

1. A data type: what name do we want to use for data of this type, and how (if at all) is this type parameterized? N.B.: A signature may actually contain many data type descriptions, but for our first couple, we’ll just have one type in each. When we get to the Game project, a GAME will have a type for moves, a type for the game-progress (either “still in play” or “over”), a type for the game-state (i.e., where the checkers are on the checkerboard, or which squares have x’s or o’s in them for tic-tac-toe).

2. Constructors: how do we create values of this type?

3. Other operations: what can we do with values of this type once we’ve constructed them?

Let’s fill in our signature based on the answers to these three questions.

1. What name do we want to use to refer to our concrete representation of stacks? For the sake of simplicity, we’ll just use stack. But that isn’t quite enough. Stacks, like lists, are containers, and so a stack must contain elements of some type. We want to be able to have stacks of numbers, stacks of strings, stacks of lists, and so on. So, we have to add a type parameter:

```ocaml
module type STACK =
sig
  type 'a stack
... end
```

Question: Can a stack be heterogenous?

Answer: No! (And that’s an adamant no.) Stacks, like lists, are containers. They contain data of one and only one type.

Question: What if we were working with stacks of tuples? Can 'a be a tuple?

Answer: Sure! (And that’s an adamant yes.)

Question: How would we do that? Like this?

```ocaml
module type TUPLE_STACK =
sig
  type ('b * 'c) stack
... end
```

Answer: Yes, you could, and indeed you should. But note that this is not actually necessary, since the type parameter 'a can be instantiated with any type, including compound types like ('b * 'c). But it is good programming practice to be as specific as possible with your type declarations. So, if your stacks contain only tuples, then this would be the definition of choice.

2. How can we create stacks? In our ADT, we’ll provide an empty stack, which we can then build on. We’ll use our type declaration together with a val declaration to do this. The latter is a way of declaring a value, by giving it a name. (We don’t actually care what the value is at this point; we only care about its type.)
module type STACK =
sig
  type 'a stack

  (* Constructs an empty stack *)
  val empty : 'a stack
end

In general, the shape of a \texttt{val} declaration is:
\[ \texttt{val} \langle \text{name} \rangle : \langle \text{type} \rangle \]
Here, \langle \text{name} \rangle can be any identifier, and \langle \text{type} \rangle can be any OCaml type (built-in or user-defined).

3. We are now ready to specify the rest of the operations on stacks. We specify their type signatures, and we also include comments above their declarations describing their purpose.

Specifically, for a stack, we want to be able to check if the stack is empty. We also want to be able to view the datum on top, and remove it from the stack. And given a stack and a datum, we want to be able to push the datum onto the stack. This leads us to the following signature:

module type STACK =
sig
  type 'a stack

  (* Constructs an empty stack *)
  val empty : 'a stack

  (* Tests whether a stack is empty *)
  val is_empty : 'a stack \rightarrow bool

  (* Pushes an item onto a stack *)
  val push : 'a \rightarrow 'a stack \rightarrow 'a stack

  (* Pops an item off a stack *)
  val pop : 'a stack \rightarrow 'a stack

  (* Peeks at the top of a stack *)
  val top : 'a stack \rightarrow 'a
end

In sum, a signature consists of a possibly (and in fact, usually) parametric type declaration, followed by several \texttt{val} declarations in which the constructor and various other operations provided by the ADT are specified in terms of the given type declaration.

As you can see, \texttt{val} declarations also allow you to declare procedures (which are, of course, values after all!).

\textbf{Question:} How do we ensure that \texttt{top} and \texttt{pop} operate only on non-empty stacks?

\textbf{Answer:} At the moment we are simply declaring the type signatures of these procedures. Type signatures do not enforce such restrictions. We will, however, enforce these restrictions in our implementations of these operations.
4.2 Modules

We’ve now specified our ADT. But how do we implement it? To do so, we’re going to use yet another OCaml construct, called modules. As you might expect, modules look very similar to signatures. In fact, we can use a signature as a template for writing a module.

Here is our signature, again:

```ocaml
module type STACK =
  sig
    type 'a stack
    val empty : 'a stack
    val is_empty : 'a stack -> bool
    val push : 'a -> 'a stack -> 'a stack
    val pop : 'a stack -> 'a stack
    val top : 'a stack -> 'a
  end
```

And here is a module template whose structure mimics that of our signature:

```ocaml
module ModuleName : STACK =
  struct
    type 'a stack = ...
    let empty = ...
    let is_empty <'a stack> : bool = ...
    let push <'a> <'a stack> : 'a stack = ...
    let pop <'a stack> : 'a stack = ...
    let top <'a stack> : 'a = ...
  end
```

Here, <'a> and <'a stack> are placeholders for arguments of those types.

Signatures and modules look similar to one another. What are the key differences between them?

- You explicitly specify which signature the module is implementing (e.g., STACK).
- The module itself has a name (e.g., ListStack).
- The second line is struct instead of sig.
- Each occurrence of val in a signature corresponds to an occurrence of let in a module.
• All types and identifiers declared in the signature must be defined in the module. Whatever else is deemed necessary (e.g., helpers) can also be defined in the module; but such definitions won’t be part of the interface.

Now, let’s fill in our template. How do we want to implement stacks? Well, we need to store an arbitrary amount of data, so let’s try using lists.

```ml
module ListStack : STACK =
struct
  type 'a stack = Stack of 'a list

  let empty = ...

  let is_empty s = (s = empty)

  let push datum (Stack l) = Stack (datum :: l)

  let pop (Stack l) =
    match l with
    | [] -> failwith "empty stack"
    | hd :: tl -> Stack tl

  let top (Stack l) =
    match l with
    | [] -> failwith "empty stack"
    | hd :: tl -> hd
end
```

Instead of defining `type 'a stack = 'a list`, we use the `Stack` constructor to protect ourselves against accidentally confusing lists and stacks inside our module. This isn’t such a big issue for stacks, but we’ll find it very useful when implementing other abstract data types whose corresponding concrete representations are not as similar as stacks and lists.

Since stack operations are so much like list operations, filling in the rest of our template is a piece of cake:

```ml
module ListStack : STACK =
struct
  type 'a stack = Stack of 'a list

  let empty = Stack []

  let is_empty s = (s = empty)

  let push datum (Stack l) = Stack (datum :: l)

  let pop (Stack l) =
    match l with
    | [] -> failwith "empty stack"
    | hd :: tl -> Stack tl

  let top (Stack l) =
    match l with
    | [] -> failwith "empty stack"
    | hd :: tl -> hd
end
```
One very cool feature of this implementation is that we pattern match on the arguments to our procedure. That is, instead of writing these three lines of code:

```ocaml
let push datum s =  
match s with  
| Stack l -> Stack (datum :: l)
```

or these three lines of code:

```ocaml
let push datum s = 
let (Stack l) = s in 
Stack (datum :: l)
```

it suffices to write just one line of code:

```ocaml
let push datum (Stack l) = Stack (datum :: l)
```

**Question:** Can you implement `pop` and `top` like this:

```ocaml
let pop (Stack hd :: tl) = Stack tl
let top (Stack hd :: tl) = hd
```

**Answer:** No, you can’t. OCaml would complain that the pattern matching is not exhaustive, and you should treat such complaints as an error.

## 5 Summary

### Ideas

- We’ve introduced abstract data types which is essentially a specification of a data type and the operations that can be performed on it.
- ADTs offers implementation neutrality - in other words, we can switch data representation without affecting the rest of our program.
- ADTs also preserve invariants. By preventing access to the internal representation of ADTs, we can ensure that our implementation maintains this invariant on all operations.

### Skills

- We learned how to write a module type/signature - which is a specification of how to use an ADT.
- We learned to use modules, which look very similar to signatures. All types and identifiers declared in the signature must be defined in the module. Other helpers can be defined within a module, but won’t be part of the interface.
• We learned how to view data within abstract data types, for testing purposes.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs017/feedback