Take, Drop, Nth, match-argument suppression

The "unit" type

Check-expect and check-error

Modules again: a Stack module

Rackette Processing in detail
Warmup

/* take
 * Input:
 * - n : a natural number
 * - lst : an 'a list of length L, with n <= L
 * Output:
 * an 'a list with the first n items of lst */
let rec take = (n: int, lst: list('a)): list('a) =>
  switch (n, lst) {
    |...
    |...
    |...
    |...
  }
Warmup

/* take
* Input :
* - n : a natural number
* - lst : an 'a list of length L, with n <= L
* Output :
* an 'a list with the first n items of lst */

let rec take = (n: int, lst: list('a)): list('a) =>
switch (n, lst) {
| (0, _) => []
| (n, []) => failwith("Tried to take more elements than available")
| (n, [hd, ...tl]) => [hd, ...take(n-1, tl)]
};
Warmup

/* take */
* Input:
* - n: a natural number
* - lst: an 'a list of length L, with n <= L
* Output:
* an 'a list with the first n items of lst */
let rec take = (n: int, lst: list('a)): list('a) =>
switch (n, lst) {
| (0, _) => []
| (_, []) => failwith("Tried to take more elements than available")
| (n, [hd, ...tl]) => [hd, ...take(n-1, tl)]
}
This week's homework

• ... provides "take" and "drop" for you
• and the List module provides "nth", which selects the nth item in a list.
Take, Drop, Nth, match-argument suppression

**The "unit" type**

Check-expect and check-error

Modules again: a Stack module

Rackette Processing in detail
The type “unit”

• (First mentioned a while ago)...
• The type “unit” has a single item (sort of the way “bool” has exactly two).
• That single item is denoted ()
• It’s actually surprisingly useful
  • `print_string` has type `string -> unit`, for instance
• Lots of built-ins that work with the operating system end up using “unit” as well.
• It’s used as the argument type for a function with no arguments
  • `let f: unit => int = () => 4;`
A function with no arguments?

• Why would you ever want one of those? Why use
  `let f = () => 4;`
  Instead of
  `let f = 4;`? It’s a constant function, after all!
• Consider
  `let f = () => 10/0;`
  Instead of
  `let f = 10/0;`
• The first produces no error (until you use f!); the second
  produces an error right away!
While we’re looking at weird things...

• What’s the type of the function `failwith`?
  ```csharp
  let dizzy : int => int = fun
  | 0 => failwith ("can’t divide by 0")
  | n => 10/n;
  let tizzy : string => string = fun
  | "abc" => failwith ("No alphabets!")
  | s => s;
  ```

• Clearly `failwith` is both an integer value and a string value:
  ```csharp
  let failwith: string => ???
  ```
While we’re looking at weird things...

• What’s the type of the function `failwith`?
  ```ocaml
  let dizzy : int => int = fun
  | 0 => failwith ("can’t divide by 0")
  | n => 10/n;
  let tizzy : string => string = fun
  | "abc" => failwith ("No alphabets!")
  | s => s;
  ```

• Clearly `failwith` is both an integer value and a string value:
  ```ocaml
  let failwith: string => 'a
  ```
What type does failwith actually produce???

• It doesn’t!
• Program terminates before it ever returns a value
• But saying it has type string => ’a means that it type-checks OK!
Let’s write checkExpect

• Specification: ???
Let’s write checkExpect

- checkExpect: ('a, 'a, string) => ???
- Do we ever *use* the value returned by checkExpect?
  - No!
  - Good reason to have it be “unit”!
- checkExpect: ('a, 'a, string) => unit
Let’s write checkExpect

- checkExpect: ('a, 'a, string) => unit
- What should checkExpect(x, y, s) do?
  1. Check whether x and y are the same
  2. If so, do nothing.
  3. Otherwise, report an error (use “print_endline: string => unit”)

let checkExpect: ('a, 'a, string) => unit = (a, b, s) =>
  if (a == b) { () } else { print_endline(s) } ;
Let’s write checkError!

- **checkError**: ('a, string) => unit
- What should `checkError(x, msg)` do?

1. Check whether `x` produces an error message `msg`; if so, do nothing.

2. Otherwise, report an error

   ```
   let checkError: ('a, string) => unit = (a, msg) =>
   if (???) { () } else { ??? } ;
   ```

- Problems:

  1. When there’s an error a string gets *printed*...but the value produced is ().
     - There’s no way to get at the printed string!

  2. The error occurs and the body of `checkError` never gets processed, because processing halts!

3. (N.B.: the TAs pointed out that our `checkError` doesn’t have a separate message to print to identify the check-error, so I removed the third argument that was present during class.)
What we want vs. what we have to live with

• We’d like to write
  `checkError(1/0, "Divide by zero");`

• We actually write
  `checkError(() => 1/0, "Divide by zero");`

• That means that the type of `checkError` is
  `let checkError : (() => 'a, string) => unit`

• The use of “unit” to make a function lets us delay evaluation of the “bad part” until we can handle it
  `checkError(()`
Wait... *how do we handle it?*

- We use a “try” expression:

  ```
  try (<something> ) {
  |...
  |...
  }
  
  Example
  try (100/x){
  |_ => 17
  }
  ```

- If x is zero, there’ll be a divide-by-zero “exception” that gets “raised”, and then “caught”; it’ll match the “_” and produce the value 17
- If x is nonzero, then the result will be 100/x as expected.
- NB: although this failed when I tried it in class, I just tried it again and it works fine; I probably added a semicolon somewhere in the wrong place, or used a smart-quote, or...
More details

Example
try (100/x){
|_ => 17
}  
• We can raise our own exceptions using “failwith”
The raised exception for failwith(“message”) is “Failure(“message”)”
try (  
{  
    failwith(“my message”);  
    17  
}) {  
| Failure(“my message”) => 22  
};  
Will produce the number 22.
More details

• Three cases
  1. The raised exception is the expected one (pass)
  2. An exception is raised, but it’s not the right one (fail)
  3. No exception is raised (fail)

• See the TA code in CS17 ...re to see details

• You never need to use try{} expressions
  • ...but it’s nice to have seen them once before you need to use them in another class.
  • In many languages: try ... catch ...
Take, Drop, Nth, match-argument suppression
The "unit" type
Check-expect and check-error

**Modules again: a Stack module**
Rackette Processing in detail
An example module type (signature) and module: stacks
Stack ADT (abstract data type)

• Represents something like a stack of playing cards

• **Defined** by allowed operations:
  • You can put something onto the top of the stack ("push")
  • You can remove something from the top of the stack ("pop")
  • You can look at the top item without removing it ("top")
    [sometimes "peek"]
  • You can create an empty stack
  • You can check whether a stack is empty

• Typically a stack contains items all of the same kind
  • ints, bools, ...
  • "processes" in your computer
  • ...
ReasonML for representing an ADT

- "Module type": says what a module must contain, but doesn't say how anything is done.

```reasonml
module type Stack =
{
  type stack('a);
  let empty: stack('a)
  let isEmpty: stack('a) => bool
  let push : ('a, stack('a)) => stack('a)
  let pop : stack('a) => stack('a)
  let top: stack('a) => 'a
};
```
A module that has the specified type: ListStack

module ListStack =
<copy and paste the module-type definition here>
A module that has the specified type: ListStack

module ListStack =
  type stack('a);
  let empty: stack('a)
  let isEmpty: stack('a) => bool
  let push : ('a, stack('a)) => stack('a)
  let pop : stack('a) => stack('a)
  let top: stack('a) => 'a
}
module ListStack =
{
    type stack('a) = Stack (list('a));
    let empty: stack('a) = Stack([]);
    let isEmpty: stack('a) => bool = s => (s == empty);
    let push: ('a, stack('a)) => stack('a) = (datum, Stack(lst)) => Stack([datum,...lst]);
    let pop : stack('a) => stack('a) = fun
        | Stack([]) => failwith("Can't pop from empty stack.")
        | Stack([hd, ...tl]) => Stack(tl);
    let top: stack('a) => 'a = fun
        | Stack([]) => failwith("Empty stack has no top element.")
        | Stack([hd, ...tl]) => hd;
};
How do you test a module?

- You want to write procedures that build examples, check that they do the right thing, etc.
- Those procedures are not part of the module type, so if you say the module has that type, you can't use those procs.
- CS17 solution
  1. rename the module to TestListStack
  2. Include/write testing functions
  3. Test like mad
  4. Then write `module ListStack = TestListStack:Stack;`
A module that has the specified type: ListStack

module TestListStack =
{
  type stack('a) = Stack (list('a));
  let empty: stack('a) = Stack([]);
  let isEmpty: stack('a) => bool = s => (s == empty);
  let push: ('a, stack('a)) => stack('a)  = (datum, Stack(lst)) => Stack([datum,...lst]);
  let pop : stack('a) => stack('a) = fun
      | Stack([]) => failwith("Can't pop from empty stack.")
      | Stack([hd, ...tl]) => Stack(tl);
  let top: stack('a) => 'a = fun
      | Stack([]) => failwith("Empty stack has no top element.")
      | Stack([hd, ...tl]) => hd;
};
module TestListStack =
{
    type stack('a) = Stack (list('a));
    ...
};
let c = TestListStack.empty;
let c1 = TestListStack.push(8, c);
print_int(TestListStack.top(c1));
  8
module ListStack:Stack = TestListStack;
let d = ListStack.empty;
let d1 = ListStack.push(8, d);
print_int(ListStack.top(d1));
Type ascription and data hiding

• When we say that ListStack meets the signature Stack, suddenly all the contents of a ListStack are hidden from us
  • Called "signature ascription"
• We can only see a ListStack through its interface
• Why would we want this?

• This lets us change our implementation of ListStack without breaking any program that uses it!
• Problems: testing is a pain
• Advantages: sometimes we find ourselves wanting to get at the underlying representation, and the "hiding" annoys us.
  • That's proof that it's doing its job!
This weekend's lab

• We'll write a signature (i.e., module **type**) for sets (Set)
• You'll implement a module, ListSet, that meets this signature
• Then you'll write another module, SortedListSet, that also meets this signature
• It'll be possible to swap them for one another in any program that needs to use a set!
• You'll write "subsets" and "set difference" and test this out.