CSci 2041
Advanced Programming Principles
L12: Programming Techniques 1

Eric Van Wyk

Fall 2014

- Hwk6 is complex; it involves many pieces.
- How should a complex program like this be structured or organized?
- What goals should we have in considering various structures?

Goals

- You will want to write your solution so that these pieces can be individually tested.

  Thus, you may want these functions to return intermediate data (lists of words) instead of simple results from comparisons of that data.

- Our programs should be easy to read.

- Programs should be easy to write, but this goal is secondary to ease in reading or comprehending.

- Makes use of the types and static analyses in the language to detect and avoid errors.

Testing

- You may consider writing a function `get_lines` that takes the filename of the potential paradelle and returns a `estring list option` value.

  Here, `None` indicates that there were not 24 lines in the file.
  Otherwise a list of 24 `estring`s are returned.

- But this is harder to test. If there is a bug we can't easily see what may have gone wrong.

- Instead, you may want to write function that returns the list of lines before trying to filter out blank lines, and

  a function that does the filtering so that we can easily inspect the output of both.

In general, write functions that return useful data, not just true or false.

Name commonly used temporary values

- REPLs (read, evaluate, print loops) are convenient for testing these functions.
- They let us just type in non-trivial expressions to test our functions: *e.g.*

```
List.sort cmp (words (take 1 (drop 6 poem_lines)))
```

- But typing long expressions can be tedious.
- Instead, define temporary values in your source file.

  ```
  let stanza1 = drop 6 poem_lines
  let ln1 = take 1 stanza1
  ```
- Then `List.sort cmp ln1` in utop.
- Think carefully about what named values you may want to experiment with when designing your functions.

Use asserts

Once a function seems to be working, define a few asserts in your source file to ensure that future edits don't break them.

```
assert (same_words ln1_stz1 ln2_stz1)
```

where these values have been named as well.

Pipelines, with errors

The paradelle problem and many others process data with a sequence of transformation or analysis steps.

- The transformations are often functions whose output is a different type than the input.

  e.g. , converting a text file to a list of lines
- The analyses check some condition and may terminate the computation with an error if it fails.

  e.g. , checking for 24 lines in a paradelle.
- Some transformation steps may also fail.
  They have both transformation and analysis components.

How can we organize programs of this type?

An important question is how we choose to represent valid results and failures.

A transformation may need to return a value indicating each possible outcome.

Similarly, an analysis needs to indicate success or failure.

What are some possible options for this data type?

- One possibility is an `option`.

  A transformation may have the type
  `type_data_1 -> type_of_data_2 option`

- Some analyses may return just `true` or `false`.

  This may just fine for checking that, for example, a poem has 24.

- But while these may be fine for individual steps, they provide no way to report errors.

A type such as the following would support the reporting of error messages.

```
type 'a 'b result = OK of 'a
                    Error of 'b
```

Here `'a` is the type of a error-free result.

And `'b` is the type for error messages.

This could be `string` or a disjoint union type that enumerate the different kind of errors.

This may be useful if our code needs to process the errors that may occurs.

Consider a compiler and how it needs to report errors.

We also want to give some consideration to the structure of the code that applies these transformations and analyses and handles the errors.

- `match` expressions are generally preferred to `if-then-else` expression.

  But simple boolean-valued checks may be fine as `if-then-else` expressions.

- Giving names to intermediate results when the function that generates them is a general purpose function helps the reader understand the program.

- Use exceptions for truly exceptional behavior, not for expected possible errors in the input.

Warning: the following code snippets are for illustration purposes, they have not be run through OCaml.

A series of error-reporting transformations

```
match transform_1 data_1 with
| None -> Error "problem in transform_1 ..."
| Some data_2 ->
    match analysis_2 data_2 with
    | false -> Error "problem in analysis_2 ..."
    | true ->
        match transform_3 data_2 with
        | Error e -> Error e
        | OK data_3 ->
            ...
```

A series of error-reporting transformations, poorly structured.

```
match transform_1 data_1 with
| Some data_2 ->
   (
    match analysis_2 data_2 with
    | true ->
       (
        match transform_3 data_2 with
        | Error e -> Error e
        | OK data_3 ->
            ...
       )
    | false -> Error "problem in analysis_2 ..."
   )
| None -> Error "problem in transform_1 ..."
```

In this case it is difficult to see the valid and erroneous options for each transformation or analysis.

So, this style is preferred.

```
match transform_1 data_1 with
| None -> Error "problem in transform_1 ..."
| Some data_2 ->
    match analysis_2 data_2 with
    | false -> Error "problem in analysis_2 ..."
    | true ->
        match transform_3 data_2 with
        | Error e -> Error e
        | OK data_3 ->
            ...
```

But if there are very many transformations the indentation can get too deep.
We could split this into a few phases, each phases implemented as a function processing some number of transformations.

```
(* phase1 : type_of_data_1 -> type_of_data_2 string result *)
let phase_1 data_1 =
  match transform_1 data_1 with
  | None -> Error "problem in transform_1 ..."
  | Some data_2 ->
      match transform_2 data_2 with
      | false -> Error "problem in analysis_2 ..."
      | true ->  OK data_2

(* phase2 : type_of_data_2 -> type_of_data_4 string result *)
let phase_2 data_2 =
    ...  similar to phase 1 ...

(* process: type_of_data_1 -> type_of_data_4 string result *)
let process data_1 =
   match phase_1 data_1 with
   | Error e -> Error e
   | OK d -> phase_2 d
```

Naming intermediate results

It is wise to give good names to intermediate results that are produced by general purpose functions.

```
let poem_lines = split text ["\n"]
in
if length poem_lines <> 24
then Error "Incorrect number of lines"
else ... continue processing ...
```

Error values

Above, we use `string` as our error value.

It is often better to be precise and provide an error type that captures the different kinds of expected errors.

Perhaps

```
type Errs = FileNotFound of string
          | IncorrectNumLines of int
          | InvalidStanzaLines of (int * int) list
          | InvalidLastStanza
```

Exceptions

- We typically should not raise exceptions for expected possible invalid input data.
- For example, raising an exception when a file is not found from a user provided file name.
- But this is always a balancing act. We also don't want possible failure handling code to cascade into deeply indented programs.

  Though we've seen techniques for dealing with this above.

Exceptions

We should raise exceptions for those cases that “really should never happen” — those that indicate an error in your program.

What about a function `check_stanza` whose input is a 6 element list of stanza lines?

The calling function checked that there were 24 lines and split them 4 ways.

Exceptions

We may write a match of the form

```
match stanza with
| l1:l2:l3:l4:l5:l6:[] ->  ... check the lines ...
```

But this is not an exhaustive match. The compiler raises warning that we should fix.

A good fix would be

```
| _ -> raise (Failure "Internal error in check_stanza")
```