Take-home Re-Exam in Advanced Programming

Deadline: Friday, February 4, 15:00

Version 1.0

Preamble

This is the exam set for the individual, written take-home re-exam on the course Advanced Programming, B1R-2021. This document consists of 14 pages; make sure you have them all. Please read the entire preamble carefully.

The exam consists of two questions. Your solution will be graded as a whole, on the 7-point grading scale, with an external examiner. The questions each count for 50%. However, note that you must have both some non-trivial working Haskell and Erlang code to get a passing grade.

In the event of errors or ambiguities in an exam question, you are expected to state your assumptions as to the intended meaning in your report. You may ask for clarifications in the discussion forum on Absalon, but do not expect an immediate reply. If there is no time to resolve a case, you should proceed according to your chosen (documented) interpretation.

What To Hand In

To pass this exam you must hand in both a report and your source code:

- *The report* should be around 3-6 pages, not counting appendices, presenting (at least) your solutions, reflections, and assumptions, if any. The report should contain all your source code in appendices. The report must be a PDF document.
- *The source code* should be in a .ZIP file called `code.zip`, archiving one directory called `code`, and following the structure of the handout skeleton files.

Make sure that you follow the format specifications (PDF and .ZIP). If you don't, the hand in **will not be assessed** and treated as a blank hand in. The hand in is done via the Digital Exam system (`eksamen.ku.dk`).

Learning Objectives

To get a passing grade you must demonstrate that you are both able to program a solution using the techniques taught in the course *and* write up your reflections and assessments of

your own work.

- For each question your report should give an overview of your solution, **including an assessment** of how good you think your solution is and on which grounds you base your assessment. Likewise, it is important to document all *relevant* design decisions you have made.
- In your programming solutions emphasis should be on correctness, on demonstrating that your have understood the principles taught in the course, and on clear separation of concerns.
- It is important that you implement the required API, as your programs might be subjected to automated testing as part of the grading. Failure to implement the correct API may influence your grade.
- To get a passing grade, you *must* have some non-trivial working code in both Haskell and Erlang.

Exam Fraud

This is a strictly individual exam, thus you are **not** allowed to discuss any part of the exam with anyone on, or outside the course. Submitting answers (code and/or text) you have not written entirely by yourself, or *sharing your answers with others*, is considered exam fraud.

You are allowed to ask (*not answer*) how an exam question is to be interpreted on the course discussion forum on Absalon. That is, you may ask for official clarification of what constitutes a proper solution to one of the exam problems, if this seems either underspecified or inconsistently specified in the exam text. But note that this permission does not extend to discussion of any particular solution approaches or strategies, whether concrete or abstract.

This is an open-book exam, and so you are welcome to make use of any reading material from the course, or elsewhere. However, make sure to use proper and *specific* citations for any external material from which you draw considerable inspiration – including what you may find on the Internet, such as snippets of code. Similarly, if you reuse any significant amount of code from the course assignments *that you did not develop entirely on your own*, remember to clearly identify the extent of any such code by suitable comments in the source.

Also note that it is not allowed to copy any part of the exam text (or supplementary skeleton files) and publish it on forums other than the course discussion forum (e.g., StackOverflow, IRC, exam banks, chatrooms, or suchlike), whether during or after the exam, without explicit permission of the author(s).

During the exam period, students are not allowed to answer questions on the discussion forum; *only teachers and teaching assistants are allowed to answer questions.*

Breaches of the above policy will be handled in accordance with the Faculty of Science's disciplinary procedures.

Question 1: The Slush template language

A *template language* is a simple, domain-specific language for specifying how structured data values should be rendered in a human-readable form. The language may target plain-text outputs, or some kind of markup language, such as HTML.

The Slush template language is inspired by the Liquid framework, popular in Ruby, and with more-or-less faithful ports to other languages.[1] It is also similar to other template engines, such as Jinja for Python.

A Slush template is simply a text document interspersed with a number of *tags* that indicate when special processing is required. Tags are either *output tags* or *logic tags*. Output tags, delimited with "`{{ ··· }}`", are used for including data values into the generated document. For example, if the template document consists of the text:

```
Hello, {{user.first_name}}!
```

and the variable `user` is set to a record, with a field named `first_name` containing the string `"John"`, the template will be rendered as,

```
Hello, John!
```

Logic tags, delimited by "`{% ··· %}`", do not themselves produce output, but may be used to specify simple computations to be performed during rendering. For example:

```
{% if order.items %}You have ordered the following items:
{% for item in order.items %}
  * {{item.name}}{% if 2 <= item.count %} (quantity {{item.count}}){% endif %}
{% endfor %}
{% else %}You haven't ordered anything yet!
{% endif %}
```

If `order.items` contains a list of records, each of which has a `name` field (a string) and a `count` field (a number), this could produce the text:

```
You have ordered the following items:

 * Widget (quantity 2)

 * Small gadget
```

(Note that each order item is both preceded and followed by a newline, because the template text between the `for ··· endfor` tags contains two such characters.) The Slush language itself is completely agnostic about the underlying document format. For example, if we were generating HTML code rather than plain text, we could instead have written the template as:

[1] No prior knowledge of Liquid is required or expected for this exam. If you are familiar with that framework, however, do note that despite the evident similarities, both the syntax and semantics of Slush may differ subtly from that of Liquid.

```
You have ordered the following items:
<table border="1">
  <tr> <th>Item name</th> <th>Quantity</th> </tr>
  {% for item in order.items %}
  <tr>
     <td>{{item.name}}</td>
     <td align="right">{{item.count}}</td>
  </tr>
  {% endfor %}
</table>
```

Using the Slush template language from a program involves two steps: first, the template document is *parsed* into an internal representation, which operation may fail with a parse error (such as mismatching tags). If the parse succeeds, the parsed template – together with any data values to potentially insert – may be *rendered* as the final document text, or possibly a rendering error (say, if an output tag refers to an undefined variable).

The two parts of this question concern, respectively, parsing and rendering of Slush templates. They are weighted approximately equally, but you should aim at answering both at least partially, rather than trying to get every detail right in just one of them. For your code submission, please use the provided skeletons under `code/slush/src/`, and in particular, modify only the files `ParserImpl.hs` and `RendererImpl.hs`. Put your testing code in separate module(s) under `code/slush/tests/`, importing from `Parser`, `ParserImpl`, `Renderer`, and `RendererImpl` as appropriate; and remember to explain in the report how to run your tests (if not by the normal "`stack test`").

We have included some sample Slush code (both raw and parsed) and corresponding data in `code/slush/examples/`, as well as a simple main program (under `code/slush/app/`) to invoke the parser, renderer, or both. For example, after a "`stack build`", the command

```
stack run -- -b examples/sample.slush examples/data.hs
```

will parse the template in `sample.slush` and render it using the data values given in `data.hs`. (In normal use of the `Renderer` module, the data for rendering will have been freshly computed by a Haskell program, not simply read from a file.)

Question 1.1: Parsing Slush

Formally, a Slush template follows the grammar in Figure 1.
The syntactic grammar is supplemented with the following specifications:

Lexical specifications

- A *literal* is any *non-empty* sequence of characters other than "`{`". Additionally, so that literals can also contain left braces, the two-character sequence "`{ `" (i.e., a left brace followed immediately by a single space, but not any other whitespace

Template	::=	ε
	|	*Frag* *Template*
Frag	::=	*literal*
	|	*Output*
	|	*Assignment*
	|	*Conditional*
	|	*Iteration*
	|	*Capture*
Output	::=	'{{' *Exp* '}}'
Assignment	::=	'{%' 'assign' *Var* '=' *Exp* '%}'
Conditional	::=	'{%' 'if' *Exp* '%}' *Template* *CondRest* '{%' 'endif' '%}'
CondRest	::=	ε
	|	'{%' 'else' '%}' *Template*
	|	'{%' 'elsif' *Exp* '%}' *Template* *CondRest*
Iteration	::=	'{%' 'for' *Var* 'in' *Exp* '%}' *Template* '{%' 'endfor' '%}'
Capture	::=	'{%' 'capture' *Var* '%}' *Template* '{%' 'endcapture' '%}'
Exp	::=	*Var*
	|	*Numeral*
	|	*Exp* '.' *Field*
	|	*Exp* '+' *Exp*
	|	*Exp* '<=' *Exp*
	|	'(' *Exp* ')'
Var	::=	*ident*
Field	::=	*ident*
literal	::=	(see text)
ident	::=	(see text)
numeral	::=	(see text)

Figure 1: The grammar of Slush

character) may be used to represent a verbatim character `'{'`. For example, the input text "`a{␣b{␣{␣␣c}␣d`" should be parsed as the Haskell string `"a{b{{␣c}␣d"` (where we have used ␣ to indicate individual space characters).

- An *ident* is any non-empty sequence of ASCII letters, digits, and underscores, starting with a letter. There are no reserved identifiers.
- A *numeral* is any non-empty sequence of decimal digits, optionally preceded by a negative sign. (Don't worry about overflowing `Int`s.)

Whitespace handling

- In *literal* text, all whitespace is significant, and should be included verbatim in the parse result (except for the space character following a literal "`{`").
- Inside tags (i.e., between "`{{ ··· }}`" or "`{% ··· %}`"), the grammar elements may be surrounded by zero or more whitespace characters (spaces, tabs, and newlines), which are ignored. However, at least one whitespace character is needed between alphabetic keywords and adjacent letters, digits, or underscores; for example, "`{% ifx <= 10 %}`" is *not* a well formed tag.

Disambiguation

- *literal* text should extend as far as possible. In other words, a *Template* must not contain two consecutive *literal Frag*s.
- '`.`' groups tighter than '`+`', which again groups tighter than '`<=`'. For example, "`x<=y+z.u`" parses like "`x<=(y+(z.u))`".
- The operator '`+`' is left-associative (as usual); e.g., "`x+3+y`" parses like "`(x+3)+y`", not like "`x+(3+y)`". '`<=`' is non-associative; e.g., "`x<=y<=z`" is a syntax error, as distinct from both "`x<=(y<=z)`" and "`(x<=y)<=z`".

The following Haskell module defines the AST of Slush:

```haskell
module Ast where

type Template = [Frag]

data Frag =
    TLit String
  | TOutput Exp
  | TAssign Var Exp
  | TIf Exp Template Template
  | TFor Var Exp Template
  | TCapture Var Template
  deriving (Eq, Show, Read)
```

```haskell
data Exp =
    EVar Var
  | ENum Int
  | EField Exp Field
  | EPlus Exp Exp
  | ELeq Exp Exp
  deriving (Eq, Show, Read)

type Var = Ident
type Field = Ident

type Ident = String
```

It should be self-evident how to represent all concrete-grammar elements in the abstract syntax, with the possible exception of conditionals, which should parse into chains of nested `TIf`s, with an empty *CondRest* equivalent to an `{% else %}` followed by an empty template. For example, both of the following lines

```
{% if x %} foo {% elsif y %} bar {% endif %} baz
{% if x %} foo {% elsif y %} bar {% else %}{% endif %} baz
```

should parse as the following `Template` value:

```haskell
  [TIf (EVar "x")
       [TLit " foo "]
       [TIf (EVar "y") [TLit " bar "] []],
   TLit " baz\n"]
```

Define (explicitly or as a synonym) a type `ParseErr` (which must be an instance of `Eq` and `Show`), suitable for reporting errors arising during parsing. Then define a top-level parsing function:

```haskell
parseString :: String -> Either ParseErr Template
```

where a result of `Right` *t* represents a successful parse of the input string as the template *t*, while `Left` *e* represents a (possibly generic) parsing error detailed in *e*.

You must use one of the monadic parser libraries ReadP or Parsec to implement your parser. If you use Parsec, then only plain Parsec is allowed, namely the following submodules of `Text.Parsec`: `Prim`, `Char`, `Error`, `String`, and `Combinator` (or the compatibility modules in `Text.ParserCombinators.Parsec`); in particular, you are *disallowed* to use `Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`.

In your report, you should describe any transformations you made to the grammar to make it suitable for processing with your chosen parser library, and how you resolved any other significant parsing-related challenges you encountered. ***

Question 1.2: Rendering Slush

A Slush context is a finite mapping of identifiers (strings) to data values. Contexts are represented as association lists (pairs of keys and corresponding data values), where the keys are required to be distinct. A data value is either a number (integer), a string, a list of values, or a record, which itself contains a context mapping field names to values. This is expressed by the following Haskell module:

```haskell
module Data where

import Ast (Ident)

type Ctx = [(Ident, Value)]

data Value =
    N Int
  | S String
  | L [Value]
  | R Ctx
  deriving (Eq, Show, Read)
```

Templates are always rendered in a context, which keeps track of the current values of all variables, either pre-existing or introduced in the template. We will define the semantics of the template language by recursively specifying the meaning of each constructor in the AST.

An expression evaluates to a value, or signals an error. The meanings of the various expression forms are as follows:

- `EVar` x : returns the current value of variable x in the context. If x does not have a value, an error is signaled.
- `ENum` n : returns the number n.
- `EField` e x : if e evaluates to a record, and that record includes a field named x, returns the corresponding field value. Otherwise, signals an error.
- `EPlus` e_1 e_2 : if both e_1 and e_2 evaluate to numbers, returns their sum. In all other cases, signals an error.
- `ELeq` e_1 e_2 : if both e_1 and e_2 evaluate to numbers, returns 1 if the first number is less-than-or-equal-to the second, and 0 otherwise. In all other cases, signals an error.

Successful execution of a template may produce some output text, and/or modify the context; alternatively, it may signal an error. Template fragments are executed in sequence from left to right, and their outputs concatenated. The meanings of the individual fragments are as follows:

- `TLit` s : outputs the string s.

- `TOutput` e : evaluates expression e in the current context, and outputs the result. Strings are output as themselves; numbers are formatted in the standard way (i.e., as unpadded strings of one or more decimal digits, possibly preceded by a negative sign). Attempting to output a whole list or record value at once signals an error.
- `TAssign` x e : evaluates the expression e in the current context, and assigns the result to the variable x. If x already had a value, it is overwritten.
- `TIf` e t_1 t_2 : evaluates the expression e, and depending on whether the result is "truthy" or "falsy", executes t_1 or t_2, respectively. Truthy values are non-zero numbers, non-empty strings, and non-empty lists; zero, the empty string, and the empty list are considered falsy. Attempting to use a record as a truth value signals an error.
- `TFor` x e t : evaluates the expression e to a list, and executes the loop body t with x set to each of the list elements in turn. (t is allowed to assign to x, but any such assignments will not affect subsequent iterations.) After the loop, x should keep the last value it was set to during the loop; that is, x is *not* local to the loop body. If the list is empty, the loop body is not executed, and x is not modified. If e evaluates to anything other than a list value, the iteration construct signals an error.
- `TCapture` x t : executes the template t in the current context, and captures the output it produces as a string value, which is assigned to x. The capture construct itself does not produce any output (but any context modifications performed by t are kept). For example, to set variable `a` to the concatenation of the variables `b` and `c`, we can use `TCapture "a" [TOutput (EVar "b"), TOutput (EVar "c")]`.

You you may assume that your rendering functions will only be called with well-formed context values (i.e., with at most one binding for each variable or record field), and they should preserve this property whenever they modify the context.

Define (explicitly or as a synonym) a type `RenderErr` (which must be an instance of `Eq` and `Show`), suitable for reporting errors arising during rendering. Further, define (explicitly, or using standard monads/transformers from `Control.Monad` and its submodules) two monads `EvalM` and `ExecM`, with associated operations (such as `ask`, `modify`, etc), suitable for computation of expression evaluations and template executions, respectively.

Be careful about how any relevant Reader, Writer, State, and Error patterns are incorporated in the two monads. Justify your choices in the report. ***

Using these monads, define two functions:

```
eval :: Exp -> EvalM Value

exec :: Template -> ExecM ()
```

implementing the above-given specifications of expressions and templates.

Finally, using the above functions, define the top-level rendering function:

```
render :: Ctx -> Template -> Either RenderErr String
```

where a result of `Right` s represents a successful rendering of the template as the string s, while `Left` e represents a rendering error detailed in e.

General instructions

All your code should be put into the provided skeleton files under `code/slush/`, as indicated. In particular, the actual code for each module `Mod` should go into `src/ModImpl.hs`. Do not change any of the other files, especially `Ast.hs` and `Data.hs`, which contain the common type definitions presented above; nor should you modify the type signatures of the exported functions in the APIs. Doing so will likely break our automated tests, which may affect your grade. Be sure that your codebase builds against the provided `app/Main.hs` with `stack build`; if any parts of your code contain syntax or type errors, remember to comment them out for the submission. Note that `stack.yaml` specifies the LTS-18.13 version of the Haskell platform (including GHC 8.10.7) for the exam, which should avoid the stray dependency problems on Windows.

Place your main tests in `tests/BlackBox.hs`, so that they are runnable with `stack test`. As the name suggests, these should be black-box tests only, i.e., not depend on any internals of your own implementation, but test only against the API specifications given above. In particular, we may run *your* black-box tests on *our* sample implementations (correct and/or incorrect). If you want to directly test some of your non-exported auxiliary functions, and/or unspecified behaviors, you can put those in `tests/suite1/WhiteBox.hs` (and uncomment the suite in `package.yaml`).

In your report, you should describe your main design and implementation choices for each module in a separate (sub)section. Be sure to cover *at least* the specific points for each module asked about in the text above (and emphasized with *** in the margin), as well as anything else you consider significant. Low-level, technical details about the code should normally be given as comments in the source itself.

For the assessment, we recommend that you use the same (sub)headings as in the weekly Haskell assignments (Completeness, Correctness, Efficiency, Robustness, Maintainability, Other). Feel free to skip aspects about which you have nothing significant to say. You may assess each module separately, or make a joint assessment in which you assess each aspect for both modules (but then be clear about which module you are referring to, when discussing any specific issues).

Question 2: Observables

In a distributed system we often set up a network of processes for broadcasting (messages about) events. Such a network consists of *observable servers*.

Terminology

An observable server can have a number of subscribers associated, and can act as a *publisher* where you ask it to *publish* an event for its subscribers. Likewise an observable server can act as a *subscriber* where it subscribes for events from a publisher (which it will then re-publish to its own subscribers). When a subscriber subscribes to a publisher, it specifies a *filter* indicating which events it is interested in to get, and can also specify a *limit* on how many events it want to receive. An *event* can be any Erlang term, we sometimes say this is the *value* of the event. A *filter* is a function that takes one argument, an event, and returns a boolean.

A *published event* may reach an observable server several times via different paths, however it should only re-publish an event once to its subscribers.

The following type declarations capture some of the terminology:

```
-type event()  :: term().
-type limit()  :: pos_integer() | infinity.
-type filter() :: fun((event()) -> boolean()).
```

Question 2.1: The `observable` module

Implement a module called `observable` for starting and interacting with observable servers in Erlang. Your module should have the following client API, where `P` and `S` are observable servers:

(a) `new()` for starting a new observable server. Returns `{ok, P}` on success where `P` is the new observable server, or `{error, Reason}` if some fault occurred.

(b) `add_subscriber(P, S, Filt, Lim)` for adding the subscriber `S` to `P`, with the filter `Filt`. The observable server `P` should only send an event `E` to `S` if `Filt(E)` is true. `Lim` is the limit for how many messages `S` should receive from `P`; `Lim` should either be the atom `infinity` if there is no upper limit on the number of messages, or it should be a positive integer (not zero). When the limit has been reached, `S` is unsubscribed from `P`.

The function `add_subscriber` should return `ok` if `S` is not already subscribed to `P`; or `{error, Reason}` if `S` is already subscribed, if `Lim` does not have the right form, or if some other error occurred.

(c) `subscribers(P)` for getting the subscribers for `P`. Returns `{ok, Subs}` on success where `Subs` is the list of subscribers for `P`, or `{error, Reason}` if some fault occurred. The order of elements in `Subs` is unspecified.

(d) `publish(P, E)` that sends the event `E` to `P`, after which `P` will publish the event to all of its relevant subscribers (who might in turn publish it to their subscribers).

This function should *not block*. Hence, the function can return before the event has reached all subscribes.

An observable server should only (re-)publish an event to its subscribers once. To achieve this, you want to give each event an internal unique identifier, and for this you might want to use the build-in function `make_ref`.

(e) `events(P)` for getting the events received by `P`. Returns `{ok, Evs}` on success where `Evs` is a list of events, or `{error, Reason}` if some fault occurred. The order of elements in `Evs` is unspecified.

Note that `Evs` should only contain *one* instance of a published event, even if it has reached `P` twice or more by different routes in the network. Also note that two published events can have the same value.

All your functions should have `-spec` annotations.

Using `observable`

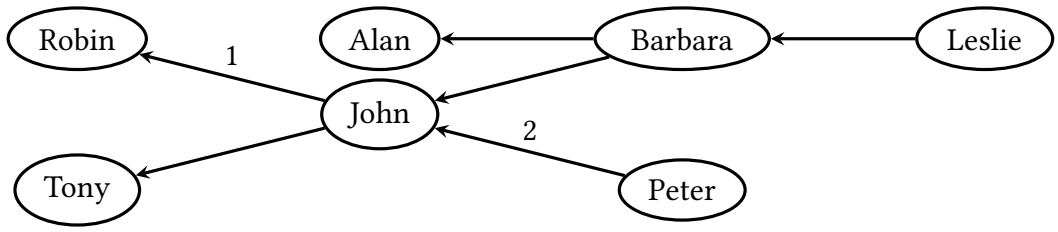

Figure 2: Network of observable servers

To demonstrate that your solution works write a test module `twinners` that exports at least three functions:

- The function `setup()` that sets up the network of observable servers as specified in the graph show in Figure 2, where an arrow from a server *A* to another server *B* means that *A* subscribes to *B*; Barbara subscribes to John, for instance. If there is a number above the arrow, that is the limit for the subscription, otherwise the limit is `infinity`. There are only two non-trivial filters: John only wants to receive events that are even integers from Robin; and Barbara only wants to receive events from John that are lists where the second element is the character `$e`. All other filters are the trivial filter that allows all events.

  The function should return the list of the observable servers Tony, Robin, John, Alan, Peter, Barbara, and Leslie in that order.

- The function `send_events(X, Y, W, Z)` that publishes the following events: the integer `5`, the integer `4`, and the atom `point` from `X`; the string `"Hello"` from `Y`; the pair of atoms `{small,talk}` from `W`; and the atom `liskov` from `Z`.

- The function `test_both()`. This function should first calls `setup()`, then call `send_events(Robin, John, Alan, Barbara)`, and finally checks the actual results of calling `events` on Leslie and Peter against the expected ones. The function should return `true` if the test is successful and `false` otherwise.

Question 2.2: Testing `observable`

Make a module `test_observable` for testing a `observable` module. We evaluate your tests with various versions of the `observable` module that contains different planted bugs and checks that your tests find the planted bugs. Thus, your tests should only rely on the API described in the exam text.

Your `test_observable` module should contain, at least, the following:

- A `test_all/0` function that runs all your tests and only depends on the specified `observable` API.
- We also evaluate your tests on your own implementation, for that you should export a `test_everything/0` function (that could just call `test_all/0`).

You may want to put your tests in multiple files (especially if you use both `eqc` and `eunit` as they both define a `?LET` macro, for instance). If you use multiple files, they must all start with the prefix `test_` (or `apqc_` if you use that). Remember that the code called from `test_all/0` must only depend on the specified `observable` API and your (if any) `test_` modules.

Topics for your report for Question 2

Your report should document:

- For Question 2.1:
  - Clearly explain if you support the complete API or not. If not, which parts you don't support.
  - If you have made any assumptions, especially simplifying assumptions. In particular if you have made any assumptions about filter functions.
  - How many processes your solution uses, what role each process has, and how they communicate.
  - What data your processes maintain.
  - One of the most interesting part of Question 2.1 is that we allow arbitrary functions as filter. Because we allow arbitrary functions as filters, the observable servers should be robust against programming mistakes in filters. Explain what assumptions you have made about filters, and what you have done to make your implementation robust.
- For Question 2.2:

- Clearly explain how you have ensured that you test all parts of your code, and explain your overall strategy for testing.
- The quality or limitations of your testing.
- Explain which of your tests are qualitative different from the minimal testing required by the `twinners` module. Here a relevant topic is the shape of the network graphs you have used for testing.