Take-home Re-exam in Advanced Programming

Deadline: Monday, January 26, 16:00

Version 1.1

Preamble

This is the exam set for the individual, written take-home re-exam on the course Advanced Programming, B1-2014. This document consists of 12 pages; make sure you have them all. Please read the entire preamble carefully.

The exam set consists of 3 questions. Your solution will be graded as a whole, on the 7-point grading scale, with an external examiner.

In the event of errors or ambiguities in the exam set, you are expected to state your assumptions as to the intended meaning in your report. You may ask for clarifications in the discussion forum, 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 re-exam you must hand in both a report and your source code:

- *The report* should be around 5–10 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, archiving one directory called `src` (which may contain further subdirectories).

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 course web page on Absalon.

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 working code for all questions.

Exam Fraud

The exam is an individual exam, thus you are **not** allowed to discuss any part of the exam with anyone on, or outside the course.

You are only allowed to discuss how a question is to be interpreted with the teachers and their assitants on the discussion forum set up on the course-page on Absalon. If you are afraid that your question does not fall into this category, you can instead send an email to `kflarsen@diku.dk`.

Specifically, but not exclusively, you are **not** allowed to discuss any part of the exam with any other student nor to copy parts of other students' programs. Submitting answers you have not written yourself, or *sharing your answers with others*, is considered exam fraud.

This is an open-book exam, and so you are welcome to make use of any reading material from the course, or elsewhere. Make sure that you use proper academic citation for the material you draw considerable inspiration from (including what you may find on the Internet, e.g., pieces of code). Also note, that these rules mean that it is not allowed to copy any part of the exam set (or supplementary skeleton files) and publish it on forums other than the course discussion forum (e.g. IRC, exam banks, chatrooms, or suchlike).

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

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

Emergency Webpage

There is an emergency web page at

`http://www.diku.dk/~kflarsen/ap-e2014/`

in case Absalon becomes unstable. The page will describe what to do if Absalon becomes unreachable during the re-exam, especially what to do at the hand-in deadline.

Question 1: Parsing ANTARESIA

ANTARESIA is sub-set of the programming language PYTHON containing declaration of names and have list comprehensions.

In this question you must use one of the three monadic parser libraries, `SimpleParse.hs`, `ReadP` or Parsec, from the course to write a parser for ANTARESIA. The parser must be implemented in a `AntaParser` module. You find Haskell skeletons for the parser and abstract syntax tree on Absalon.

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`), in particular you are *disallowed* to use `Text.Parsec.Token`, `Text.Parsec.Language`, and `Text.Parsec.Expr`.

The grammar for ANTARESIA is given on the following page. Furthermore,

- ϵ is the empty sequence.
- *integer* is an arbitrary-precision integer constant.
- *Name* is a nonempty sequence of letters, digits, and underscores (_), starting with a letter, that is not a reserved word.
- The reserved words are `True`, `False`, `range`, `for`, `if`, `in`, and `not`. VER. 1.1
- The arithmetic operators `*`, `%` and `//` have highest priority, followed by the arithmetic operators `+` and `-`, which have higher precedence than the relational operator `==`, which has higher precedence than the relational membership operators `in` and `not in`. The five arithmetic operators are left-associative, and the three relational operators are right-associative.

Alpha-numeric tokens (*Name*s and reserved words) are separated by at least one whitespace (spaces, tabs, and newlines), symbolic tokens are separated by arbitrary whitespace.

It is the intention that a syntactically correct ANTARESIA program is also a syntactically correct PYTHON program (but because PYTHON is whitespace sensitive and ANTARESIA is not, there might be some corner cases where it is not the case).

In your report, make sure to detail all transformation(s) of the grammar you make as part of your parser construction.

Advice for your solution

If you have difficulties making your parser work for the full language, then try to make it work for a simpler subset of the language. If you make such restrictions make sure to clearly documenting them in your assessment, and explain why the disallowed language constructs cause you problems.

Grammar

```
    Program  ::= Decls
      Decls  ::= ϵ
              |  Decl Decls
       Decl  ::= Name '=' Expr
      Args3  ::= Expr
              |  Expr ',' Expr
              |  Expr ',' Expr ',' Expr
      Exprs  ::= ϵ
              |  Expr CommaExprs
 CommaExprs  ::= ϵ
              |  ',' Expr CommaExprs
       Expr  ::= integer
              |  'True'
              |  'False'
              |  Name
              |  'range' '(' Args3 ')'
              |  Expr '+' Expr
              |  Expr '-' Expr
              |  Expr '*' Expr
              |  Expr '//' Expr
              |  Expr '%' Expr
              |  Expr '==' Expr
              |  Expr 'in' Expr
              |  Expr 'not' 'in' Expr
              |  '[' ListComp ']'
              |  '(' Expr ')'
   ListComp  ::= Exprs
              |  Expr ListFor
    ListFor  ::= 'for' Name 'in' Expr ListIter
   ListIter  ::= ϵ
              |  ListFor
              |  'if' Expr ListIter
```

Abstract syntax trees

Your parser must construct abstract syntax trees represented with the data types defined in `AntaAST.hs`. The mapping from grammar to constructors should be straightforward.

Thus, you should implement module `AntaParser` with the following interface: a function `parseString` for parsing a Antaresia program given as a string:

```
parseString :: String -> Either Error Program
```

Where you decide and specify what the type `Error` should be. The type `Error` must also be exported from the module. The handed-out skeleton code already has the exports set up correctly.

Likewise, you should implement a function `parseFile` for parsing an Antaresia program given in a file located at a given path:

```
parseFile :: FilePath -> IO (Either Error Program)
```

Where `Error` is the same type as for `parseString`.

You should not change the types for the abstract syntax trees unless there is an update on Absalon telling you explicitly that you can do so.

Together with your parser you must also hand in a test-suite to show that your parser works (or where it does not work).

Question 2: Interpreting ANTARESIA

This question is about writing an interpreter for the ANTARESIA language defined in Question 1. It is the intention that the semantics of an syntactical correct ANTARESIA is the same as it would be as a PYTHON program (since all syntactical correct ANTARESIA programs should also be syntactical correct PYTHON programs) up to a few simplifications with respect to type coercions, which is detailed below.

We recommend that you read through the whole question before you start implementing anything, and read the commented example in Appendix A, which also contains abstract syntax trees if your parser is not completely working.

Semantics of ANTARESIA

The semantics of most of ANTARESIA should be straightforward, below some of the more sticky points are elaborated.

- Names can only be refereed to after they have been declared (thus there are no recursive declarations).

- Names can be redeclared and will shadow earlier declarations.

- In contrast to PYTHON there are no type coercions in ANTARESIA. Thus, it is illegal to, say, multiply an integer and a list.

  Both arguments to arithmetic operators must be integers. Likewise, both arguments to the == operator should have the same type.

  As ANTARESIA computations are dynamically typed there are several ways type errors can occur. From the description of the syntax it is possible to infer what these type error conditions are. If you are in doubt document your interpretation succinctly in your report.

- The operator `//` is integer division. It is an error to divide or take modulus by zero.

- The build-in function `range(start, stop, step)` is used for generating lists. The arguments `start`, `stop`, and `step` must be integers. The full form returns a list of integers [`start`, `start` + `step`, `start` + 2 ∗ `step`, . . .]. If `step` is positive, the last element is the largest `start` + i ∗ `step` less than `stop`; if `step` is negative, the last element is the smallest `start` + i ∗ `step` greater than `stop`. `step` must not be zero (or else it is an error similar to dividing by zero).

  If only two arguments are given `step` is the value one, and if only one argument is given `start` is the value zero and `step` is the value one.

- A list comprehension

  [e_1 `for` n `in` e_2 *iter*]

  consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the list are those that would be produced by evaluation e_1 in environments with n bound to each element in e_2 (thus, e_2 must be a list) plus the bindings stemming from *iter*.

Just like the Haskell expression

$[e_1$ `|` n `<-` e_2`,` $iter_H]$

where each `if` clause in *iter* is a boolean expression in $iter_H$.

For example, the following ANTARESIA program (part of the example program in Appendix A):

```
noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
primes = [x for x in range(2, 50) if x not in noprimes]
```

means the same as the following Haskell program

```
noprimes = [j | i <- [2..(8-1)], j <- [i*2, i*2+i .. (50-1)]]
primes = [x | x <- [2..(50-1)], not (x `elem` noprimes)]
```

Note that the bindings from `for` clauses are only in scope in the list comprehension nested left to right. For example, the binding of n is available in `iter` (and in e_1), but the binding from *iter* is not available in e_2 (but they are in e_1).

- The result of evaluation an ANTARESIA expression is a value which is either an integer, a boolean, or a list of values. We represent values by the following Haskell data type:

  ```
  data Value = IntVal Integer
             | TrueVal | FalseVal
             | List [Value]
             deriving (Eq, Show)
  ```

  which is declared in `AntaAST.hs`. Thus, the result of evaluating an ANTARESIA program is a list of names and values, the type `Result`:

  ```
  type Result = [(Name, Value)]
  ```

  also declared in `AntaAST.hs`.

If your interpreter encounter an error, it should terminate with an well-defined error type. That is, **not** by calling the built-in Haskell function `error`.

Your Task

The main objective of this question is that you should demonstrate that you know how to write an interpreter using monads for structuring your code. Thus you should structure your solution along the following lines:

(a) Define a module `AntaInterpreter` that exports a function `runProg` and a type `Error`.

(b) Define a function `runProg`

```
runProg :: Program -> Either Error Result
```

`runProg` *p* runs program *p*, yielding either a runtime error, or the result of the program (as described previously). You will need to define the type `Error` as well.

(c) See the handed-out `AntaInterpreter.hs` for a *strongly recommended* skeleton for your solution. This file consists of incomplete and commented-out definitions – the latter are not part of the core skeleton, but are provided as guidance and inspiration for your own helper functions.

Your solution should implemented as module `AntaInterpreter` that exports at least `Error` and `runProg`. The handed-out skeleton code already has the exports set up correctly.

Once you have implemented the parser and interpreter, the file `Anta.hs` can be used to run Antaresia programs, as follows.

```
% runhaskell Anta.hs program.at
```

You should *not* need to modify `Anta.hs`.

Advice for your solution

If you have difficulties making your interpreter work for the full Antaresia language, then try to make it work for a simpler subset of the language. For instance, by allowing only one `for` clause in list comprehensions, allowing only `range` with positive a step argument, or disallowing `if` clause in list comprehensions, and so on. If you make such restrictions make sure to clearly documenting them in your assessment, and explain why the disallowed language constructs cause you problems.

Question 3: Ant Colony

The task in this question is to create a framework in Erlang for controlling a colony of ants on a big canvas, where each ant has a small pen on its tail. When the pen is down, it draws on the canvas.

We use the following terminology:

- A picture is represented as a list of line segments:

  ```
  Position = {integer(), integer()}
  LineSeg  = {Position, Position}
  Picture  = [LineSeg]
  ```

  The order of segments does not matter, thus two pictures are equal if they contains the same segments. (However, two pictures may be visually similar, but are not equal, if one of the pictures contains duplicate segments, for instance.)

- An ant has a position, a boolean specifying whether the pen is down, and an angle in degrees specifying which direction it is pointing. Degrees are one of the integers 0, 90, 180, or 270, where zero points east and rotation is counterclockwise so north is 90.

- A colony consists of a set of ants, and each ant belong to exactly one colony.

- The framework should be able to handle concurrent colonies, and the colonies should be completely independent.

Implement a module `anttalk` for working with colonies and ants. The client API of the module can be divided into a colony-related part and an ant-related part.

The ant-related part of the API is the following functions, where *A* is always an ant ID:

(a) A function `forward(`*A*`, `*N*`)` to move an ant forward. For example, if the ant is at position `{`*X*`, `*Y*`}` with angle 90 and it is moved forward *N* paces, then the new position is `{`*X*, *Y* + *N*`}`, and likewise for the other angles. Note that *N* must be an integer greater than or equal to zero, and the function should check this.

(b) The functions `left(`*A*`, `*D*`)` and `right(`*A*`, `*D*`)`; where `left` pivots the ant *D* degrees counter-clockwise, and `right` pivots the ant *D* degrees clockwise. Note that *D* must be one of the integers 0, 90, 180, or 270, and the functions should check this.

(c) A function `setpen(`*A*`, `*P*`)`, where *P* is either `up` or `down`, for setting the state of the ant's pen.

(d) A function `clone(`*A*`, `*N*`)` for creating *N* new ants. The new ants belong to the same colony as the original ant, they start in the same position, and have the same angle, but with empty pictures. Note that *N* must be an integer greater than or equal to zero, and the function should check this.

    On success the function should return `{ok, `*AL*`}` where *AL* is the list with ant IDs of the *new* ants.

(e) A function `position(`*A*`)` for getting the position of an ant. The function should return `{ok,` *P*`}` if it succeeds, where *P* the position.

(f) The functions `forward`, `left`, `right`, and `setpen` should be asynchronous in the following sense: if the ant is alive the functions should return immediately and without error, even if wrong arguments are given. If the functions are called with wrong arguments the ant should die.

The colony-related part of the API is the following functions, where *C* is a colony ID:

(g) A function `start()` for starting a new colony process. The function should return `{ok,` *C*`}` if it succeeds.

(h) A function `blast(`*C*`)` for stopping a colony process and all its ants.

(i) A function `new_ant(`*C*`)` for starting a new ant process, with position `{0,0}` and angle 0. The function should return `{ok,` *A*`}` if it succeeds.

(j) A function `picture(`*C*`)` for getting the picture painted by all living ants on the canvas. The final picture is the concatenation of the individual pictures painted by the ants. The function should return `{ok,` *P*`}` if it succeeds.

(k) A function `ants(`*C*`)` for getting the ant IDs of all living ants and a count of dead ants. Thus, on success the function should return `{ok, {`*LA*`,` *N*`}}` where *LA* is the list of live ants and *N* is the count of dead ants.

(l) A function `graveyard(`*C*`)` for getting the picture painted by all dead ants who have been on the canvas, the pictures by currently livings ants should not be included. The final picture is the concatenation of the individual pictures painted by the ants.

(m) The framework must be robust at least in the following senses: the picture drawn by a ant should never get lost as long as the colony is alive (even if the ant dies), if one of the ant processes fails (and thus dies), the colony process should detect the failure, but the colony (and all other ants) should continue. No ant processes should survive if the colony process stops for one reason or another.

Note: The functions `picture`, `ants`, and `graveyard` are somewhat vulnerable to race conditions, you can ignore that aspect.

For visualising pictures you can use the function `pictureToSvg` in Appendix B that takes a `Picture` and create an SVG image that can be displayed by most browsers.

Demonstrate that your program works by making a systematic test suite, and explain what you test and why.

Appendix A: Squares and Primes

Appendix A.1: Antaresia program

```
S = [x*x for x in range(10)]
M = [x for x in S if x % 2 == 0]

noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
primes = [x for x in range(2, 50) if x not in noprimes]
```

Appendix A.2: Abstract Syntax Tree

```
[ ("S", ListComp (ListFor (Mult (Name "x") (Name "x"))
                 ("x",Range (A1 (IntConst 10)),Nothing)))
, ("M", ListComp (ListFor (Name "x")
                 ("x",Name "S",Just (ListIf (Equal (Modulus (Name "x") (IntConst 2))
                                                   (IntConst 0)) Nothing))))
, ("noprimes", ListComp (ListFor (Name "j")
                        ("i",Range (A2 (IntConst 2) (IntConst 8)),
                        Just (ListForIter ("j",
                                    Range (A3 (Mult (Name "i") (IntConst 2))
                                               (IntConst 50) (Name "i")),
                                    Nothing)))))
, ("primes", ListComp (ListFor (Name "x")
                      ("x",Range (A2 (IntConst 2) (IntConst 50)),
                      Just (ListIf (NotIn (Name "x") (Name "noprimes")) Nothing))))]
```

Appendix B: Erlang SVG Writer

```
pictureToSvg (Picture) ->
    Points = lists:append(tuple_to_list(lists:unzip(Picture))),
    {Minx,Maxy} = lists:foldl (fun({X1,Y1}, {X2,Y2}) ->
                                       {min(X1, X2), max(Y1, Y2)} end,
                               hd(Points), tl(Points)),
    Xdir = 1 + if Minx < 0 -> abs(Minx); true -> 0 end,
    Ydir = 1 + Maxy,
    Lines = lists:map(fun svgline/1, Picture),
    lists:flatten(["<svg xmlns=\"http://www.w3.org/2000/svg\">",
                   string_format("<g transform=\"translate(~B, ~B) scale(1,-1)\">~n",
                                 [Xdir, Ydir]),
                 Lines,
                 "</g></svg>"]).


svgline ({{X1,Y1}, {X2,Y2}}) ->
    string_format("<line style=\"stroke-width: 2px; stroke:black; fill:white\"\
      \ x1=\"~B\" x2=\"~B\" y1=\"~B\" y2=\"~B\" />~n" , [X1, X2, Y1, Y2]).

string_format(S, L) ->
    lists:flatten(io_lib:format(S, L)).
```