Take-home Exam in Advanced Programming

Deadline: Friday, November 10, 16:00

Version 1.0

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2017. This document consists of 20 pages; make sure you have them all. Please read the entire preamble carefully.

The exam consists of 2 questions. Your solution will be graded as a whole, on the 7-point grading scale, with an external examiner. The weight between the two questions is that Question 1 counts for 60% and Question 2 counts for 40%. However, note that you must have both some working Haskell code and some working 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 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 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 constitues 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 *specfic* academic citation for any material from which you draw considerable inspiration – including what you may find on the Internet, such as snippets of code. 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: APLE: A simple computer algebra system

Computer algebra system (CASs), such as MATHEMATICA or MAPLE, are domain specific languages for manipulating – often interactively – mathematical expressions, for purposes such as algebraic simplification or symbolic differentiation.

Modern CASs actually have little, if any, knowledge of common mathematical syntax or semantics hard-wired into their core program code. Instead, they are based on relatively general-purpose term-rewriting engines that can be instantiated to many different tasks. In particular, such engines support repeatedly applying simple equations from algebra and calculus to transform a term into a desired form. As a special case, this rewriting can include just evaluating a closed arithmetic expression to a numeric result (like in a functional language), but CASs can also work meaningfully with terms containing unbound variables. In this question, you will implement a simple CAS called APLE (pronounced like "maple", but without the initial 'm').

Informal presentation of APLE

An APLE term is either a variable, an (unbounded) integer, or a function symbol applied to a list of arguments. Some binary functions may be optionally declared as infix operators with specified precedence and associativity conventions. For example, `f(x,y)+2` is a valid term.

APLE rewriting rules are written as algebraic equations between terms, considered as oriented from left to right. For example, here are some possible simplification rules (file `tiny.ap`) governing addition and multiplication (assuming that `+` and `*` have been declared as left-associative infix operators):

```
1  0 + t = t.
2  t + 0 = t.
3  t1 + (t2 + t3) = (t1 + t2) + t3.
4  t + t = 2 * t.
5  0 * t = 0.
```

(The rule numbers in the margin are just for reference in the following.) Rewriting a term consists of repeatedly matching the left-hand side (LHS) of a rule against a part of the term (by suitably instantiating variables in the rule) and replacing that subterm with the right-hand side (RHS) of the rule (with the same instantiation applied).

For example, with the previous five rules, we can pose a query to simplify an expression:

```
> 3 * (x + (0 + y)) ?
3*(x+y)
```

Note that this simplification can be achieved either by applying rule 1 to the subexpression `0 + y` (instantiating `t` in the rule as `y`), or by first applying rule 3 (with `t1` taken as `x`, `t2` as `0`, and `t3` as `y`), and then rule 2 (with `t` as `x`).

In general, especially for complex rule sets, simplifying an expression may give different results depending on the strategy used for choosing which rules to apply, and where. To see how a result was derived, we can request *verbose* output from the query:

```
> (0+x)+(x+0) ??
0+x+(x+0) =
0+x+x+0 =
0+x+x =
x+x =
2*x
```

(Note that terms are output without redundant parentheses, given that `+` is declared as left-associative; but rules only match according to the underlying tree structure of the term. For example, we could *not* use rule 4 to rewrite the term `0+x+x+0` to `0+2*x+0`, because the term is actually `((0+x)+x)+0`, which does not have `x+x` as a subterm.)

It is useful to think of the term on the LHS of a rule as a *pattern* in a functional language. However, unlike in Haskell (but like in Erlang or Prolog), the pattern may contain multiple occurrences of the same variable, in which case the rule will only match if all occurrences are instantiated to identical terms. For example, rule 4 above would rewrite `x*y + x*y` to `2*(x*y)`, but it it would not match, e.g., `x+y` or `(x+1)+(1+x)`.

Also, like in a functional language, all variables occurring on the RHS of the rule must be bound by the rule; otherwise an error is reported:

```
> f(x) = y+1.
> f(z)?
f(z)
Error: Unbound var: y
```

Sometimes we only want to apply a rule in particular circumstances. For example, if we want to move numeric constants to the end of an expression, we might add a *conditional* rule,

```
n + t = t + n | num(n).
```

This says that we should only swap the arguments of an addition when the rule variable `n` is instantiated as a numeric constant (as checked by the built-in *predicate* `num`), while `t` can be an arbitrary term. (Note that this rule will still result in infinite rewriting on a term like `3 + 4`.) A more general form of conditional rules also allows variables to be bound by computation:

```
n1 + n2 = n3 | num(n1), num(n2), add(n1, n2; n3).
```

This says that we can constant-fold a `+`-expression: the built-in predicate `add` binds its third argument to the arithmetic sum of the first two, which must be numerals. (The semicolon separates input and output arguments to the predicate.) Adding this rule *before* the commutativity one will ensure that 3+4 gets rewritten to 7, rather than to 4+3.

Finally, as demonstrated previously, it is possible to accidentally specify rule sets that will lead to infinite rewriting in some cases. APLE has two mechanisms for coping with this. First, if successive rewriting steps ever reach a term seen before, further rewriting would be pointless since (APLE's strategy being deterministic), we'd just go into an infinite loop. So, with the rules above (except the constant folding), we'd get:

```
> 3 + (4 + y) ??
3+(4+y) =
3+4+y =
4+3+y =
4+3+y
Error: Loop
```

Second, any rewriting sequence will stop after a configurable number of steps. (For simplicity, this is just a constant `maxSteps` in `AST`; a more general approach would be to include it in `GEnv` so that it could be changed interactively.) For example,

```
> t(x) = t(x+1) | num(x).
> t(5)?
t(505)
Error: Too many steps
```

Note that each recursive call involves two rewriting steps: `t(5) = t(5+1) = t(6) = t(6+1) = ...`. That is why we stop at 505, not at 1005.

A larger set of rules can be seen in Figure 1. With these definitions, a sample interaction could be:

```
> mypoly(x,y)=(x+y)**3.
> mypoly(3,4)?
343
> D(a,mypoly(a,b))?
a*a+a*b+a*a+a*b+b*a+b*b+b*a+b*b+a*a+a*b+b*a+b*b
```

With this introduction, we now present the details of the assignment.

Part 1: Syntax of APLE

The full grammar of APLE is shown in Figure 2

This grammar can be parsed into the following AST:

```
module AST where

type ErrMsg = String  -- human-readable error messages
type VName = String   -- variable names
type FName = String   -- function (including operator) names
```

```
n1 + n2 = n3 | num(n1), num(n2), add(n1,n2;n3).
n1 * n2 = n3 | num(n1), num(n2), mul(n1,n2;n3).

0 + t = t.
t + 0 = t.
t1 + (t2 + t3) = t1 + t2 + t3.

t1 - t2 = t1 + ~1 * t2.

0 * t = 0.
1 * t = t.
(t1 + t2) * t3 = t1 * t3 + t2 * t3.

t * 0 = 0.
t * 1 = t.
t1 * (t2 + t3) = t1 * t2 + t1 * t3.

t ** 0 = 1.
t ** n = t * t ** (n + ~1) | num(n).

D(x,n) = 0 | num(n).

D(x,x) = 1.
D(x,y) = 0 | var(y), lexless(x,y).
D(x,y) = 0 | var(y), lexless(y,x).

D(x,t1+t2) = D(x,t1) + D(x,t2).
D(x,t1*t2) = t1*D(x,t2) + t2*D(x,t1).
```

Figure 1: Sample rule collection, `rules.ap`

$$
\begin{array}{rcl}
\mathit{Term} & ::= & \mathit{vname} \\
& | & \mathit{number} \\
& | & \mathit{fname}\ \text{`('}\ \mathit{Termz}\ \text{`)'} \\
& | & \mathit{Term}\ \ \mathit{oper}\ \ \mathit{Term} \\
& | & \text{`('}\ \mathit{Term}\ \text{`)'} \\
\mathit{Termz} & ::= & \epsilon \\
& | & \mathit{Terms} \\
\mathit{Terms} & ::= & \mathit{Term} \\
& | & \mathit{Term}\ \text{`,'}\ \mathit{Terms} \\
\mathit{Cond} & ::= & \mathit{pname}\ \text{`('}\ \mathit{Termz}\ \text{`)'} \\
& | & \mathit{pname}\ \text{`('}\ \mathit{Termz}\ \text{`;'}\ \mathit{Terms}\ \text{`)'} \\
\mathit{Conds} & ::= & \mathit{Cond} \\
& | & \mathit{Cond}\ \text{`,'}\ \mathit{Conds} \\
\mathit{Rule} & ::= & \mathit{Term}\ \text{`='}\ \mathit{Term}\ \text{`.'} \\
& | & \mathit{Term}\ \text{`='}\ \mathit{Term}\ \text{`|'}\ \mathit{Conds}\ \text{`.'} \\
\mathit{Cmd} & ::= & \mathit{Rule} \\
& | & \mathit{Term}\ \text{`?'} \\
& | & \mathit{Term}\ \text{`??'} \\
\mathit{Cmds} & ::= & \epsilon \\
& | & \mathit{Cmd}\ \ \mathit{Cmds}
\end{array}
$$

Nonterminals:

- *vname*, *fname*, *pname*: any non-empty sequence of letters and digits, starting with a letter. There are no reserved names.
- *number*: any non-empty sequence of decimal digits, optionally preceded (without intervening whitespace) by a tilde character (‘~’), representing a negative sign.
- *oper*: any non-empty sequence of characters from the set “`!@#+-*/\<>=`”, *except* for a single ‘=’ (which is reserved.

Whitespace:

- All tokens may be separated by arbitrary whitespace (spaces, tabs, and newlines).
- There are no comments.

Disambiguation: see text.

Figure 2: Grammar and lexical specification of APle

```haskell
type PName = String   -- predicate names

data Term =
    TVar VName
  | TNum Integer
  | TFun FName [Term]
  deriving (Eq, Ord, Show, Read)

data Cond = Cond PName [Term] [Term]
  deriving (Eq, Show, Read)

data Rule = Rule Term Term [Cond]
  deriving (Eq, Show, Read)

data Cmd =
    CRule Rule
  | CQuery Term Bool{-verbosity flag-}
  deriving (Eq, Show, Read)

data Fixity = FLeft | FRight | FNone
  deriving (Eq, Show, Read)

data OpTable = OpTable [(Fixity, [FName])]
  deriving (Eq, Show, Read)

-- the remaing definitions only relate to the Semantics part
data Rsp = Rsp [Term] (Maybe ErrMsg)
  deriving (Eq, Show)

maxSteps :: Int
maxSteps = 1000
```

Since APLE is a general-purpose system, none of the operators are hardcoded in the grammar. Rather, the available operators are specified in a separate *operator table*, such as `arithmetic.op`:

```
OpTable
  [(FNone, ["<=", "<"]),
   (FLeft, ["+", "-"]),
   (FLeft, ["*"]),
   (FRight, ["**"])]
```

This table lists all the operators, grouped by increasing order of precedence. To avoid ambiguities, we specify that all operators at a given precedence level have the same associativity (as in Haskell's `infixl`, `infixr`, and `infix` declarations). You may assume that the operator table is well formed, i.e., that all operator names are lexically valid, and that any operator name occurs at most once in the tale.

Question 1.1: Parsing

Implement a parser for APLE, in the file `ParserImpl.hs` The parser must provide the following top-level functions:

```
parseStringTerm :: OpTable -> String -> Either ErrMsg Term
parseStringCmds :: OpTable -> String -> Either ErrMsg [Cmd]
```

You may use Parsec or ReadP for 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`). In particular you are *disallowed* to use `Text.Parsec.Token` and `Text.Parsec.Expr`.

Hint If you cannot get the general `OpTable`-parameterized parser to work, make the exported parser functions check that their first argument is precisely the fixed table in `arithmetic.op`, and hard-code this collection of operators in your grammar. Be sure to document the restriction in your report.

In fact, writing such a hard-coded parser may be a useful first step towards producing a general one.

Question 1.2: Pretty-printing

Implement a pretty-printer for terms. The output should be a syntactically valid APLE *Term*. It should contain the minimal number of parentheses, and no whitespace. For example, with the standard operator table, the term

```
TFun "*" [TNum 3,
          TFun "+" [TFun "+" [TVar "x",
                              TFun "f" [TVar "y",
                                        TNum 4]],
                    TVar "z"]]
```

should pretty-print as the string "`3*(x+f(y,4)+z)`".

The printer, in `PrinterImpl.hs`, must provide the following function:

```
printTerm :: OpTable -> Term -> String
```

Hint If you cannot get the `OpTable`-parameterized pretty-printer to work, make a first version that simply includes enough parentheses in the output that operator precedences and associativities do not matter. As a second version, make one that handles the fixed set of operators from `arithmetic.op` correctly, and prints all others with redundant parentheses, so that the output is still parseable. You might find the version that prints many parentheses useful in your testing.

Question 1.3: QuickChecking the syntax handling

In addition to the normal unit testing of the Parser and Printer modules separately, you should devise one or more QuickCheck tests, in file `SyntaxQC.hs`, that verify some non-trivial properties of the parser and/or printer.

It is important that your quick-check tests are *black-box*: they should only refer to the above-mentioned three top-level functions, as exported by the `Syntax` module. That is, your tests would also work (and potentially find bugs in) alternative implementations of the `Syntax` API. In the report, discuss *briefly* what kind of errors your QuickCheck tests would and would not be likely to find in someone else's implementations of the Parser and Printer.

Part 2: Semantics of APLE

We elaborate on the intended behavior of APLE (beyond the previously given informal overview) in terms of concrete syntax for readability, but it is *not* a requirement that you have a functioning parser or printer to complete this part.

Rule-application terminology A rule consists of a LHS term, a RHS term, and zero or more conditions. We say that a rule *matches* a term if we can consistently bind the rule's variables to terms, such that the LHS instantiated according to those bindings becomes identical to the term we're trying to match. The rule *applies* to the term if it matches, and additionally all the rule conditions (if any) are satisfied (as detailed below). The *result* of the application is the RHS of the rule, with all variables instantiated according to the bindings. (It is an error for the RHS to contain unbound variables.)

Finally, a rule applies *inside* a term if it applies to a proper subterm of the term. In that case, the result of the whole application is the result of applying the rule to the subterm, placed into the subterm's original context, as in normal equational reasoning about algebraic expressions.

Rule selection For a given term, several rules may apply to it, or inside it. APLE specifies a deterministic strategy for where and how to apply rules in each rewriting step:

- If two potential rule applications are nested within each other the *outermost* is chosen. For example, using the rules from `tiny.ap` in the informal presentation, the term `3 + 0 * (0 + x)` is rewritten by rule 5 to `3 + 0` (and not by rule 1 to `3 + 0 * x`).
- If two potential rule applications are independent (non-nested), the *leftmost* is chosen. For example, the term `(3 + 0 * x) * (x + 0)` is rewritten by rule 5 to `(3 + 0) * (x + 0)` (and not by rule 2 to `(3 + 0 * x) + x`).
- If multiple rules apply at the same position in the term, the *first* one (in the order listed) is chosen. For example, the term `3 * (0 + 0)` is rewritten by rule 1 to `3 * 0` (and not by rule 4 to `3 * (2 * 0)`).

Rule conditions In a rule with conditions, the conditions are evaluated left-to-right, after any variable bindings arising from matching the LHS of the rule against the term. (It is an error for an input argument of a condition to contain unbound rule variables.) The conditions are all invocations of a collection of built-in predicates. A successful predicate invocation may further bind rule variables in the output arguments of the condition. Such bindings apply to any subsequent conditions in the list, as well as to the RHS of the rule.

The built-in predicates are:

- `num(t)` succeeds iff `t` is a numeric constant.
- `var(t)` succeeds iff `t` is a variable.
- `add(t1,t2;t3)` succeeds if `t1` and `t2` are numeric constants, and `t3` can be bound to (or already is) their sum. For example, `add(3,4;7)` simply succeeds; `add(3,4;x)` (where `x` is not already bound) succeeds while binding `x` to `7`; and `add(3,4,8)` fails. If `t1` and/or `t2` are not numbers, the predicate signals an error.
- `mul(t1,t2;t3)` is analogous to `add`, only computing the product of the numbers, instead of their sum.
- `lexless(t1,t2)` succeeds if the term `t1` is *lexicographically* less than `t2` (as defined by Haskell's automatically derived `Ord` class instance on the type `Term`). In particular, if `t1` and `t2` are both numbers, the lexicographic ordering coincides with the usual arithmetic one; and if they are both variables, the ordering coincides with the string ordering on the variable names.

Any attempted invocations of predicates other than the above, or with the wrong argument counts, signal an error.

Question 1.4: A rewriting engine for APLE

Your engine, in `SemanticsImpl.hs` should be organized as specified below. It is very important that you do not modify the types of any of the intermediate functions you are asked to implement, as they will be subjected to automated testing (as well as human inspection).

The `Global` monad and related functions

```
type GEnv = [Rule]
newtype Global a = Global {runGlobal :: GEnv -> Either (Maybe ErrMsg) a}
instance Monad Global where ...

getRules :: Global [Rule]
failS :: Global a
failH :: ErrMsg -> Global a
tryS :: Global a -> Global a -> Global a
```

The `Global` monad organizes the general rewriting process, not related to any particular rule application. The `GEnv` type contains the list of all currently available rewrite rules, accessible at any time with the `getRules` function.

We also formalize that the rewriting can fail, in one of two ways: (1) *soft* failures, signaled by `failS`, which represent recoverable conditions, such as a particular rule LHS not matching a term, or a rule condition not being satisfied; and (2) *hard* failures, signaled by `failH`, which indicate that the rewriting process has encountered an irrecoverable error condition, such as referencing an unbound variable or predicate, or a rewriting loop. Soft failures carry no further data, while hard failures come with an error message to be reported; this correspond to the `Maybe ErrMsg` type in the `Left` branch of `Global`.

The function `tryM m1 m2` runs `m1`; if that succeeds, its result is the result of the whole expression, and `m2` is not used. If `m1` signals a *soft* failure, `m2` is run instead. On the other hand, if `m1` signals a *hard* failure, then that will be returned, and `m2` is again not used.

Complete the instance declaration of `Global` as a `Monad` (as well as `Functor` and `Applicative`, as usual), and define the related functions. The remainder of your code should never invoke the `Global` term constructor directly, but only through the above-defined functions.

The `Local` monad and related functions

```
type LEnv = [(VName, Term)]
newtype Local a = Local {runLocal :: LEnv -> Global (a, LEnv)}
instance Monad Local where ...

inc :: Global a -> Local a
askVar :: VName -> Local Term
tellVar :: VName -> Term -> Local ()
```

The `Local` monad keeps track of variable bindings in the context of applying a single rule. The local environment `LEnv` is an extend-only association list (i.e., once a variable has been bound, it cannot be further modified). `inc m` views a Global computation `m` as a special case of a Local computation that simply does not access or modify the local environment. `askVar v` returns the current binding of `v`, or signals a *hard* failure (with a suitable error message) if the variable is unbound. `tellVar v t` binds `v` to `t`, if `v` is currently unbound. If `v` is already bound to `t`, `tellVar` does nothing; whereas if `v` is bound to some term other than `t`, `tellVar` signals a *soft* failure.

Again, the remainder of your code should not use the `Local` term constructor directly, but only the above-defined functions.

Matching and instantiation

```
matchTerm :: Term -> Term -> Local ()
instTerm :: Term -> Local Term
```

The function `matchTerm p t` attempts to match the subject term `t` against the pattern term `p`, potentially binding variables in `p` to the corresponding subterms of `t` (but not the other

way around). If the match fails (possibly because of already existing conflicting variable bindings), `matchTerm` signals a *soft* failure.

`instTerm t` replaces all variables in `t` with their current values from the local environment. If `t` contains an unbound variable, a *hard* failure is signaled.

Conditions and rule application

```
evalCond :: PName -> [Term] -> Global [Term]
applyRule :: Rule -> Term -> Global Term
```

`evalCond pn ts` evaluates the predicate `pn` on the input arguments `ts` (which are assumed to have already been instantiated according to the local environment). If the predicate succeeds, it returns the values of its output arguments (if any), to be matched against the pattern terms (usually just variables) in the predicate invocation. If the predicate does not hold for the input arguments, it signals a *soft* failure. On the other hand, if the predicate is not defined, or invoked on illegal arguments, `evalCond` signals a *hard* failure.

`applyRule r t` attempts to apply the rewrite rule `r` to the term `t`. If this is not possible (e.g., because the rule LHS does not match `t`, or because one or more conditions in the rule is not satisfied), `applyRule` signals a *soft* failure. On the other hand, if attempting to apply the rule leads to a *hard* failure (e.g., because the rule RHS contains an unbound variable, or a condition mis-invokes a predicate), `applyRule` fails likewise.

Single-step term rewriting

```
rewriteTerm :: Term -> Global Term
```

`rewriteTerm t` attempts to rewrite the term once, by applying a rule anywhere within `t` according to the strategy defined for APLE (i.e., outermost, leftmost, first). If no rule applies, `rewriteTerm` signals a *soft* failure. If attempting to apply a rule signals a *hard* failure, so does `rewriteTerm`.

Top-level interaction

```
processCmd :: Cmd -> GEnv -> (Rsp, GEnv)
```

`processCmd c ge` processes the command `c` in the global environment `ge`, and returns a response and a possibly updated new global environment.

The command `CRule r` simply adds `r` to the end of the rule list in the global environment and returns an empty response (no terms and no message).

The command `CQuery t True` returns a response with the list of the successive rewritings of `t` (starting with `t` itself). If the rewriting stopped because no further rules applied, there is no message. If rewriting stopped because of an error (either during a rewriting step, or because of a detected loop or timeout), the message will say so; see the examples in the informal presentation. In any case, the global environment is unchanged. The command `CQuery t False` is similar, but includes only the *last* term in the rewriting list (i.e., the final result, or the term at which the error occurred), not all the preceding ones.

Hints Each group of functions above is expected to require around 15–25 lines of code; if you are using significantly more than that, you may be doing something needlessly complicated.

If you don't get through all groups, do as many as you can: you will get partial credit for each correctly working (and properly tested/assessed) group.

General instructions

Each of the two parts (Syntax and Semantics) will be weighted roughly equally in the assessment. However, as a matter of exam technique, keep in mind that it's probably significantly easier to produce incomplete but acceptable solutions to both parts, than a near-perfect solution to only one of them.

Put all your files in the `src/APle/...` directory. Please do use the provided skeleton files: they have all the types, imports and exports set up correctly, which should significantly decrease the risk of your solution failing our automated tests. Remember to assess your code in the report (including a description of how you tested it, and how we can reproduce your tests), and to include your source code in appendices.

For your convenience, we have also provided a `APle.hs` with a simple interactive top-level loop that you may use to experiment with APLE. Note that this file is *not* part of the assignment, and your code or tests should not depend on it. Usage is:

```
$ runhaskell APle operators.op rules1.ap ... rulesn.ap
```

where `operators.op` contains the operator table (as a Haskell term), and the $\texttt{rules}_i\texttt{.ap}$ files contain any rule definitions to be preloaded before the interaction starts.

Question 2: RoboTA

This question is about making RoboTA, an alternative to the renowned OnlineTA system. For teachers, a RoboTA server can hold a number of *assignments*, each assignment consists of a number of *parts* and each part has a *grader*. For students, it is possible to submit a *submission* for an assignment to a RoboTA server to get the submission analysed and get *feedback* based on the analysis.

Terminology:

- An assignment consists of a list of parts. A part is a pair `{Label, Grader}`, where `Label` is an atom and `Grader` is a grader. All labels on parts in an assignment should be different.

  An assignment can either be available or unavailable. It's only possible to add parts to an assignment when it's unavailable, and it's only possible to submit submissions for grading when it's available.

- A submission consists of a list of labeled answers. A labeled answer is a pair `{Label, Answer}`, where `Label` is an atom and `Answer` is an appropriate Erlang term.

  We say that a submission is *valid* with respect to an assignment, if the labels in the submission are a subset of the labels in the assignment. (The order doesn't matter.) If a submission contains multiple answers with the same label, only one of them is graded and it is unspecified which of the answers is graded.

  We say that a submission is *partial* if it does not contain answers to all parts of an assignment.

- Feedback is a list of labeled results. A labeled result is a pair `{Label, Result}`, where `Label` is an atom and `Result` is an appropriate Erlang term.

  Feedback with respect to an assignment contains results for all parts of the assignment. For a partial submission, the result of missing parts should be the atom `missing`.

- A grader analyses a labeled answer and gives a result of the analysis. A grader can either be a *simple* grader or a *composed* grader made of other graders, each dealing with different aspects of the analysis.

  For a labeled answer, `{Label, Answer} = LAnswer`, we have the following graders:

  - A forgiving grader is the atom `abraham`. The result of a forgiving grader is always the atom `looks_good` no matter the answer.
  - An angry grader is the atom `niels`. The result of an angry grader is always the atom `failed` no matter the answer.
  - A matching grader is the tuple `{mikkel, Expect}` where `Expect` is an Erlang term. If `Answer` is exactly equal to `Expect` (compared with `=:=`) then the result is `looks_good` otherwise it is `failed`.
  - A testing grader is the tuple `{simon, Arg, Expect}` where `Arg` and `Expect` are Erlang terms. The grader expects `Answer` to be a function and if `Answer(Arg)`

is exactly equal to `Expect` (compared with `=:=`) then the result is `looks_good` otherwise it is `failed`.

You may assume that `Answer` does not start any processes. But that is the only assumption you should make about the behaviour of `Answer`.

- A modular grader is the tuple `{andrzej, Callback, Init}` where `Callback` is an atom denoting a *grading callback module* and `Init` is the argument to the `setup` function from `Callback`. The result of the grader is the result of calling the `grade` function from `Callback` with the argument `LAnswer`.

  Grading callback modules are described later in this question.
- A concurrent grader is the tuple `{troels, Gs}` where `Gs` is a list and each element in `Gs` is a grader. The result is the list of results obtained by using each grader in `Gs` on `LAnswer` in the order as `Gs`. The graders cannot rely on any order of execution, thus they can be executed concurrently. If any grader fails, all graders should be exited and the result should be `grader_failed`.

  Remember that a grader can have side-effect, such as writing to the file system.

A grader can *fail* if it calls a function that exits in any non-normal fashion, or if the result takes longer than 3 seconds to be computed (in which case all computations in the grader should be stopped). The result of a failed grader is the atom `grader_failed`. Note that, a grader can successfully give the result `failed` for a submission, which is different from `grader_failed`.

- A grader module should export (at least) the functions:
  - `setup(Init)` for performing any initialisation necessary for the module. Returns `{ok, State}` on success, and `{error, Reason}` if some error occurred.
  - `grade(State, LAnswer)` for grading a labeled answer. Here, `State` is what is returned by `setup/1`, and `LAnswer` is a labeled answer. Returns `{ok, Result}` on success, and `{error, Reason}` if some error occurred and the grader failed.
  - `unload(State)` for performing any module level cleanup. Returns `ok` on success, and `{error, Reason}` if some error occurred. If any processes were started by the module, it is the responsibility of `unload/1` to stop them.

  You can find an example grader module in Appendix A.

Part 1: The RoboTA module

Implement an Erlang module `robota` with the following API:

- `get_the_show_started()` for starting a RoboTA server. Returns `{ok, RoboTA}` on success or `{error, Reason}` if some error occurred.
- `new(RoboTA, Name)` for creating a new assignment with the name `Name`. Returns `{ok, AssHandler}` on success or `{error, Reason}` if some error occurred. Here, `AssHandler` is the authorised handle to the assignment for the teacher. It is the teacher's responsibility to keep `AssHandler` secret. The assignment starts as unavailable.

- `add_part(AssHandler, Label, Grader)` for adding a part labeled `Label` to `AssHandler` with the grader `Grader`. Returns `ok` on success or `{error, Reason}` if some error occurred, where `Reason` should be `{Name, is_available}` if the assignment is available, and `Name` is the name of the assignment. `Reason` should be `{Label, not_unique}` if `Label` is not unique in the assignment.
- `del_part(AssHandler, Label)` for deleting the part labeled `Label` from `AssHandler`. This should be an non-blocking function.

  The function has no effect if the assignment is available or if there is no part labeled `Label` in the assignment.
- `available(AssHandler)` for making the assignment available. When an assignment is made available all modular graders should be initialised. Returns `ok` on success or `{error, Reason}` if some error occurred, for instance that a `setup` function from a modular grader threw an exception. If some error occurred and the assignment was unavailable then the assignment stays unavailable.
- `unavailable(AssHandler)` for making an assignment unavailable. When an assignment is made unavailable it stops accepting new submissions for grading, and finishes grading all submissions currently being graded. This function returns when there are no submissions under grading, and all modular graders in the assignment have been unloaded. Returns `ok` on success or `{error, Reason}` if some error occurred, for instance that an `unload` function from a modular grader threw an exception.

  Even if some error occurred the assignment should be unavailable after this function returns.
- `status(AssHandler)` for examining the state of an assignment. Returns `{Status, Parts}` where `Status` is one of the atoms `available` or `unavailable`, and `Parts` is the list of parts in the assignment. The order of `Parts` is unspecified.
- `grade(RoboTA, Name, Submission, Pid)` for sending a submission for grading to the assignment named `Name` (the argument given to `new/2`). Returns `Ref` quickly, where `Ref` is a reference. Quickly means before the submission is graded.

  When the result of the grading is ready it is sent as an Erlang message to `Pid` on the form `{Ref, Response}`, where `Ref` is the reference returned by `grade`, and `Response` is either `{ok, Result}` if the submission was successfully graded or `{error, Reason}` if some error occurred. If the assignment is unavailable `Reason` should be `{Name, is_unavailable}`.

Desirable properties:

- We want to minimise the latency of the grading of submissions. Thus, the grading of a submission should not block the grading of other submissions.
- RoboTA should be robust against malicious input from students. Meaning that it should not be possible for students to make a submission that crashes RoboTA.
- RoboTA should be robust against incompetence from teachers. Meaning that it should not be possible for teachers to make a grader that crashes RoboTA.

Robustness in the desirable properties should be interpreted within reason and scope of the course. Thus, you should not start to look into, say, sandboxing of processes or trying to control which libraries a process or function can call. Don't overthink it, but make sure that you document what your interpretation of robustness mean.

Part 2: Use RoboTA

Write a test program that consists of two modules, `demo` and `demo_grader`:

- `demo` starts a RoboTA server.
- there should be a teacher process that creates a new assignment with multiple parts, where there is at least one of each kind of grader, including a modular grading.
- `demo_grader` is a grading callback module.
- the assignment should be made available
- a submission should submitted for grading
- the result of the grading should be successfully received
- the assignment should be made unavailable.

Part 3: QuickCheck RoboTA

Make a module `robota_qc` that uses QuickCheck for testing a `robota` module. We will test your `robota_qc` module with our special version of the `robota` module, so your tests should only rely on the API described in the exam text.

Your module should export:

- A generator of graders named `grader/0`.
- A generator of assignments named `assignment/0` that generates a list of parts.
- At least two properties related to `robota` that starts with the prefix `prop_`. For these, you'll probably need other generators than just `grader/0` and `assignment/0`.
- If you have properties that are specific to *your* implementation of `robota` (perhaps they are related to an extended API or you are testing a sub-modules of your implementation), they should start with the prefix `myprop_`, we will test these with your own implementation of `robota`.

General comments

Part 1 counts for 70% of this question, Part 2 counts for 10%, and Part 3 counts for 20% of this question.

Document which properties your module provides (and under which assumptions). Likewise you should document if you have implemented all parts of the question. Remember to detail in your report how you have tested your module. In general, as always, remember to test your solution and include your test in the hand-in.

Appendix A: Example Grader Module

An example grader module that gives seemingly random, but consistent grades. Meaning, that the same answer will always get the same grade while the assignment is available.

```
-module(consistentgrader).

-export([setup/1, grade/2, unload/1]).

setup(_) ->
    Now = erlang:system_time(microsecond),
    Bin = term_to_binary(Now),
    <<Salt:8, _/binary>> = erlang:md5(Bin),
    {ok, Salt}.

grade(Salt, {_, Answer}) ->
    Bin = term_to_binary(Answer),
    <<Desider:8, _/binary>> = erlang:md5(Bin),
    {ok, case (Desider + Salt) rem 2 of
             0 -> looks_good;
             _ -> failed
         end}.

unload(_Salt) ->
    ok.
```