Take-home Exam in Advanced Programming

Deadline: Friday, November 6, 16:00

Version 1.0

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2020. This document consists of 24 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 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 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 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 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: APQL: A Perplexing Query Language

An important application domain for declarative languages is the design and implementation of *deductive databases*. Such databases generalize the relational model by allowing tables to be built not only by the standard relational operators, such as joins and unions, but also by recursive *rules*. In this part of the exam, we will look at a simple deductive-database engine, and its associated query language, collectively called APQL.

Introduction

An APQL database consists of a number of *relations*, or *predicates* (also known as *tables* in database terminology), each of which contains a finite collection of tuples (or *rows*) of data values. All tuples are of the same length, the *arity* of the predicate. When talking about a predicate, we will usually show its arity by a slash after the name. For simplicity, the only data values supported are character strings.

An APQL predicate can be *extensional*, meaning that it's directly represented as an (often large) concrete set of tuples; or *intensional*, meaning that its contents is given by (typically only one or a few) rules for membership, which rules may make use of variables, logical connectives (conjunction, disjunction, negation), as well as (potentially mutually recursive) references to other predicates. For example, in a simple genealogy database, we may have the following extensional predicates:

```
father/2 :                    mother/2 :
  ("Bob", "Alice")              ("Alice", "Carol")
  ("Bob", "Eve")                ("Alice", "Ivan")
  ("Bob", "Frank")              ("Judy", "Rupert")
  ("Frank", "Judy")             ("Mallory", "Eve")
  ("Ivan", "Bob")               ("Mallory", "Frank")
  ...                           ...
```

An APQL *program* defines a collection of intensional predicates, each determined by a collection of *rules*. For example, we can specify the following predicates:

```
parent(x,y) if father(x,y).
parent(x,y) if mother(x,y).

sibling(x,y) if father(f,x) and father(f,y) and mother(m,x) and mother(m,y)
                and x is not y.

ancestor(x,y) if parent(x,y) or (ancestor(x,z) and parent(z,y)).

paradox(x) if ancestor(x,x).

myquery(x) if sibling("Frank", x) and not ancestor(x, "Rupert").
```

(The last predicate is not meant as a general definition, but as an example of a query we

may want to pose, in this case, "Are there any siblings of Frank who are *not* ancestors of Rupert?")

APQL thus looks much as Prolog (with a slightly different syntax), but there are some important differences:

- There are no *structured* values (e.g., lists or trees): all variables range over atomic values (i.e., strings) only.
- All predicates (whether intensional or extensional) only contain finitely many tuples. In particular, if a goal `p(x,y)` succeeds, both `x` and `y` must be(come) fully instantiated.
- Logical formulas in rules may use not only conjunctions and disjunctions, but also negation, of both equality tests (as in `sibling`/2) and of general formulas (as in `myquery`/1).
- Rules may be recursive (as in `ancestor`/2). However, even if the underlying data contains cycles (as in the example), all programs terminate. For example, (if the database does not contain any more tuples in the "..." parts), `paradox`/1 will contain precisely the 1-tuples `("Alice")`, `("Bob")`, and `("Ivan")`, each of which is nominally their own grandparent. (This could actually happen if we consider parenthood a *legal*, rather than a *biological*, relationship, and in particular allow adoptions.)

To guarantee these properties, we must impose some additional conditions on APQL programs. That is, not all *syntactically* well-formed programs will be allowed. Fortunately, these conditions can all be checked *statically*, i.e., by looking only at the program, at not the extensional predicates it may refer to. In fact, all of the rules above are valid.

The execution of an APQL program consists simply of computing extensional representations of all the intensionally specified predicates. Unlike in Prolog, which uses a top-down, or goal-directed, search strategy, APQL uses a bottom-up model, which starts by determining the contents of the more basal predicates (e.g., `parent`/2, which depends only on extensional predicates), and works its way up to the higher-level ones (e.g, `myquery`/1), which may exploit that its dependencies have already been evaluated, so we can tell immediately whether or not, e.g., `ancestor(x, "Rupert")` holds for some particular `x`. The contents of recursive predicates, such as `ancestor`/2 are computed incrementally, starting from the tuples that must definitely be present, because they come from extensional or already computed predicates. For example, by the left disjunct in the rule, the tuple `("Bob", "Alice")` must be in `ancestor`/2 because it is in `parent`/2. But therefore, taking `x` = `"Bob"`, `y` = `"Alice"`, and `z` = `"Carol"` in the right disjunct, we also get that `("Bob","Carol")` is in `ancestor`/2. This process continues, until no more elements can be added.

In the following subsections, we will consider the three main modules of the APQL implementation: the *Parser*, which reads a program in the notation outlined above, and converts it to an abstract syntax tree; the *Preprocessor*, which checks that the rules in the program satisfy some constraints about variable and predicate occurrences, and also converts the program to a restricted form, better suited for execution; and the *Engine* which actually computes the contents of the intensional predicates, given the preprocessed program and the contents of the extensional ones. The system also comes with a simple

main program, which ties the three modules together, and allows APQL programs to be run from the command line.

Hint: We recommend that you try to develop some *minimally* working code (and tests demonstrating both what does and does not yet work) for each module, and then repeatedly address what you consider the most serious remaining flaws or limitations in your implementation, possibly switching to working on something else if one problem proves too time-consuming.

Question 1.1: APQL syntax and parsing

The abstract syntax of APQL programs is given by the following types (omitting some standard `derives`-clauses), as found in the module `Types`:

```
type Program = [Rule]

data Rule = Rule Atom Cond

data Cond =
    CAtom Atom
  | CTrue
  | CEq Term Term
  | CAnd Cond Cond
  | COr Cond Cond
  | CNot Cond

data Atom = Atom PName [Term]

data Term =
    TVar VName
  | TData Data

type PName = String
type VName = String
type Data = String
```

Note in particular that a `Term` can only be a variable or a data (i.e., string) constant.

The concrete syntactic grammar is shown in Figure 1. This grammar is supplemented by the following specifications:

Lexical tokens The grammar references a couple of non-literal terminal symbols:

- ***pName*** and ***vName***. Predicate and variable names are *both* specified to be sequences of *ASCII* letters (upper- and lowercase), digits, or underscores, starting with a letter. (Unlike in Prolog, variables are allowed to start with a lowercase letter, and predicates with an uppercase one; this causes no ambiguity, because predicate names are always

Program	::=	ϵ
	|	*Rule* '.' *Program*
Rule	::=	*Atom*
	|	*Atom* 'if' *Cond*
	|	*Atom* 'unless' *Cond*
Cond	::=	*Atom*
	|	*Term* 'is' *Term*
	|	*Term* 'is' 'not' *Term*
	|	'true'
	|	'false'
	|	'not' *Cond*
	|	*Cond* 'and' *Cond*
	|	*Cond* 'or' *Cond*
	|	*Cond* 'implies' *Cond*
	|	'(' *Cond* ')'
Atom	::=	*pName* '(' *Termz* ')'
Termz	::=	ϵ
	|	*Terms*
Terms	::=	*Term*
	|	*Term* ',' *Terms*
Term	::=	*vName*
	|	*Constant*
Constant	::=	*stringConst*

Figure 1: Concrete grammar of APQL

followed by an open parentheses, while variable names never are.) The *keywords* of the grammar (`and`, `false`, `if`, `implies`, `in`, `is`, `not`, `or`, `true`, `unless`) cannot be used as variable or predicate names.

- ***stringConst***. A string constant consists of zero or more *printable ASCII* characters enclosed in double-quotes, e.g, `"Hello, world!"`. If the string constant is to itself include double-quote characters, they must be *doubled*; e.g., the concrete syntax `""foo""bar""` represents the 7-character string `foo"bar` .

Whitespace and comments All tokens may be surrounded by *whitespace* (spaces, tabs, and newlines), which is generally ignored (except inside string constants). However, some whitespace is *required* between keywords/names and any adjacent letters or digits; for example, "`isnot`" is a name, while "`is not`" is a sequence of two keywords.

Comments are written between the delimiters "`(*`" and "`*)`". They may contain an arbitrary sequence of characters (not necessarily ASCII, and not necessarily printable), except for the comment-closing sequence; in particular, comments do not nest. Comments are not recognized within strings, but may otherwise occur wherever whitespace is allowed, and are treated as whitespace for the purpose of token separation, e.g. "`is(**)not`" is again two keywords, not a single name.

Disambiguation of connectives The raw grammar for logical connectives (operators) is ambiguous. It is to be disambiguated as follows, from lowest to highest precedence:

1. '`implies`': right-associative.
2. '`or`': left-associative
3. '`and`': left-associative
4. '`not`': nestable (e.g., `not not true` is allowed)

As usual, these precedences and associativites can be overridden by parentheses.

Representation as abstract syntax The correspondence between concrete and abstract syntax should largely be evident. However, the concrete grammar contains a few constructions that don't have dedicated abstract counterparts; instead, they should be desugared as follows:

- In the alternatives for *Rule*, the rule form "*Atom*" should be represented as if it had been written "*Atom* `if true`" (i.e., as `Rule` *a* `CTrue`, where *a* is the AST for the *Atom*); and the rule "*Atom* `unless` *Cond*" should be represented like "*Atom* `if not` (*Cond*)".
- In the alternatives for *Cond*, the condition "`false`" should be represented like "`not true`"; the condition "$Term_1$ `is not` $Term_2$" like "`not` ($Term_1$ `is` $Term_2$)"; and the condition "$Cond_1$ `implies` $Cond_2$" like "`not` ($Cond_1$) `or` ($Cond_2$)".

For example, the concrete-synax rule "`a(x) unless b(x) and c() implies true`" should parse the same as "`a(x) if not (not (b(x) and c()) or true)`", i.e.,

```
Rule (Atom "a" [TVar "x"])
     (CNot (COr (CNot (CAnd (CAtom (Atom "b" [TVar "x"]))
                            (CAtom (Atom "c" []))))
                CTrue))
```

The Parser API consists of a single function,

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

where `parseString` *s* returns either the result of successfully parsing s, or an error message. (See the dedicated instructions on error messages in a later section).

You must implement your parser using either the `ReadP` or the `Parsec` parser-combinator library (as supplied with the course LTS version). 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`.

Hint: Try to quickly get your parser into a shape where it can at least correctly parse some of the programs in the `examples/` subdirectory of the handout (possibly with minor tweaks). If you cannot easily get the finer points of string and comment syntax, token separation, exact operator precedences and associativites, etc. to work, they can wait.

Question 1.2: Preprocessor

The `Preprocessor` module contains two, largely independent, pieces of functionality:

1. Converting program rules into a more restricted form known as clauses, and checking some conditions on variable occurrences in those clauses.
2. Checking that (clausified) programs involving negation can in fact be executed coherently, and if so, determining the order in which to evaluate the intensional predicates.

Each of these functions is described further in the following.

Question 1.2.1: Clausification

Even after desugaring of, e.g., `implies`-connectives to more basic forms, APQL program rules may not be directly suited for execution. The main problem is that negations may be applied to complex subformulas, which significantly complicates the bottom-up evaluation model. Therefore, as a first step, we convert each rule to a number of simpler *clauses*, in which negations can only be only applied to atomic conditions and equality tests. At the same time, we eliminate disjunctions from conditions entirely, by transforming each rule to potentially more than one clause, much like ; can be eliminated from Prolog programs.

More precisely, an APQL clause is given by the following abstract syntax:

```
data Clause = Clause Atom [Atom] [Test]

data Test =
   TNot Atom
 | TEq Term Term
 | TNeq Term Term
```

Like a rule, a clause has a head (the first `Atom`) and a body, which represents a conjunction of primitive conditions. Those conditions are separated into a list of *positive* (i.e., non-negated) references to (the same or other) predicates, and a list of *tests*, where a test is either a *negative* (i.e., negated) reference to another predicate, or an assertion that two terms (typically involving at least one variable) are either equal or unequal.

The transformation from rules to clauses is based on some familiar logical equivalences. We write them here in concrete-syntax notation, but of course the transformations actually work with the abstract syntax.

First, we have some simple equivalences for negation:

$$\texttt{not (}\mathit{Cond}_1\ \texttt{and}\ \mathit{Cond}_2\texttt{)} \iff \texttt{(not}\ \mathit{Cond}_1\texttt{) or (not}\ \mathit{Cond}_2\texttt{)}$$
$$\texttt{not (}\mathit{Cond}_1\ \texttt{or}\ \mathit{Cond}_2\texttt{)} \iff \texttt{(not}\ \mathit{Cond}_1\texttt{) and (not}\ \mathit{Cond}_2\texttt{)}$$
$$\texttt{not (not}\ \mathit{Cond}\texttt{)} \iff \mathit{Cond}$$

Using these, any *Cond* can be transformed into an equivalent form, where the `not`-operator may only be applied to logical atoms, equality tests (`is`), or `true`.

Next, we can employ the following equivalences:

$$\mathit{Cond}_1\ \texttt{and false} \iff \texttt{false}$$
$$\mathit{Cond}_1\ \texttt{and (}\mathit{Cond}_2\ \texttt{or}\ \mathit{Cond}_3\texttt{)} \iff \texttt{(}\mathit{Cond}_1\ \texttt{and}\ \mathit{Cond}_2\texttt{) or (}\mathit{Cond}_1\ \texttt{and}\ \mathit{Cond}_3\texttt{)}$$

(as well as the symmetric versions where the Cond_1 is on the right-hand side of the and). Using these, any *Cond* from above can be put in a form where, additionally, no argument of an `and` contains an inner `or` or `false`.

Finally, any rule of the form

Atom `if false.`

can simply be discarded (since the body condition will never be satisfied); and a single rule of the form

Atom `if` Cond_1 `or` Cond_2`.`

is equivalent to the pair of rules

Atom `if` Cond_1`.`
Atom `if` Cond_2`.`

Using these equivalences, we can transform any program to one containing exclusively rules of the form "Atom `if` Cond", where the *Cond* can only be a conjunction of atoms,

negated atoms, (possibly negated) `is`-comparisons, and `true`s. But a rule in such a form is effectively the same as a `Clause`, because all the non-negated atoms in the rule body can be put into the first list; the negated atoms and the two forms of comparisons can be turned into tests in the second list, and any `true` conjuncts can simply be discarded.

After the transformation to clausal form, to guarantee that each intensional predicate can only represent a finite set of tuples, we finally need verify that every variable occurring in the clause head, and/or in one or more of the tests, *also* occurs at least once in the list of non-negated atoms in the body. Otherwise, the clause (and thus the entire transformed program) should be rejected.

Note, incidentally, that there is a subtle difference between the following two rules/-clauses:

```
p1(x) if x is "a".
p2("a").
```

Although they have the same logical meaning, `p2` (trivially) satisfies the variable-occurrence conditions, while `p1` does not. (It is in fact possible to eliminate all non-negated equality tests from clauses, by a suitable substitution of variables, but that is not part of the task.)

Converting a program into clausal form yields an *intensional database*:

```
data IDB = IDB [PSpec] [Clause]

type PSpec = (PName, Int)
```

Since some program rules may correspond to zero clauses, the database also includes a list of all intensional predicates defined in the original programs by at least one rule. Note that we consider predicates with the same name but different arities as *unrelated*; thus, a proper predicate specification consists of both a name and an arity.

The API for this part of the Preprocessor consists of a single function

```
clausify :: Program -> Either ErrMsg IDB
```

The call `clausify` *p* should convert the APQL program *p* into clausal form (which is always possible), and then check that the resulting clauses all obey the variable restrictions (which might fail, if the original program was invalid). In the latter case, the function should return a suitable error message. *If* the original program lists all the rules for a predicate together (which is otherwise not a requirement), the output of `clausify` should preserve that order in both the `PSpec` list and the `Clause` list of the `IDB`.

For example, the APQL program

```
p(x) if q(x) and not (r(x) and x is not a).
p() if p("b").
```

should be converted to:

```
DB [("p",1), ("p",0)]
   [Clause (Atom "p" [TVar "x"])
           [Atom "q" [TVar "x"]]
           [TNot (Atom "r" [TVar "x"])],
    Clause (Atom "p" [TVar "x"])
           [Atom "q" [TVar "x"]]
           [TEq (TVar "x") (TData "a")],
    Clause (Atom "p" [])
           [Atom "p" [TData "b"]]
           []]
```

Note that you are not required to implement the above-described transformations *literally*, as long as the end result is the same. For example, instead of working with `Cond`-formulas everywhere, you may introduce your own tailored intermediate data structures, e.g., representing a tree of conjuncts or disjuncts as a flattened list; and you may exploit that the order of elements in such a list is not significant. However, you should *not* perform any other transformations as part of clausification, even if they are meaning-preserving (such as the `is`-elimination mentioned above).

Hint: If you cannot implement the full functionality of `clausify`, note that it is feasible to do a wide range of partial implementations. For example, if the original program was effectively *already* in clausal form (i.e.., the `Cond` in each `Rule` contains no `COr`-connectives, and each `CNot` is only applied to a `CAtom` or `CEq`), transforming such rules into clauses should be quite straightforward – especially if you impose the further requirement that the `CAnds` in the `Cond` are all left-associated (as they would be if the rule was written in the concrete syntax without extra parentheses).

Similarly, you may decide to skip the verification of the variable conditions. Just be sure to clearly document any simplifications or restrictions you impose in your report and tests.

Question 1.2.2: Stratification

The second restriction on APQL programs has to do with negations in recursive rule specifications. If we allow arbitrary negation, we can write a program like following:

```
p("a") if not p("b").
p("b") if not p("a").
```

(Note that, since this program contains no variables at all, it trivially satisfies the occurrence conditions.) Which values should the extensional representation of `p/1` contain? Clearly the empty set is not acceptable, because it violates both the rules. Including `"a"` but not `"b"`, or vice versa, would be consistent, but the rules give no logical justification to prefer one solution over the other. And including both `"a"` and `"b"` would make neither rule applicable, and thus there is no support for including either of the two in the first place (nor to exclude `"c"` for that matter).

To avoid this, and many similar, problems, we require that APQL programs must be *stratifiable*. That is, it must be possible to partition all intensional predicates into groups,

called *strata*, such that the strata can be evaluated consecutively, without logical vicious circles.

More precisely, stratum 0 is considered to contain all the extensional predicates, while the intensional ones are put in strata $i \geq 1$. Clauses for all predicates in stratum i may refer positively (i.e., non-negated) to other predicates in i and all lower strata, but any negative references from stratum i must be to strictly lower strata only. In particular, the program above (or any program including those rules) is *not* stratifiable, because whatever stratum `p/1` is put in, the clauses refer negatively to a predicate (namely `p/1` itself) in the same stratum.

In general, each stratum might as well include as many predicates as possible. That is, after already having completed all strata less than i (initially, $i = 1$), and there are still predicates left to be placed, we initially put all the remaining predicates into the next stratum i. Then we repeatedly remove predicates from i if either (a rule for) the predicate refers negatively to a predicate not in a strictly lower stratum, or if it refers positively to a predicate previously removed from i. Note that removals may cascade, so removing one predicate may require additional ones to be removed, and so on. Once the stratum stabilizes (i.e., no more predicates need to be removed), we go on to the next one. However, if *all* predicates were removed (i.e., the stratum becomes completely empty), it means that the program was unstratifiable, and we can report the failure.

For example, consider the following program in clausal form. (For readability, we use the concrete syntax of rules; and since stratification only looks at the predicates themselves, we have omitted the argument lists.)

```
(* r is extensional *)

p() if q() and r().
p() if p() and not r().

q() if q() and not s().

s() if r().
```

First, we let stratum 0 be the list (or set) `[r]`, since that is the only extensional predicate.

Then, we tentatively take stratum 1 to be `[p, q, s]`. The clauses for `p` do not immediately cause problems (since `r` is in a lower stratum for the negative occurrence), and the positive references to `p` and `q` are ok. But because `q` refers negatively to `s`, which is *not* in a lower stratum, we must remove `q`, leaving `[p, s]`. Now, however, the first clause for `p` refers positively to `q`, which is no longer in the stratum, so we must remove `p` as well, leaving `[s]`. Since the clause for `s` is unproblematic, we end up with `[s]` as stratum 1.

Next, we start out with stratum 2 as the remaining predicates, i.e.. `[p, q]`. Both clauses for `p` are still ok, but now the one for `q` is also acceptable, because `s` is in the lower stratum 1. Thus, we can keep stratum 2 as `[p, q]`, and since we have placed all predicates, the final strata of the intensional predicates are 1: `[s]`, and 2: `[p, q]`.

The API for the stratification part of the Preprocessor consists of the function

```
stratify :: IDB -> [PSpec] -> Either ErrMsg [[PSpec]]
```

This is invoked as `stratify` *idb* *eps* where *idb* is an intensional database (as returned by `clausify`), and *eps* is a list of extensional predicates (i.e., those that can go into stratum 0); it is an error for a predicate to appear in both *idb* and *eps*. The function should return a correct stratification of all the predicates in *idb*, listed in ascending order by stratum, or a suitable error message. For example, for the *idb* from the example above, a correct output from "`stratify` *idb* `[("r",0)]`" would be `[[("s",0)], [("p",0), ("q",0)]]` The order of predicates within each stratum does not matter, but every predicate in *idb* should end up in exactly one stratum, and there should be no empty strata.

Again, you are not required to implement exactly the above algorithm, as long as you construct a correct stratification when one exists, and fail otherwise; in particular, your stratification is allowed to contain more strata than necessary.

Hint: As before, if you struggle implement the stratifier in full generality, consider recognizing and handling some easier special cases. For example, if there are no clauses with negated atoms at all (negated equalities are ok), then the program is trivially stratifiable (by putting all predicates into the same stratum). Conversely, if there is no *mutual* recursion between the predicates, and they are already ordered suitably in the original program (i.e., all the clauses for a predicate occur together, and each clause only refers to previously defined predicates, and possibly also positively to the predicate itself), then each predicate can be put into its own stratum. Again, if you go for such a partial (but still likely to quite usable in many circumstances) implementation, document the restrictions in your report and tests.

Question 1.3: Execution engine

The final module consists of the execution engine for APQL programs, in clausified and stratified form. This module takes as input the tuple sets for all the extensional predicates, and computes analogous sets for the intensional ones.

The computation is done stratum-wise, i.e., we first compute all the intensional predicates in stratum 1, then stratum 2, and so on. This way, *negative* references to atoms become simple non-membership tests in already computed sets, just like equality and non-equality tests become simple comparisons.

Within each stratum, the extensional representations are built up incrementally, starting from the empty sets of tuples for each, and repeatedly adding more tuples that arise from clause applications, until no more tuples need to be added. (Note that this is essentially the opposite process than the one suggested for `stratify`, where we repeatedly *removed* predicates from the stratum we were building.)

Applying a clause for a predicate consists of finding instantiations, or bindings, for its variables, to match already computed extensional tuples for the positive atoms (in the complete tables for predicates in lower strata, or in the ones being built in the current stratum), such that those instantiations satisfy any further tests specified in the clause. Then the contributions of that clause are given by the clause's head atom, again after applying the relevant bindings of all variables.

Unlike in Prolog, an APQL variable can only be bound/instantiated to a data value, not to another variable. Thus, the variable-occurrence conditions guarantee that the instantiated

clause head will indeed be a ground tuple, ready to be added to a table.

In each round of computations within a stratum, we compute the contributions of all the clauses, to obtain a set of tuples to be added to each predicate table. Note that some of these tuples may be present already, because in general a tuple may be derived by multiple rules. However, if *all* of the newly computed additions were already present (i.e., none of the tables actually got larger), we have added everything that can be deduced from the rules in the current stratum, and we can go on to the next one (if any).

Note that, when computing the contributions of a clause, it may not be necessary to revisit *all* the tuples in the tables for the positive atoms it refers to. In particular, if only one of the referenced predicates has changed since the last round, it suffices to look at only at the newly added tuples for that predicate, as those are the only ones that can lead to additional conclusions. You are encouraged, but not required, to take advantage of this optimization opportunity in your engine. Describe your strategy in the report

Concretely, an *extensional database* consists of a collection of named tables, each of which is associated to a set of tuples (rows):

```
type Row = [Data]

type ETable = Data.Set.Set Row

type EDB = [(PSpec, ETable)]
```

The API for the engine is then the following code:

```
execute :: IDB -> [[PSpec]] -> EDB -> Either ErrMsg EDB
```

`execute` *idb pss edb* takes a set of clauses, a stratification, and a database with tables for the extensional predicates; it returns an extended database, which now also includes tables for the intensional ones. Your function can assume that the clauses in *idb* satisfy the variable-occurrence conditions and are correctly stratified by *pss*, and that the tuples in *edb* have the correct lengths for their predicates.

Hint: Once again, there are ample opportunities for partial implementations. For example, if none of the clauses are recursive, then all the contributions of a rule in a stratum can be found in a single round, with no iteration necessary. In that case, the program is effectively just computing joins and/or unions of relations, which is an important special case, for which you will get substantial partial credit.

Error messages

The `ErrMsg` type, seen several times above, has some additional structure:

```
data ErrMsg =
    EUser String
  | EInternal String
  | EUnimplemented String
```

When reporting errors from your API functions, you should classify them appropriately:

- **User errors** indicate errors made by the user of the APQL system. This would include syntax errors in an APQL program, violations of the variable-occurrence and/or stratification conditions, or similar. The associated error message should explain what went wrong, ideally with as much context as it is *easily* possible to provide, e.g., the location and details of the syntax error if using a `Parsec` parser (whereas a `ReadP` one could just say `"no parse"`). User errors should *never* be reported with Haskell's `error` function.
- **Internal errors** indicate that something went wrong inside the implementation, for if example a `ReadP`-based parser encountered an ambiguous grammar, or if some helper function that expected to be called with a non-empty list was called with an empty one. The associated message will probably not make sense to the of user the system, but should ideally help the developer locate and diagnose the problem. Internal errors are a gentler alternative to simply crashing out with a Haskell error, but the latter might be acceptable in some circumstances, especially when the error is not discovered in a monadic context. (Of course, in the first instance, you should attempt to *fix* any internal errors; the message is for circumstances that you were not able to resolve in time.)
- **Missing-implementation errors** are intended for situations where a particular feature or aspect of a function has simply not been implemented (yet). Reporting such an error is much better than returning a known-incorrect result. Note that if an API function is missing completely, you should just leave it as `undefined`; but it's *much* preferable (from a partial-credit perspective) to at least implement some of "easy" cases.

In general, there may be some room for interpretation about which kind of error to use. A guiding principle should be that user errors are reserved for cases where (it's most likely that) the user of the entire system (as opposed to the caller of any particular API function) did something wrong. For example, if a bug in `stratify` causes `execute` to fail because a needed predicate has not yet been computed, that should be reported as an internal error, rather than an user one (because `stratify` was expected to complain if the original, user-provided program was actually not stratifiable). However, do not fret too much over which kind of message to use; just pick whatever seems the most appropriate, and possibly justify your choice in the report, if you think that it might be controversial.

Driver program

To help with your informal testing and experimentation, we have provided a simple driver program, which shows how the APQL modules are expected to fit together, and additionally has some simple support for reading and writing database tables in a textual format. You can build it with `stack build`, and run it with "`stack run -- args`". Running it without any arguments prints a brief usage message.

For example, to run an APQL variant of the InstaHub predicates (except for level 3) from Assignmen 3, you can try the following (with all arguments on one line):

```
stack run -- -i person 1 examples/g1-person.txt
             -i follows 2 examples/g1-follows.txt
             -x examples/instahub.apql
             -p ignorant 2
```

Note that the driver program is not particularly robust; if you use it on malformed inputs, it's likely to just crash. The textual representation of extensional predicates is one tuple per line, with each tuple represented as space-separated strings. Beware that these strings use the standard Haskell conventions, *not* the ones for string constants in APQL programs. In particular, backslashes are used for escaping special characters, while quote-doubling is *not* used.

General instructions

You should use the skeleton files provided. *Do not* modify the types in the abstract syntax or APIs, as this will likely prevent your solution from working with our automated tests.

For each of the three modules, there is both a file `Module.hs`, which exports only the required API, and `ModuleImpl.hs`, which exports everything. You should place your code in the latter only. Then, if you want to do white-box testing of some of your internal functions, you may import them from the relevant implementation modules.

Your implementation modules should *not* import directly from each other. If you have a non-trivial type definition or function that you want to share between two or more modules, place it in `Utils.hs` and import it from both. Note that `Utils` is also considered an implementation module, since its contents is not specified by the assignment.

We provide some sample programs and input database files in the `examples/` directory. Note that these *do not* exercise all the relevant functionality of the parser, preprocessor, and engine, so you should also test with your own (probably smaller, and targeted) sample programs and databases. Do consider whether any parts of the system would be particularly suitable for property-based testing (QuickCheck), and if relevant, implement some such tests as well.

Testing

As usual, you should test your programs thoroughly, especially since there will be no OnlineTA oracle. Note that we may run your test suite not only on your own implementations of the APQL modules, but also on some of our own (correct and/or buggy) implementations. This means that you should pay particular attention to the following guidelines.

As a general rule, the bulk of your tests should be **black-box**, i.e., test the API functions against their specifications in the exam text. While you may certainly look at your own implementation code when deciding what test cases would be particularly relevant, it should be the case that *any* correct implementation of the specification (as you understand it) would pass all your tests. This means that, when the specification says that several correct outputs are possible (e.g., particular wordings of error-message strings would normally not be specified, and elements of sets may generally be listed in any order), you should try (where this can be done with reasonable effort) to accommodate *any* such output.

And your black-box tests should never *expect* a function to return an `EInternal` (let alone `EUnimplemented`) error; when the specification says that a function may assume something about its inputs, an implementation of that function shouldn't go to any *extra* effort to explicitly verify that the promised property does indeed hold.

However, there may also be parts of your code that would be difficult to properly test through only the specified APIs. This includes, in particular, any non-trivial auxiliary functions (subparsers, etc.) you may have defined. Thus, where you believe it would significantly improve your (and others') confidence in the correctness or other aspects of your code (e.g., performance or robustness), you may additionally include one or more *separate* **white-box** test suite(s) specific your code. Such tests are allowed to import from your implementation modules, and/or expect particular behavior from your functions, in cases or areas where the official specification is silent. Note that you are not necessarily expected to do any particular white-box testing at all; if you believe that you can reasonably base your assessment on the black-box tests alone, that's perfectly fine.

Question 2: Mail Filter

An anonymous MegaCorp, we'll use the code name "G", is offering a free-to-use email service. G have found that analysing their users' email is good for their other business (undisclosed). Thus they are making a tender to find a developer for a new mail-analysis framework.

General comments

This question consists of two sub-questions: Question 2.1 about implementing an API for starting a mailfilter server and for setting up mail analysis, and Question 2.2 about writing QuickCheck tests against this API. Question 2.1 counts for 70% and Question 2.2 counts for 30% of this question. Note that Question 2.2 can be solved with a simple partial (or even without) implementation of the `mailfilter` module from Question 2.1.

In Appendix A you can find an example on how to use the API.

Remember that it is possible to make a partial implementation of the API that does not support all features. If there are functions or features that you don't implement, then make them return the atom `not_implemented`.

There is a section at the end of this question, on page 23, that lists expected topics for your report.

Terminology

A *mail-filter server* fundamentally manages two things: a set of *mails* to be analysed and a set of *filters* for analysing the mails. Multiple filters will be used to analyse the same mail, and not all filters should be used for analysing all mails. Furthermore, a mail-filter server has a *capacity* (which may be infinite) and it should try to execute as many filters concurrently as possible up to a given capacity. The capacity is global for all mails at a given mail-filter server. The capacity is a positive integer larger than zero, or the atom `infinite`, that denotes how many filters may be running at the same time.

As filters may *transform* the mail during analysis, your framework should not depend on the representation of mails.

Each filter is associated with a label; the result of each filter is stored under the given label. Your framework should not depend on the representation of labels. A result is either: the pair `{done, Data}`, meaning that the filter is done analysing the mail with result `Data`, where `Data` can be anything; or the atom `inprogress`, meaning that the filter is not done analysing the mail.

For each mail, the mail-filter server should maintain the (perhaps transformed) mail and a *configuration* which is the *labelled results* of the filters that have analysed the mail. It should always be possible to ask for the current configuration for a mail, even if there are ongoing filters analysing the mail.

Note that configurations should always be *consistent*. That is, when you get the configuration for a mail, all `done` results should be for the same version of the mail. So if one filter

transforms the mail it invalidates the results of other filters, which will then need to be stopped and restarted or run again with the transformed mail.

We use the following types to represent mails, data and filter functions:

```
-type mail() :: any().
-type data() :: any().
-type label() :: any().
-type result() :: {done, data()} | inprogress .
-type labelled_result() :: {label(), result()}.
-type filter_result() :: {just, data()}
                       | {transformed, mail()}
                       | unchanged
                       | {both, mail(), data()}.
-type filter_fun() :: fun( (mail(), data()) -> filter_result() ).
```

The meaning of `filter_result` is: `{just, Data}` means that the filter is just returning some data, and is not transforming the mail; `{transformed, Mail}` means that the mail has been transformed to `Mail`; `unchanged` means that the result is the initial data given and that the mail is not transformed; and `{both, Mail, Data}` means that the mail has been transformed to `Mail` and that the filter returns `Data`.

A filter function may run for an unlimited time and it may throw a normal exception, an error, try to exit or return something of the wrong form. Any kind of exceptions, exits or errors, or returning something of the wrong form is considered equal to returning `unchanged`.

Filters have the type `filter()`:

```
-type filter() :: {simple, filter_fun()}
                | {chain, list(filter())}
                | {group, list(filter()), merge_fun() }
                | {timelimit, timeout(), filter()}.

-type merge_fun() :: fun( (list(filter_result() | inprogress)) ->
                            filter_result() | continue ).
```

What the different tuples mean:

- `{simple, Fun}` for a filter that just runs `Fun` and updates the mail and result accordingly.
- `{chain, Filts}` for running the filters in `Filts` in sequential order. The result of this filter is the result of the last filter in `Filts`. The filter is allowed to transform the mail during the chain, without the changes being reflected in the final result. If `Filts` is the empty list, the result is `unchanged`.
- `{group, Filts, Merge}` for running the filters `Filts` concurrently, with the merge function `Merge` (see the type `merge_fun()`). Each time a filter in `Filts` completes execution, `Merge` is called with a list of the results of the filters in `Filts`. This list

is the same length as `Filts`: the i-th element of the list is either the result of the i-th filter in `Filts` or the atom `inprogress` if the i-th filter has not yet completed. If `Merge` returns `continue` it means that more results should be collected; otherwise the filter completes with the result of `Merge`.

You may assume that `Merge` will always terminate, and won't cause exceptions of any kind.

- `{timelimit, Time, Filt}` for running `Filt` with a timeout. If `Filt` has not completed within `Time` milliseconds it may be stopped and the result is unchanged. The timeouts are best effort. That is, they need not be exact, but must be at least as long as requested. Note that `Time` may be the atom `infinity` for no timeout.

  The filter is timed from when it first uses capacity at the server.

If a process is executing a filter function you may assume that it is safe to stop the process in any way that you want.

Question 2.1: The `mailfilter` module

Implement an Erlang module `mailfilter` with the following API, where `MS` denotes a reference to a mail-filter server and `MR` denotes a reference to a mail under analysis (you decide the representation). All functions are allowed to return `{error, Reason}` if some unspecified fault occurred.

- `start(Cap)` for starting a mail-filter server with `Cap` as the capacity. Returns `{ok, MS}` on success, or `{error, Reason}` if some fault occurred.
- `stop(MS)` for stopping a mail-filter server; this includes stopping all ongoing filters. Returns `{ok, State}` after all filters have been stopped. Where `State` has type: VER. 1.0

  ```
  State :: list({mail(), list({label(), result()})})
  ```

  That is, `State` is the current configuration for all mails registered for analysis at `MS`, represented as a list of pairs `{Mail, Labelled}`, where `Labelled` is the labelled results which is also represented as a list of pairs.

  The orders of the lists are unspecified.

- `default(MS, Label, Filt, Data)` for registering a default filter, with initial data `Data` and label `Label`, for all mail added to `MS`. Only mails added to `MS` after `Filt` has been registered will be analysed by `Filt`.

  If `Label` is already registered, this function has no effect.

  This function is non-blocking, thus the return value is unspecified.

- `add_mail(MS, Mail)` for adding mail to be analysed by the mail-filter server.

  Returns `{ok, MR}` on success, or `{error, Reason}` if some fault occurred.

- `get_config(MR)` for getting the current configuration for `MR`. Returns `{ok, Config}` on success, or `{error, Reason}` if some fault occurred.

- `enough(MR)` for stopping the analysis of `MR`. This also removes `MR` from the originating mail-filter server.

  This function is non-blocking, thus the return value is unspecified.

- `add_filter(MR, Label, Filt, Data)` for registering the filter `Filt` for `MR` under the label `Label` and with initial data `Data`.

  If `Label` is already registered, this function has no effect.

  This function is non-blocking, thus the return value is unspecified.

How to get started

Try to start by implementing a version of the `mailfilter` library that only runs a single simple filter at a time, perhaps even without spawning a separate process. When this works, you can extend it to work with different kinds of filters and then to work with multiple concurrent filters.

Think about how you want to represent a reference to a mail-filter server and how to represent a reference to a mail.

If you find it hard to complete the whole API here are some suggestions for simplifications:

- Only work with `infinite` capacity.
- Only support the API functions `start/1`, `stop/1`, `default/4` and `add_mail/2`.
- Only support `simple` filter.

Question 2.2: Testing `mailfilter`

Make a module `test_mailfilter` that uses `eqc` QuickCheck for testing a `mailfilter` module. We evaluate your tests with various versions of the `mailfilter` 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_mailfilter` module should contain, at least, the following:

- A QuickCheck generator `wellbehaved_filter_fun/0` that generates filter functions (that is, of type `filter_fun()`) that are well-behaved. That is, they will always terminate (within 1s), will not throw exceptions, will not call `exit`, will not cause errors, etc.
- A parameterised QuickCheck generator `filter(FunGen)` that takes a generator for filter functions as argument, and generates filters (that is, of type `filter()`).

  One possible argument for this generator could be `wellbehaved_filter_fun/0`.
- A QuickCheck property `prop_mail_is_sacred()` that checks that the length of the result of `stop/1` is equal to the number of mails added to the server with `add_mail/2`, minus the number of mails removed with `enough/1`.

  Note that the specification of this property leaves many decisions up to you. For instance: should the `add_mail/2` and `enough/1` calls be interleaved? should they all be successful? Should other API calls (which?) be interleaved? And so on. Remember to explain and motivate your choices.
- A QuickCheck property `prop_consistency()` that checks that when two mail-transforming filters, `F1` and `F2`, are applied to some mail, the resulting configuration returned via `stop/1` is consistent. That is, if both filters have completed when `stop/1` is called, then the mail and the labelled result correspond to either applying `F1` and then `F2` on the mail *or* applying `F2` and then `F1` on the mail.

  Again you have quite some leeway in deciding exactly how to formulate this property. But you might want to start out by writing a generator for generating relevant filters.

  Remember that while your implementation for Question 2.1 should not depend on the representation of, say, mail, your testing *is* allowed to use a specific representation.
- A QuickCheck property of your own choice. Should start with the prefix `prop_`.

  If you have a hard time coming up with good properties are here a few for inspiration:

  - For mail-filter servers with capacity N, no more than N filter are executing at the same time.
  - The number of labelled results in the result of `stop/1` is equal to the number of registered filters.
  - Non-terminating filters do not disturb the functionality of the server.
- A `test_all/0` function that runs all your tests and only depends on the specified `mailfilter` API. Your tests should involve the required properties in this module, but should also test aspects and functionality not covered by the required properties.

- 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_`. Remember that the code called from `test_all/0` must only depend on the specified `mailfilter` API and your (if any) `test_` modules.

You are welcome (even encouraged) to make more QuickCheck properties than those explicitly required. Properties that only depend on the specified `mailfilter` API should start with the prefix `prop_`. If you have properties that are specific to *your* implementation of the `mailfilter` library (perhaps they are related to an extended API or you are testing sub-modules of your implementation), they should start with the prefix `myprop_`, so that we know that these properties most likely only work with your implementation of `mailfilter`.

Topics for your report for Question 2

Your report should document:

- Which kinds of filters you support, and if you have made any assumptions about filter functions.
- How many processes your solution uses, what role each process has, and how they communicate.
- Clearly explain if you support the complete API or not. If not, which parts you don't support.
- Explain if you support server capacity or not. If you do, also explain how you have assessed the quality of your implementation of this feature.
- The quality or limitations of your testing. Especially the QuickCheck parts explicitly required in Question 2.2. Explain how you have measured this quality.

In general, as always, remember to test your solution, and include your tests and the result of running your test in your report (perhaps in your appendix).

Appendix A: Example use of `mailfilter`

The following example demonstrates how to use the `mailfilter` API.

```
-module(simplefilter).
-export([try_it/0]).
importance(M, C) ->
  Important = binary:compile_pattern([<<"AP">>, <<"Haskell">>, <<"Erlang">>]),
  case binary:match(M, Important, []) of
    nomatch -> {just, C#{spam => true}};
    _ -> {just, C#{importance => 10000}} end.

oneway(Mail) ->
  {ok, MS} = mailfilter:start(infinite),
  mailfilter:default(MS, importance, {simple, fun importance/2}, #{}),
  {ok, MR} = mailfilter:add_mail(MS, Mail),
  timer:sleep(50), % Might not be needed,
                   % but we'll give mailfilter a fighting chance to run the filters
  {ok, [{importance, Res}]} = mailfilter:get_config(MR),
  {MS, Res}.

another(Mail) ->
  {ok, MS} = mailfilter:start(infinite),
  {ok, MR} = mailfilter:add_mail(MS, Mail),
  mailfilter:add_filter(MR, "importance", {simple, fun importance/2}, #{}),
  mailfilter:add_filter(MR, ap_r0cks, {simple, fun(<<M:24/binary, _>>) ->
                                                   {transform, M} end}, flap),
  timer:sleep(50), % Might not be needed
  {ok, [{M, Config}]} = mailfilter:stop(MS),
  [Res] = [ Result || {Label, Result} <- Config, Label == "importance"],
  {M, Res}.

try_it() ->
  Mail = <<"Remember to read AP exam text carefully">>,
  {MS, Res1} = oneway(Mail),
  {_M, Res2} = another(Mail),
  {ok, _} = mailfilter:stop(MS),
  Translate = fun (Result) ->
                  case Result of
                    {done, #{spam := true}} -> "We git spam - nom nom nom";
                    {done, #{importance := N}} when N > 69 -> "IMPORTANT!";
                    {done, _} ->                  "Can wait for tomorrow";
                    inprogress ->                  "Need a new computer!"
                  end end,
  io:fwrite("One way: ~s~nor another: ~s~n" , [Translate(Res1), Translate(Res2)]).
```