Take-home Exam in Advanced Programming

Deadline: Friday, 11 November 2022 at 15:00

Version 0.99

Preamble

This is the exam set for the individual, written take-home exam on the course Advanced Programming, B1-2022. This document consists of 26 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: APpy: A simple parser generator

Don't worry, be 'APpy!

In the course we have looked primarily at language-integrated parsing in Haskell. In many other languages, parser combinators cannot be expressed so concisely and elegantly; in particular, backtracking may be quite hard to accommodate in a language that cannot immediately support it as an instance of the monadic framework. In such languages, it is common to instead write parsers and/or lexers in a dedicated, domain-specific language. Such a language provides a concrete, machine-processable notation for expressing context-free grammars, together with a way to specify how to build the intended structured representation of the the parsed text, typically as some form of abstract syntax tree

This approach was popularized for C in the 1970s by the `yacc` and `lex` tools, subsequently reimplemented as the Free variants `bison` and `flex`, and with more or less direct ports to other languages, both imperative and functional. A parser specification in this style combines a grammar specification with snippets of host-language code that get embedded directly into the generated parser, and get executed when that parser is then run on some concrete input.

A main advantage of a dedicated parser-specification language is that it may allow more grammars to be expressed directly, or to improve efficiency. A disadvantage is that it complicates the development process somewhat, in that it is now necessary to first run the parser-generator to obtain the parser program, and then compile and the parser program to actually get a runnable parser. This means that any syntactic or semantic issues with the embedded host-language snippets are only discovered when the constructed parser is compiled, at which time it may be hard to trace back a compile-time error to the specific location in the parser-specification file that caused it, and that needs to be fixed.

In this task, you will implement a simple parser generator for Haskell along these lines. An open-source parser generator for Haskell, with an interface quite similar to the `yacc` tool (but with many Haskell-specific adaptations) exists under the name under the name of HAPPY. Accordingly, our much less ambitious version will be called APPY. Note, however, that apart from the similar name and overall goal, APPY is very substantially different from both `yacc` and HAPPY; in particular, it is highly unlikely that you will find anything of value for this task in either the HAPPY documentation or its source code.

A tour of APPY

The following section gives an informal overview of the APPY features from a user's point of view. The next section goes into more detail on what you are asked to implement.

Basic BNFs The grammar is specified in a syntax similar to the one in the parser notes, with literal tokens written between double quotes, and all rules terminated by a period:

```
-- A simple expression grammar:
Exp  ::= Exp "+" Term | Exp "-" Term | Term.
Term ::= num | "(" Term ")".
```

A nonterminal name can by any sequence of letters, digits, and underscores, starting with a letter. (The *case* of the letter matters; see the paragraph "tokens and whitespace" below.)

A literal can be any *non-empty* sequence of *printable* characters between double-quotes. If the literal itself contains a double-quote, it must be doubled; for example, the 3-character token `="=` would be written as `"=""="`.

ϵ-alternatives are written as simply an empty sequence of symbols, e.g.

```
Digits ::= Digit Digitz.
Digitz ::= | Digit Digitz.
```

(Note that these examples are not yet valid APpy inputs because they are missing the semantic actions, discussed next.)

Comments in the grammar specification start with `--` and end with the next newline.

Semantic actions The purpose of APpy is to generate a parser that not only checks whether an input string can be generated by a grammar, but to construct the semantic meaning of the string, typically as an abstract syntax tree. For this purpose, every alternative in a rule ends with a *semantic action* specifying how to construct the meaning of this alternative from the meanings of the constituents in the sequence. For example, we could specify:

```
Exp  ::= Exp "+" Term         {Plus _1 _3}
      |  Exp "-" Term         {Minus _1 _3}
      |  Term                 {_1}.
Term ::= num                  {Cst _1}
      |   "(" Term ")"        {_2}.
```

A semantic action is simply an arbitrary Haskell expression, in which each special variables `_1`, `_2`, ... will be bound to the meaning constructed by the corresponding element of the preceding sequence of symbols.

The action will always be treated as a complete, parenthesized expression, and should not rely on indentation; instead, if needed, semicolons should be used to separate bindings in a do or `let`, alternatives in a `case`, etc. The semantic action can contain any number of arbitrary characters, and will not be modified or analyzed by APpy itself. However, if an action itself is to contain the characters `{` or `}` anywhere (even in string constants, etc., they must be doubled. For example,

```
Exp ::= ...
    |  "let" id "=" Exp "in" Exp  { Let {{var = _2, def = _4, body = _6}} }.
```

Such doubled braces are replaced by single ones in the actual Haskell code. A sole `}` immediately terminates the action, whereas a sole `{` is illegal.

Literal tokens, such as `"+"`, have the semantic meaning `()`. They would normally not be explicitly referenced in the semantic action, but they are still counted for the purpose of numbering.

Conversely, if some phrase has no semantic meaning (for example, a comment, as discussed), its semantic action should be given as `{()}`.

All semantic actions in a a single rule must be of the same type. (If the exact semantic action for a particular alternative has not been determined yet, it may be written as `{undefined}`, which makes it typable in any context.)

If a sequence of grammar elemets in an alternative is of length exactly one, the action may be omitted and corresponds to simply propagating the meaning of the single element in the sequence, as if the action had been written as `{_1}`.

Finally, mainly to aid the human reader (since Haskell's type inferencer can almost always deduce it), the type of a nonterminal's meaning can be optionally given on the left-hand side of the rule, as follows:

```
Exp {:Expr} ::= Exp "+" Term {Plus _1 _3} | ...
```

Again, the text in `{:···}` can be any well formed Haskell type, using the same convention for brace-doubling as in the semantic actions.

Nested choices Sometimes it is convenient to specify a simple choice in the middle of a grammar rule, as in EBNF, by enclosing the alternatives in parentheses. Correspondingly, in APpy we can write:

```
Exp ::= ...
     | Exp ("+" {Plus} | "-" {Minus}) Term {_2 _1 _3}
```

Variables in nested actions refer to constituents of the preceding sequence, *not* those in the outer rule. For example, in

```
Addr ::= reg ({\r -> RegPlain r} | "+" num {\r -> RegOffset r _2}) {_2 _1}
```

the first occurence of `_2` refers to the value of the numeric constant `num`, whereas the second `_2` is the function resulting from the parenthesized choice.

Also like in EBNF, there are convenient shorthands for introducing optional and repeated grammar elements. Their meanings have `Maybe` and list types, respectively. For example:

```
Exps{:[Expr]} ::= Exp Exp* {_1 : _2}.
Addr ::= reg ("+" num {_2} | "-" num {negate _2})?
  {case _2 of Nothing -> RegPlain _1; Just n -> RegOffset _1 n}.
```

(Note how we must separate the `case`-branches with `;` instead of indentation in the semantic action.)

Thus, the EBNF notation "[···]" can be expressed as `(···)?` in APpy, and "{···}" as `(···)*`.

Predicated and negated parsers Semantic actions and nested choices do not change what kinds of languages we can parse; they merely allow us to express the meanings of various phrases as we parse them, or specifying the syntax more compactly. However, APPY also contains two extensions to the context-free paradigm specifically meant for expressing practical parsers.

The first, *predicated* parsers, allow the semantic *meaning* of a phrase to affect whether it is *syntactically* well formed. It allows any element in a sequence to be followed by a *predicate* (expressed in general Haskell, like the semantic actions). For example,

```
var ::= Word{? `notElem` ["let", "in", "if", "then", "else"]}.
Word ::= Char{?isAlpha} *.
```

The Haskell expression in `{? ··· }` uses the same conventions for brace-doubling as semantic actions. It should have type *t* `->` `Bool` where *t* is the type of the meaning of the element it follows.

Since predicates (like `?` and `*`) are always attached to the preceding grammar element, the convention is to write them immediately next to the preceding element without intervening whitespace (but such whitespace is still formally allowed).

The second extension is negated parsers. A sequence element can be preceded by an exclamation mark (`!`), meaning that it should *not* be possible to successfully call that parser at that point in the input. For example, we can say

```
Num ::= Digit Digit* !Digit {_1:_2}
```

This will parse a maximal sequence of digits (similarly to `munch1 isDigit` in ReadP).

Again, the negation is conventionally written immediately before the element it negates. Only a single `?`, `*`, or `!` decoration is allowed on a grammar element, since combinations of them very rarely make sense in practice.

For efficiency the parser to be negated should be simple, often just a single character. The semantic value of a negated parser is always `()` (like for token literals), regardless of the type of the parser being negated.

In a grammar specification, predication groups tighter than the prefix/postfix decorations; for example, `!A{?p}` parses as `!(A{?p})`, not `(!A){?p}` (which wouldn't make much sense, anyway). Nested predications *are* allowed, as in as `Char{?isAscii}{?isLetter}`.

Tokens and whitespace Like parser combinators, APPY can be used to specify both lexical and syntactic grammars in a unified framework. This requires careful attention to where whitespace can or must occur. APPY uses the standard convention of letting designated *token parsers* (only) skip any following whitespace. Tokens in the grammar are either literals (i.e., strings between double quotes) or symbols with names starting with *lowercase* letter. Note that whitespace-skipping is added to each token parser at its *definition* (like with `lexeme`), not at its *uses*.

Character parsers Since double-quoted (i.e. token) literals skip following whitespace, we also need a way to express that whitespace is *not* allowed after some characters, such as the digits within a number, or the letters in an identifier. For that purpose, APpy allows *character literals*, written between single quotes:

```
num ::= '-'? Digits {maybe _2 (\_ -> negate _2} _1}
```

(where `Data.Maybe.maybe` is a concise way of writing a `case`-expression over a `Maybe` type.) Here, there can be no whitespace between the optional sign and the digits.

A character literal consists of exactly one *printable* character between single quotes. (So a literal single-quote character can be written directly as `'''` without any doubling or escaping.)

Quite commonly, we want to allow a choice between a fair-sized number of characters. While we could write, e.g., `('0'|'1'|`…`|'9')|` (with … replaced by an actual enumeration of the remaining digits, this gets cumbersome. Instead, we can use predication with a general single-character parser:

```
Digit ::= @{?isDigit}.
```

Here, `@` accepts any single character, further constrained by the predicate. This also allows us to specify individual non-printable characters by escaping to Haskell's general character syntax, e.g.,

```
Newline ::= @{?=='\n'}. Bell ::= @{?=='\7'}.
```

Token-separation parser The character parsers can be used to express the precise whitespace and comment rules of the grammar as a noterminal. Then, this designated *token-separation parser* named `_` (suggesting whitespace) is automatically called after each token parser, as well as initially. It is not possible to invoke `_` directly as part of a grammar rule.

However, there is a complication in that certain literal tokens (typically keywords) often must not be followed by additional letters or digits, while others (such as symbols) have no such restrictions; yet this difference is not evident from the token literals, and we don't want to hard-code any particular conventions int the parser generator.

Therefore, as a special case, the separation parser is told which literal it follows, so that it can make any special arrangements. The literal is availablle in the special String-typed variable `_0`, and may be used in predicates to influence the behavior of the parser. When the separation parser is invoked after a nonterminal designated as a token, as well as when skipping initial whitespace, `_0` is bound to the empty string (recall that token literals must be non-empty.)

For example, to ensure that keyword literals are not immediately followed by a letter, we can say:

```
_ ::= !Letter{?\_ -> _0 `elem` keywords) (@{?isSpace})* {()}.
```

When `_0` is *not* a keyword, the predicate `\_ -> ...` will always return `False` regardless of its argument, so the predicated parser `Letter{?...}` will fail, and therefore the *negated* parser `!Letter{?...}` will succeed, and go on to skip any following space characters. Similarly, if `_0` is a keyword but the next character is *not* a letter, the negated parser will succeed.

Buth when `_0` is a keyword (e.g., `"not"`) and next character in the input is (e.g., `'x'`), the negated parser will correctly fail, since the input string `"notx"` should not parse as a negation, bot only as an identifier.

Note that `_0` is *not* bound in any parsers invoked from the separation parser; this includes any parsers arising from nested choices or sequences.

Preamble and top-level parsers In many cases, the semantic actions in the parser specification use task-specific abstract-syntax datatypes, which need to be defined or imported. Similarly, predicates often refer to the various character-classification `isXxx` predicates from `Data.Char`. Finally, the generated parser will usually be a separate module with a suitable header and export list, which includes the generated parsers.

To accommodate this, a APpy grammar specification starts with a *preamble*, which is an arbitrary fragment of Haskell code that will be copied verbatim (without even treating open/close braces specially) to the generated parser file. For example,

```
module Expressions (Exp(..), parseExp) where

import qualified Text.ParserCombinators.ReadP as RP -- needed by backend
import Data.Char (isDigit)

data Exp = Cst Int | Plus Exp Exp | Minus Exp Exp
  deriving Show

parseExp :: String -> Either String Exp
parseExp = parseTop p_E
---
E ::= T {_1} | E "+" T {_1 `Plus` _3} | E "-" T {_1 `Minus` _3}.
T ::= num {Cst _1} | "(" E ")" {_2}.
num ::= Digit Digit* {read (_1:_2) :: Int}.
Digit ::= @{?isDigit}.
_ ::= ' '* {()}.
```

The preamble is separated from the grammar specification by a line containing only `---` (three consecutive dashes). The generated code expects `ReadP` (or a work-alike) to be available as `RP`.

The code in the preamble can use the function

```
parseTop :: RP.ReadP a -> String -> Either String a
```

to invoke one of the generated parsers, with `p_` prepended to its name. (For debugging, all parsers can be invoked directly with `readP_to_S` as usual. For ease of testing (since the object grammars might be ambiguous), the top-level parser will return the `Left`-tagged error message "ambiguous" (instead of calling error, as is usually recommended). Additionally, the generated code may contain some internal type and function definitions, all with names ending in a final underscore, which will hopefully not clash with any names introduced in the preamble.

Implementing APpy

The APpy application is divided into four main modules:

1. The Parser, which parses a grammar specification into an abstract syntax (or complains if the specification is ill-formed).
2. The Transformer, which transforms the user-written grammar into a form that can be directly used as the basis for an implementation, for instance by eliminating left recursion.
3. The Generator, which converts the suitably transformed grammar into a parser using the ReadP parser combinators as back end. (A back end using Parsec or some other combinator library would in principle also be possible, but would often require additional transformations, such as judicious insertions of try.)
4. The Main program which handles command-line processing, file IO, and invoking the relevant parts of the other modules.

A suitable a Generator and main program are already given, so you only need to implement the other two modules. The are weighted equally, but demonstrate different skills, so you should work on both

All modules share the same mechanism for reporting errors:

```
type ErrMsg = String    -- human-readable error messages
type EM = Either ErrMsg -- can be used directly as a Monad
```

Question 1.1: The Parser module

The concrete grammar of an APpy specification is shown in Figure 1. For consistency, we use the basic APpy syntax for literals and rules, but none of the other features, such as semantic actions, nested choices, etc. The lexical rules are also specified in English text, not as a formal grammar.

The corresponding abstract grammar is as follows (omitting the standard deriving-clauses):

```
type EGrammar = [ERule]
type ERule = (RLHS, ERHS)

type RLHS = (NTName, RKind, Maybe Type)
type NTName = String
data RKind = RPlain | RToken | RSep
```

```
Spec ::= preamble ERules.

ERule ::= ERule | ERule ERules.

ERule ::= LHS "::=" Alts ".".

LHS ::= name OptType | "_".

OptType ::= | "{:" htext "}".

Alts ::= Seq | Seq "|" Alts.

Seq ::= Simple | Simplez "{" htext "}".

Simplez ::= | Simple Simplez.

Simple ::= Simple0 | Simple0 "?" | Simple0 "*" | "!" Simple0.

Simple0 ::= Atom | Simple0 "{?" htext "}".

Atom ::= name | tokLit | "@" | charLit | "(" Spec ")".
```

`preamble` Any sequence of characters, terminated by `---` on a single line. (The terminator is not included in the preamble text).

`name` Any sequence of (Unicode) letters, digits, and underscores, starting with a letter. There are no reserved names.

`htext` Any sequence of arbitrary characters, including any leading whitespace. Characters `{` and `}` to be included in the sequence must be written as `{{` and `}}`, respectively. When following an opening `{`, the `htext` must not *start* with a `:` or `?`.

`tokLit` Any *non-empty* sequence of *printable* characters, enclosed in double-quotes. If the sequence is itself to contain a double-quote, it must be written as `""`.

`charLit` Any *printable* character (including `'`), enclosed in single-quotes.

Tokens may be surrounded by arbitrary whitespace and comments (from `--` up to and including the terminating newline).

Figure 1: Concrete syntax of APpy specifications

```
data ERHS =
    ESimple Simple
  | ESeq [ERHS] HText
  | EBar ERHS ERHS
  | EOption ERHS
  | EMany ERHS
  | EPred ERHS HText
  | ENot ERHS

data Simple =
    SLit String
  | SNTerm String
  | SAnyChar
  | SChar Char
  | SPred Simple HText
  | SNot Simple
  | SDummy

type HText = String
```

The left-hand side of a rule contains a nonterminal name, its *kind* (“plain”, “token”, or “separator”), and an optional type. The right-hand side covers all the ways of combining grammar elements, some of which are grouped in the auxiliary data type `Simple`. Your parser should use the `EPred` and `ENot` constructors for the abstract syntax trees. (The `S`-variants may be used in the final BNF grammar.)

Your parser should export just a single function

```
parseSpec :: String -> EM (Preamble, EGrammar)
```

That is, given an input string (typically read from a file), it should either return as a parse result the preamble and the abstract syntax of the grammar specification. Alternatively, it should fail with an (not necessarily particularly informative) error message.

For implementing your parser, you may use either ReadP or Parsec. 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`. As always, don’t worry about providing informative syntax-error messages if you use ReadP.

Note, however, that since the grammar specification consists of a sequence of relatively short grammar rules, it may be possible for even a ReadP-based parser to give an error message with some indication of how much of the input string it was able to process before encountering a syntax error. This is *not* a requirement, but you may want to consider it to aid your own fault-finding. If you do something clever, you may want to briefly mention it in the report. (Note that this is for *failed* parses; if `parseSpec` returns a normal result, it should mean that the entire string was successfully and unambiguously processed.) ***

Question 1.2: The Transformer module

The Transformer has three main tasks:

1. Conversion from EBNF to BNF.
2. Left-recursion elimination (LRE)
3. Left-factorization.

Question 1.2.1: Conversion

The first task of the transfomer is to convert grammar specifications that may contain EBNF extensions (nested sequences, choices, and repetitins) to plain BNF. At the same time, it should perform some basic validaton: check that all nonterminals referenced in the grammar are actually defined, but only once. (It should not check that any parsers referenced in the preamble are actually defined in the grammar; that's the user's responsibility.) Also, the grammar should always contain a token-separator definition. (If whitespace-skipping is not relevant for a particular grammar, the user must explicitly says so by defining "`_ ::= {()}.`".)

Note that even such a validted input grammar may be ambiguous, or otherwise unsuitable for top-down parsing. That is the user's responsibility; the conversion function should not attempt to detect or correct that.

The BNF grammar has a much more regular structrure than the EBNF one

```
type Grammar = [Rule]
type Rule = (RLHS, [([Simple]{-seq-}, Action)]{-alts-})
```

That is, a rule RHS contains a number of alternatives (possibly just one), each of which is a sequence of simple elements and a corresponding semantic action.

The actions are given by the following datatype

```
data Action =
    AUser HText
  | AVar String
  | ALam String Action
  | AApp Action Action
  | ACst String

type HText = String   -- Haskell text from user
type Type = Action
```

`AUser` represents Haskell text taken from directly from the input gramma, and may contain occurences of _*i* variables. It will always be parenthesized when used. The other constructors are used for wrapping the user actions in extra lambda-abstractions and/or applications, as well as closed Haskell terms, such as `Just`, which is used when transforming away EOption. Constructors with multiple arguments, such as (:) can be handled by nested uses of AApp.

When producing the parsers, the Generator will take care of inserting parentheses, spaces, etc. in accordance with the rules of Haskell's lambda-abstraction and application syntax. ACst should be able to be inserted in any context (so it should include its own outer parentheses if needed), and should not use any _-variables.

For this subtask, your Transformer module hould export a function

```
convert :: EGrammar -> EM Grammar
```

If the input grammar does not use any of the EBNF features, the conversion should be completely straightforward. But when, e.g., a nested EBar occurs inside a ESeq, a new auxilary nonterminal with a corresponding rule must be generated. For example, if the input grammar has the shape

```
A ::= B (C | D E {...(a)...}) F G {...(b)...}
```

the transformed one should look like

```
A ::= B A' F G {...(b)...}.
A' ::= (C | D E {...(a)...}).
```

where A' is some fresh nonterminal (which doesn't need to have a name related to A, but must be globally unique in the output grammar). The semantic actions (a) and (b) are simply moved along with the sequences the are attached to.

Note that lexical specification of the APpy grammar does not allow primes in nonterminal names, so you can use them in freshly generated names to be sure that they do not clash with those of unrelated nonterminals. Hoewver, you may still need to introduce multiple unique names, e.g., if the original rule contains more than one embedded choice.

Optional and repeated elements are handled analogously. For example,

```
A ::= ... B? ...
C ::= ... D* ...
```

should be converted to

```
A  ::= ... A' ...
A' ::= B {Just _1} | {Nothing}.
C  ::= ... C' ...
C' ::= D C' {_1:_2} | {[]}.
```

(Again A' and C' are some fresh names). More generally, B and D may themselves contain nested choices or sequences; for example

```
A ::= ... (B1 | B2 | B3)? ...
C ::= ... (D1 D2 {(_1,_2)})* ...
```

can be transformed to

```
A  ::= ... A' ...
A' ::= B1 {Just _1} | B2 {Just _1} | B3 {Just _1} | Nothing.
C  ::= ... C' ....
C' ::= D1 D2 C' {(_1,_2):_3} | {[]}.
```

Question 1.2.2: Left-recursion elimination (LRE)

An essential transformation to make many grammars suitable for use as parsers is elimination of left recursion. Recall from that a grammar of the form

$$A ::= A\,\alpha_1 \mid \cdots \mid A\,\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

where none of the sequences β_i start with the nonterminal A, can be transformed to the grammar

$$\begin{aligned} A &::= \beta_1\,A' \mid \cdots \mid \beta_m\,A' \\ A' &::= \alpha_1\,A' \mid \cdots \mid \alpha_n\,A' \mid \epsilon \end{aligned}$$

where A' is a fresh nonterminal name. (Note that the alternatives in the original grammar may have the α- and β-alternatives interleaved, not nicely separated.) This transformation will fail if $m = 0$ (no base cases), or if one of the α_i is empty (in which case the resulting grammar will still be left-recursive).

When the grammar contains semantic actions, we also have to transform those to match the transformed syntax. Consider the concrete example:

```
E ::= E "+" num {Plus _1 (Cst _3}) | var {Var _1}.
```

After transformation it should become something of the form

```
E ::= var E' {???}.
E' ::= "+" T E' {???} | {???}.
```

What should the new semantic actions be? In the transformed parsers with attributes, the parser for `E'` takes an extra accumulating parameter containing the expression built up so far, and extends it with zero or more further addends. In our formulation, the semantic meaning of an `E'` -phrase should therefore itself be a function from expressions to expressions, which we can apply to the accumulator:

```
E  ::= var E'       {_2 (Var _1)}.
E' ::= "+" num E'  {\e -> _3 (Plus e (Cst _2)}
    |               {\e -> e}.
```

However, this seemingly requires us to do a substantial modification to the original semantic actions, which could be arbitrarily complex Haskell code. This is because, in the transformed code, the positions of the constituents got shifted when we removed the initial left-recursive call to E.

However, with a simple trick, we can make the transformed grammar have almost the same structure as the original one, by introducing a dummy nonterminal symbol D with no meaningful semantics:

```
E  ::= var E'        {_2 (Var _1)}
E' ::= D "+" T E'    {\_1 -> _4 (Plus _1 (Cst _3)}
    |  D E'          {\_1 -> _1}.
D  ::=               {undefined}.
```

Here, the semantic actions for the two alternatives in E' will not actually depend on the meaning of the initial D in _1, because that variable is immediately shadowed by the inner lambda-binding of _1. Now the semantic actions Var _1 and Plus _1 (Cst _3) can now be used *verbatim* (only with some wrappers around them, which can be expressed with ALam, AApp and AVar) in the semantic actions of the transformed grammar. This transformation generalizes straightforwardly to the general case of a self-left recursive grammar where n and/or m may be greater than 1, and the sequences α_i and β_j can likewise be of any length.

In practice, there's no need to introduce a separate 'D' for every instance of the transformation; they can all share the same, “built-in” element EDummy.

Your Transformer module should export a function

```haskell
lre :: Grammar -> EM Grammar
```

that eliminates all left-recursion in the grammar, as outlined above. It is only expected to handle *direct* (self) left-recusion, not more complicated instances, such as when the left-recursive reference happens through a chain of other nonterminals, or when the recursive invocation is not syntactically leftmost in the sequence, but is preceded by other elements that may all succeed without reducing the input string. If the input grammar contains no left-recursive rules, lre should not modify it.

The function should return a suitable error message (via EM, not by error!) when the transformation fails. This includes at least the situation in which there is no base case (i.e. $m = 0$), or when an α_i is empty. It is also *allowed* to detect other cases where the grammar is patently left-recursive in the general sense (and hence unsuitable for top-down parsing), but your code is unable to transform it, such as:

```
A ::= B | "a".
B ::= A | "b".
```

Question 1.2.3: Left-factorization

A second transformation that is frequently relevant in practice, even if not as essential as lre (at least in a fully backtracking parser such as ReadP), is left-factorization. Again, we only consider the case where opportunity is syntactically evident, such as a rule of the form

$$A ::= \alpha\beta_1 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m.$$

where α is non-empty, $n \geq 2$ (but $m \geq 0$), not all the β_i start with the same element, and the γ_i do not start with α. (As for LRE, the alternatives need not be nicely listed in that order.) Here, a top-down parser may repeatedly re-parse the sequence alpha, even if only the last alternative β_n succeeds. When alpha itself may involve (non-leftmost, and possibly indirect) recursive references to A, this duplication of work easily cause exponential slowdowns. To address this, we transform the grammar to parse the α only once and share it across all the βs:

$$\begin{aligned} A &::= \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_m \\ A' &::= \beta_1 \mid \cdots \mid \beta_n \end{aligned}$$

Again, when the grammar contains semantic actions, we also need to transform those correspondingly. For a simple example, consider:

```
T ::= base                   {RBase _1}
   |  base "->" T            {Arrow (LBase _1) _3}
   |  "(" T "," T")"         {Prod _2 _4}.
```

Here we can again use the trick of introducing dummy elements in the transformed sequences, corresponding to the shared part, and passing the actual semantic values instead as function arguments:

```
T ::=   base T'              {_2 _1}
   |   "(" T "," T")"        {Prod _2 _4}.
T' ::=  D                    {\_1 -> RBase _1}
   |    D "->" T             {\_1 -> Arrow (LBase _1) _3).
```

Note that the common prefix may contain more than element; in that case, we just use multiple `D`s as placeholders. For example, the grammar

```
T ::= base "." num           {RBaseN _1 _3}
   |  base "." num "->" T    {Arrow (LBaseN _1 _3) _5}.
```

is transformed into:

```
T ::=  base "." num T'      {_4 _1 _2 _3}.
T' ::= D D D                {\_1 _2 _3 -> RBaseN _1 _3}
   |   D D D "->" T         {\_1 _2 _3 -> Arrow (LBaseN _1 _3) _5}.
```

Again, we can use the original semantic actions unmodified, only with some wrappers. (Recall that, in Haskell, `\x y -> E` is just syntactic sugar for `\x -> \y -> E`, and `f E1 E2` for `(f E1) E2`, so we can express everything with just single-argument ALam and AApp.)

Note also that, since the semantic actions in the original grammar did not use _2 (i.e., the semantic value of the ".", which is just ()), we don't strictly need to pass it to the function denoted by `T'`, but we do so for uniformity.

Finally, keep in mind that – unlike for left-recursion – there may be multiple opportunities for left-factorization within a single rule. For example, in the original pattern, some of the γ_i may also share a separate common prefix, which can be independently factored out. More subtly, α may share a common prefix with a γ_i, or some (but not all) βs may start with the same symbol. For example, consider the following grammar:

```
T ::= "(" ")" "->" T          {Arrow0 _4}
   |  "(" T ")" "->" T        {Arrow1 _2 _5}
   |  "(" T "," T ")" "->" T  {Arrow2 _2 _4 _7)
   |  base                    {Base _1}
```

Here, the first 3 alternatives share the prefix `"("`, but only the last 2 share the longer `"(" T`, so we can left-factorize the common parts in two stages:

```
T   ::=  "(" T'                 {_2 _1}
     |   base                   {TBase _1}.
T'  ::=  D ")" "->" T           {\_1 -> Arrow0 _4}
     |   D T T''                {\_1 -> _3 _1 _2}.
T'' ::= D D ")" "->" T          {\_1 _2 -> Arrow1 _2 _5}
     |   D D "," T ")" "->" T   {\_1 _2 -> Arrow2 _2 _4 _7}.
```

Clearly, performing this left-factorization by hand is both error prone and leaves the result rather less readable and maintainable than the original grammar (even if we remove the `D`s and renumber the variables in the semantic actions), which is precisely why it should be automated.

Your Transformer module should export a function

```
lfactor :: Grammar -> EM Grammar
```

that performs as much left-factorization as possible. (If there are no opportunities, it should leave the grammar unmodified.) You may assume that the input grammar does not contain any left recursion.

`lfactor` is only expected to factor each rule locally, independently of the others. Sometimes it may be possible to introduce more factoring opportunities by expanding (possibly recursively) the definitions of some nonterminals used in a rule. But that pre-transformation can (if relevant) be performed by the user already in the original grammar.

Although left-factorization cannot strictly speaking fail (it just may do nothing), it should recognize and report *blatant* ambiguities in the grammar, i.e., when two or more alternatives in a rule consist of exactly the same (possibly empty) sequence of elements, regardless of whether the associated semantic actions are the same or not. (Even if the actions are syntactically identical, a ReadP-based parser will succeed with the same result more than once, which still gets treated as ambiguous, especially since the parse results need not be `Eq`-types.)

In your report, you should briefly explain how you implemented left-factorization, for example if you used some kind of auxiliary data structure for keeping track of the tree of prefixes. If you only implement some restricted patterns of left-factorization, document it; even those may help, such as in the first example with a right-associative infix operator. ***

Your transformations should preserve any optional type signatures of nonterminals in the input grammars. They are *not* required to provide type annotatons for any auxiliary nonterminals the introduce as part of the transformations (but it they do, those annotation must be correct, i.e., not introduce any type errors in the generated Haskell programs).

For all three functions, your code should – in addition to being correct – adhere to the general principles for separating concerns. In particular, they should use suitable monad(s) as you deem appropriate. Think carefully about RWSE. and justify your choice in the report. ***

While the three functions can be implemented completely independently of each other, you may find it convenient to share some infrastructure (such as fresh-name generation) between them.

Generator and driver

The generator (which is already provided) exports a single function and a string

```
render :: Grammar -> String
prelude :: String
```

The function `render` generates straightforward parser-combinator code from a grammar, essentially as sketched in the parser notes. The `prelude` (not to be confused with the user-supplied *preamble*) contains a few auxiliary definitions that may be used by the code generated by `render` and/or referenced from the preamble (such as `parseTop`).

Note that `render` performs no sanity checking of its own; hence the output is not of `EM` type. If the input Grammar is erroneous (for example, uses undefined or multiply defined nonterminals, or contains flawed Haskell code in the semantic actions or predicates), the resulting code (when combined with the prelude and preamble) may fail to even compile.

For your own *development* and *testing* work, you may modify the `Generator` module as you see fit (for example, make it print debugging output, or ignore the semantic actions); we will not use it for anything in our own tests. Note, however, that the output produced by your Transformer still needs to work correctly with the *standard* Generator module, so you cannot introduce any incompatible changes to the semantics of the parser combinators, for instance.

If you believe that there is a significant design or implementation flaw in the Generator that causes it to generate incorrect or inappropriate parser code from your otherwise correct Grammar, you may ask for clarification on the forum, just like for any other dubious-looking aspects of the specification in this document.

The Driver module is completely optional, and will not be used for anything. Again, you may modify it entirely as you see fit for your own development process.

How to get started

As usual, it is not a recommended strategy to work only on Parser (or Transformer); although they both work in the domain of parsing, they demonstrate complementary skills, and it's easier to collect the low-hanging fruits in each than to code up a near-perfect implementation of one module. And also as usual, try to work in a breadth-first manner, implementing (and testing!) some basic functionality before adding more complicated features.

For the Parser, start with the basic grammar specifications, i.e., rules with alternatives, sequences, alternatives, (non-token) nonterminals, (string) literals, and semantic actions. Unless you immediately see how to handle then, wait with the default actions, embedded sequences and choices, and predicated, negated, character, and whitespace parsers. Also, don't fret about non-essential lexical issues such as quote- and brace-doubling, comments, preamble, etc.

For the Transformer, start by getting the basic functionality of `convert` to work, i.e., the cases with no embedded choices or sequences, so that you don't need to generate multiple output rules for a single input rule. Leave that, and possibly (some of the) error-checking, for later.

Second, implement `lre`. This will give you practice in working with rule generation and transforming semantic actions in a relatively simple setting.

Then, do at least some simple instances of `lfactor`, for example, if all the alternatives start with the same sequence (i.e., if m = 0), or only considering α has length 1. Generalize as you see fit, interleaving with implementing more of the `convert` functionality.

For all three functions, work out some specific examples of input and output grammars on paper before you try to write the code to do so automatically,

Testing

As usual, to support your assessment, you should do systematic testing of your code. Your tests should be predominantly (or exclusively) black-box, but we have made accommodations for white-box testing if there are parts or aspects of your code that you cannot test adequately otherwise. Specifically, your actual code should go in the ParserImpl and TransformerImpl modules, which export everything; then the actual Parser and Transformer are just wrappers then import from the implementation modules, and re-export only the API functionality. Your black-box tests (in tests/BlackBox.hs) should only import from the API modules, while the ones in tests/WhiteBox.hs also have access to any internal definitions.

Moreover, the white-box tests are allowed to assume particular choices for the otherwise unspecified aspects of the desired functionality, such as the exact names of fresh nonterminals, or the order in which rules or alternatives are listed. But note that this is a two-edged sword at best: tying the tests too closely to your implementation risks not detecting genuine problems due to a misreading of the specification, because the test case just replicates whatever your code happens to be doing. Also, even minor changes to your code (for example, to fix an unrelated problem) may cause a large number of white-box tests to spuriously fail, requiring them to be reworked.

Note that we *may* evaluate the quality of your black-box tests separately from the the code they are testing, such as whether they correctly find planted bugs in a sample implementation of the Parser or Transformer, as well as not reporting spurious errors for correct implementation. In particular, if you do anything clever when testing the mostly relational specifications of the Transformer, for example property-based testing, or symbolic evaluation of our generated parsers, you should mention it in the report.However, your primary focus and priority should still be on correctness and completeness of your own code, not of its test suite. ***

For your amusement, and to demonstrate a real-word use of the system, in the `code/appy/examples` directory we have included the Warmup and BoaParser grammars from Assignment 3, re-expressed as APPY specifications. The latter exercises much, but far from all, of the desired functionality. While you may use the Boa parser as a final acceptance test (it should produce a fully functional parser), it is *not* suitable for test-driven development of either the Parser or the Transformer module, especially if you also use A3 OnlineTA to run the tests. For one thing, it makes for a fairly long turnaround time, but it's also quite hard to use a failed test to isolate and diagnose the problem. Write your own, smaller tests that exercise each feature separately; you will need these for the test suite anyway.

Keep in mind that many problems in the Transformer will manifest themselves by the generated parser containing type (or, if you mangle the actions or predicates too badly, syntax) errors. It will generally be easier to see what went wrong from by looking at the resulting Grammar, not the messy Haskell parser code that the Generator produces from it.

General instructions

All your code should be put into the provided skeleton files under `code/appy/`, as indicated. In particular, the actual code for each module `Mod` should go into `src/ModImpl.hs`. Do not change any of the other files, especially `Definitionss.hs`, which contains the common type definitions presented above; nor should you modify the type signatures of the exported functions in the APIs. Doing so will likely break our automated tests, which may affect your grade. Be sure that your codebase builds against the provided `app/Generator.hs` with `stack build`; if any parts of your code contain syntax or type errors, be sure to comment them out for the submission.

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

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

For assessment of performance and robustness of especially the Transformer module, it is potentially relevant to consider these aspects for both the transformation code itself, and for the parser generated from output. (On the other hand, correctness of a transformer function intrinsically includes correctness of the resulting parser.)

Question 2: Stock Exchange

The task in this question is to implement a simple stock exchange called Erl Street. It allows users to open accounts that can hold money and stocks. The fundamental event is that of a *trade*, which happens when one user has made an offer at the exchange while another user has installed a trader that accepts the given offer.

General comments

This question consists of two sub-questions: Question 2.1 about implementing an API for starting an Erl Street server and for trading, 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 an) implementation of the `erlst` module from Question 2.1.

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 26, that lists expected topics for your report.

Terminology

A *stock exchange* maintains a set of *accounts*, active *offers*, and *traders*. Each item has a corresponding unique identifier: every account has an account ID, etc.

The *holdings* of an account consist of an amount of money, measured in ISK, and a finite set of stocks together with amounts of each stock. For example, an account may consist of 2000 ISK, 3 shares of Haskell Ltd. and 5 shares of Erlang Inc.

An *offer* is described by the name of a stock and an asking price. Offers are always for selling 1 share of the stock for the given price. For example, there might be an offer to sell a Haskell Ltd. share for 50 ISK.

A *trader* is described by a strategy, a function from offers to decisions. The trader is presented with each offer in turn and makes decisions based on its strategy. For example, we could have a trader that is only willing to buy Erlang Inc. shares and only for at most 42 ISK per share.

```
-type account_id() :: term().
-type offer_id() :: term().
-type trader_id() :: term().
-type stock() :: atom().
-type isk() :: number().
-type stock_amount() :: pos_integer().
-type holdings() :: {isk(), [{stock(), stock_amount()}]}.
-type offer :: {stock(), isk()}.
```

```
-type decision :: accept | reject.
-type trader_strategy() :: fun((offer()) -> decision()).
```

Question 2.1: The `erlst` module

Implement an Erlang module `erlst` with the following API.

- `launch()` for starting a `erlst` server. Returns `{ok, S}` on success, or `{error, Reason}` if some fault occurred.
- `shutdown(S)` stops the stock exchange. All accounts are closed and all traders are stopped. It returns the total number of trades that have been executed by the stock exchange.
- `open_account(S, Holdings)` for opening an account with the stock exchange. The account holdings `Holdings = {Money, Stocks}` initially consist of an amount of ISK given by `Money` as well as the stocks given by `Stocks`. Returns `Acct`, a fresh account ID.
- `account_balance(Acct)` for querying the balance of an account. Returns (a snaphot of) the holdings for the account `Acct` in the same format as the second parameter of `open_account/2`. It should return immediately and not wait for ongoing trades to be settled.
- `make_offer(Acct, Offer)` adds `Offer` to the stock exchange and associates it with account `Acct`. Returns `{ok, OfferId}` or `{error, Reason}` if `Offer` is not a valid trade offer.
- `rescind_offer(S, OfferID)` removes the offer identified by `OfferID` from the stock exchange. It is a non-blocking operation.
- `add_trader(Acct, Strategy)` installs a trader for the account `Acct` and returns `TraderID`, a fresh trader ID. The trader function `Strategy` maps offers to decisions (`accept` or `reject`). Throwing an exception counts as a rejection. It is assumed to be deterministic, i.e. for a given offer `Offer` the call `Strategy(Offer)` returns the same decision every time.

  A trade is executed whenever the following conditions are met:

  - There is an offer on the exchange.
  - A trader accepts the offer.
  - The account that made the offer holds at least 1 unit of the stock being offered.
  - The account that installed the trader has at least the amount of ISK requested by the offer.

  Executing a trade means transferring 1 unit of the offered stock from the seller to the buyer, moving the requested amount of ISK from the buyer to the seller, and removing the offer from the stock exchange. At any given point in time it is possible that multiple traders satisfy the requirements for executing a trade involving a particular offer. In this case it is unspecified who gets to trade.

- `remove_trader(Acct, TraderID)` removes the trader identified by `TraderID`. It is a non-blocking operation.

The stock exchange should be robust in the face of trader strategies that take a long time, or even forever, to compute a decision. This includes, but is not limited to, the following:

- Shutting down the stock exchange should not be delayed because of slow traders.
- If a trader has accepted and is able to execute a trade, the stock exchange should execute the trade without waiting for the decisions of other—possibly slow—traders.
- Any execution of a trader strategy function should be stopped when the outcome no longer has any relevance, e.g. if the offer has been rescinded or the trader has been removed.

How to get started

Start by implementing `launch/0`, `shutdown/1`, `open_account/2`, `make_offer/2` and `add_trader/2` as this subset of the API allows meaningful testing. Most of the complexity in the system comes from the behaviour of traders, so consider making some simplifications initially. For instance, your first version might ignore the problem of robustness. You could also start by assuming that all traders are installed before any offers are made (or vice versa).

Consider developing your test-suite along side your implementation. This will help you greatly, should you see the need for refactoring your code.

Question 2.2: Testing `erlst`

Make a module `test_erlst` that uses `eqc` QuickCheck for testing a `erlst` module. We evaluate your tests with various versions of the `erlst` module that contain different planted bugs and check that your tests find the planted bugs. Thus, your tests should only rely on the API described in the exam text.

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

- `mkstrategy(Opr)` a helper function that generates a terminating deterministic trader strategy function from the argument `Opr`. You decide which values are valid for `Opr`. We will not call this function directly, only through symbolic calls generated by your generators.
- A QuickCheck generator `reliable_strategy()` that generates a symbolic call which can be evaluated to a trader strategy function.

  For instance, four samples from this generator could be:

  ```
  {call,test_erlst,mkstrategy,[buy_everything]}
  {call,test_erlst,mkstrategy,[{buy_cheap, 20}]}
  {call,test_erlst,mkstrategy,[{buy_only, [haskell_ltd, boa_llc]}]}
  {call,test_erlst,mkstrategy,[{both, [{buy_cheap, 8}, {buy_only, [erlang_inc]}]}]}
  ```

  Again, remember that the arguments to `mkstrategy/1` are up to you to decide. The given samples are just for inspiration.
- A QuickCheck property `prop_value_preservation` which checks that the total amount of money and total amount of each stock on the exchange does not change as long as no new accounts are opened.
- A QuickCheck property `prop_total_trades` which checks that the number of trades returned by `shutdown/1` is less than or equal to the number of calls to `make_offer/2`.
- A `test_all/0` function that runs all your tests and only depends on the specified `erlst` 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_` (or `apqc_` if you use that). Remember that the code called from `test_all/0` must only depend on the specified `erlst` 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 `erlst` API should start with the prefix `prop_`. If you have properties that are specific to *your* implementation of the `erlst` 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 `erlst`.

Topics for your report for Question 2

Your report should document:

- For Question 2.1:
  - Clearly explain if you support the complete API or not. If not, which parts you don't support.
  - If you have made any assumptions, especially simplifying assumptions. In particular if you have made any assumptions about traders.
  - How many processes your solution uses, what role each process has, and how they communicate.
  - What data your processes maintain.
  - Overall, the interesting parts of Question 2.1 are the interactions of making new offers, creating new traders and executing trades. You should make sure that your report cover these parts.
- For Question 2.2:
  - Clearly explain if you have implemented all required parts or not. If not, which parts you don't support.
  - Explain the idea behind and how your properties works.
  - The quality or limitations of your testing. Especially the QuickCheck parts explicitly required in Question 2.2. Explain how you have measured this quality.
  - Summarise how you have tested aspects and functionality not covered by the required properties in Question 2.2.

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