CS 381, Spring 2025 — Homework 2 (Abstract Syntax)

Important Reminders!

1. Put your solution into a single Elm file (ending in `.elm`), and convert that file into a pdf file (e.g., via printing). Upload the pdf file to the Pensieve grading web site:
   `https://tutor.pensieve.co/student/classes/oregonstateuniversity_cs381_sp25/external-assignments`
2. **Only submit files that compile without errors!** (Put all non-working parts in comments.)
3. You must do all homework assignments by yourself, without the help of others. Also, you must not use services such as Chegg or Course Hero. If you need help, simply ask on Canvas, and we will help!
4. You can work in teams of ***up to four*** students to create and submit a common homework solution. To this end, first create an alphabetically sorted (by last name) list of the team members, and add this information at the beginning of the submitted Elm file, like so.

```
{- GROUP:
   Helena Eagen
   Dylan George
   Mark Scout
-}
```

   The other group members **must not** submit a solution. All group members will receive the same grade.

Exercise 1. Mini Logo

Mini Logo is an extremely simplified version of the Logo language for programming 2D graphics. The idea behind Logo and Mini Logo is to describe simple line graphics through commands to move a pen from one position to another. The pen can either be "up" or "down." Positions are given by pairs of integers. Macros can be defined to reuse groups of commands. The syntax of Mini Logo is as follows (nonterminals are typeset in italics, and terminals are typeset in typewriter font).

cmd ::= `pen` *mode*
| `moveto (`*pos*`,`*pos*`)`
| `def` *name* `(` *pars* `)` *cmd*
| `call` *name* `(` *vals* `)`
| *cmd*`;` *cmd*

mode ::= `up` | `down`

pos ::= *num* | *name*

pars ::= *name*`,` *pars* | *name*

vals ::= *num*`,` *vals* | *num*

(a) Define the abstract syntax for Mini Logo as Elm types. Remember that unspecified nonterminals, such as *num* and *name*, should be represented by corresponding predefined Elm types, such as `Int` and `String`.

(b) Write a Mini Logo macro `vector` that draws a line from a given position `(x1,y1)` to a given position `(x2,y2)` and represent the macro in abstract syntax, that is, as an Elm data type value.

Note: What you should actually do is write a Mini Logo program that defines a vector macro. Using *concrete syntax*, the answer would have the following form.

```
def vector (...) ...
```

It might be a good idea to write the solution in concrete syntax first. But then you should write the same Mini Logo program in *abstract syntax*, that is, you should define a value built with Elm constructors that starts as follows (assuming `Def` is the constructor that represents the `def` production in the Elm type used for *cmd*).

```
vector = Def "vector" ... ...
```

You only need to submit this Elm definition of the value `vector` as part of your Elm program. (If you like, you can include the concrete syntax as a comment, but it is not required.)

Remember: The Elm program you submit doesn't have to draw anything. It only needs to contain the abstract syntax of the macro `vector`.

Exercise 2. Grammar Grammar

Consider the following grammar that describes the syntax of the language for grammar definitions.

grammar	::=	*prod* `;` ... `;` *prod*
prod	::=	*nt* `::=` *rhs* `|` ... `|` *rhs*
rhs	::=	*symbol**
symbol	::=	*nt* | *term*

A grammar is given by a list of (grouped) productions (*prod*), each of which consists of a nonterminal *nt* and a list of alternative right-hand sides (*rhs*). A right-hand side of a production is given by a sequence of terminal (*term*) and nonterminal (*nt*) symbols.

Note carefully the difference between the object language symbols `::=` and `|` (typeset in blue typewriter font) and the similar-looking symbols ::= and | that belong to the grammar metalanguage.

(a) Give Elm type (alias) definitions for the types `Grammar`, `Prod`, `RHS`, and `Symbol` to represent the abstract syntax for the above language. As part of your definitions, use the following type aliases `NonTerm` and `Term`.

```
type alias NonTerm = String
type alias Term    = String
```

(b) Consider the following grammar for a small imperative language Imp that we already encountered in class.

cond	::=	`T` | `not` *cond* | `(` *cond* `)`
stmt	::=	`skip` | `while` *cond* `do` `{` *stmt* `}` | *stmt*`;` *stmt*

Represent this grammar by an Elm value of type `Grammar` defined in part (a).

```
imp : Grammar
imp = ...
```

Note: It might be useful to break this definition down into smaller parts and create a few auxiliary definitions. For example, you can define a separate name for each `Prod` value. These can then be used in the definition of the value `imp`. You may even want to consider separate definitions for each right-hand side.

(c) Define the following two functions for extracting all defined nonterminals and all used terminals in a grammar.

```
nonterminals : Grammar -> List NonTerm
terminals : Grammar -> List Term
```

For the value `imp` defined in part (b), the functions would produce the following results.

```
> nonterminals imp
["cond","stmt"]

> terminals imp
["T","not","(",")","skip","while","do","{","}",";"]
```

Note: Depending on your representation chosen in (a), it might be beneficial to use the list functions `map` and `concat`. (You can import them from the module `List`.)

Note also that while the definition of `nonterminals` might be straightforward, the definition of `terminals` will probably require more effort, since terminal symbols are scattered over RHSs of multiple productions. It might therefore be a good idea to define the function `nonterminals` with a number of auxiliary functions that each can extract nonterminals from symbols, RHSs, and productions, respectively.

Exercise 3. Regular Expressions

A regular expression defines a language, that is, a set of words (or strings) over some basic alphabet Σ. For the purpose of this exercise, Σ contains all characters of the Elm built-in type `Char`. The following grammar defines the syntax of regular expressions.

regex ::= ϵ | `.` | *char* | *regex*`?` | *regex*`*` | *regex*`+` | *regex regex* | *regex*`|`*regex* | `(`*regex*`)`

Note carefully that the bar symbol `|` in the second-to-last production for alternatives is part of the concrete syntax and must not be confused with the bar symbol | that is part of the grammar meta notation. Note also that the nonterminal *char* ranges over characters of the alphabet Σ; it should be represented in the abstract syntax definition by the Elm type `Char` (in the same way as *num* is represented by `Int`).

(a) Define the abstract syntax for regular expressions as an Elm type `RegEx`.

```
type RegEx = ...
```

(b) Different regular expressions can denote the same language, and we can often simplify regular expressions by exploiting those equivalences. For example, for any given regular expression *e*, `((`*e*`)*)*` and `(`*e*`)*` denote the same language. We can express this fact as `((`*e*`)*)*` $\equiv$ `(`*e*`)*`. Similarly, we have `((`*e*`)+)*` $\equiv$ `(`*e*`)*`, `((`*e*`)*)+` $\equiv$ `(`*e*`)*`, `((`*e*`)+)+` $\equiv$ `(`*e*`)+`, and `(`*e*`|`*e*`)` $\equiv$ *e*.

Define an Elm function `simplify` that uses these equivalences to simplify regular expressions.

```
simplify : RegEx -> RegEx
```