Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Assignment P2

Andres Löh, João Pizani, David van Balen

December 5, 2023

The goal of this assignment is to implement a little domain-specific programming language. Programs comprise instructions for a little spaceship called *Arrow* that flies around in a bounded two-dimensional space. The space is not empty, but inhabited with various flying objects such as asteroids, lambdas and debris. By interpreting programs, we can let Arrow fly through space and perform certain tasks such as finding a way through an asteroid field and cleaning up debris.

Credits

This assignment is inspired by the *Kara* programming system, and in particular by Frank Huch's paper "Learning Programming with Erlang" that appeared in the proceedings of the 2007 ACP SIGPLAN workshop on Erlang.

Alex and Happy

For this task, you will use the Alex lexer generator and the Happy parser generator. Their documentation is available at

`http://haskell.org/alex/`

`http://haskell.org/happy/`

Running *cabal build* on the starting framework should not throw errors, but *cabal run* should throw an error from the lexer. This means that you have everything set up correctly. When implementing exercises 1 and 3, delete the existing code in Lexer.x and Parser.y: they are just there to get the template to compile.

General remarks

- For some reason, this project does not compile when you put it in a OneDrive folder. Just don't. Also, having spaces or ampersands in the folder names can cause errors.

- The starting framework includes a readme, describing the file structure and how to work with Alex and Happy.
- Make sure your program compiles.
- Include *useful* comments in your code. Do not paraphrase the code, but describe the structure of your program, special cases, preconditions, choices you made, etc.
- Try to write readable and idiomatic Haskell. Style influences the grade! The use of existing higher-order functions such as *map*, *foldr*, *filter*, *zip* – just to name a few – is explicitly encouraged. The use of non-parsing related libraries is allowed (as long as the program still compiles with the above invocation).
- Copying solutions from the internet is not allowed.
- You may work alone or with one other person. A team must submit a single assignment and put both names on it. You do not need to work in the same team as in the previous assignments.

The Arrow programming language

The concrete syntax of the Arrow language is given by the following grammar with start symbol *Program*:

$$
\begin{array}{lll}
\mathit{Program} & \to & \mathit{Rule}^* \\
\mathit{Rule} & \to & \mathit{Ident}\ \texttt{->}\ \mathit{Cmds}\ \texttt{.} \\
\mathit{Cmds} & \to & \varepsilon \mid \mathit{Cmd}\ (\texttt{,}\ \mathit{Cmd})^* \\
\mathit{Cmd} & \to & \texttt{go} \mid \texttt{take} \mid \texttt{mark} \mid \texttt{nothing} \\
 & & \mid\ \texttt{turn}\ \mathit{Dir} \\
 & & \mid\ \texttt{case}\ \mathit{Dir}\ \texttt{of}\ \mathit{Alts}\ \texttt{end} \\
 & & \mid\ \mathit{Ident} \\
\mathit{Dir} & \to & \texttt{left} \mid \texttt{right} \mid \texttt{front} \\
\mathit{Alts} & \to & \varepsilon \mid \mathit{Alt}\ (\texttt{;}\ \mathit{Alt})^* \\
\mathit{Alt} & \to & \mathit{Pat}\ \texttt{->}\ \mathit{Cmds} \\
\mathit{Pat} & \to & \texttt{Empty} \mid \texttt{Lambda} \mid \texttt{Debris} \mid \texttt{Asteroid} \mid \texttt{Boundary} \mid \texttt{\_}
\end{array}
$$

A program is a sequence of rules. Think of rules as procedures. A name is bound to a sequence of commands. Rules are terminated by a period.

Commands are separated by commas. There is a fixed number of commands. These are instructions for *Arrow*. Informally, `go` means "move in the current direction if possible", `take` means "pick up whatever is here", `mark` means "leave a lambda in the current spot", `nothing` means "do nothing", `turn` takes a direction and causes Arrow to turn left or right. The `case` command takes a direction and performs a sensor reading in that direction. Depending on what is sensed, different actions may be taken. Finally, another rule can be invoked by naming it.

In a `case` construct, multiple alternatives can be provided (separated by semicolons) that map patterns to rules. Patterns correspond to the things that can be located in a certain position, and there is a catch-all pattern called `_`.

Note that unlike in Haskell, `case` expressions are terminated by an `end` keyword.

The lexical syntax of a program is described as follows: the program text consists of a (possibly space-separated) sequence of tokens.

$$
\begin{aligned}
\mathit{Token} \rightarrow\ & \texttt{->} \mid \texttt{.} \mid \texttt{,} \mid \texttt{go} \mid \texttt{take} \mid \texttt{mark} \mid \texttt{nothing} \mid \texttt{turn} \mid \texttt{case} \mid \texttt{of} \mid \texttt{end} \\
\mid\ & \texttt{left} \mid \texttt{right} \mid \texttt{front} \mid \texttt{;} \\
\mid\ & \texttt{Empty} \mid \texttt{Lambda} \mid \texttt{Debris} \mid \texttt{Asteroid} \mid \texttt{Boundary} \mid \texttt{\_} \\
\mid\ & \mathit{Ident} \\
\mathit{Ident} \rightarrow\ & (\mathit{Letter} \mid \mathit{Digit} \mid \texttt{+} \mid \texttt{-})^{+}
\end{aligned}
$$

A token is either symbolic, a command keyword, a pattern keyword, or an identifier. It is implicitly understood that an *Ident* must not be any of the keyword tokens and must not be directly followed by another character that could occur in an identifier.

Furthermore, comments may occur in programs between tokens. These are introduced by `--` and extend to the end of the line, like in Haskell.

1 (1 pt)**.** Write a lexer/scanner for the language using Alex. Define a datatype to represent tokens and let the scanner return such tokens. Study the online documentation of Alex to find out what the syntax of Alex specification files is. Start in Chapter 2, the introduction should already contain most of what you need. The use of the `basic` wrapper is sufficient for this task. If you want to use one of the more advanced wrappers, that is also fine, but not required. In particular, terminating with an exception on a lexing error is allowed in this assignment. Remember to test your implementation!

Write the datatype in `Model.hs`, and the lexer in `Lexer.x`.

2 (1 pt)**.** Define a suitable abstract syntax for the Arrow language in `Model.hs`. Call the type corresponding to a whole program *Program*.

3 (1 pt)**.** Write a parser for the language using Happy, in `Parser.y`. Again, study the online documentation to find out about the syntax of Happy specification files. Again, Chapter 2 is a good start and should contain most of the required information. Use the datatype of tokens delivered by the lexer as input, and produce values of the abstract syntax as results of the parser. It is not required to use any of the advanced error-handling functionality of Happy, nested lexing, or monadic parsing functionality. Again, failing with an exception on a parse error is allowed, and remember to test your implementation!

4 (0.5 pt)**.** What can you find out from the Happy documentation about Happy's handling of left-recursive and right-recursive grammars. How does this compare to the situation when using parser combinators? Include your answer in `open-questions.md`.

5 (1 pt)**.** Define an algebra type and a fold function for your abstract syntax type, in `Algebra.hs`.

6 (1 pt)**.** Define one or more algebras that describe an analysis of the program that (besides possibly required additional information) performs the following sanity checks on a given program:

- There are no calls to undefined rules (rules may be used before they are defined though).
- There is a rule named `start`.
- No rule is defined twice.
- There is no possibility for pattern match failure, i.e., all `case` expressions must either contain a catch-all pattern `_` or contain cases for all five other options.

Use the algebras and fold functions to define a function

$$\mathit{check} :: \mathit{Program} \to \mathit{Bool}$$

that combines all of the above checks and returns true iff a program is sane.

An interpreter for Arrow programs

Arrow lives on a rectangular board that we call "space" and represent using a finite map (dictionary), from the module *Data.Map*:

$$\begin{array}{ll}
\textbf{type } \mathit{Space} & = \mathit{Map\ Pos\ Contents} \\
\textbf{type } \mathit{Size} & = \mathit{Int} \\
\textbf{type } \mathit{Pos} & = (\mathit{Int}, \mathit{Int}) \\
\textbf{data } \mathit{Contents} & = \mathit{Empty} \mid \mathit{Lambda} \mid \mathit{Debris} \mid \mathit{Asteroid} \mid \mathit{Boundary}
\end{array}$$

We assume that there always is a rectangular area of positions with non-negative row- and column-coordinates contained in the finite map, including position $(0, 0)$. We leave the size of the space open though, and functions can use *findMax* to find the maximum key and hence the maximum position in a given space.

We define an input format for spaces where contents are represented by characters:

contents	character
empty	`.`
lambda	`\`
debris	`%`
asteroid	`O`
boundary	`#`

We specify the format by example:

```
(7,7)
........
....%...
..%%%%..
....%%%.
...%%%..
....%.%%
....%%%%
........
```

The first line contains the maximum valid row-column-coordinate for the board. Here, we thus have a space with 8 rows and 8 columns. The rows are then specified line by line, starting with row 0 end ending with row 7. The example space contains a field of debris, but otherwise just empty space.

A parser for the input format is included in the starting framework:

```
parseSpace :: Parser Char Space
parseSpace =
  do
    (mr, mc) ← parenthesised
                 ((,) <$> natural <* symbol ',' <*> natural) <* spaces
    — read mr + 1 rows of mc + 1 characters
    css      ← replicateM (mr + 1) (replicateM (mc + 1) contents)
    — convert from a list of lists to a finite map representation
    return $ M.fromList $ concat $
             zipWith (λr cs →
             zipWith (λc d  → ((r, c), d)) [0 ..] cs) [0 ..] css
```

We also have the parser *contents* that parses a single character and maps it to the appropriate constructor of type *Contents*:

```
contents :: Parser Char Contents
contents =
  choice (Prelude.map (λ(f, c) → f <$ symbol c) contentsTable) <* spaces
contentsTable :: [(Contents, Char)]
contentsTable =
  [(Empty, '.'), (Lambda, '\\'), (Debris, '%'), (Asteroid, 'O'), (Boundary, '#')]
```

7 (0.5 pt)**.** Write a printer for *Space* that produces the output format just shown, in `Interpreter.hs`.

8 (0.5 pt)**.** Assuming that *Ident* is the Haskell type representing an identifier, and *Commands* represents a sequence of commands, we represent a program as an environment during execution:

type *Environment* = *Map Ident Commands*

Write a function

toEnvironment :: *String* → *Environment*

in `Interpreter.hs` that first lexes and then parses a string, checks the resulting *Program* using *check*, and, if the check succeeds, translates the *Program* into an environment.

9 (1.5 pt)**.** During the execution of a program, we have to maintain state. The state contains the current space, the position of Arrow, its heading, and a stack of commands.

type *Stack* = *Commands*
data *ArrowState* = *ArrowState Space Pos Heading Stack*

Implement a function in `Interpreter.hs` that performs a single execution step:

step :: *Environment* → *ArrowState* → *Step*

where *Step* encodes the possible results of one execution step:

data *Step* = *Done Space Pos Heading*
| *Ok ArrowState*
| *Fail String*

The function implements the following semantics. The top item on the command stack is analyzed:

- On `go`, Arrow moves forward one step using its current heading, as long as the target field is empty or contains a lambda or debris. Otherwise, it stays where it is.
- On `take`, Arrow picks up lambda or debris, leaving an empty space at its current position.
- On `mark`, Arrow places a lambda at its current position regardless of what was there before (debris is removed).
- On `nothing`, nothing changes.
- On `turn`, Arrow changes its heading by 90 degrees to the left or right as indicated. Turning `forward` is possible, but has no effect.
- On a `case`, Arrow makes a sensor reading. Depending on the direction specified as an argument to `case`, Arrow will take a look at the position that – according to its current heading – is to the front, left, or right. The pattern of each alternative is then analyzed in turn until one matching alternative is found. The instructions on the right hand side are then prepended to the command stack and execution continues. If no alternative matches, execution fails. An alternative matches if the pattern corresponds to the contents. Positions that are not stored in the finite map are implicitly assumed to contain `Boundary`. A catch-all pattern matches always.

- On a rule call, the code stored with that rule in the environment is prepended to the command stack. If the rule is not defined, execution fails.
- If the command stack is empty, a *Done* result is produced.

10 (0.5 pt)**.** Rules can be recursive. Note how recursion affects the size of the command stack during execution. Does it matter whether the recursive call is in the middle of a command sequence or at the very end of the command sequence? Include your observations in `open-questions.md`.

11 (0.5 pt)**.** Write two drivers

$$\begin{aligned} &\mathit{interactive} :: \mathit{Environment} \to \mathit{ArrowState} \to \mathit{IO}\ () \\ &\mathit{batch} :: \mathit{Environment} \to \mathit{ArrowState} \to (\mathit{Space}, \mathit{Pos}, \mathit{Heading}) \end{aligned}$$

in `Main.hs` that – given an environment and an initial state – run the program.

The interactive driver should print in every step at least the board and ask for some form of user confirmation. After getting the user input, the driver should invoke the next step and continue from the beginning. This driver should recognize abnormal and successful terminations of the reduction and treat them sensibly. The batch driver should run the program and return the final state, in which there are no more steps to take.

Write proper main programs that lets you read in a space and a program from a file, specify a start position and heading, and runs the program in either mode.

Example: Remove debris

The following example program removes all debris in a connected component of the space. So, for instance, running this program on the example space shown above with the ship starting in any position filled with debris should ultimately clear all the debris in the space, then stop.

```
start -> take,
         case front of
           Debris -> go, start, turn right, turn right,
                     go, turn right, turn right;
           _      -> nothing
         end,
         turn right,
         s2.

s2    -> take,
         case front of
           Debris -> go, start, turn right, turn right,
                     go, turn right, turn right;
           _      -> nothing
         end,
         turn right,
         s3.


s3    -> take,
         case front of
           Debris -> go, start, turn right, turn right,
                     go, turn right, turn right;
           _      -> nothing
         end,
         turn right,
         s4.

s4    -> take,
         case front of
           Debris -> go, start, turn right, turn right,
                     go, turn right, turn right;
           _      -> nothing
         end,
         turn right.
```

Example: Adding natural numbers

Here is another example program that adds two natural numbers:

```
start       -> turn right, go, turn left, firstArg.

turnAround  -> turn right, turn right.

return      -> case front of
                 Boundary  ->  nothing;
                 _         ->  go, return
               end.

firstArg    -> case left of
                 Lambda  ->  go, firstArg, mark, go;
                 _       ->  turnAround, return, turn left,
                             go, go, turn left,
                             secondArg
               end.

secondArg   -> case left of
                 Lambda  ->  go, secondArg, mark, go;
                 _       ->  turnAround, return, turn left,
                             go, turn left
               end.
```

The program expects its input as rows of lambdas, as in the following example:

```
(4,14)
\\\\\..........
..............
\\\\\\\........
..............
..............
```

The first number here is 5, the second 7.

If you start arrow facing east in the upper left corner of the space (i.e., at position $(0, 0)$), then the result of adding the two numbers is written below the two inputs:

```
(4,14)
\\\\\..........
..............
\\\\\\\........
..............
\\\\\\\\\\\...
```