Assessed Coursework

Course Name	Programming Languages (H)			
Coursework Number	1			
Deadline	Time:	16:30	Date:	14th March 2025
% Contribution to final course mark	20			
Solo or Group ✓	Solo	X	Group	
Anticipated Hours	20			
Submission Instructions	Submit a zip file containing only your Interp.java and Typecheck.java to Moodle. See the handout for details.			
Please Note: This Coursework cannot be Re-Assessed				

You must complete an "Own Work" form via https://studentltc.dcs.gla.ac.uk/ for all coursework

Programming Languages (H) Assessed Exercise
An Interpreter and Typechecker for Diggleby

v2 (February 28, 2025)

Introduction

This assignment is due Friday 14th March at 4:30PM.

Please read over this handout carefully and look over the supplied code before beginning work, as some of your questions may be answered later. Please let us know if there are any apparent errors or bugs. We will update this handout to add clarifications and fix any issues; such updates will be announced on Teams. The handout is versioned and the most recent version will always be available from the Moodle page.

To aid our marking, please do not modify any of the function signatures in the skeleton code. You may of course add your own helper functions.

Submission When you are ready to submit, please create a ZIP file containing only your `Interp.java` and `Typecheck.java` files, and a `status.txt` file detailing the exercises you have attempted. The `status.txt` does not need to include any more information and will not be marked.

Please submit a single `zip` file to the Moodle submission box, where the name of the file corresponds to your student ID (for example, `2600000F.zip`). Your submission will be assessed according to the usual University policies on lateness and plagiarism. Remember that this is *individual* work and you must not share code with others.

There are a total of 65 marks available. You can attempt the two parts of the assignment in either order; there are no dependencies between the two.

Important: *We expect your code to compile. Submissions that do not compile because of syntax or type errors will be capped at 20 marks.*

Working on the Lab Machines You are welcome to work on either your own machine or the lab machines.

The Diggleby programming language

In this assignment, you will implement an interpreter and typechecker for a small expression-based language called Diggleby from a formal specification. The general implementation strategy follows on from the work set in the labs (so I'd recommend looking at the labs if you haven't yet).

Diggleby is like many of the small languages we have seen in the lectures and labs, including features like variables, binary operators, let-expressions, conditionals, and function calls. Additionally, it has support for strings and string concatenation, a Boolean negation operator, console IO, and type casts.

Unlike the languages we have seen in the lectures and labs, Diggleby programs consist of a sequence of *function definitions*, which are accessible to both the interpreter and typechecker, rather than having first-class (lambda) functions. Function definitions are mutually-recursive: a function can call any other function. Definitions can also have *multiple* parameters.

The following Diggleby program calculates the factorial of 10:

```
def fac(x: Int): Int {
  if (x == 0) {
    1
  } else {
    let minusone = x - 1 in
    x * fac(minusone)
  }
}

fac(10)
```

Here we introduce a definition `fac` which takes a parameter `x` of type `Int`, where the definition has return type `Int`. The body checks whether `x` is equal to 0, returning 1 if so; otherwise it multiplies `x` by the factorial of `x - 1` using a recursive call.

The body of the program `fac(10)` calls the `fac` function with the argument `10`.

Since Diggleby also supports console IO and type casts, we can generalise our program to retrieve the argument from the user via the console:

```
def fac(x: Int): Int {
  if (x == 0) {
    1
  } else {
    let minusone = x - 1 in
    x * fac(minusone)
  }
}

def getBound(): Int {
  (getInput() : Int)
}

let result = fac(getBound()) in
print((result : String))
```

Here we keep the `fac` function as before, but also implement a `getBound` function that gets some input from the user via the built-in `getInput` function, and casts it to an integer. Finally, the program casts the result back to a string and prints it to the console using the `print` command.

We can also write programs with mutually-recursive functions. The following program (inefficiently!) tests whether the number `11` is even:

```
def isEven(x: Int): Bool {
  if (x == 0) { true } else { isOdd(x - 1) }
}

def isOdd(x: Int): Bool {
  if (x == 0) { false } else { isEven(x - 1) }
}

isEven(11)
```

Syntax of types and expressions

$$
\begin{array}{lrcl}
\text{Types} & A, B & ::= & \text{Int} \mid \text{Bool} \mid \text{String} \mid \text{Unit} \\
\text{Variables} & x, y, \ldots & & \\
\text{Function names} & f, g, \ldots & & \\
\text{Integers} & n & & \\
\text{Strings} & s & & \\
\text{Booleans} & b & ::= & \texttt{true} \mid \texttt{false} \\
\text{Constants} & c & ::= & n \mid b \mid s \mid \texttt{()} \\
\text{Operators} & \odot & ::= & + \mid - \mid * \mid / \mid \texttt{\%} \\
 & & \mid & > \mid < \mid \texttt{==} \mid \texttt{\&\&} \mid \texttt{||} \\
\text{Expressions} & L, M, N & ::= & x \mid c \mid M \odot N \\
 & & \mid & \texttt{!}M \\
 & & \mid & \texttt{let}\ x = M\ \texttt{in}\ N \\
 & & \mid & \texttt{if}\ (L)\ \{\ M\ \}\ \texttt{else}\ \{\ N\ \} \\
 & & \mid & f(\overrightarrow{M}) \\
 & & \mid & \texttt{concat}(\overrightarrow{M}) \\
 & & \mid & \texttt{print}(M) \mid \texttt{getInput}() \\
 & & \mid & (M\ :\ A)
\end{array}
$$

Syntax of definitions and programs

$$
\begin{array}{lrcl}
\text{Function Definition} & F & ::= & \texttt{def}\ f(\overrightarrow{x : A}) \colon B\ \{M\} \\
\text{Programs} & D & ::= & \overrightarrow{F}\ M
\end{array}
$$

Figure 1: Abstract syntax of Diggleby

Notational Preliminaries

The assignment will involve implementing Diggleby from a formal specification. We make use of the following notation. We write $\overrightarrow{X}$ for a (potentially-empty) ordered sequence of some syntactic object X (in practice X will be specialised to e.g., $\overrightarrow{M}$ to denote a sequence of expressions).

When it is useful to talk about individual members of of a sequence or a set, we use indexing notation: for example given the sequence $\overrightarrow{M} = \texttt{true}, \texttt{false}, 5, 10$, by writing $(M_i)_{i \in I}$ we have that $M_1 = \texttt{true}, M_2 = \texttt{false}, M_3 = 5, M_4 = 10$. Where it is useful to refer to the size of the sequence, we write $(M_i)_{i \in 1..n}$ (and with the above example, $n = 4$).

Diggleby Abstract Syntax

Figure 1 shows the abstract syntax for Diggleby; we have chosen the abstract syntax to closely resemble the concrete syntax. Types, ranged over by A, B, include the integer type Int, Boolean type Bool, string type String, and also the unit type Unit. There is a single value `()` that has the unit type Unit; it is used as a result type for side-effecting operations that do not have a result type (for example, **`print`**).

We let x, y, z range over variable names, and f, g range over function names. Constants include integers n, strings s, Booleans b, and the unit value `()`.

Binary operators include the usual arithmetic and Boolean operators. The `%` operator refers to the `mod` operation (e.g., `5 % 2 = 1`). The == operator returns **`true`** if both operands evaluate to the same value.

Expressions, ranged over by L, M, N include variables x, constants c, binary operations $M \odot N$, and unary negation $\texttt{!}M$. Additionally, expressions include let-binders $\texttt{let}\ x = M\ \texttt{in}\ N$; conditionals $\texttt{if}\ (L)\ \{\ M\ \}\ \texttt{else}\ \{\ N\ \}$, function calls $f(\overrightarrow{M})$; (indicating a call to function f with arguments $\overrightarrow{M}$); string concatenation $\texttt{concat}(\overrightarrow{M})$; console output $\texttt{print}(M)$; console input $\texttt{getInput}()$; and type casts $(M\ :\ A)$.

A function definition F has the form $\texttt{def}\ f(\overrightarrow{x : A}) \colon B\ \{M\}$, where f is the name of the function; $\overrightarrow{x : A}$ is a (potentially empty) sequence of type-annotated parameters; B is the return type of the function; and M

is the function body.

A program P consists of a sequence of function definitions followed by a “main” expression to be evaluated when the program is run.

Skeleton Code

The skeleton code is organised as follows. You only need to edit files marked with a star $(\star)$.

- `src/antlr/Diggleby.g4`: The Diggleby ANTLR grammar
- `src/ast`: The Java classes representing the abstract syntax tree. Of these:
  - All files beginning with `E` represent expressions. For example, `ELet.java` represents a let-expression. `Expr.java` is the abstract superclass of all expressions.
  - All files beginning with `V` represent values, which are produced as a result of evaluating an expression. For example, `VInt.java` represents an integer value. `Value.java` is the abstract superclass of all expressions.
  - All files beginning with `Ty` represent types. For example, `TyInt` represents the Int type. Obtain an instance of a type using the `.type()` static method, for example `TyInt.type()`.
    `Type.java` is the abstract superclass of all types.
  - `AnnotatedParam.java` is a helper class, representing a parameter paired with a type. It is used in representing function definitions.
  - `FnDef.java` represents a function definition.
  - `Program.java` represents a program, consisting of a list of function definitions and a main expression.
- `src/CastException.java` and `src/TypeErrorException.java` are exceptions that should be thrown when a cast fails, and when typechecking fails, respectively.
- `DigglebyASTGeneratorVisitor.java` is the visitor that constructs a Diggleby AST from an ANTLR parse tree.
- `Environment.java` represents an immutable map. Its key is a string (representing a variable), but it is generic in its value. We will use this class to represent environments for both types (when typechecking) and values (when evaluating). It has two methods: `lookup`, which looks up a variable; and `extend`, which returns a copy of the current environment, extended with the new mapping.
- `Interp.java` $(\star)$ is the interpreter, which evaluates programs and expressions down to values.
- `Main.java` is the main entry point of the program. It parses command line arguments, loads in and parses a Diggleby source file, and runs the typechecker and interpreter.
- `Typecheck.java` $(\star)$ is the typechecker, which typechecks programs and expressions.

We have also included a few example Diggleby programs in the `examples` directory. These are not exhaustive; we recommend writing some of your own when you are testing.

- `concatIO.dig`: inputs a number n from the user, then inputs n strings, returning the concatenation of all of them.
- `fac.dig`: computes the factorial of 10.
- `facIO.dig`: inputs a number n from the user, then outputs the factorial of n.
- `isEven.dig`: mutually-recursive implementation of evenness checking for positive integers.
- `memoFib.dig`: efficient implementation of calculating the nth Fibonacci number.
- `naiveFib.dig`: inefficient implementation of calculating the nth Fibonacci number.

Running the Skeleton Code

To reduce reliance on IDEs, we have included `.bat` scripts (for Windows) and `.sh` scripts (for Linux / Mac).

To compile and run the skeleton code, use the following workflow from a terminal (assuming Windows; replace with the `.sh` versions on Mac and Linux):

- Run `antlr.bat` to run ANTLR and generate the lexer, parser, and base visitor files.
- Run `compile.bat` to compile the project.
- Run `run.bat <filename>` to run the typechecker and interpreter on the Diggleby file `<filename>`. It is possible to pass the flags `-interp` and `-typecheck` to run only the interpreter or typechecker respectively.

  For example, run `.\run.bat -typecheck examples\fac.dig`
  (or `./run.sh -typecheck examples/fac.dig` on Mac/Linux) to run the typechecker on the `fac.dig` file.

Part 1: Interpreter

This part of the assignment involves editing `src/Interp.java`.

Environment-based Interpreters

In the lectures and labs, we have seen *substitution-based* interpreters, where whenever we wish to bind variable x to value V in expression M, we substitute all free occurrences of x for V.

While this is close to the theory, it is inefficient in terms of both space and time: every substitution requires a linear traversal of the syntax tree, and we need to duplicate a value each for every occurrence of a variable.

Instead, we will implement the interpreter for Diggleby using an *environment-based* interpreter. The idea is that instead of substituting whenever we bind a variable x to a value V, we instead record a mapping from x to V and then look up x in the environment when x is encountered during evaluation.

As an example, consider expression $\texttt{let}\ x = 5\ \texttt{in}\ x + 10$. Here is the derivation tree for a substitution-based interpreter (using the semantics that we have discussed in the lectures and labs):

$$\dfrac{\dfrac{}{5 \Downarrow 5} \qquad \dfrac{\dfrac{}{5 \Downarrow 5} \qquad \dfrac{}{10 \Downarrow 10}}{5 + 10 \Downarrow 15}}{\texttt{let}\ x = 5\ \texttt{in}\ x + 10 \Downarrow 15}$$

In contrast, consider the following derivation tree in an environment-based interpreter. Here, the judgement $\rho,\ M \Downarrow V$ requires an (initially-empty) environment ρ that maps variables to values. When evaluating the continuation of the **`let`** expression, we record that x refers to 5, and then look this up when evaluating x:

$$\dfrac{\dfrac{}{\cdot,\ 5 \Downarrow 5} \qquad \dfrac{\dfrac{}{x \mapsto 5,\ x \Downarrow 5} \qquad \dfrac{}{x \mapsto 5,\ 10 \Downarrow 10}}{x \mapsto 5,\ x + 10 \Downarrow 15}}{\cdot,\ \texttt{let}\ x = 5\ \texttt{in}\ x + 10 \Downarrow 15}$$

Since Diggleby programs also need to refer to function definitions when evaluating function calls, we use the evaluation judgement $\rho,\ M \Downarrow_{\mathcal{D}} V$ which can be read "given a mapping $\mathcal{D}$ from function names to definitions, and a value environment ρ mapping variables to values, expression M evaluates to value V".

Programs

The first part of the assignment involves implementing the `public Value interpProgram(Program prog)` function that runs a program. In essence, this function generates a mapping $\mathcal{D}$ from function names f to their definitions, and then evaluates the main function body under an empty value environment. Specifically, let us write $\texttt{name}(F)$ to project the name from a function definition:

$$\texttt{name}(\textbf{def}\ f(\overrightarrow{x : A})\colon B\ \{M\}) = f$$

Program evaluation can then be written:

$$\frac{\mathcal{D} = \{\texttt{name}(F) \mapsto F \mid F \in \overrightarrow{F}\} \qquad \cdot,\ M \Downarrow_{\mathcal{D}} V}{\overrightarrow{F}\ M \Downarrow V}$$

To implement this function, you will need to construct a Java map of type `Map<String, FnDef>`, where each `String` refers to a function name and `FnDef` refers to a function definition, and call the `interpExpr` function with this mapping and an empty value environment.

Exercise 1 (`interpProgram`). *Implement the `interpProgram` method following the rule above.*
[5 marks]

Expressions

Next, we will implement the main expression evaluator. This will involve implementing the
`public Value interpExpr(Map<String, FnDef> fnDefs, Environment<Value> valEnv, Expr e) { .. }`
method by implementing cases for each type of expression.

Recall the evaluation judgement $\rho,\ M \Downarrow_{\mathcal{D}} V$: in the function signature, $\mathcal{D}$ is represented by `fnDefs`; environment ρ is represented by `valEnv`, and expression M is represented by `e`.

The formal evaluation rules are as follows:

Evaluation rules for expressions $\boxed{\rho,\, M \Downarrow_{\mathcal{D}} V}$

$$\frac{x \mapsto V \in \rho}{\rho,\, x \Downarrow_{\mathcal{D}} V} \qquad \frac{}{\rho,\, c \Downarrow_{\mathcal{D}} c} \qquad \frac{\rho,\, M \Downarrow_{\mathcal{D}} V \qquad \rho,\, N \Downarrow_{\mathcal{D}} W}{\rho,\, M \odot N \Downarrow_{\mathcal{D}} V \mathbin{\hat{\odot}} W} \qquad \frac{\rho,\, M \Downarrow_{\mathcal{D}} V}{\rho,\, \texttt{!}M \Downarrow_{\mathcal{D}} \neg V}$$

$$\frac{\rho,\, M \Downarrow_{\mathcal{D}} V \qquad \rho[x \mapsto V],\, N \Downarrow_{\mathcal{D}} W}{\rho,\, \textbf{let}\, x = M \,\textbf{in}\, N \Downarrow_{\mathcal{D}} W}$$

$$\frac{\rho,\, L \Downarrow_{\mathcal{D}} \textbf{true} \qquad \rho,\, M \Downarrow_{\mathcal{D}} V}{\rho,\, \textbf{if}\,(L)\,\{\,M\,\}\,\textbf{else}\,\{\,N\,\} \Downarrow_{\mathcal{D}} V} \qquad \frac{\rho,\, L \Downarrow_{\mathcal{D}} \textbf{false} \qquad \rho,\, N \Downarrow_{\mathcal{D}} V}{\rho,\, \textbf{if}\,(L)\,\{\,M\,\}\,\textbf{else}\,\{\,N\,\} \Downarrow_{\mathcal{D}} V}$$

$$\frac{\mathcal{D}(f) = \textbf{def}\, f((x_i : A_i)_{i \in I})\colon B\,\{N\} \qquad (\rho,\, M_i \Downarrow_{\mathcal{D}} V_i)_{i \in I} \qquad (x_i \mapsto V_i)_{i \in I},\, N \Downarrow_{\mathcal{D}} W}{\rho,\, f((M_i)_{i \in I}) \Downarrow_{\mathcal{D}} W}$$

$$\frac{(\rho,\, M_i \Downarrow_{\mathcal{D}} s_i)_{i \in 1..n}}{\rho,\, \textbf{concat}((M_i)_{i \in 1..n}) \Downarrow_{\mathcal{D}} s_1 + \ldots + s_n} \qquad \frac{\rho,\, M \Downarrow_{\mathcal{D}} s \qquad \text{(print } s \text{ to console)}}{\rho,\, \textbf{print}(M) \Downarrow_{\mathcal{D}} \texttt{()}}$$

$$\frac{\text{(input string } s \text{ from console)}}{\rho,\, \textbf{getInput}() \Downarrow_{\mathcal{D}} s} \qquad \frac{\rho,\, M \Downarrow_{\mathcal{D}} V}{\rho,\, (M\ :\ A) \Downarrow_{\mathcal{D}} \texttt{cast}(V, A)}$$

Let us look at the rules in turn.

Variables and operators. The variable rule looks up x in the environment ρ, returning the stored value. As usual, constants c evaluate to themselves (for example, an integer literal `EInt` would reduce to a corresponding value `VInt`). Binary operators $M \odot N$ work by evaluating arguments M and N to values V and W, and then performing the underlying operation (e.g., $\rho,\ (5 + 10) * 10 \Downarrow_{\mathcal{D}} 150$). Unary negation $\texttt{!}M$ evaluates M down to a Boolean value and returns its logical negation.

Let-binders and conditionals. To evaluate $\textbf{let}\, x = M \,\textbf{in}\, N$, we firstly need to evaluate M to a value V. Next, we need to extend our current value environment with a mapping $x \mapsto V$, and then return the result of evaluating N under the new environment.

To evaluate $\textbf{if}\,(L)\,\{\,M\,\}\,\textbf{else}\,\{\,N\,\}$ we firstly need to evaluate L: if the result is **true** then we evaluate M, and otherwise we evaluate N.

Function calls. Evaluating a function call is a bit more involved. To evaluate a function call $f((M_i)_{i \in 1..n})$ we need to:

- Lookup function name f in $\mathcal{D}$ to get the function definition $\textbf{def}\, f((x_i : A_i)_{i \in 1..n})\colon B\,\{N\}$.
- Evaluate each argument M_i down to a value V_i.
- Create a new value environment that maps each parameter x_i to each evaluated argument V_i.
- Evaluate the function body using the new environment, returning the result.

For example, suppose we had a definition mapping $\mathcal{D}$ containing function definition *add3* that adds three numbers:

$$\textbf{def}\, \mathit{add3}(x : \mathsf{Int}, y : \mathsf{Int}, z : \mathsf{Int})\colon \mathsf{Int}\,\{x + y + z\}$$

Suppose then we wanted to evaluate the function call $\mathit{add3}(5{*}10, 3{+}4, 5)$. We would need to evaluate the arguments down to values: in this case, $50, 7, 5$. Then, we would need to create the value environment $\rho = x \mapsto 50, y \mapsto 7, z \mapsto 5$ and finally compute $\rho,\ x + y + z \Downarrow_{\mathcal{D}} 62$.

String concatenation. To evaluate $\texttt{concat}((M_i)_{i \in 1..n})$, we need to evaluate each term M_i to a string s_i, and then return the string resulting from concatenating all of the strings together.

Console IO. To evaluate $\texttt{print}(M)$, we firstly evaluate M down to a string s, and then print it to the console using `System.out.println()`. We then need to return a `VUnit` value.

To evaluate $\texttt{getInput}()$ you will need to input a string from the console (see here for some methods of doing this), and then return the retrieved string.

Casts. Finally, to implement a cast $(M : A)$, evaluate M down to a value V and then call the `cast` function (which you will complete in the next exercise).

Exercise 2 (`interpExpr`)**.** *Implement the `interpExpr` method following the rules above.*

[25 marks]

Casts

Casting allows us to convert a value from one type to another. The partial function $\texttt{cast}(V, A)$, used in the evaluation rule for the casting construct, is defined as follows:

$$
\begin{array}{lcl}
\texttt{cast}(n, \text{String}) & = & (\text{string representation of } n) \\
\texttt{cast}(\textbf{true}, \text{String}) & = & \texttt{"true"} \\
\texttt{cast}(\textbf{false}, \text{String}) & = & \texttt{"false"} \\
\texttt{cast}(s, \text{String}) & = & s \\
\texttt{cast}(\texttt{()}, \text{String}) & = & \texttt{"()"} \\
\\
\texttt{cast}(s, \text{Int}) & = & (\text{conversion of } s \text{ to an integer, if } s \text{ contains only numbers}) \\
\texttt{cast}(\textbf{true}, \text{Int}) & = & 1 \\
\texttt{cast}(\textbf{false}, \text{Int}) & = & 0 \\
\texttt{cast}(n, \text{Int}) & = & n \\
\\
\texttt{cast}(\texttt{()}, \text{Unit}) & = & \texttt{()} \\
\\
\texttt{cast}(b, \text{Bool}) & = & b \\
\texttt{cast}(\texttt{"true"}, \text{Bool}) & = & \textbf{true} \\
\texttt{cast}(\texttt{"false"}, \text{Bool}) & = & \textbf{false}
\end{array}
$$

Any other casts are invalid, and your interpreter should raise a `CastException`.

Exercise 3 (Cast)**.** *Implement the `cast` method, following the rules above.*

[5 marks]

Part 2: Typechecker

Programs and Function Definitions

Since programs consist of a sequence of function definitions and a main expression, we need to write typing rules for programs and function definitions.

Typing rule for programs $\boxed{\vdash \overrightarrow{F}\ M}$

$$\frac{\mathcal{D} = \{\mathsf{name}(F) \mapsto F \mid F \in (F_i)_{i\in I}\} \\ (\vdash_{\mathcal{D}} F_i)_{i \in I} \qquad \Gamma \vdash_{\mathcal{D}} M : A}{\vdash (F_i)_{i\in I}\ M}$$

Typing rule for function definitions $\boxed{\vdash_{\mathcal{D}} F}$

$$\frac{(x_i : A_i)_{i\in I} \vdash_{\mathcal{D}} M : B}{\vdash_{\mathcal{D}} \texttt{def}\ f((x_i : A_i)_{i\in I})\colon B\ \{M\}}$$

To typecheck a program $(F_i)_{i\in I}\ M$ (described by the first rule), we need to:

- Construct a mapping from function names to function definitions, in the same way as we did for the interpreter.
- Typecheck all function definitions.
- Typecheck the main expression.

To typecheck a function definition (described by the second rule), we need to typecheck its body under an environment extended with all of its parameter types, and check that the return type matches the return type annotation.

Exercise 4 (`typecheckProgram` and `typecheckDef`). *Implement the* `typecheckProgram` *and* `typecheckDef` *methods, following the rules above. You will need to call the* `typecheckExpr` *method.*
[5 marks]

Expressions

Finally, we will implement the expression typing rules. The typing judgement is of the form $\Gamma \vdash_{\mathcal{D}} M : A$, which can be read "Under typing environment Γ and given function definitions $\mathcal{D}$, term M has type A".

The corresponding function that you will implement, again by writing cases for each expression, is `public Type typecheckExpr(Map<String, FnDef> fnDefs, Environment<Type> tyEnv, Expr e) {..}`.

In our judgement, $\mathcal{D}$ is represented by `fnDefs`; type environment Γ is represented by `tyEnv`; and expression M is represented by `e`. The return type of the method corresponds to A in the judgement.

Typing rules for expressions $\boxed{\Gamma \vdash_{\mathcal{D}} M : A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash_{\mathcal{D}} x : A} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} n : \mathsf{Int}} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} \texttt{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} \texttt{false} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} s : \mathsf{String}}$$

$$\frac{}{\Gamma \vdash_{\mathcal{D}} \texttt{()} : \mathsf{Unit}} \qquad \frac{\mathtt{ty}(\odot) = (A, B) \to C \qquad \Gamma \vdash_{\mathcal{D}} M : A \qquad \Gamma \vdash_{\mathcal{D}} N : B}{\Gamma \vdash_{\mathcal{D}} M \odot N : C} \qquad \frac{\Gamma \vdash_{\mathcal{D}} M : \mathsf{Bool}}{\Gamma \vdash_{\mathcal{D}} \texttt{!}M : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} M : A \qquad \Gamma, x : A \vdash_{\mathcal{D}} N : B}{\Gamma \vdash_{\mathcal{D}} \texttt{let}\ x = M\ \texttt{in}\ N : B} \qquad \frac{\Gamma \vdash_{\mathcal{D}} L : \mathsf{Bool} \qquad \Gamma \vdash_{\mathcal{D}} M : A \qquad \Gamma \vdash_{\mathcal{D}} N : A}{\Gamma \vdash_{\mathcal{D}} \texttt{if}\ (L)\ \{\ M\ \}\ \texttt{else}\ \{\ N\ \} : A}$$

$$\frac{\mathcal{D}(f) = \texttt{def}\ f((x_i : A_i)_{i\in I})\colon B\ \{N\} \qquad (\Gamma \vdash_{\mathcal{D}} M_i : A_i)_{i\in I}}{\Gamma \vdash_{\mathcal{D}} f((M_i)_{i\in I}) : B} \qquad \frac{(\Gamma \vdash_{\mathcal{D}} M_i : \mathsf{String})_{i\in I}}{\Gamma \vdash_{\mathcal{D}} \texttt{concat}((M_i)_{i\in I}) : \mathsf{String}}$$

$$\frac{\Gamma \vdash_{\mathcal{D}} M : \mathsf{String}}{\Gamma \vdash_{\mathcal{D}} \texttt{print}(M) : \mathsf{Unit}} \qquad \frac{}{\Gamma \vdash_{\mathcal{D}} \texttt{getInput}() : \mathsf{String}} \qquad \frac{\Gamma \vdash_{\mathcal{D}} M : B}{\Gamma \vdash_{\mathcal{D}} (M\ :\ A) : A}$$

$$
\begin{aligned}
&\texttt{ty}(\texttt{==}) = (A, A) \to \mathsf{Bool} \quad (\text{for any type } A)\\
&\texttt{ty}(+) = (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}\\
&\texttt{ty}(-) = (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}\\
&\texttt{ty}(*) = (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}\\
&\texttt{ty}(/) = (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}\\
&\texttt{ty}(\texttt{\%}) = (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Int}\\
&\texttt{ty}(\texttt{\&\&}) = (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}\\
&\texttt{ty}(\texttt{||}) = (\mathsf{Bool}, \mathsf{Bool}) \to \mathsf{Bool}\\
&\texttt{ty}(<) = (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Bool}\\
&\texttt{ty}(>) = (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Bool}
\end{aligned}
$$

Variables and constants. The typing rule for variables looks up a variable's type in the type environment. Integer literals have type Int, Boolean literals have type Bool, string literals have type String, and the unit value `()` has type Unit.

Operators. To type a Boolean operator $M \odot N$ we need to firstly determine the type of the arguments $\texttt{ty}(\odot) = (A, B) \to C$, given in the table above, and then check that M has type A and N has type B, with the resulting type being C.

In particular the equality operator == has type Bool if both arguments have the same type.

Let-bindings and conditionals. To type a let-binding $\textbf{let}\ x = M\ \textbf{in}\ N$, we firstly need to typecheck M to determine its type A. Then, we can extend our type environment with a mapping $x : A$ and return the result of typechecking N under the new environment.

Function calls. To typecheck a function call $f((M_i)_{i \in I})$, we firstly need to look up the function name in $\mathcal{D}$ to get the definition $\textbf{def}\ f((x_i : A_i)_{i \in I}) \colon B\ \{N\}$. Note that the number of arguments and parameters must be the same.

Next, we need to check that each argument has the matching type as given in the parameter list: that is, checking that M_1 has type A_1, and so on.

The final type is the result type of the called function (in this case B).

Note: you do *not* need to typecheck the function body N – this has already been done in the previous exercise.

String concatenation. Typing string concatenation involves ensuring that all arguments are strings; if so, the result type is also of type String.

Console IO. The $\textbf{print}(M)$ construct has type Unit if its argument M has type String. The $\textbf{getInput}()$ construct always has type String.

Casts. To type a cast $(M : A)$, we need to firstly typecheck term M to make sure it is well-typed. The return type is then A.

Exercise 5 (`typecheckExpr`)**.** *Implement the `typecheckExpr` method. Throw a* `TypeErrorException` *whenever you encounter a type error.*

[25 marks]

Changelog

- v1: Initial release
- v2: Fix typo before Exercise 1 (`typecheckExpr` corrected to `interpExpr`)