COMP3161/9164 22T3 Assignment 1
MinHs

Version 2.7.1

Marks : 17.5% of the mark for the course.

Due date: **Friday, 28th of October 2022, 12 noon Sydney time**

Overview

In this assignment you will implement an interpreter for MinHs, a small functional language similar to ML and Haskell. It is fully typed, with types specified by the programmer. The assignment consists of a base compulsory component, worth 70%, and five additional components, each worh 10%, which collectively are worth 50%, meaning that only three must be completed to earn full marks. Each additional bonus task completed will earn 0.5% more marks for the course. For example, if a student completes all tasks correctly, they will earn the 17.5% allocated to this assignment and an additional 1% on their course mark.

- **Task 1** (70%)
  Implement an interpreter for the MinHs language presented in the lectures, using an environment semantics, including support for recursion and closures.

- **Task 2** (10%)
  Extend the interpreter to support partially applied prim-ops.

- **Task 3** (10%)
  Extend the interpreter to support n-ary functions

- **Task 4** (10%)
  Extend the interpreter to support multiple bindings in the one `let` form.

- **Task 5** (10%)
  Extend the interpreter to support `let` bindings that take parameters, definining non-recursive functions.

- **Task 6** (10%)
  Extend the interpreter to support mutually recursive bindings.

The front end of the interpreter (lexer, parser, type checker) is provided for you, along with the type of the *evaluate* function (found in the file `Evaluator.hs`). The function *evaluate* returns an object of type *Value*. You may modify the constructors for *Value* if you wish, but not the type for *evaluate*. The return value of *evaluate* is used to check the correctness of your assignment.

You must provide an implementation of *evaluate*, in `Evaluator.hs`. It is this file you will submit for Task 1. No other files can be modified.

You can assume the typechecker has done its job and will only give you type-correct programs to evaluate. The type checker will, in general, rule out type-incorrect programs, so the interpreter does not have to consider them.

Please use the Ed forum for questions about this assignment.

Submission

Submit your (modified) `Evaluator.hs` using the CSE give system, by typing the command

```
give cs3161 Eval Evaluator.hs
```

or by using the CSE give web interface.

1 Task 1

This is the core part of the assignment. You are to implement an interpreter for MinHs. The following expressions must be handled:

- variables. v, u

- integer constants. $1, 2, ..$

- boolean constants. *True*, *False*

- some primitive arithmetic and boolean operations. $+, *, <, <=, ..$

- constructors for lists. *Nil*, *Cons*

- destructors for lists. *head*, *tail*

- inspectors for lists. *null*

- function application. *f* *x*

- if e then e_1 else e_2

- let $x :: \tau_x = e_1$; $y :: \tau_y = e_2$; ... in e_2

- `recfun` $f :: (\tau_1 \rightarrow \tau_2)\, x = e$ expressions

These cases are explained in detail below. The abstract syntax defining these syntactic entities is in `Syntax.hs`. You should understand the data type *Exp* and *Bind* well.

Your implementation is to follow the dynamic semantics described in the lectures, and this document. You are *not* to use substitution as the evaluation strategy, but must use an environment/heap semantics. If a runtime error occurs, which is possible, you should use Haskell's *error* :: *String* $\rightarrow$ *a* function to emit a suitable error message (the error code returned by *error* is non-zero, which is what will be checked for – the actual error message is not important).

1.1 Program structure

A program in MinHs may evaluate to either an integer, a list of integers, or a boolean, depending on the type assigned to the *main* function. The *main* function is always defined (this is checked by the implementation). In Task 1 programs, you need only consider the case of a single top-level binding for *main*, like so:

```
main :: Int = 1 + 2;
```

or

```
main :: Bool
  = let x :: Int = 1;
    in if x + ((recfun f :: (Int -> Int) y = y * y) 2) == 0
       then True
       else False;
```

1.2 Variables, Literals and Constants

MinHs is a spartan language. We only have to consider 4 types:

```
Int
Bool
t1 -> t2
[Int]
```

The only literals you will encounter are integers. The only non-literal constructors are *True* and *False* for the *Bool* type, and *Nil* and *Cons* for the *[Int]* type.

1.3 Function application

MinHs is, by virtue of its Haskell implementation, a non-strict language. An argument to a function is only evaluated when needed — when the function tries to inspect its value. This does not add a great deal of complexity to your implementation — it will occur naturally as you will be writing the interpreter in Haskell, which is also a non-strict language.

The result of a function application may in turn be a function.

1.4 Primitive operations

You need to implement the following primitive operations:

```
+       :: Int -> Int -> Int
-       :: Int -> Int -> Int
*       :: Int -> Int -> Int
/       :: Int -> Int -> Int

negate  :: Int -> Int

>       :: Int -> Int -> Bool
>=      :: Int -> Int -> Bool
<       :: Int -> Int -> Bool
```

```
<=        :: Int -> Int -> Bool

==        :: Int -> Int -> Bool
/=        :: Int -> Int -> Bool

head :: [Int] -> Int
tail :: [Int] -> [Int]
null :: [Int] -> Bool
```

These operations are defined over *Int*s, [*Int*]s, and *Bool*s, as usual. *negate* is the primop representation of the unary negation function, i.e. -1. The abstract syntax for primops is defined in `Syntax.hs`.

1.5 `if-then-else`

MinHs has an `if` e `then` e_1 `else` e_2 construct. The types of e_1 and e_2 are the same. The type of e is *Bool*.

1.6 `let`

For the first task you only need to handle simple `let` s of the kind we have discussed in the lectures. Like these:

```
main :: Int
    = let
        x :: Int = 1 + 2;
      in x;
```

or

```
main :: Int
    = let f :: (Int -> Int)
            = recfun f :: (Int -> Int) x = x + x;
      in f 3;
```

For the base component of the assignment, you do not need to handle `let` bindings of more than one variable at a time (as is possible in Haskell). Remember, a `let` may bind a recursive function defined with `recfun` .

1.7 `recfun`

The `recfun` expression introduces a new, named function value. It has the form:

```
(recfun f :: (Int -> Int) x = x + x)
```

A `recfun` value is a first-class value, and may be bound to a variable with `let` . The value 'f' is bound in the body of the function, so it is possible to write recursive functions:

```
recfun f :: (Int -> Int) x =
   if x < 10 then f (x+1) else x
```

Be very careful when implementing this construct, as there can be problems when using environments in a language allowing functions to be returned by functions.

4

1.8 Evaluation strategy

We have seen in the tutorials how it is possible to evaluate expressions via substitution. This is an extremely inefficient way to run a program. In this assignment you are to use an environment instead. You will be penalised for an interpreter that operates via substitution. The module `Env.hs` provides a data type suitable for most uses. Consult the lecture notes on how environments are to be used in dynamic semantics. The strategy is to bind variables to values in the environment, and look them up when requried.

In general, you will need to use: *empty*, *lookup* and *addAll* to begin with an empty environment, lookup the environment, or to add a binding to the environment, respectively. As these functions clash with functions in the `Prelude`, a good idea is to import the module `Env` qualified:

```
import qualified Env
```

This makes the functions accessible as *Env.empty* and *Env.lookup*, to disambiguate from the Prelude versions.

2 Dynamic Semantics of MinHs

Big-step semantics

We define a relation $\Downarrow$ which relates an environment mapping variables to values[1] Γ and an expression E to the resultant value of that expression V. Our value set for V will, to start with, consist of:

- *Machine integers*

- *Boolean values*

- *Lists of integers*

We will also need to add *closures*, or *function values* to our value set to deal with the `recfun` construct in a sound way. See the section on Function Values for details.

Environment

The environment Γ maps variables to values, and is used in place of substitution. It is specified as follows:

$$\Gamma ::= \ \cdot \mid \Gamma, x = v$$

Values bound in the environment are closed – they contain no free variables. This requirement creates a problem with function values created with `recfun` whose bodies contain variables bound in an outer scope. We must bundle them with their associated environment as a *closure*.

---

[1]Or, possibly, to an unevaluated computation of the value, but in Haskell these two things are indistinguishable.

Constants and Boolean Constructors

$$\overline{\Gamma \vdash \mathtt{Num}\ n \Downarrow n} \quad \overline{\Gamma \vdash \mathtt{Con\ True} \Downarrow \textit{True}} \quad \overline{\Gamma \vdash \mathtt{Con\ False} \Downarrow \textit{False}}$$

Primitive operations

$$\frac{\Gamma \vdash e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\Gamma \vdash \mathtt{Plus}\ e_1\ e_2 \Downarrow v_1 + v_2}$$

Similarly for the other arithmetic and comparison operations (as for the language of arithmetic expressions)

Note that division by zero should cause your interpreter to throw an error using Haskell's `error` function.

The abstract syntax of the interpreter re-uses function application to represent application of primitive operations, so `Plus` e_1 e_2 is actually represented as:

$$\mathtt{App}\ (\mathtt{App}\ (\mathtt{Prim\ Plus}\ e_1)\ e_2)$$

For this first part of the assignment, you may assume that prim-ops are never partially applied — that is, they are fully saturated with arguments, so the term `App` (`Prim Plus` e_1) will never occur in isolation.

Evaluation of *if*-expression

$$\frac{\Gamma \vdash e_1 \Downarrow \textit{True} \quad \Gamma \vdash e_2 \Downarrow x}{\Gamma \vdash \mathtt{If}\ e_1\ e_2\ e_3 \Downarrow x}$$

$$\frac{\Gamma \vdash e_1 \Downarrow \textit{False} \quad \Gamma \vdash e_3 \Downarrow x}{\Gamma \vdash \mathtt{If}\ e_1\ e_2\ e_3 \Downarrow x}$$

Variables

$$\frac{\Gamma(x) = v}{\Gamma \vdash \mathtt{Var}\ x \Downarrow v}$$

List constructors and primops

$$\overline{\Gamma \vdash \mathtt{Con\ Nil} \Downarrow []} \quad \frac{\Gamma \vdash x \Downarrow v_x \quad \Gamma \vdash xs \Downarrow v_{xs}}{\Gamma \vdash \mathtt{Cons}\ x\ xs \Downarrow v_x : v_{xs}}$$

$$\frac{\Gamma \vdash x \Downarrow v : vs}{\Gamma \vdash \mathtt{head}\ x \Downarrow v} \quad \frac{\Gamma \vdash x \Downarrow v : vs}{\Gamma \vdash \mathtt{tail}\ x \Downarrow vs} \quad \frac{\Gamma \vdash x \Downarrow v : vs}{\Gamma \vdash \mathtt{null}\ x \Downarrow \textit{False}}$$

$$\frac{\Gamma \vdash x \Downarrow []}{\Gamma \vdash \mathtt{head}\ x \Downarrow \mathtt{error}} \quad \frac{\Gamma \vdash x \Downarrow []}{\Gamma \vdash \mathtt{tail}\ x \Downarrow \mathtt{error}} \quad \frac{\Gamma \vdash x \Downarrow []}{\Gamma \vdash \mathtt{null}\ x \Downarrow \textit{True}}$$

For the first part of the assignment, you may assume that `Cons` is also never partially applied, as with prim-ops.

Variable Bindings with Let

$$\frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma, x{=}v_1 \vdash e_2 \Downarrow v_2}{\Gamma \vdash \texttt{Let } e_1 \ (x.e_2) \Downarrow v_2}$$

Function values

To maintain soundness with function values, we need to pair a function with its environment, forming a *closure*. We introduce the following syntax for function values: (the types are included for completeness, but are not required at runtime):

$$\langle\!\langle \Gamma; \texttt{Recfun } \tau_1 \ \tau_2 \ f.x.e \rangle\!\rangle$$

You will need to decide on a suitable representation of closures as a Haskell data type.

Now we can specify how to introduce closed function values:

$$\frac{}{\Gamma \vdash \texttt{Recfun } \tau_1 \ \tau_2 \ f.x.e_1 \Downarrow \langle\!\langle \Gamma; \texttt{Recfun } \tau_1 \ \tau_2 \ f.x.e_1 \rangle\!\rangle}$$

We also re-use the `recfun` syntax, only without arguments, to construct infinite structures like infinite lists. Here, the recursive reference is provided but no function value is necessary:

$$\frac{\Gamma, f = v \vdash e \Downarrow v}{\Gamma \vdash \texttt{Recfun } \tau \ f.e \Downarrow v}$$

Function Application

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 \Downarrow v_1 & v_1 = \langle\!\langle \Gamma'; \texttt{Recfun } \tau_1 \ \tau_2 \ f.x.e_f \rangle\!\rangle \\ \Gamma \vdash e_2 \Downarrow v_2 & \Gamma', f{=}v_1, x{=}v_2 \vdash e_f \Downarrow r \end{array}}{\Gamma \vdash \texttt{App } e_1 \ e_2 \Downarrow r}$$

---

3 Additional Tasks

*In order to get full marks in the assignment, you must do **three** of the following five tasks*:

3.1 Task 2: Partial Primops

In the base part of the assignment, you are allowed to assume that all primitive operations (and the constructor `Cons`) are fully saturated with arguments. In this task you are to implement *partial* application of primitive operations (and `Cons`), which removes this assumption. For example:

```
main :: Int
    = let inc :: (Int -> Int)
            = recfun inc :: (Int -> Int) = (+) 1;
      in inc 2;  -- returns 3
```

Note that the expression `(+) 1` partially applies the primop `Plus` to 1, returning a *function* from `Int` to `Int`.

You will need to develop a suitable dynamic semantics for such expressions and implement it in your evaluator. The parser and type checker are already capable of dealing with expressions of this form.

3.2 Task 3: n-ary functions

In this task you are to implement n-ary functions. In other words, you should modify the interpreter to handle bindings of functions of more than 1 argument.

```
main :: Bool
    = let eq :: (Int -> Int -> Bool)
            = recfun eq :: (Int -> Int -> Bool)
                         x y = x == y;
      in eq 3 4;
```

We haven't discussed the semantics of such functions in the lectures so you will need to work out a reasonable dynamic semantics for n-ary functions on your own, based on the semantics for unary functions. The parser and type checker are once again already capable of handling expressions of this form, so the only extension necessary is in the evaluator component.

Hint: Is the following example semantically different to the previous?

```
main :: Bool
    = let eq :: (Int -> Int -> Bool)
            = recfun eq :: (Int -> Int -> Bool)
                         x = recfun eq2 :: (Int -> Bool)
                                        y = x == y;
      in eq 3 4;
```

3.3 Task 4: Multiple bindings in **let**

In the base part of the assignment, we specify that `let` expressions contain only one binding. In this task, you are to extend the interpreter so that `let` expressions with multiple bindings, like:

```
main :: Int
    = let a :: Int = 3;
          b :: Int = 2;
      in a + b;
```

are evaluated the same way as multiple nested `let` expressions:

```
main :: Int
    = let a :: Int = 3;
      in let b :: Int = 2;
            in a + b;
```

Once again the only place where extensions need to be made are in the evaluator, as the type checker and parser are already capable of handling multiple `let` bindings.

3.4 Task 5: `let` bindings declare functions

In this task you are to extend the `let` construct further so that `let` bindings can take parameters — defining *non-recursive* functions. This makes programming in MinHs much less verbose, as `recfun` is only necessary for defining recursive functions. For example:

```
main :: Int
    = let y :: Int = 3;
        in let f :: (Int -> Int) x = x + 1;
            in f y;  -- returns 4
```

Note that these bindings are *non-recursive*, so `let x = x` is a scope error, as the x in the binding is not in scope.

3.5 Task 6: Mutually recursive bindings

Currently bindings must be specified in dependency order. However, in Haskell, the order of declarations is irrelevant, which allows *mutually recursive* bindings. Implement Haskell-style mutually recursive bindings, like so:

```
main :: Int
    = letrec a :: Int = b;
            b :: Int = c;
            c :: Int = 7;
        in c + a;
```

Syntax for `letrec` bindings is already defined, typechecked, and parsed by the provided code. To implement this part, you simply need to implement the evaluator portion.

4 Testing

Your assignments will be tested *very* rigorously: correctness is a theme of this subject, after all. You are encouraged to test yourself. `minhs` comes with a regress tester script, and you should add your own tests to this.

The tests that come with this assignment tarball, which are also run on submission as a dryrun, cover the *base part* (the first 70%) of the assignment only. You will be responsible for testing each extension adequately.

5 Building `minhs`

`minhs` (the compiler/interpreter) is written in Haskell, and requires GHC 7.4 or higher, and the `cabal` build tool included in the Haskell Platform, or the `stack` tool that is also popular for building Haskell projects. If you are using CSE machines, follow instructions for `cabal`, however if you are working on your own machine you may find it more convenient to use `stack`.

5.1 Building with `cabal` on CSE machines

All testing will occur on standard CSE Linux machines. Make sure you test your program on a CSE Linux machine. Once in a COMP3161 subshell, MinHS can be built with:

- `cabal update` to set up the package database.

- `cabal configure` to set up the build environment

- `cabal install --only-dependencies` to install the libraries on which the interpreter depends.

- `cabal build` to build the compiler

- `cabal run minhs-1 -- --help` will help you find any useful debugging options.

To run the interpreter on a file `foo.mhs`:

```
$ cabal run minhs-1 -- foo.mhs
```

You may wish to experiment with some of the debugging options to see, for example, how your program is parsed, and what abstract syntax is generated.

To run the test driver, a short shell script is provided. For usage information, type:

```
$ ./run_tests_cabal.sh --help
```

5.2 Building with `stack`

You should be able to build the compiler by simply invoking:

```
$ stack build
```

To see the debugging options, run (after building):

```
$ stack exec minhs-1
```

To run the compiler with a particular file, run:

```
$ stack exec minhs-1 -- foo.mhs
```

And to run all of our tests, type:

```
$ ./run_tests_stack.sh
```

6 Late Penalty

You may submit up to five days (120 hours) late. Each day of lateness corresponds to a 5% reduction of your total mark. For example, if your assignment is worth 88% and you submit it two days late, you get 78%. If you submit it more than five days late, you get 0%.

Course staff cannot grant assignment extensions—if you need an extensions, you have to apply for special consideration through the standard procedure. More information here: `https://www.student.unsw.edu.au/special-consideration`

7 Plagiarism

Many students do not appear to understand what is regarded as plagiarism. This is no defense. Before submitting any work you should read and understand the UNSW plagiarism policy `https://student.unsw.edu.au/plagiarism`.

All work submitted for assessment must be entirely your own work. We regard unacknowledged copying of material, in whole or part, as an extremely serious offence. In this course submission of any work derived from another person, or solely or jointly written by and or with someone else, without clear and explicit acknowledgement, will be severely punished and may result in automatic failure for the course and a mark of zero for the course. Note this includes including unreferenced work from books, the internet, etc.

Do not provide or show your assessable work to any other person. Allowing another student to copy from you will, at the very least, result in zero for that assessment. If you knowingly provide or show your assessment work to another person for any reason, and work derived from it is subsequently submitted you will be penalized, even if the work was submitted without your knowledge or consent. This will apply even if your work is submitted by a third party unknown to you. You should keep your work private until submissions have closed.

If you are unsure about whether certain activities would constitute plagiarism ask us before engaging in them!

References

[1] *Report on the Programming Language Haskell 98*, eds. Simon Peyton Jones, John Hughes, (1999) `http://www.haskell.org/onlinereport/`

[2] Robert Harper, *Programming Languages: Theory and Practice*, (Draft of 19 Sep 2005), `https://people.cs.uchicago.edu/~blume/classes/aut2008/proglang/text/offline.pdf`.

[3] The Implementation of Functional Programming Languages, Simon Peyton Jones, published by Prentice Hall, 1987. Full text online (as jpg page images).

[4] Simon Peyton-Jones, *Implementing Functional Languages : a tutorial*, 2000.

A Lexical Structure

The lexical structure of MinHS is an small subset of Haskell98. See section 2.2 of the Haskell98 report [1]. The lexical conventions are implemented by the Parsec parser library, which we use for our Parser implementation.

B Concrete syntax

The concrete syntax is based firstly on Haskell. It provides the usual arithmetic and boolean primitive operations (most of the Int-type primitive operations of GHC). It has conventional `let` bindings. At the outermost scope, the `let` is optional. As a result, multiple outer-level bindings are treated as nested `let` bindings down the page. It is required that a distinguished `main` function, of atomic type, exist. There is an `if-then-else` conditional expression. The primitive types of MinHS are `Int`,`Bool` and `[Int]`. MinHS also implements, at least partially, a number of extensions to MinML: inline comments, n-ary functions, infix notation, more primitive numerical operations and a non-mutually recursive, simultaneous `let` declaration (treated as a nested-`let`). Function values may be specified with `recfun` .

The concrete syntax is described and implemented in the `Parser.hs` module, a grammar specified using the Parser combinator library Parsec.

Features of Haskell we do not provide:

- No nested comments

- No layout rule. Thus, semi-colons are required to terminate certain expressions. Consult the grammar.

C Abstract syntax

The (first-order) abstract syntax is based closely on the MinHs syntax introduced in the lectures. It is implemented in the file `Syntax.hs`. Extensions to the MinHs abstract syntax take their cue from the Haskell kernel language. Presented below is the abstract syntax, with smatterings of concrete syntax for clarity.

D Static semantics

The static semantics are based on those of the lecture, and of MinML, from Bob Harper's book. They are implemented by the module `TypeChecker.hs`.

D.1 n-ary functions

Functions may be declared to take more than 1 argument at a time.

E Environments

Environments are required by typechecker and possibly by the interpreter. The typechecker needs to map variables to types, and the interpreter might need to map variables to functions or values (like a heap). This latter structure is used to provide a fast alternative to substitution.

$$
\begin{array}{lrcl}
\text{Types} & \tau & \rightarrow & \texttt{Int}\,|\,\texttt{Bool}\,|\,\tau \rightarrow \tau \\[6pt]
\text{Literals} & n & \rightarrow & \ldots\,|\,\texttt{0}\,|\,\texttt{1}\,|\,\texttt{2}\,|\,\ldots \\
& b & \rightarrow & \texttt{True}\,|\,\texttt{False} \\[6pt]
\text{Primops} & o & \rightarrow & \texttt{+}\,|\,\texttt{-}\,|\,\texttt{*}\,|\,\texttt{/}\,|\,\texttt{\%}\,| \\
& & | & \texttt{>}\,|\,\texttt{>=}\,|\,\texttt{==}\,|\,\texttt{/=}\,|\,\texttt{<}\,|\,\texttt{<=} \\[6pt]
\text{Expressions} & exp & \rightarrow & \texttt{Var}\,x \\
& & | & \texttt{Lit}\,n \\
& & | & \texttt{Con}\,b \\
& & | & \texttt{Apply}\,e_1\,e_2 \\
& & | & \texttt{Let}\,decl\,exp \\
& & | & \texttt{Recfun}\,decl \\
& & | & \texttt{If}\,exp\,exp_1\,exp_2 \\[6pt]
\text{Decl} & decl & \rightarrow & \texttt{Fun}\,f\,\tau\,[arg]\,e \\
& & | & \texttt{Val}\,v\,\tau\,e
\end{array}
$$

Figure 1: The expression abstract syntax of MinHS

We provide a general environment module, keyed by identifiers, in `Env.hs`.

Environments are generally simpler in MinHs than in real Haskell. We still need to bind variables to partially evaluated functions, however.

F   Dynamic semantics

The dynamic semantics are described in this document, the lectures, and resemble that of Harper [2]. Implemented in the module `Evaluator.hs`.

F.1   Interpreter

The interpreter is the backend that runs by default. It should implement the dynamic semantics of MinHs.

G   Interfaces

The basic types are found in `Syntax.hs`, which contains definitions for the structure of terms, types, primOps, and others.

Printing

Most structures in MinHS need to be printed at some point. The easiest way to do this is to make that type an instance of class Pretty. See `Pretty.hs` for an example.

Testing

```
./run_tests_cabal.sh
```

Check directories may have an optional 'Flag' file, containing flags you wish to pass to minhs in that directory, or the magic flag, 'expect-fail', which inverts the sense in which success is defined by the driver.