Interpreters

An interpreter is a program that takes another program as an input and evaluates it. Many modern languages such as Java\(^1\), Javascript, Python, and PHP are *interpreted*. Although interpreted languages are generally slower than compiled ones such as Haskell, OCaml, and C, they offer many advantages such as platform independence.

Haskell’s type system makes it an excellent *metalanguage*. A metalanguage is a language that is used to reason about a different language. In this assignment, you will implement an interpreter for a simple imperative language in Haskell.

Meet SImPL

SImPL is the Simple Imperative Programming Language that you will be interpreting. This language is comprised of seven kinds of statements: assignments, increments, if statements, while loops, for loops, sequences, and skips. The only kinds of values that exist in SImPL are integers. Integers can be represented in expressions as literals (ie -1, 0, 5), variables, or the result of a computation involving two sub-expressions. An example SImPL program that calculates the value of B to the power of E is:

\(^{1}\) Java is first compiled to bytecode which is then interpreted by the Java Virtual Machine
Out := 1;
for (I := 0; I < E; I++) {
    Out := Out * B
}

At the end of the program's execution, the value of B to the power E is stored in the variable Out. This SImPL program is nicely readable by humans, however it is not so easy for Haskell to work with.

In Haskell, we will represent SImPL programs using Abstract Syntax Trees or ASTs. An AST is just a Haskell datatype that represents the structure of a piece of code. The datatypes that we will be working with in this assignment are defined in HW03.hs. The SImPL program above could be represented in Haskell as follows:

Sequence
  (Assign "Out" (Val 1))
  (For (Assign "I" (Val 0))
    (Op (Var "I") Lt (Var "E"))
    (Incr "I")
    (Assign (Var "Out") (Op (Var "Out") Times (Var "B"))))

Before you continue, make sure you understand the correspondence between the SImPL program and its representation in Haskell. All of this assignment will be based on working with and manipulating the AST.

**Exercise 1** Before we can start evaluating Expressions and Statements we need some way to store and look up the state of a variable. We define a State to be a function of type String -> Int. This makes it very easy to look up the value of a variable; to look up the value of "A" in state, we simply call state "A". Whenever we assign a variable, we want to update the program State. Implement the following function:

```
extend :: State -> String -> Int -> State
```

*Hint:* You can use the input State as a black box for variables other than the one you are assigning.

*Example:*

```
let st' = extend st "A" 5
in st' "A" == 5
```

In addition to the ability to extend States, we need to have an empty State that we can use to evaluate programs where no variables are assigned. In the empty State, it might make sense to throw
an exception when you try to access a variable that has not been defined. Unfortunately, SimPL has no notion of exceptions\(^2\). For this reason, we will define the empty State to set all variables to 0. Define the empty State:

\[
\text{empty} :: \text{State}
\]

**Example:**

\[
\text{empty} \; "A" \; == \; 0
\]

**Exercise 2** We are now ready to evaluate expressions! This can be implemented as a fairly straightforward recursive function. The value that an Expression evaluates to depends on the state of the variables that appear in the Expression. For this reason, a State must be provided along with the Expression that is being evaluated. Implement the function:

\[
\text{evalE} :: \text{State} \to \text{Expression} \to \text{Int}
\]

*Note:* some of the binary operators (Bops) in SimPL are \text{Int} \to \text{Int} \to \text{Int} functions, but others have type \text{Int} \to \text{Int} \to \text{Bool}. Since SimPL only has integer types, we need to modify these functions to return Ints. We will do so by returning 0 in place of False and 1 in place of True.

**Example:**

\[
\text{evalE} \; \text{empty} \; (\text{Val} \; 5) \; == \; 5
\]

**Example:**

\[
\text{evalE} \; \text{empty} \; (\text{Op} \; (\text{Val} \; 1) \; \text{Eql} \; (\text{Val} \; 2)) \; == \; 0
\]

**Exercise 3** It turns out that SimPL isn’t so simple after all. The syntax of SimPL contains some repetition. For example, Incrs are just special cases of Assigns. Syntax like this makes it easier for programmers to reason about code, however it makes the internal representation of the language more complicated. This is called Syntactic Sugar\(^3\). SimPL is a little too sweet for my taste, so before we evaluate SimPL Statements we are going to desugar them.

A DietStatement is a new datatype that we will use to represent desugared SimPL Statements. You may notice that DietStatements are very similar to Statements. In fact, the DietStatement type is the same as the Statement type with two constructors removed: Incr and

\(^2\) SimPL has no type errors either; everything is an integer, so typechecking isn’t necessary!

\(^3\) Haskell has plenty of Syntactic Sugar. We will learn much more about this when we learn about Monads in a few weeks. Before Haskell gets compiled to native machine code, it goes through an intense desugaring process. The result is a language called GHC Core.
For Loop in SimPL
   \texttt{for (A := 0; A < N; A++) {...}}

For Loop represented as an AST
   \texttt{For (Assign "A" (Val 0)) (Op (Var "A") Lt (Var "N")) (Incr "A") (...)}

Explanation
   \begin{itemize}
     \item \textbf{Initialization}: Executed once at the very beginning
     \item \textbf{Loop Condition}: Expression dictating whether or not the body of the loop should be executed.
     \item \textbf{Update}: Executed every iteration after the body of the loop
   \end{itemize}

Now, implement the following function:

\texttt{desugar :: Statement -> DietStatement}

Example:
\begin{verbatim}
  desugar (Incr "A") == DAssign "A" (Op (Var "A") Plus (Val 1))
\end{verbatim}

\textbf{Exercise 4} In this exercise, you will write a function that evaluates desugared Statements. Unlike Expressions, Statements do not evaluate to a single value. Instead, they perform actions that mutate the initial State. Since we cannot actually mutate the State in Haskell, we will return a new State that is the result of evaluating the program. Implement the function:

\texttt{evalSimple :: State -> DietStatement -> State}

\textit{Note:} If statements and While loops both represent conditionals as an Expression. Since Expressions in SimPL are integers, we will consider 0 to be False and every other value to be True\textsuperscript{4}.

Example:
\begin{verbatim}
  let s = evalSimple empty (DAssign "A" (Val 10))
  in s "A" == 10
\end{verbatim}

We also want to be able to run programs that have not already been desugared. Implement the function:

\texttt{run :: State -> Statement -> State}

\textit{Note:} \texttt{run} should be defined in terms of \texttt{desugar} and \texttt{evalSimple}. It should not be implemented from scratch.

\textsuperscript{4} This is the standard convention in C. There is no boolean type in C either.
Running Programs

Congratulations! You have written an interpreter for a simple imperative language. But what good is an interpreter without any programs to test it on? We have provided three programs to test your code on: factorial, square root, and fibonacci.

You can use these programs to test your interpreter. In each case, the input is assumed to be assigned to the variable "In" and the output is assign to "Out". So, for example, if you wanted to run the factorial program with input 4, you should run the program with a State that binds the variable "In" to 4.

Example:

let s = run (extend empty "In" 4) factorial
in s "Out" == 24