Programming Assignment

An Interpreter for a Simple Programming Language

In this programming assignment, you will implement an interpreter for a simple programming language we will call LANG. You will need to write a parser that takes a LANG program (as a `String`) and returns a parse tree for the program. Then, you will write the interpreter that executes that parse tree.

Included with this assignment are two Haskell source files:

- `Parsing.hs` is a module that defines the basic monadic parsing API we studied in the section on monad transformers. You will use these parsing primitives to implement a recursive-descent parser for LANG programs.
- `AST.hs` is a module that defines the data types representing the abstract syntax of a LANG program. Your parser should use these types as its output and your interpreter should for its input.

1 The Language

The structure of a LANG program is reflected in the types defined in `AST.hs`:

- A program is a list of statements.
- A statement is one of the following:
  - A print statement for printing numeric values: `print (1 + 1)`
  - A print statement for string literals: `sprint "Hello world!"`
  - A read statement to read a number from the console and store that value in a variable: `read x`
  - An assignment statement: `x = 42`
  - A while loop that executes a statement for as long as a control expression evaluates to a true value:

```
while x < 10 {
    x = x + 1
    print x
}
```

- An if-then statement that executes a statement if a control expression evaluates to a true value:

```
if x < 10 then
    sprint "Less than ten"
```

- An if-then-else statement that executes one statement if a control expression evaluates to a true value but executes a second statement otherwise:

```
if x < 10 then
    sprint "Less than ten"
else
    sprint "Greater than or equal to ten"
```

- A compound statement, surrounded by braces, that can contain zero or more other statements. Since the abstract syntax for `if` and `while` statements only admits a single statement body, compound statements allow for more complex constructs. A compound statement is seen in the sample `while` loop above.

- An expression is one of the following:
  - A variable reference that evaluates to the variables current value in the global name scope.
  - A literal integer constant. You only need to support decimal notation with optional negation using (e.g., `6` for negative six).
  - Binary operator application. See the description of binary operators below.
  - Unary operator application. The only unary operator you need to support is `!` for logical negation.

- `read` statements, assignment statements, and variable references require the notion of identifiers. Your parser should support identifiers that start with an alphabetic character followed by zero or more alphanumeric characters.
- `sprint` uses string literals. Your parser does not need to support escape sequences in string literals.

1.1 Expressions, Arithmetic, and Logic

- LANG does not support binary operators with different precedence or associativity. As such, binary expressions must be surrounded with parentheses if they are subexpression of a larger expression.
- All arithmetic in LANG is with integers, specifically Haskell's arbitrary-precision integers. The division operator should perform integer division only.
- Relational and comparison operators should return `1` for true and `0` for false.
- Logical operators (including unary logical negation) should treat non-zero values as true and `0` as false.

The following binary operators should be supported:

- `+` – Addition
- `-` – Subtraction
- `*` – Multiplication
- `/` – Division
- `^` – Exponentiation
- `%` – Modulus
- `==` – Equality
- `>` – Comparison: greater than
- `<` – Comparison: less than
- `>=` – Comparison: greater than or equal to
- `<=` – Comparison: less than or equal to
- `&&` – Logical AND
- `||` – Logical OR

2 Implementation

Copy the starter files (`Parsing.hs` and `AST.hs`) into a directory for your implementation. Create a Haskell source file `Main.hs` with a function `main :: IO ()` to hold your program's main entry point.

Create separate modules for your parser and interpreter and import those into your `Main.hs`. The parsing module will need to import `Parsing`, the module defined in `Parsing.hs` and the `AST` module in `AST.hs`. Your interpreter module will just need the `AST` module.

You main function should use the `System.Environment` module to read your program's command-line arguments. If exactly one argument was given, treat that as the file name of the source file to interpret. Load the the contents of that file as a string and parse it. If the parse was successful, interpret the resulting `Program` value. If parse was not successful, report the message to the user and exit.

You should be able to compile your program into an executable with:

```
> ghc Main.hs
```

You should also be able to run your program without compiling it with:

```
> runghc Main.hs
```

2.1 Implementing the parser

The `Parsing` module defines the `Parser` type, a state monad with the `ExceptT` transformer adding the possibility of failure with `ParseError` values. Your ultimate goal is to write a parser of type `Parser Program` in terms of lower-level parsers for statements, expressions, identifiers, etc.

Composing parsing primitives into more complex parsers allows us to avoid an explicit lexing step, where a stream of characters is broken down into a stream of important tokens like identifiers, string and integer literals, punctuation,

etc. However, this means your parser needs to explicitly ignore whitespace and newlines. Use the `token`, `strToken`, and `whitespace` parsers to help with this. Try to move whitespace handling to the lowest-level functions so that higher-level parsers are more readable.

Be sure to consider what happens when your parser finishes parsing a single statement and the remaining input cannot be parsed. This is probably erroneous. Consider how you might use the `eoi` (end-of-input) parser to require that the program parse the input buffer in its entirety before succeeding.

2.2 Implementing the interpreter

Because LANG programs need to read and write to the console while also manipulating the global name scope, you will need to compose the `IO` monad with the `StateT` transformer. You will also need to write a function that can execute an action in this combined monad.

Use the `Map` type from the `Data.Map` module to map `String` identifiers to their values.

Consider writing a function for interpreting a single statement that uses pattern matching to consider each of LANG's statement types. How might you implement `while`-loop iteration in Haskell? How can you interpret a `Program`, i.e., a list of `Statement`s, without explicit recursion?

Consider writing some basic primitive operations for your composed `IO`/`StateT` monad for loading the value of an variable, storing a new value in a variable, reading and writing integers, and echoing strings to the console. Write your interpreter function in terms of these primitives.