ARITHMETIC$_{\text{LANG}}$

POPL Instructors

September 9, 2019

Contents

In this assignment, we build an interpreter for the ARITHMETIC language. Here are the various components of the implementation. The ARITHMETIC language implements simple arithmetic operations. The specifications for the language are described here.

1 Instructions

1.1 Versions And Formatting

- Use GHC version 8.4.2 or greater. Ping me for download issues.

- Your language will be evaluated in the GHCi REPL, by loading the file like so:

```
> :l arithmetic.hs
> run <expression>
```

- You may write code for each of the components under the given headings, and tangle it all in a file called `arithmetic.hs`, or you may write the code wherever you would like. The code should, however, be in this file and tangle out to `arithmetic.hs`.
- If orgmode or emacs is posing a problem, start writing it in a `.hs` file, and put it in orgmode later. Do not delay starting this assignment because of orgmode issues.
- the `haskell-mode` and `dante` emacs packages provide Haskell support in emacs.

1.2 Help, Hints and Evaluation

The instructions for this assignment:

- No code snippets or boilerplate is given for this assignment. This is because Haskell as a language has very little boilerplate, and also to allow you the freedom of implementing the interpreter using any feature of Haskell you feel like using.
- However, since this class is unfamiliar with Haskell, you will be given code snippets and help…

  … if you ask for it. This is how it will go:

  – Each student has a reserve of 60 bonus points. Ten bonus points for implementation of each of the following components:
    - Expressible values
    - Operators
    - AST
    - Concrete Syntax
      - Parser
    - Evaluator
    - Running the interpreter (or: putting it all together)

  - There will be a separate hints thread on the course forum for this assignment. Any student who wishes to ask for a hint will post "hint" + <name of component>. I will privately send them the hint by e-mail, and dock ten bonus points from their reserve.
    * Hints for functions will be given as type annotations.
    * Hints for types will be more vague- eg: "implement this as a recursive type", et cetera.
  - No requests for hints will be responded to beginning from 12 hours before the assignment deadline. Start early. If you have an issue with starting early- clashing assignments, etc- contact me early.
  - There will be a time gap of 12 hours for each student before they can ask for the next hint.
  - The intent is to encourage you to play around with the features of the language and write your own code. Remember: points that might be docked for incorrect/partially working implementations can be redeemed in bonus points. However, code for each component should be there and at least partially working during the eval- you will not get bonus points for an unworkable or absent interpreter.
  - Help on theory- eg: "What is an interpreter", "how to write an AST", etc, will not reduce your bonus points.

- Evaluation will be manual- as in, I will ask you to show how you have implemented each of the features within your code, and demonstrate a working example of your language.

- Any plagiarism- from online sources or from each other- will be caught. If a significant portion of your code is deemed to be copied, you will get zero for this assignment.

1.3 "Can I do/use this library/feature?"

You are not allowed to import any external packages. Haskell has a `base` package that includes a number of basic libraries- `http://hackage.haskell.org/package/base` - and importing and using any one of these is allowed. The one exception: do not use any of the libraries under `Text.ParserCombinators`.

Language extensions were not taught in class and should not be needed for this assignment. However, if you have the knowledge, feel free to use language extensions, under the stipulation that I will ask you to explain what that language extension does.

As for using any other features of Haskell- if you can stay within the above two rules, then the answer is "yes". I would, however, recommend not getting too complicated, and restricting yourself to the features that have been taught in class.

2 Semantic Domains

2.1 Expressible Values

The language expresse two types of values:

- 1. Numbers
- 2. Booleans

2.2 Primitive operations

The language has syntactic support for expressing the following operations:

1. **Addition** takes two numbers and returns a number which is the sum of the two arguments.
2. **Multiplication** takes two numbers and returns a number whichi is the product of the two arguments.
3. **Subtraction** takes two numbers and returns a number which is the difference of the two numbers.
4. **Division** takes two numbers, the second being non-zero, and returns a number which is the quotient of the two arguments.
5. **Equality** takes two value and determines if they are equal.
6. **zero?** takes a number and determines if it is equal to zero.

The evaluation throws an error if the number or types of the arguments do not match those expected by the operators.

3 Concrete Syntax

The concrete syntax is ‘Racket like’ and uses parentheses.

The syntax may be written using BNF (Backus Naur Form):

```
<exp> ::=  <num> | <bool> | <prim-app>
<prim-app> ::= (<op> <exp> ...)

<num> ::= Haskell’s =Int= type
<bool> ::= Haskell’s =Bool= type
<op> ::= + | * | - | / | = | IsZero
```

The `op` type can be expressed in the `ARITHMETIC` language as symbols, or it can be expressed as words (`Add` instead of ’+’, etc.)

3.0.1 Examples of expressions

(Note if you’re reading this in emacs: the `=` signs around the expression are orgmode syntax, not part of the language.)

1. `(5)`
2. `(/ 3 2)` or `(Div 3 2)`
3. `(True)`
4. `( + 4 (* 2 5))` or `( Add 4 (* 2 5))`
5. `( = 3 5)` or `(Equals 3 5)`
6. `(IsZero 4)`

4 Parser

The parser takes an expression- a string- and returns an AST.

Haskell will automatically provide parsing capability for you if you define `Show` and `Read` instances for your types. You can do that if you have a sufficiently good grasp on these typeclasses, or you can write your own parser.

5 Evaluator

The evaluator takes an ast, evaluates it and returns a value.

6 Run

This is your interpreter function. Make it a REPL (Read-Eval-Print-Loop) if you want to.