Computer Science 3MI3 – 2020 assignment 1

A language of expressions

Mark Armstrong

September 21st, 2020

Contents

Introduction

This assignment asks you to construct interpreters for a simple language of mathematical expressions. To put it another way, you are asked to construct simple calculators.

We provide an informal description of the language below. Your task is to then represent the terms of the language in Scala and Prolog, using the languages' facilities for representing such data.

Updates and file history

September 21st

- Original version published.

October 13th

- **A documentation requirement has been added!**
  - As part of the boilerplate, which is now in place.
- A submission checklist was added near the end of this file to help bring attention to the documentation requirement.
- The testing code is now given for the Scala portions.
  - The scripts and the Prolog code should come soon.

- The instructions to use a "`Simple`" constructor in the Scala types of parts 3 and 4 have been removed. Instead, the constructors from part 1 should be repeated.
  - (I had been over-eager in trying to avoid repetition, and somehow failed to realise that the use of the "`Simple`" constructor would rule out most interesting expressions .)
- Instructions are now given at the beginning of each part as to the filenames.
- The naming of the types and the interpretation functions/predicates has also been standardised.
- The word "task" has been replaced by "part".

Boilerplate

Documentation

In addition to the code for the assignments, you are required to submit (relatively light) documentation, along the lines of that found in the literate programs from lectures and tutorials.

- Those occasionally include a lot of writing when introducing concepts; you do not have to introduce concepts, so your documentation should be similar to the *end* of those documents, where only the purpose and implementation details of types, functions, etc., are discussed.

This documentation **must be** in the literate style, with (nicely typeset) English paragraphs alongside code snippets; comments in your source code do not count. The basic requirement is

- the English paragraphs must use non-fixed width font, whereas
- the code snippets must use fixed width font.
- For example, see these lecture notes on Prolog:
  - `https://courses.cs.washington.edu/courses/cse341/98sp/logic/prolog.html`

But you are encouraged to strive for nicer than just "the basic requirement". (the ability to write decent looking documentation is an asset!

You are free to present your documentation in any of these formats:

- an HTML file,
  - (named `README.html`)
- a PDF (for instance, by writing it in LaTeX using the `listings` or `minted` package for your code blocks),
  - (named `README.pdf`), or
- rendering on GitLab (for instance, by writing it in markdown or Org)
  - (named `README.md` or `README.org`.)

If you wish to use another format, contact Mark to discuss it.

Not all of your code needs to be shown; only portions which are of interest are needed. Feel free to omit some "repetitive" portions. (For instance, if there are several cases in a definition which look almost identical, only one or two need to be shown.)

Submission procedures

The same guidelines as for homework (which can be seen in any of the homework files) aapply to assignments, except for the differences below.

Assignment naming requirements

Place all files for the assignment inside a folder titled `an`, where `n` is the number of the assignment. So, for assignment 1, use the folder `a1`, for assignment 2 the folder `a2`, etc. Ensure you do not capitalise the `a`.

Each part of the assignments will direct you on where to save your code for that part. Follow those instructions!

If the language supports multiple different file extensions, you must still follow the extension conventions noted in the assignment.

Incorrect naming of files may result in up to a 5% deduction in your grade.

This is slightly decreased from the 10% for homeworks.

Proper conduct for coursework

Refer to the homework code of conduct available in any of the homework files. The same guidelines apply to assignments.

Informal description of the expression language

The language consists of integer constants and seven *prefix* operators.

- The unary operator `const` which operates on integers.
- The unary operators `neg` (negative) and `abs` (absolute value) which operate on expressions of this language.
- The binary operators `plus`, `times`, `minus` and `exp` (exponent) which operator on expressions of this language.

The fact that all the operators are prefix means that this language does not admit ambiguity; that is, even without using parentheses, there is only one possible reading of a given expression.

Example 1

As an example of a syntactically correct expression, consider

```
abs minus abs const 5 abs const 6
```

which, if we parenthesised the expression and dropped the `const`'s, would read

```
(abs (minus (abs 5) (abs 6)))
```

or, in the usual notation, $|(|5| - |6|)|$

Example 2

As an additional example, consider

```
plus plus plus exp const 1 const 2 exp const 2 const 2
    exp const 3 const 2 exp const 4 const 2
```

which, if we parenthesised the expression and dropped the `const`'s, would read

```
(plus (plus (plus (exp 1 2) (exp 2 2)) (exp 3 2)) (exp 4 2))
```

or, in the usual notation, $(((1^2 + 2^2) + 3^2) + 4^2)$

Part 1 – Interpreter written in Scala [20 marks]

Place your code for this part in a file `a1p1.sc`.

1.1 Representation [5 marks]

Create a representation of the above described expressions in Scala using the standard approach for algebraic datatypes, and call this type `Expr`.

Name your constructors after the operators, of course capitalising the names.

So the expression

```
abs minus abs const 5 abs const 6
```

should be represented using your type as

```
Abs(Minus(Abs(Const(5)), Abs(Const(6))))
```

1.2 Interpreter [15 marks]

Define a method `interpretExpr` acting on `Expr` which calculates the value of an `Expr`.

Part 2 – Interpreter written in Prolog [20 marks]

Place your code for this part in a file `a1p2.pl`.

1.1 Expression recogniser [5 marks]

Define a predicate `isExpr` in Prolog which recognises trees which represent the above defined expressions.

The labels on the trees should be the name of the operation appended by `E` (in order to avoid any clash with builtin predicates of the same name.)

So the expression

```
abs minus abs const 5 abs const 6
```

would be represented in Prolog as

```
absE(minusE(absE(constE(5)), absE(constE(6))))
```

and the query `isExpr(absE(minusE(absE(constE(5)), absE(constE(6)))))` would result in the response `true`.

1.2 Interpreter [15 marks]

Define a binary predicate `interpretExpr` which relates these expressions to their numerical values.

That is, querying `interpretExpr(e,X)` should receive the response `X = n` where `n` is the integer value of the expression.

Do not concern yourself with the other direction; that is, we are not expecting queries of the form `interpretExpr(X,n)` to result in an answer. (If you can, "fail gracefully", i.e., either throw a more meaningful exception or avoid an exception altogether but this is not expected even as a bonus.)

Part 3 – Variables and substitution [30 marks]

Place your code for this part in files `a1p3.sc` and `a1p3.pl`.

Part 3 description

In this part, we wish to add variables to our expressions.

This can be done by adding a new operator `var` to our expressions, which operates on symbols representing variable names (prolog has atoms which can be used a symbols, and Scala has a `Symbol` type. The reason why symbols are a better candidate than strings will be discussed when we discuss types.)

However, adding this operator introduces a problem with our `interpretExpr` method/predicate: how do we interpret a variable when we don't know its value?

Our solution in this assignment is to introduce a *substitution* operator, which takes three values:

1. an expression to perform the substitution on,
2. the variable to be substituted, and
3. the expression to substitute for the variable.

For instance,

```
subst var x x const 6
```

would be written, using our usual mathematical syntax, as

```
x[x   6]
```

It should interpet to just `6`.

(An alternate solution, instead of building variable substitution into the language, would be to add a *state* argument to `interpretExpr`, which maps variables to values. We will use states in later assignments for this purpose.)

The part 3

In Scala, create a new algebraic datatype `VarExpr` and a new interpretation method `interpretVarExpr` in Scala ([15 marks].) Your new constructors should be called `Var` and `Subst`. ~~You should reuse your existing `Expr` type when defining `VarExpr`, by including a constructor `Simple` which takes a `Expr` argument.~~ (Edited October 13th.) You should also repeat the constructors from the part 1 as constructors for this new type.

In Prolog, create a new recognising predicate `isVarExpr` which recognising valid expressions in this extended language, with the new labels being the operator names given in the description, and a new predicate `interpretVarExpr` which relates expressions to their interpreted value ([15 marks].)

Pay attention when implementing the interpretation of substitution! Review the concept of variable binding, and do not substitute instances of the variable name which are bound elsewhere!

(This note added October 13th.) Note: you are allowed and encouraged to collect any helper methods, for instance a definition of exponentiation, in a separate file which is imported into your files for the different parts of this assignment. Just ensure you use Ammonite-compatible import directives in your Scala code.

Hint (added October 13th)

You may need to define a helper function/predicate before you can define interpretation here.

Part 4 – Boolean expressions [30 marks]

Place your code for this part in files `a1p4.sc` and `a1p4.pl`.

Part 4 description

In this part, we create an alternate extension to our first language of expressions (that is, we build on to `Expr`, not `VarExpr`.)

Our goal here is to add a second *type* of expressions to the language. Namely, we are adding booleans.

The new operators are the 0-ary `tt` and `ff` (0-ary meaning taking no arguments), the unary `bnot` and the binary `band` and `bor`.

For example, we have the new expression

```
bnot band tt bor ff tt
```

which in our usual notation would be written

```
¬ (true  (false  true))
```

These expressions cannot legally be allowed to mix with integer expressions; that is, trying to apply a integer operator to a boolean expression or vice versa is not legal.

The part 4

In Scala, create a new algebraic datatype `MixedExpr` and a new interpretation method `interpretMixedExpr` in Scala ([15 marks].) Your new constructors should be named `TT`, `FF`, `Band`, `Bor` and `Bnot`. ~~You should again reuse your existing `Expr` type when defining `TypedExpr`, by including a constructor `IntExpr` which takes a `Expr` argument.~~ Your interpreter should return an `Option[Either[Int,Boolean]]`. The `Option` type is used to handle failure in the case of a "mixed" expression. The `Either` type is used to handle two possible return types.

(Edited October 13th.) As in part 3, you will need to repeat the constructors from the `Expr` type in part 1 in this type's definition.

In Prolog, create a new recognising predicate `isMixedExpr` which recognising valid expressions in this extended language, with the new labels being the operator names given in the description, and a new predicate `interpretMixedExpr` which relates expressions to their interpreted value ([15 marks].)

Part 5 – Bonus: parsing [10 bonus marks]

Place your code for the bonus, if you attempt it, in `a1p5.sc` and/or `a1p5.pl`.

Create a *parser* for the first language of expressions in this assignment, both in Scala and in Prolog (partial marks for implementing it in just one language.)

A parser will take as argument a string such as

```
abs minus abs const 5 abs const 6
```

and return an `Expr` representing the expression in that string.

You will likely want to first define a *lexer* for the language, that converts a string to a list of lexemes (to do this, you will need to represent lexemes somehow.) Such a list is far easier to match over than a string.

Submission checklist

For your convenience, this checklist is provided to track the files you need to submit. Use it if you wish.

```
- [ ] Documentation; one of
  - [ ] README.html
  - [ ] README.pdf
  - [ ] README.md
  - [ ] README.org
- [ ] Part 1
  - [ ] a1p1.sc
  - [ ] a1p1_test.sc tests passed!
- [ ] Part 2
  - [ ] a1p2.pl
  - [ ] a1p2_test.plt tests passed!
- [ ] Part 3
  - [ ] a1p3.sc
  - [ ] a1p3.pl
  - [ ] a1p3_test.sc tests passed!
  - [ ] a1p3_test.plt tests passed!
- [ ] Part 4
  - [ ] a1p4.sc
  - [ ] a1p4.pl
  - [ ] a1p4_test.sc tests passed!
  - [ ] a1p4_test.plt tests passed!
- [ ] Part 5 (Bonus)
  - [ ] a1p5.sc
  - [ ] a1p5.pl
```

Testing

The tests

Scala

- Framework a1_testframework.sc

```
/* Given an expected result and a computed result,
   check if they are equal in value.
   If so, return 0. Otherwise, inform the user, and
↪  return 1,
   so the number of failures can be counted. */
def test[A](given: A, expected: A, the_test: String) =
  if (!(given equals expected)) {

    ↪  println("+-------------------------------------------------")
    println("| " + the_test + " failed.")
    println("| Expected " + expected + ", got " + given +
    ↪  ".")

    ↪  println("+-------------------------------------------------")
    1
  } else {
    0
  }

def runTests[A](tests: List[Tuple3[A,A,String]]): Unit =
↪  {
  // Apply test to each element of tests, and sum the
  ↪  return values.
  // This is essentially a for loop.
  val failed = tests.foldLeft(0) {
    (failures, next) => next match {
      // Deconstruct the tuple to get its parts
      case (given, expected, the_test) => failures +
      ↪  test(given, expected, the_test)
    }
  }
```

```
    println("+-------------------------------------------------")
  println("| " + failed + " tests failed")

    println("+-------------------------------------------------")
}
```

- Part 1 a1p1_test.sc

```
import $file.a1_testframework, a1_testframework._
import $file.a1p1, a1p1._

val tests = List(
  (interpretExpr(Const(0)), 0, "Constant expression 0"),
  (interpretExpr(Plus(Const(5),Const(4))), 9, "Additive
  ↪  expression 5 + 4"),
  (interpretExpr(Exp(Const(5),Const(4))), 625,
  ↪  "Exponential expression 5 ^ 4"),
  )

runTests(tests)
```

- Part 2 There is no Scala code for part 2.

- Part 3 a1p3_test.sc

```
import $file.a1_testframework, a1_testframework._
import $file.a1p3, a1p3._

val tests = List(
  (interpretVarExpr(Const(0)), 0, "Constant expression
  ↪  0"),
  (interpretVarExpr(Plus(Const(5),Const(4))), 9,
  ↪  "Additive expression 5 + 4"),
  (interpretVarExpr(Exp(Const(5),Const(4))), 625,
  ↪  "Exponential expression 5 ^ 4"),
  // New tests for this part specifically.
  (interpretVarExpr(Subst(Var('x), 'x, Const(0))), 0,
                    "Simple substitution x[x := 0]
                     ↪  expression"),
  (interpretVarExpr(Subst(Subst(Plus(Var('x),Var('y)),
                                 'x,
```

```
                                    Const(5)),
                                'y,
                                Const(4))),
                            9, "Nested substitution ((x + y)[x :=
                            ↪  5])[y := 4]"),
  )

  runTests(tests)
```

- Part 4 a1p4_test.sc

```
import $file.a1_testframework, a1_testframework._
import $file.a1p4, a1p4._

val tests = List(
  (interpretMixedExpr(Const(0)), Some(Left(0)), "Constant
  ↪  expression 0"),
  (interpretMixedExpr(Plus(Const(5),Const(4))),
  ↪  Some(Left(9)), "Additive expression 5 + 4"),
  (interpretMixedExpr(Exp(Const(5),Const(4))),
  ↪  Some(Left(625)), "Exponential expression 5 ^ 4"),
  // New tests for this part specifically.
  (interpretMixedExpr(TT),Some(Right(true)),"Constant
  ↪  expression true"),

  ↪  (interpretMixedExpr(Band(TT,FF)),Some(Right(false)),"And
  ↪  expression true && false"),
  )

runTests(tests)
```