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.

Boilerplate

:TODO: (It will be similar to the homework boilerplate.)

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, $(((1^2 + 2^2) + 3^2) + 4^2)$

Task 1 – Interpreter written in Scala [20 marks]

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`.

Task 2 – Interpreter written in Prolog [20 marks]

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 `interpret(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 `interpret(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.)

Task 3 – Variables and substitution [30 marks]

Task 3 description

In this task, 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 `interpret` 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, is to add a *state* argument to `interpret`, which maps variables to values. We will use states in later assignments for this purpose.)

The task 3

In Scala, create a new algebraic datatype `VarExpr` and a new interpretation method `varInterpret` 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.

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 `varInterpret` which relates expressions to their interpreted value ([15 marks].) (In Prolog, there is no need to use a label for "simple" expressions like we use the `Simple` constructor in Scala.)

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!

Task 4 – Boolean expressions [30 marks]

Task 4 description

In this task, 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 task 4

In Scala, create a new algebraic datatype `TypedExpr` and a new interpretation method `typedInterpret` 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.

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 `mixedInterpret` which relates expressions to their interpreted value ([15 marks].)

Task 5 – Bonus: parsing [10 bonus_marks]

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.

Testing

:TODO: This will be posted ASAP.