Computer Science 3MI3 – 2020 homework 6

Representing and interpreting `Expr` in Ruby

Mark Armstrong

October 23rd, 2020

Contents

Introduction

Recall the `Expr` language of integer expressions using prefix operators from part 1 of assignment 1.

In this homework, we task you with representing this type using an object-oriented approach in Ruby.

Boilerplate

Submission procedures

Submission method

Homework should be submitted to your McMaster CAS Gitlab respository in the `cs3mi3-fall2020` project.

Ensure that you have **pushed** the commits to the remote repository in time for the deadline, and not just committed to your local copy.

Naming requirements

Place all files for the homework inside a folder titled `hn`, where `n` is the number of the homework. So, for homework 1, use the folder `h1`, for homework 2 the folder `h2`, etc. Ensure you do not capitalise the `h`.

Unless otherwise instructed in the homework questions, place all of your code for the homework in a single file in the `hn` folder named `hn.ext`, where `ext` is the appropriate extension for the language used according to this list:

- For Scala, `ext` is `sc`.
- For Prolog, `ext` is `pl`.
- For Ruby, `ext` is `rb`.
- For Clojure, `ext` is `clg`.

If multiple languages are used in the homework, submit a `hn.ext` file for each language.

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

Incorrect naming of files may result in up to a 10% deduction in your grade.

Do not submit testing or diagnostic code

Unless you are instructed to do so in the homework questions, **you should not submit testing code with your homework submission**.

This includes

- any `main` function,
- any `print` statements which output information **that is not directly requested as console output in the homework questions**.

If you do not wish to remove diagnostic print statements manually, you will have to find a way to ensure that they disabled in your final submission.

For instance, by using a wrapper on the print function or macros.

Due date and allowance for technical difficulties

Homework is due on the second Sunday following its release, by the end of the day (midnight). Submissions past 00:00 may not be considered.

If you experience technical difficulties leading up to the submission time, please contact Mark **ASAP** with the details of the problem and, if possible, attach the current state of your homework to the communication. This information will help ensure we are able to accept your submission once the technical difficulties are resolved.

Proper conduct for coursework

Individual work

Unless explicitely stated in the homework questions, all homework in this course is intended to be *individually completed*.

You are welcome to discuss the content of the homework in the public forum of the class Microsoft Teams team homework channel, though obviously solutions or partial solutions should not be posted or described.

Private discussions about the homework cannot reasonably be forbidden, but such discussions should follow the same guidelines as public discussions.

Inappopriate collaboration via private discussions which is later discovered by course staff may be considered academic dishonesty.

When in doubt, make the discussion private, or report its contents to the course staff by making a note of it in your homework.

To clarify what is considered appropriate discussions of homework content, here are some examples:

1. Discussing the language features introduced or needed for the homework.
    - Such as relevant builtin datatypes and datatype definition methods and their general use.
    - Code snippets that are not partial solutions to the homework are welcome and encouraged.
2. Questions of the form "What is meant by `x`?", "Does `x` really mean `y`?" or "Is there a mistake with `x`?"
    - Of course, questions of those form which would be answered by partial solutions are not considered appropriate.
3. Questions or advice about errors that may be encountered.
    - Such as "If you see a `scala.MatchError` you should probably add a catch-all `_` case to your `match` expressions."

Language library resources

Unless explicitely stated in the questions, it is not expected that you will use any language library resources in the homeworks.

Possible exceptions to this rule include implementations of datatypes we discuss in this course, such as lists or options/maybes, if they are included in a standard library instead of being builtin.

Basic operations on such types would also be allowed.

- For instance, `head`, `tail`, `append`, etc. on lists would not require explicit permission to be used.
- More complex operations such as sorting procedures would require permission before you used them.

Additionally, the standard *higher-order* operations including `map`, `reduce`, `flatten`, and `filter` are permitted generally, unless the task is to implement such a higher-order operator.

Part 0: Example written in C++ [0 points]

There are at least two approaches to representing the `Expr` type in an object-oriented style; either by

1. using a single `Expr` class which uses fields to track, for a given object of the class, which kind of expression is the top-level expression, or by
2. using an `Expr` interface or class which has as a *subclass* for each kind of expression.

Approach 1: A single class with fields

We use standard output to print the expressions.

```
#include <stdio.h>
```

There is not a basic exponent operator in C++; we simply define one ourselves.

```
int pow(int x, int y){
  int r = 1;
  while (y-- > 0) {
    r *= x;
  }
  return r;
}
```

Our class contains several types (`enum`'s, `union`'s and `struct`'s) for internal use in tracking the kind of expression used. In Ruby, most of this setup is not necessary due to the dynamic type system. The `UOp` and `BOp` enumerations are outside the class, as they will be used as arguments to the constructors.

```
enum UOp { neg, abs };
enum BOp { plus, times, minus, exp };

class IExpr {

  enum Arity { constant, unary, binary };

  // An operator is either a UOp or a BOp.
  // Do be careful with such an untagged union!
  union Op {UOp uop; BOp bop;};

  // An IExpr2 is a pair of IExpr's.
  struct IExpr2 {IExpr* fst; IExpr* snd;};

  // The arguments are either:
  // - a single integer (in case of constant),
  // - a single IExpr (in case of unary operator), or
  // - two IExpr's (in case of binary operator).
  // Again, be careful with untagged union type!
  union Arg {int value; IExpr* subexpr; IExpr2 subexprs;};
```

Now, each object has an arity, a (top-level) operator, and an argument (which, remember from the definition of `Arg` just above, might be an integer, and `IExpr`, or a pair of `IExpr`'s.)

```
  Arity ar;
  Op op;
  Arg arg;
```

The only methods are the constructors and the `interpret` method.

```
  public:
    IExpr(int c);
    IExpr(UOp op, IExpr* e);
    IExpr(BOp op, IExpr* e1, IExpr* e2);
    int interpret();
};
```

The constructors are fairly uninteresting.

```
IExpr::IExpr(int c) {
  ar = constant;
  arg.value = c;
}

IExpr::IExpr(UOp uop, IExpr* e) {
  ar = unary;
  op.uop = uop;
  arg.subexpr = e;
}

IExpr::IExpr(BOp bop, IExpr* e1, IExpr* e2) {
  ar = binary;
  op.bop = bop;
  arg.subexprs.fst = e1;
  arg.subexprs.snd = e2;
}
```

The interpreter uses `switch` statements on the arity and then on the top-level operator.

```
int IExpr::interpret() {
  switch(ar) {
    case constant : { return arg.value; }

    case unary : {
      // Single subexpression to evaluate
      int r = arg.subexpr->interpret();
      switch(op.uop) {
        case neg : { return - r; }
        case abs : { return r > -r ? r : -r; }
      }
    }
    case binary : {
      // Two subexpressions to evaluate
      int r1 = arg.subexprs.fst->interpret();
      int r2 = arg.subexprs.snd->interpret();
      switch(op.bop) {
        case plus  : { return r1 + r2; }
        case times : { return r1 * r2; }
        case minus : { return r1 - r2; }
        case exp   : { return pow(r1,r2); }
      }
    }
  }
}
```

We include a main method with some example uses of the interpreter. As usual, do not include such code with your submission!

```
int main() {
  IExpr five(5);
  IExpr six(6);
  IExpr e1(neg, &five);
  IExpr e2(exp, &five, &six);

  printf("neg 5 evaluates to %d\n", e1.interpret());
  printf("exp 5 6 evaluates to %d\n", e2.interpret());
  return 0;
}
```

Approach 2: Subclasses for each kind of expression

As above, we use standard input to print out test results, and implement our own exponent method.

```
#include <stdio.h>

int pow(int x, int y){
  int r = 1;
  while (y-- > 0) {
    r *= x;
  }
  return r;
}
```

The `IExpr` (abstract) class is now quite barebones; we specify that it has a `virtual` method `interpret`, which will need to be implemented for each subclass.

```
class IExpr {
  public:
    virtual int interpret() = 0; // The = 0 makes this a pure
    ↪  virtual method.
    // And the presence of a pure virtual method makes this an
    ↪  abstract class.
    // (It cannot be constructed, only used as a superclass.)
};
```

Now, we give a subclass for each kind of expression, implementing constructor and interpret methods as we go. These methods are nice an small, since no logic is really needed; we know what kind of expression we are looking at.

```
class Cons : public IExpr {
    int val;
  public:
    Cons(int c);
    int interpret();
};

Cons::Cons(int c) {
  val = c;
}

int Cons::interpret() {
  return val;
}

class Neg : public IExpr {
    IExpr* subexpr;
  public:
    Neg(IExpr* e);
    int interpret();
};

Neg::Neg(IExpr* e) {
  subexpr = e;
}

int Neg::interpret() {
  return - subexpr->interpret();
}

class Abs : public IExpr {
    IExpr* subexpr;
  public:
    Abs(IExpr* e);
    int interpret();
};

Abs::Abs(IExpr* e) {
  subexpr = e;
}

int Abs::interpret() {
  int r = subexpr->interpret();
  return r > -r ? r : -r;
}

class Plus : public IExpr {
    IExpr* subexpr1;
    IExpr* subexpr2;
  public:
    Plus(IExpr* e1, IExpr* e2);
    int interpret();
};

Plus::Plus(IExpr* e1, IExpr* e2){
  subexpr1 = e1;
  subexpr2 = e2;
}

int Plus::interpret() {
  int r1 = subexpr1->interpret();
  int r2 = subexpr2->interpret();
  return r1 + r2;
```

```
}

class Times : public IExpr {
    IExpr* subexpr1;
    IExpr* subexpr2;
  public:
    Times(IExpr* e1, IExpr* e2);
    int interpret();
};

Times::Times(IExpr* e1, IExpr* e2){
  subexpr1 = e1;
  subexpr2 = e2;
}

int Times::interpret() {
  int r1 = subexpr1->interpret();
  int r2 = subexpr2->interpret();
  return r1 * r2;
}

class Minus : public IExpr {
    IExpr* subexpr1;
    IExpr* subexpr2;
  public:
    Minus(IExpr* e1, IExpr* e2);
    int interpret();
};

Minus::Minus(IExpr* e1, IExpr* e2){
  subexpr1 = e1;
  subexpr2 = e2;
}

int Minus::interpret() {
  int r1 = subexpr1->interpret();
  int r2 = subexpr2->interpret();
  return r1 - r2;
}
```

```
class Exp : public IExpr {
    IExpr* subexpr1;
    IExpr* subexpr2;
  public:
    Exp(IExpr* e1, IExpr* e2);
    int interpret();
};

Exp::Exp(IExpr* e1, IExpr* e2){
  subexpr1 = e1;
  subexpr2 = e2;
}

int Exp::interpret() {
  int r1 = subexpr1->interpret();
  int r2 = subexpr2->interpret();
  return pow(r1, r2);
}
```

And as before, we include a `main` function to test it out.

```
int main() {
  Cons five = Cons(5);
  Cons six = Cons(6);
  Neg e1 = Neg(&five);
  Exp e2 = Exp(&five, &six);

  printf("neg 5 evaluates to %d\n", e1.interpret());
  printf("exp 5 6 evaluates to %d\n", e2.interpret());
  return 0;
}
```

Part 1: A representation and interpreter in Ruby [40 points]

In Ruby, create a class `Expr` to represent the type of integer expressions from part 1 of assignment 1.

In order to ensure that your code is compatible with the tests provided, please also create methods `construct_const`, `construct_neg`, `construct_abs`, `construct_plus`, `construct_times`, `construct_minus`, and `construct_exp` which, given either an integer, `Expr`, or two `Expr` arguments as appropriate, returns the corresponding expression.

(The body of these methods should simply be call to an `Expr` constructor; requiring you implement these simply ensures the tests will be compatible with whatever constructors you have.)

Then write an `interpret` method for `Expr`'s which returns the integer represented by that expression.

Part 2: Add variables and substitution [10 bonus points]

Repeat part 2 of the assignment, adding variables and a substitution operator to the language of expressions `Expr`, in Ruby.

Call your new type `VarExpr`, and submit both it code to test it in a file `h6b.rb`.

Testing

Unit tests for the requested methods/functions are available here. The contents of the unit test file are also repeated below.

The tests can be run by placing the `h6t.rb` file in the same directory as your `h6.rb` file, and running the following command.

```
ruby h6t.rb
```

You may also use `irb h6t.rb`, which will echo the script as it is run.

If you wish to add more tests yourself, see the documentation for Ruby.

Automated testing via Docker

The Docker setup and usage scripts are available at the following links. Their contents are also repeated below.

- Dockerfile
- docker-compose.yml
- setup.sh
- run.sh

Place them into your `h6` directory where your `h6.rb` file and the `h6t.rb` (linked to above) file exist, then run `setup.sh` and `run.sh`.

You may also refer to the README for this testing setup and those files on the course GitHub repo.

Note that the use of the `setup.sh` and `run.sh` scripts assumes that you are in a `bash` like shell; if you are on Windows, and not using WSL or WSL2, you may have to run the commands contained in those scripts manually.

The tests

The contents of the testing script are repeated here.

h6t.rb

```ruby
require_relative "h6"
require "test/unit"

class SimpleTests < Test::Unit::TestCase

  def test_constant_expression_zero
    e = construct_const(0)
    assert_equal(0, e.interpret)
  end

  def test_additive_expression
    e1 = construct_const(5)
    e2 = construct_const(4)
    e = construct_plus(e1,e2)
    assert_equal(9, e.interpret)
  end

  def test_exponential_expression
    e1 = construct_const(5)
    e2 = construct_const(4)
    e = construct_exp(e1,e2)
    assert_equal(625, e.interpret)
  end
end
```

The Docker setup

The contents of the Docker setup files and scripts are repeated here.

Dockerfile

```
FROM ruby:2.7.2-buster

# Set the name of the maintainers
MAINTAINER Habib Ghaffari Hadigheh, Mark Armstrong
↪  <ghaffh1@mcmaster.ca, armstmp@mcmaster.ca>

# Set the working directory
WORKDIR /opt/h6
```

docker-compose.yml

```
version: '2'
services:
  service:
    build: .
    image: 3mi3_h6_docker_image
    volumes:
      - .:/opt/h6
    container_name: 3mi3_h6_container
    command: ['ruby', 'h6t.rb']
```

setup.sh

```
docker-compose build --force-rm
```

run.sh

```
# Run the container
docker-compose up --force-recreate
# Stop the container after finishing the test run
docker-compose stop -t 1
```

README.md

```
# Instructions for automated testing using Docker

We have already created a `Dockerfile` here which specifies
all the necessary packages, etc., for compiling and running
↪  your code.
You only need to follow the instructions below to see
the results of unit tests designed to check your
↪  implementation.
```

~~~
## Setup
We use `docker-compose` and its configuration file to build
 ↪  the image.
Assuming you have `docker` and `docker-compose` installed,
simply execute
```shell script
./setup.sh
```
to generate the image.

## Prepare your code for the running the tests
You only need to place the `h5t.rb` unit test file and
the `run.sh` file in the same directory as your `h5.rb` source
 ↪  file.

## Running the tests
As with the build process, we have already put
the configuration needed for running the test inside
 ↪  `docker-compose.yml`.
Simply execute
```shell script
./run.sh
```
to run and see the results of the tests.
~~~