Advanced Programming 2025

Assignment One

Interpreting Arithmetic Expressions

Due: 2025-9-14

Synopsis: Implementing an interpreter for a small language of arithmetic expressions.

1 Introduction

In this assignment you will be adding various new features to the language implemented during the exercise classes. If you wish, you can base the assignment on your own solutions to the exercises, rather than ours, but only do so if you are quite sure that they are correct. Specifically, you will be implementing the following:

- Support for `for`-loops in the interpreter.
- Support for functions in the interpreter.
- Support for `try-catch` in the interpreter.
- A pretty-printer for printing expressions in conventional (infix) notation.

For all tasks you are expected to add appropriate tests to the files `AST_Tests.hs` and `Eval_Tests.hs`. Do not rename any of the definitions already present in the handout. Do not change the types of any exported functions.

Most of the tasks are intended to test your ability to extend a Haskell datatype and corresponding recursive functions.

2 Tasks

Task 1: Loops

For this task you must augment APL with `for` loops. A `for` loop is represented by the following constructor:

```
data Exp
  = ...
  | ForLoop (VName, Exp) (VName, Exp) Exp
```

An expression

```
ForLoop (p,initial) (i,bound) body
```

is evaluated as follows:

1. Evaluate expression `initial` to a value v.
2. Evaluate expression `bound` to an integer n. It is an error `"Non-integral loop bound"` if `bound` evaluates to a non-integer.
3. Bind variable `i` to the integer 0.
4. Bind variable `p` to v.
5. While `i` $< n$: evaluate expression `body` and bind `p` to the result. Increment `i`.
6. Return the final value of `p`.

Essentially, `for`-loops evaluate the given `body` expression a number of times given by `bound`, each time binding the loop parameter `p` to the result of the body.

Your task: Add the `ForLoop` constructor to `Exp`, implement evaluation of `ForLoop` in `APL.Eval`, add appropriate tests in `APL.Eval_Tests`.

Examples

```
> eval envEmpty (ForLoop ("p", CstInt 0) ("i", CstInt 10)
                         (Add (Var "p") (Var "i")))
Right (ValInt 45)
```

Task 2: Functions

In this task you will implement function expressions, function application, and function values in APL. A lambda expression and function application is represented by the following `Exp` constructors:

```
data Exp
  = ...
  | Lambda VName Exp
  | Apply Exp Exp
```

A function value is represented by the following `Value` constructor:

```
data Val
  = ...
  | ValFun Env VName Exp
```

A `Lambda` comprises a parameter name and a body expression. When evaluated, `Lambda` produces a function value, represented by `ValFun`. Apart from storing the parameter and body, a value function also *captures* the environment at the point at which it was constructed.

An `Apply` applies a function expression to an argument expression. The function expression must evaluate to a `ValFun`. The argument expression can evaluate to an argument value of any type. Application then happens by starting from the environment stored in the `ValFun`, extending it with a binding of the parameter name to the argument value, then using this environment to evaluate the body. The result of evaluating the body is the result of the application. The order of evaluation of the `Apply` subexpressions is not specified.

Your task: Add the necessary constructors to `Exp` and `Val`, implement the appropriate cases in `APL.Eval`, add tests in `APL.Eval_Tests`.

Examples

```
> eval [] (Let "x" (CstInt 2)
                   (Lambda "y" (Add (Var "x") (Var "y"))))
Right (ValFun [("x",ValInt 2)] "y"
              (Add (Var "x") (Var "y")))
> eval [] (Apply (Let "x" (CstInt 2)
```

```
                  (Lambda "y" (Add (Var "x") (Var "y"))))
                 (CstInt 3))
Right (ValInt 5)
```

Task 3: `try-catch`

In this task you are implementing a mechanism for error handling. We will

add the following constructor to `Exp`:

```
data Exp
  = ...
  | TryCatch Exp Exp
```

When evaluating an expression `TryCatch e1 e2`, we start by evaluating `e1`. If `e1` evaluates successfully, then that is the result of the `TryCatch` expression. If `e1` encounters a failure (`Left`), then we evaluate `e2`, and its result is the overall result of evaluation.

Your task: Add the necessary constructors to `Exp` and `Val`, implement `TryCatch` in `APL.Eval`, add tests in `APL.Eval_Tests`.

Examples

```
> eval [] (TryCatch (CstInt 0) (CstInt 1))
Right (ValInt 0)
> eval [] (TryCatch (Var "missing") (CstInt 1))
Right (ValInt 1)
```

Task 4: Pretty-printer

This task tests your ability to write recursive functions from scratch. The function

```
printExp :: Exp -> String
```

produces a representation of the provided expression in Haskell-like notation, including infix notation for binary operators.

Specifically:

- The binary operators are printed as `x + y`, replacing + with the appropriate operator. Exponentiation is written with the ** operator.
- An `If` expression is printed as `if x then y else z`.
- A `Let` expression is printed as `let x = y in z`.
- A `ForLoop (p,initial) (i,bound) body` expression is printed as `loop p = initial for i < bound do body`.
- A `Lambda` expression is printed as `\x -> y`.
- A `Apply` expression is printed as `x y`.
- A `TryCatch` expression is printed as `try x catch y`.
- A `CstInt` is printed as a decimal integer (use `show`).
- A `CstBool` is printed as `true` or `false` (note: lowercase).

Make sure to add parentheses as appropriate. It is acceptable to introduce more parentheses than strictly necessary. In fact, we recommend that you enclose every expression in parentheses, as this is much easier to implement. Parentheses *must* be inserted in the following cases:

- Any argument in an `Apply` must be parenthesized unless that argument is a constant or variable.
- The function part of an `Apply` must be parenthesized unless it is a constant, variable, or another `Apply`.

Your task: Implement the function `printExp` in `APL.AST`, add appropriate tests in `APL.AST_Tests`.

3 Code handout

The code handout consists of the following nontrivial files.

- `a1.cabal`: Cabal build file. **Do not modify this file.**
- `runtests.hs`: Test execution program. **Do not modify this file.**

- `src/APL/AST.hs`: The APL AST definition.
- `src/APL/AST_Tests.hs`: Placeholder for AST tests.
- `src/APL/Eval.hs`: Evaluation functions for APL.
- `src/APL/Eval_Tests.hs`: Evaluation tests. Some tests are already included. You are expected to add more.

4 Your Report

You are expected to comment on the *interesting* details of your implementation. You are *not* expected to give a line-by-line walkthrough of your code. Most importantly, you are expected to reflect on the *quality* of your code:

- Do you think it is functionally correct? Why or why not?
- Is there some improvement you'd have liked to make, but didn't have the time?

It is more important to be aware of the strengths or shortcomings of your solution, than it is to have a complete solution.

4.1 The structure of your report

Your report must be structured exactly as follows:

Introduction: Briefly mention general concerns, any ambiguities in the problem text and how you resolved them, and your own estimation of the quality of your solution. Briefly mention whether your solution is functional, which test cases cover its functionality, which test cases it fails for (if any), and what you think might be wrong.

A section answering the following numbered questions:

1. What is the observable evaluation order for `Apply`, what difference does it make, and which of your test(s) demonstrate this?

2. Would it be a correct implementation of `eval` for `TryCatch` to *first* call `eval` on *both* subexpressions? Why or why not? If not, under which assumptions might it be a correct implementation?
3. Is it possible for your implementation of `eval` to go into an infinite loop for some inputs?

All else being equal, **a short report is a good report**.

5 Deliverables for This Assignment

You must submit the following items:

- A single PDF file, A4 size, no more than 5 pages, describing each item from report section above.
- A single zip/tar.gz file with all code relevant to the implementation, including at least all the files from the handout. For this assignment it is not necessary to add additional files.

Remember to follow the general assignment rules listed on the course homepage.

6 Assessment

You will get written qualitative feedback, and points from zero to four. There are no resubmissions, so please hand in what you managed to do, even if you have not solved the assignment completely.