Advanced Programming 2025

Assignment Three

Parsing Arithmetic Expressions

Due: 2025-9-28

Synopsis: Implementing a parser for a small arithmetic language.

1 Introduction

In this assignment you will be extending the parser developed during the exercises to support all of APL. Specifically you will add these features to the parser in the `Parser` module:

- Function application
- Additional operators (== and **).
- Syntax for `print`, `put`, and `get`.
- Lambdas, `let`-binding, loops, and `try-catch`.

For all tasks you are expected to add appropriate tests to the `Parser_Tests` module. Do not rename any of the definitions already present in the handout, and do not change their types.

The grammar supported by the handout is as follows, without left-factorisation and handling of operator priority. Each task describes an extension of the grammar which you must implement. You may also do other grammar transformations you find necessary.

```
Atom  ::=  var | int | bool | "(" Exp ")"
FExp  ::=  Atom
LExp  ::=  FExp
        |  "if" Exp "then" Exp "else" Exp
Exp   ::=  LExp
        |  Exp "+" Exp
        |  Exp "-" Exp
        |  Exp "*" Exp
        |  Exp "/" Exp
```

The grammar has some redundancies (mainly the FExp nonterminal) when compared to the one developed during the exercises, but it recognises the same language. The redundancies serve to make it easier to extend in the tasks below. You may find it necessary to adjust the code handout to completely match the grammar.

You must refactor the grammar to handle left recursion and operator precedence as appropriate.

2 Tasks

Task 1: Function application

In this task, you must implement function application by juxtaposition, similar to Haskell notation.

We extend the grammar with the following production:

$$\text{FExp} \quad ::= \quad \ldots \mid \text{FExp FExp}$$

Just as in Haskell, function application is left associative and binds tighter than any infix operator (this part is already expressed in the grammar), meaning `a b c` is parsed as `(a b) c`. In the AST, application is represented by the `Apply` constructor.

Examples

```
> parseAPL "" "x y z"
Right (Apply (Apply (Var "x") (Var "y")) (Var "z"))
> parseAPL "" "x(y z)"
Right (Apply (Var "x") (Apply (Var "y") (Var "z")))
> parseAPL "" "x if x then y else z"
Left ...
```

Task 2: Equality and power operators

In this task you must implement two new operators.

We extend the grammar with the following productions:

$$\begin{array}{lll} \text{Exp} & ::= & \ldots \\ & \mid & \text{Exp “==” Exp} \\ & \mid & \text{Exp “**” Exp} \end{array}$$

The disambiguation rules are as follows:

- == is left-associative and has the lowest precedence of all operators.
- ** is right-associative and has the highest precedence of all operators.

```
> parseAPL "" "x*y**z"
Right (Mul (Var "x") (Pow (Var "y") (Var "z")))
> parseAPL "" "x+y==y+x"
Right (Eql (Add (Var "x") (Var "y")) (Add (Var "y") (Var "x")))
```

2.1 Task 3: Printing, putting, and getting

In this task you must implement the `print`, `put`, and `get` operations.

We extend the grammar with the following productions:

```
Exp  ::=  ...
       |  "print" string Atom
       |  "get" Atom
       |  "put" Atom Atom
```

The `string` terminal denotes string literals. These take the form of double quotes enclosing zero or more characters, excluding double quotes. You must implement parsing of these yourself.

Examples

```
> parseAPL "" "put x y"
Right (KvPut (Var "x") (Var "y"))
> parseAPL "" "get x + y"
Right (Add (KvGet (Var "x")) (Var "y"))
> parseAPL "" "getx"
Right (Var "getx")
> parseAPL "" "print \"foo\" x"
Right (Print "foo" (Var "x"))
```

Task 4: Lambdas, let-binding, loops, and try-catch

In this task you implement the final bits of APL syntax. We extend the grammar with the following productions:

```
LExp  ::=  ...
        |  "\" var "->" Exp
        |  "try" Exp "catch" Exp
        |  "let" var "=" Exp "in" Exp
        |  "loop" var "=" Exp "for" var "<" Exp
```

Examples

```
> parseAPL "" "let x = y in z"
Right (Let "x" (Var "y") (Var "z"))
> parseAPL "" "let true = y in z"
Left ...
> parseAPL "" "x let v = 2 in v"
Left ...
```

3 Code handout

The code handout consists of the following nontrivial files.

- `a3.cabal`: Cabal build file. **Do not modify this file.**
- `src/APL/AST.hs`: AST definition. **Do not modify this file.**
- `src/APL/Eval.hs`: An incomplete evaluator corresponding to the solution to the week 2 exercises. Feel free to replace this module with one of your own.
- `src/APL/Parser.hs`: The incomplete parser where you will do most of your work.
- `src/APL/Parser_Tests.hs`: A parser test suite where you will add plentiful tests.
- `src/runtests.hs`: Test runner. **Do not modify this file.**
- `src/apl.hs`: Interpreter program that ties together parser and evaluator for running APL programs from files.

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. Show the final and complete grammar with left-recursion removed and all ambiguities resolved.
2. Why might or might it not be problematic for your implementation that the grammar has operators * and ** where one is a prefix of the other?

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.