Advanced Programming
Home Assignment 3

Carlo Rosso, Chinar Shah

Contents

1 Introduction

This assignment presented a mix of challenges and easier tasks, but this time, we worked more collaboratively to complete it. Initially, we struggled with the first task, which required a lot of time and effort. However, the second task was simpler, as it involved replicating operations, such as division. The final two exercises were more mechanical—once we understood the pattern, it was straightforward to apply. Additionally, we used ChatGPT to help generate extra test cases (the test which has a comment over them are AI generated) and refine the wording of the report. While it didn't write the report for us, it was instrumental in making our text more concise.
Following, I describe how to run the tests:

1. unzipping `a1-handout.zip`;
2. entering the `a1-handout` directory through the terminal;
3. running `cabal test` in the terminal.

Note that you can run `cabal --enable-coverage test` to see the coverage of the codebase and you can also see what lines are not covered by the tests.

2 Task - Function application

During the first task, we encountered several doubts, primarily revolving around the design of the grammar and operator handling. These were the key issues, ranked by difficulty:

1. Defining the grammar and determining the appropriate priority for each operator;
2. Deciding on the precedence rules for the operators;
3. Implementing the parser itself, which posed a considerable challenge, though slightly less complex than the earlier stages.

Our Approach:

- We adopted a test-driven development (TDD) approach, allowing us to validate our implementation step-by-step.
- Before diving into coding, we tried to structure the grammar as proposed in the Course Notes.

Finally, we managed to complete the task and the proposed test cases passed successfully.

3 Task - Equality and power operators

1. We renamed the `pExp<i>` parsers to `pExp<i+1>`;
2. We implemented the `pExp0` parser for handling the Equality operator;
3. We implemented the `pExp3` parser to handle the Power operator, ensuring it had a higher precedence than the basic arithmetic operators.

The `pExp3` parser drew inspiration from the pExp2 parser. We also modified `pExp3` to properly handle right-associativity, as required for the new precedence rule.

4 Task - Printing, putting, and getting

- We faced some difficulty initially in determining where to insert the first parser into the grammar to ensure it worked correctly.

- We found that using the do notation made the code simpler and more readable compared to other styles. For example, compare the following implementations of the `pKvGetExp` parser:

```haskell
pKvGetExp2 :: Parser Exp
pKvGetExp2 = KvGet <$> (lKeyword "get" *> pAtom)

-- vs. do notation

pKvGetExp :: Parser Exp
pKvGetExp = do
  lKeyword "get"
  KvGet <$> pAtom
```

5 Task - Lambdas, let-binding and try-catch

The final task was quite mechanical compared to the previous ones. Once we understood the patterns from earlier tasks, the implementation became straightforward and repetitive in nature.
Additionally, we note that by the definition of a lambda in our implementation, it can only take one argument, while in Haskell, a lambda can take as many arguments as needed.

6 Asnwers to Questions

Show the final and complete grammar with left-recursion removed and all ambiguities resolved.

```
Atom ::= var
       | int
       | bool
       | "(" Exp ")"

LExp ::= "if " Exp "then " Exp "else " Exp  -- If-then-else
        | "let " vname "=" Exp "in " Exp    -- Let binding
        | "\" vname "->" Exp                -- Lambda
        | Atom (Atom)*                      -- Apply (* represents 0
or more)
        | "print " '"' string '"' Atom      -- Print
        | "get " Atom                       -- Get
        | "put " Atom Atom                  -- Put
        | "try " Exp "catch " Exp           -- Try-catch

Exp3 ::= LExp
        | LExp ("**" Exp3)                  -- Power

Exp2 ::= Exp3
        | "*" Exp3 Exp2
        | "/" Exp3 Exp2

Exp1 ::= Exp2
        | "+" Exp2 Exp1
        | "-" Exp2 Exp1

Exp0  ::= Exp1
        | Exp1 "==" Exp0                    -- Equality

Exp   ::= Exp0
```

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?

Since we are using `lString` to parse both the * and ** operators, the only feasible solution is to parse the ** operator first. Otherwise, any occurrence of ** would be incorrectly parsed as a * operator followed by another *. This issue arises because the * operator is a prefix of the ** operator.

One possible solution to this problem would be to use `lKeyword` for parsing both * and **. However, this would impose a restriction where the language would no longer recognize expressions like `4*2` without spaces, and instead, it would only accept `4 * 2`.