Home Assignment 3
Advanced Programming

qhz731

September 29, 2024

1 Introduction

In this assignment, I first rewrote the grammar to eliminate left recursion and ambiguities. `FExp` is rewritten to eliminate left recursion, as well as to keep the left associativity of function application. A hierarchy of `Exp1` to `Exp5` is introduced to handle operators of different precedence levels:

- `Exp1` handles ==
- `Exp2` handles + and -
- `Exp3` handles * and /
- `Exp4` handles **
- `Exp5` includes `LExp` and other special forms (print, get, put)

Each `Exp` level has a corresponding `Exp'` to handle left recursion elimination.

Then I added parsing support for function application, equality and power operators, printing, putting, and getting operations, as well as lambda expressions, let bindings, and try-catch. The design of the parser follows a recursive descent approach. I defined expression parsing functions of different priorities to handle grammatical structures. I paid special attention to the precedence and associativity of operators to ensure that operators such as * and ** can be parsed correctly. I think this parser is of good quality, it could handle basic arithmetic operations correctly, and can also parse more advanced language features. The parser has a clear structure and is easy to understand and extend.

To run the tests, please run the command `cabal test --test-show-details=always` in the command line under the same directory as the a3.cabal.

2 Task: Function application

Since function application has the highest precedence, I put its processing at the bottom of the parser. In my implementation, the parsing of function application happens in the `pFExp` function, which is directly above `pAtom`. This design ensures that function

application is parsed before any infix operators are processed, thus maintaining its high precedence.

The `pFExp` function uses a helper function `pFExp'` that recursively handles the successive function applications, ensuring correct left associativity. My implementation is functionally correct.

3 Task: Equality and power operators

In my implementation, I adjusted the parser structure to reflect the precedence rules. Specifically, the parsing of the == operator is placed in the `pExp1` function, which is processed as the lowest priority operator. The parsing of the ** operator is placed in the `pExp4` function, ensuring that it is processed before all other operators. My implementation should be functionally correct.

4 Task: Printing, putting, and getting

In my implementation, these new operations are placed in the `pExp5` function for parsing. For the string in the print operation, first use `lKeyword` to identify the print keyword, then use `lString` to consume \", then call `lVName` to parse `VName`, and then use `lString` to consume the next \". My implementation should be functionally correct.

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

These new grammar structures are parsed in the `pLExp` function. During the implementation, I paid special attention to the nesting capabilities of these structures. For example, the body of a lambda expression can be an arbitrarily complex expression, including nested let bindings or other lambda expressions. Similarly, the expression in a let binding and the two expression parts in a try-catch can be arbitrarily complex expressions. Overall, this implementation is functionally correct.

6 Question Answers

6.1 Grammar

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

FExp ::= Atom FExp'
FExp' ::= FExp FExp'
        | epsilon  -- epsilon (empty string)

LExp ::= FExp
       | "if" Exp "then" Exp "else" Exp
       | "\" var "->" Exp
       | "try" Exp "catch" Exp
```

```
       | "let" var "=" Exp "in" Exp

Exp ::= Exp1

Exp1 ::= Exp2 Exp1'
Exp1' ::= "==" Exp2 Exp1'
        | epsilon

Exp2 ::= Exp3 Exp2'
Exp2' ::= "+" Exp3 Exp2'
        | "-" Exp3 Exp2'
        | epsilon

Exp3 ::= Exp4 Exp3'
Exp3' ::= "*" Exp4 Exp3'
        | "/" Exp4 Exp3'
        | epsilon

Exp4 ::= Exp5 Exp4'
Exp4' ::= "**" Exp4
        | epsilon

Exp5 ::= LExp
       | "print" string Atom
       | "get" Atom
       | "put" Atom Atom
```

6.2 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?

In my implementation the problem that the presence of operators * and ** where one is a prefix of the other is handled correctly. My parser implements different precedence levels for * and **: * is handled in `pExp3`, ** is handled in `pExp4`, which has higher precedence. In `pExp4`, the parser first tries to match ** before falling back to other options. This ensures that ** is always matched before *.

7 Reference

I used the AI tool Cursor to generate some of the test cases for the Parser_Tests.hs. The generated tests are carefully checked to ensure the reliability.