Assignment 3

Advance Programming

Name: Kazi Sohrab Uddin Titu, Group Number: 58

1 Introduction:

This assignment consists of some features like parser, evaluator, and type checker for various constructs such as arithmetic expressions, function applications, and error handling mechanisms. I estimate the quality of my solution to be quite satisfactory, as the core functionalities work as expected; however, some edge cases and error handling scenarios may not have been fully addressed. I have tried to ensure coverage of different cases. Though there are some edge cases that could be refined further but I tried to cover them up. To run test cases, run cabal test in the terminal.

2 Task: Function Application

Solution Functionality:
The function application was implemented using juxtaposition, similar to Haskell's function application syntax. The Apply constructor is applied to functions in a left associative manner. Due to the left associative nature, expressions like x y z are parsed as Apply (Apply (Var "x") (Var "y")) (Var "z"). I have done this to make sure the application binds tighter than infix operators with logical precedence across operations.

Failures or Limitations:
I have given some test cases with propagating error also like x if x then y else z are rejected, ensuring the parser doesn't confuse function application with conditional expressions.

Potential Issues:
I believe the parsing mechanism could be improved by handling of precedence between function application and other constructs. A potential solution could be doing better differentiate between function applications and expressions that include control flow structures.

3 Task: Equality and Power Operators

Solution Functionality:
I have added support for equality (==) and power (**) operators. The challenge was ensuring correct precedence, especially with the power operator being a subset of multiplication (*). The parser handles expressions like x*y**z correctly, confirming that ** has higher precedence than *. Expressions like x**y**z are also correctly parsed as Pow (Var "x") (Pow (Var "y") (Var "z")), with right associativity for power. I have ensured that ** binds tighter than other arithmetic operators by explicitly managing precedence and associativity in the grammar as power operations need to be evaluated before multiplication or addition.

Failures or Limitations:
I have checked some test cases but I think it could suffer for any misuse of these operators like chaining them incorrectly or missing operands while combining multiple.

Potential Issues:
I think we can suffer issue like ambiguities due to the operator's precedence. As an example, for same expression higher precedence is making them 2 different ways.

4 Task: Printing, Putting and Getting

Solution Functionality:
Here I have implemented the print, put, and get operations to extend the interpreter's capabilities.

- **Print**: The syntax "print" string Atom allows for printing string literals with expressions.
- **Put**: The syntax "put" Atom Atom enables storing values associated with a key in a key-value store which is also an atom.
- **Get**: The syntax "get" Atom retrieves values associated with a specified key.

To parse these constructs, I have defined parsers pPrint, pPut, and pGet, which utilize the existing atom parser pAtom for capturing various types of expressions. The string literals are parsed with the lStringLiteral function, ensuring that the input matches the required format for string terminals. This design aligns with the language's syntax and ensures a coherent approach to expression evaluation.

Failures or Limitations:
The implementation operates correctly for the defined syntax but has limitations. The parser appropriately rejects incorrect forms of the operations like missing arguments for put or print. However, the EvalM monad currently does not manage or track the state of the key-value store.

As a result, while parsing and syntactical validation succeed, the runtime behavior of storing and retrieving values is not functional, which could lead to confusion or errors during program execution.

Potential Issues:
The absence of state management for the key-value store is a potential issue. Without this functionality, the put and get operations do not perform any meaningful interaction with stored values. To improve the implementation, a state management system should be integrated within the EvalM monad to maintain the key-value pairs across evaluations. This change would allow the operations to not only validate syntax but also to provide dynamic storage and retrieval capabilities.

5 Task: Lambdas, Let-Binding, and Try-Catch

Solution Functionality:
The implementation of lambdas, let-bindings, and try-catch constructs was successfully integrated into the APL interpreter. The grammar was extended to include the following productions:

- **Lambdas:** are represented by the syntax "\\" var "->" Exp, allowing the creation of anonymous functions. In the AST, these are represented using the Lambda constructor. For example, parsing the expression "\ x -> x + 1"results in a lambda function that takes x and returns x + 1.
- **Let-Bindings:** allow for defining variables in a scoped manner using the syntax "let" var "=" Exp "in" Exp. The Let constructor handles the binding of a variable to an expression, and its scope is limited to the expression that follows the in keyword. For instance, parsing let x = y in z yields Right (Let "x" (Var "y") (Var "z")), indicating that x is bound to the value of y within the expression z.
- **Try-Catch:** functionality was also implemented, enabling error handling during expression evaluation. The syntax "try" Exp "catch" Exp allows for the execution of an expression with a fallback if an error occurs. This was represented using the TryCatch constructor in the AST.

I have tried to ensure clear syntax representation, maintain scoping rules, and provide structured error handling within the APL interpreter by implementing these lambdas, let binding and try catch.

Failures or Limitations: The implementation works well for valid expressions, but there are certain failure cases. For example, the parser correctly rejects invalid let bindings, such as let true = y in z, which leads to a type error. However, the parser could struggle with nested let bindings or improperly scoped variables, leading to ambiguities in variable resolution. Additionally, while

the try-catch mechanism allows for basic error handling, it does not differentiate between various error types, which could hinder effective debugging.

Potential Issues:

1. **Variable Scope**: The handling of variable scopes in nested let bindings may lead to unintended behaviors due to variable shadowing, where an inner binding might unintentionally obscure an outer binding. This could complicate the evaluation of nested expressions.
2. **Error Handling**: The current implementation of the try-catch mechanism is quite generic and may not provide detailed information on specific errors encountered. Enhancing the error reporting could improve user experience and debugging capabilities.

6 Questions:

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

```
Exp ::=
    | "print" String Atom
    | "get" Atom
    | "put" Atom Atom
    | "\\" Var "->" Exp
    | "try" Exp "catch" Exp
    | "let" Var "=" Exp "in" Exp
    | If Exp "then" Exp "else" Exp
    | Atom FunctionArgs
    | Atom

FunctionArgs ::=
    | Atom FunctionArgs
    | ε  // ε
Atom ::=
    | CstInt
    | CstBool
    | Var
    | "(" Exp ")"
```

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?

While having * and ** where one is a prefix of the other could introduce parsing ambiguities, my implementation mitigates this issue by defining separate parsing rules for different precedence levels and using try to handle backtracking. Therefore, it should not be problematic in my case, as the parser correctly distinguishes between the two operators.