AP Assignment 5: Property-Based Testing

Nikolaj Gade (qhp695), Sebastian Larsen Prehn (vpz655), Alba Dekens (dmb904)

1 Introduction

2 Task 2: A better generator

Our renewed generator uses a function `genVar` to generate variables between 2 and 4 characters, which also aren't keywords. `genVar` is used in the generation of `Let`, `Lambda`, and `Var`.

We also included a parameter `vars` of type `[VName]` in the `genExp` function. This keeps track of valid variables in the scope. When generating `Let` or `Lambda` statements, the body of those statements is generated using an updated `vars`, which includes the new variable name. We also made a new `Var` generator, which only uses variable names in scope.

3 Task 3: A property for parsing/printing

For our implementation of `parsePrinted`, we check that printing an expression and parsing the resulting string gives back the original expression by simply comparing the two. This resulted in us finding three errors that are described more thoroughly in the Questions section's question 2, but involved us having to use the existing `parens` function to make sure that context was more easily discerned. We also implemented a `genInt`, that only generated non-negative integers, which seems to be the intent of the language given week 3's description of the language grammar.

4 Task 4: A property for checking/evaluating

Our implementation uses a simple `case` statement on the results of evaluating the expression. If the evaluation returns an error, the error is checked again the result of `checkExp`.

The test fails on expressions with the form `(Apply (TryCatch (Lambda VAR X) (Y)) (Z))`, where X has some error. This is discussed further in question 3.

5 Questions

1. The generator cannot loop indefinitely, because of the `size` parameter. The `size` parameter of a `genExp` call is *always* less than the `size` parameter of the parent call, which means it will eventually reach 0, which can only result in a terminal.

2. We got three counter-examples in total. The first counter-example was related to negative numbers in the generator, and we fixed this by implementing `genInt` in the generator which only generates non-negative numbers. As explained in Section 3, we assume this to be the intent of the language.

   The second issue stemmed from `apply`, where the implementation of `printExp` resulted in the second parameter of `apply` where it was not clear which order it should be resolved in. The function was modified to use parentheses via the existing function `parens` to make it clear in which order the function should be resolved by putting parentheses around the whole `apply` statement.

   The final issue arose when you had an expression and a `TryCatch`, where if there was a binary operator where `TryCatch` was the first parameter, then it printed it as if `TryCatch` was the outermost expression. Our solution here was to add the existing `parens` function to put parentheses around the whole function.

3. The mistaken assumption in `checkExp` is that the "try" part of a `TryCatch` expression should not be checked, since an error there will be ignored in favor of the "catch" part. This is untrue in the specific case where the "try" part is a `Lambda` expression with an error in its body. The `Lambda` will without error create the `ValFun`, even if the body of the function has an error. This means that the "try" will exit successfully, returning a "toxic" `ValFun` value. An `Apply` expression can then use that `ValFun`, which then returns the error. In this case, `CheckExp` would not have found the error.