Assignment 3: Implementation of the *supersub* language interpreter

1 Assignment Overview

In this assignment we implement the sematics of the `supersub` language. The semantics involves implementing transformations to the parse trees that were generated in the previous assignment. This assigment will have very subtle implementation points. Read the description of this assignment carefully. This project is due on Thursday, **March 23, 2017 before 3:30pm** (Before class).

2 The `set` Command

The set command in this language is provided **<M>** rule in the grammar. This command provides a mechanism to control various aspects of how the interpreter parses commands. The user may set any variable, but for the present implementation only the three values that you are required to implement are shown in table 1. The meaning of these variables will become clear in the subsequent descriptions.

Table 1: Variables for Set Command

Variable	Default	Explanation
`maxEvalSteps`	10000	Sets maximum number of substitution steps in evaluation
`preOrderEvaluate`	1	Nonzero means use pre-order evaluation, otherwise post-order
`printLevel`	1	Nonzero means print each step during evaluation

3 Substitution Semantics

In the *supersub* language the slash-dot operator describes a substitution operation. The space is used to describe an application of a substitution. Thus a doubling substitution rule applied to "a" would give

```
evaluate ({\x. (x x)} A) ;
```

gives the output `Evaluates to: (A A)`. The `A` is substituted for `x` in the slash-dot expression. Note that the expression tree for this circumstance is shown in figure 1. The substitution in this case is simple, the expression following the "." is modified whereby the `x` is replaced by `A`. The resulting expression is returned.

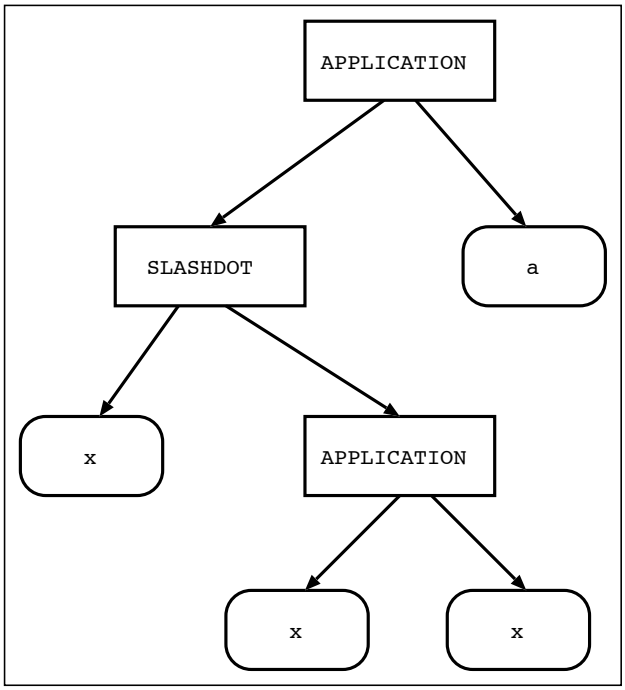

Figure 1: Parse Tree for Substitution Action

For more complex expressions, care must be taken to perform the substitutions correctly. This will be subsequently described.

3.1 Bound Variables

The name the precedes the "." in the slashdot operator is the bound variable name of the expression that follows. In the previous example, `x` was the bound variable in the expression and the `A` is the unbound variable. Note that the name of the bound variable is not significant to the meaning of the substitution rule, it is instead a placeholder used to describe the substitution. Thus {`\x.(x x)`} is the same as {`\y.(y y)`}. Variables that are not tied to any slashdot operator are called free variables. One potential concern that we have for our substitution language is that it can be possible for free variables to "capture" bound variables. For example consider the expression:

```
evaluate ({\x. {\y.(x y)}} y) ;
```

In this case it would seem that it would be valid to just substitute the `x` in the for `y` resulting in the expression {`\y. (y y)`} . However, this confuses the meaning of the substitution since the first `y` is a bound variable while the second `y` is a free variable. The derived substitution rule does not capture the intended meaning, instead the substitution should work the same regardless of the name given to the bound variable. Thus the evaluation should evaluate to something like {`\z. (y z)`} . A variable that is not bound is called a free variable. Note, that a variable may be a free variable in the context of one slashdot expression, but bound

by a slashdot expression higher in the tree. In order for substitutions to be performed in a consistent way, any bound-free variable conflicts will need to be resolved by renaming the bound variables (such as renaming `y` to `z` in the example above. The simplest way to accomplish this is to always rename variables such that each bound variable has a unique name. I am providing a helper utility that can be used to help perform this renaming facility. It is called:

```
expressionP canonicalRenameBoundVars(expressionP e, int &count) ;
```

This function will change all unbound variables in expression e to a variable that starts with an uppercase and change all bound variables to a unique lower case name. A count assigned to zero is passed into the function, and it returns with an expression where all the bound variables are unique names, all the unbound variables are changed to uppercase, and the count is incremented by the number of unbound variables. A routine like this one will be needed in order to ensure that in substitutions we do not encounter problems of variable "capture" described earlier.

3.2 Combinators

A combinator is a special type of substitution rule that contains no free variables. Thus {`\x.\y.  x x y`} is an example of a combinator, while {`\x.\y.  x x z`} is not because it contains the free variable `z`. When a combinator command is executed it should first perform a canonical rename of the bound variables, and then check to see if any of the variables are unbound. If they are, then a warning should be issued that it is not a combinator. If the expression is a valid combinator, then the expression should be installed into the combinator dictionary.

If the `evaluate` tag is provided then the expression will have the substitutions evaluated using a method described in the next subsection.

3.3 Evaluation

The evaluation of the expressions in the *supersub* language will require searching through the parse tree to find APPLICATION nodes where the left branch of the operator is either a COMBINATOR variable or a SLASHDOT operator. In some cases there will be multiple APPLICATION nodes that fit match this description. In this case there will be several different possible orders in which the substitutions can be carried out. In this project we will implement two possible orderings that will be further discussed once we describe how to resolve an APPLICATION node that meets substitution pattern.

If the APPLICATION node has a COMBINATOR variable on the left side then this variable can be replaced with the expression from the combinator dictionary for that variable. If the variable cannot be found then this should be reported as an error. Note, when making the substitution, a copy of the expression that the combinator is obtained using the `getItem(name)` method. The variables in the copied expression are then renamed to unique variable names and then the left branch of the APPLICATION node is updated using the `updateLeft` method. This method will delete the expression currently associated with the left branch and replace it with the suitably adjusted copy from the combinator dictionary. This operation is illustrated in figure 2.

If the APPLICATION node has a SLASHDOT operator on the left hand side, then the operation is a bit more involved. The APPLICATION node itself will be replaced with the right branch of the SLASHDOT node where any variable that matches the name given in the name of the SLASHDOT node will be replaced with copies of the right branch of the APPLICATION node. See figure 3 for an example of how this works when executed on the shaded APPLICATION node. Note that when making copies of the expression from

the right branch of the APPLICATION node, it is important to rename all bound variables of the expression to unique names to avoid variable capture problems.

The evaluation will be implemented as a series of steps. In each step an APPLICATION node will be identified for substitution (either the combinator, or slashdot operators). The evaluation will continue until either a maximum number of steps is reached or until no more APPLICATION nodes can be identified. In the latter case, then expression will be completely evaluated into what is known as a normal form. The search for the next APPLICATION node to execute will occur using either a pre-order or a post-order search of the parse tree. In the pre-order search the tree will be searched recursively. When it finds an APPLICATION node it will test the left branch to see if it is either a SLASHDOT or COMBINATOR case. If it is not then it will continue to recurse the left side first for a suitable APPLICATION node, and if not found on the left branch it will continue to the right branch. The first time an APPLICATION node is found that is suitable for substitution, the substitution will be performed and the resulting tree returned to the evaluate iteration where the current step may be optionally printed. In a post-order search the left and right branches of each APPLICATION will be searched first. If either branch finds the APPLICATION node then the substitution for the current iteration will be found and the process will return back to the evaluation iteration. However, if not, then the current APPLICATION node will be checked to see if it is suitable for substitution. The user will select which type of evaluation will be used using the set variable facility. Examples of the first substitution for the pre-order search is shown in figure 3, while the post order is shown in figure 4.

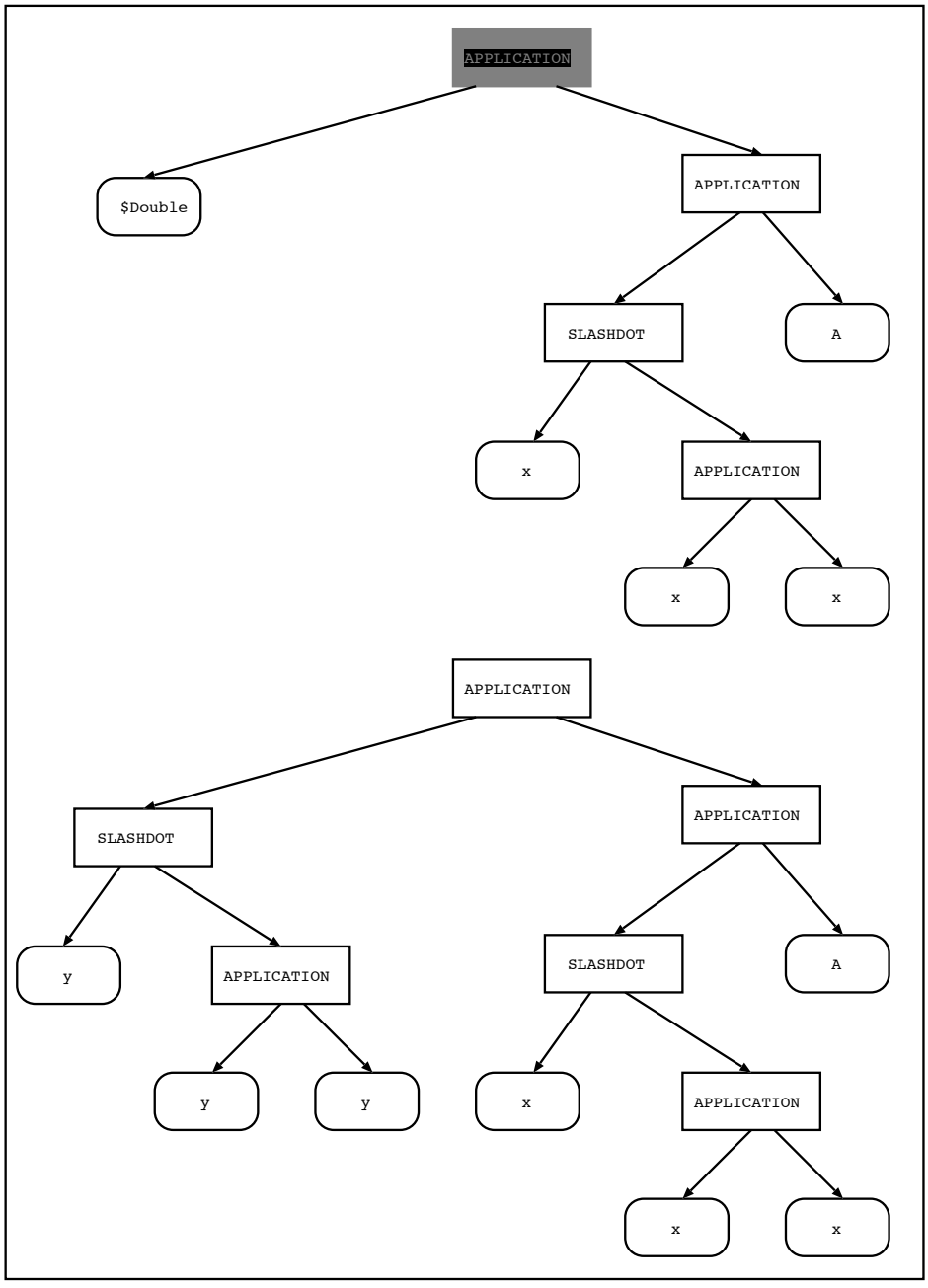

Figure 2: Example of combinator variable substitution

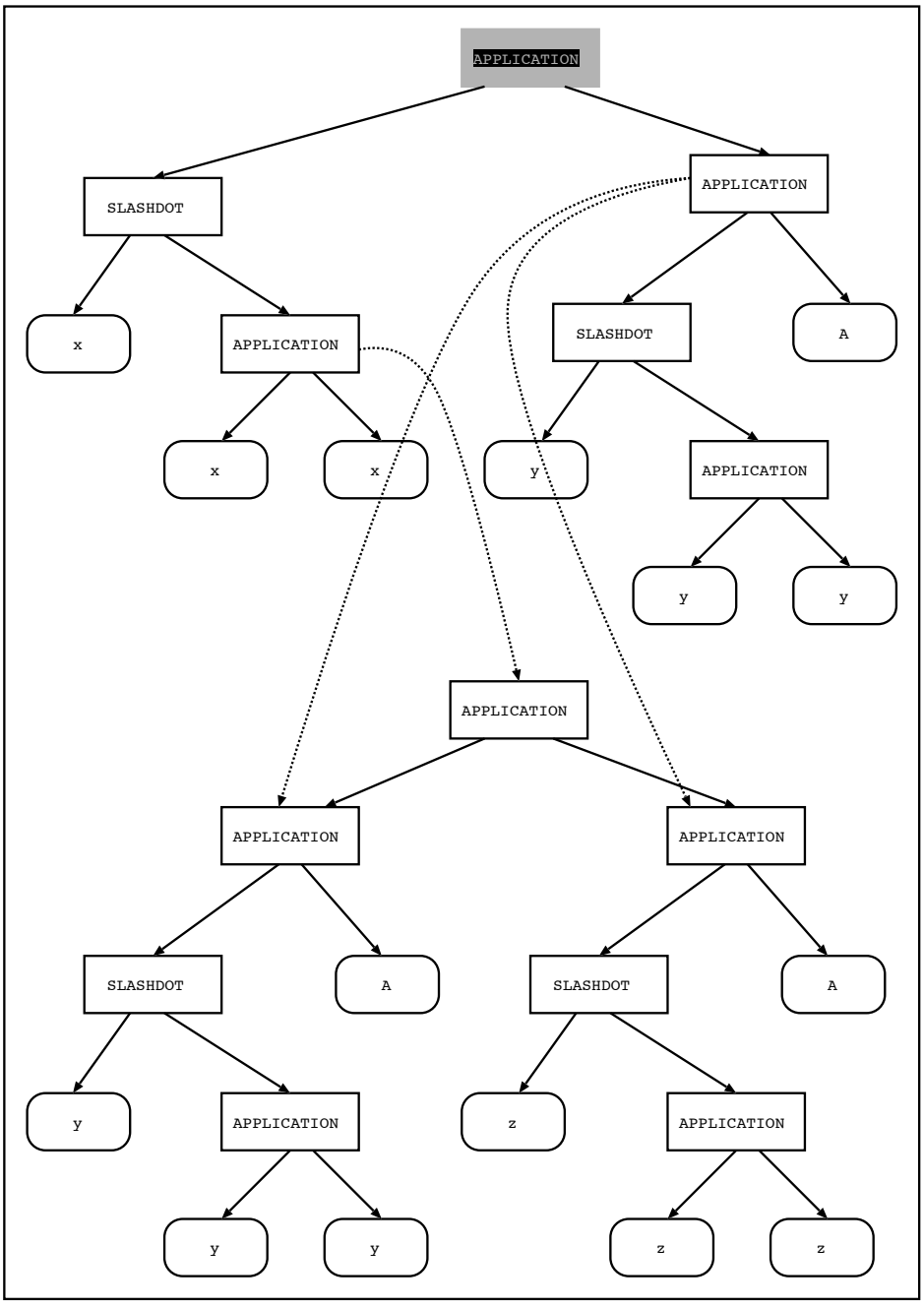

Figure 3: Example of first step of pre-order substitution.

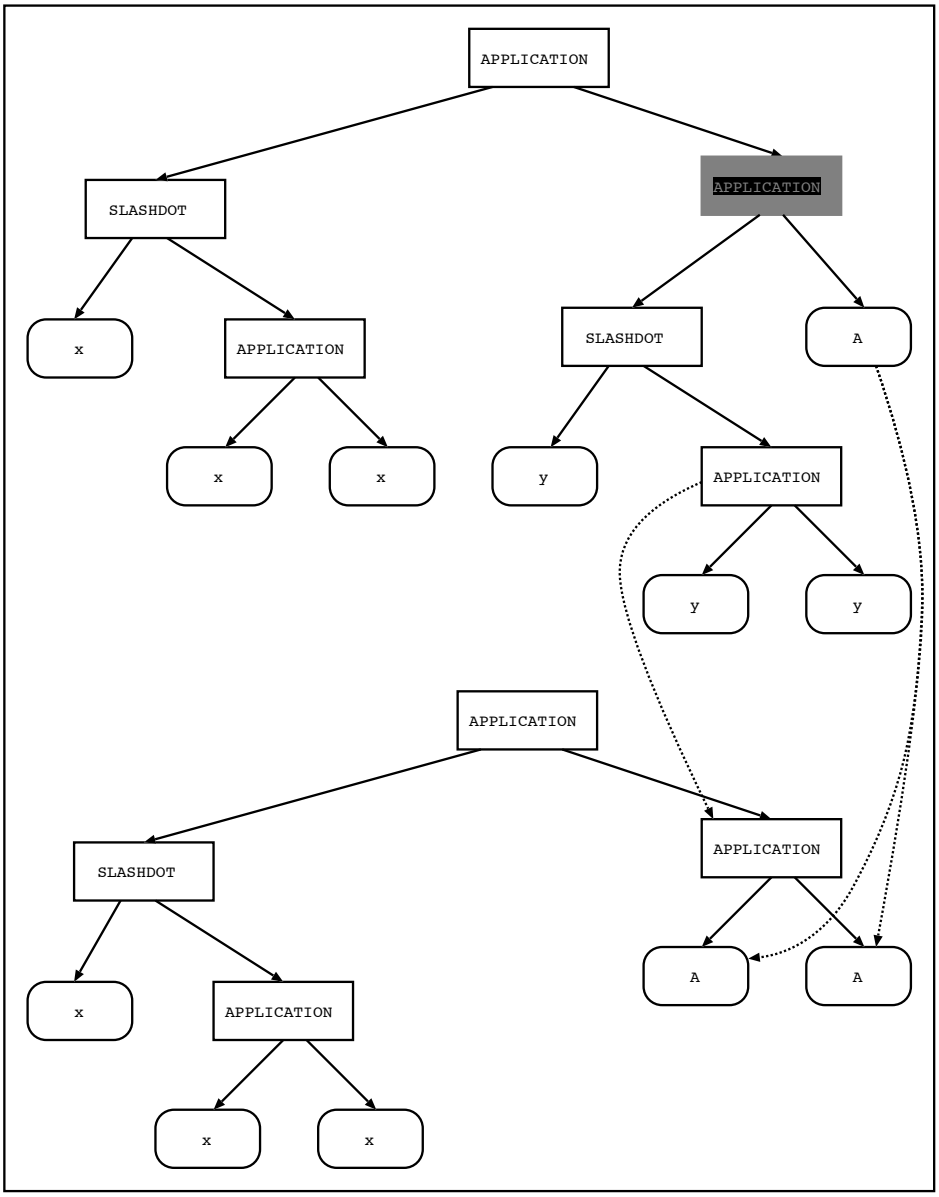

Figure 4: Example of first step of post-order substitution.

4 The `combinator` Command

The `combinator` command is extended somewhat from the first assignment. First the parse tree that is constructed will be modified by the `canonicalRenameBoundVars()` function with an initial count of `0` passed in as the second argument. This function will rename all bound variables to a unique name and change all unbound variables so that they start with an upper case character. Then the expression will be scanned for any unbound variables. If the expression has any unbound variables then it is not a combinator. A warning will be printed and the expression will not be registered to the combinator dictionary.

In the case that the `combinator` command is followed by `evaluate` then the expression will be evaluated using the pre-order substitution form of evaluate. This will be silent evaluation (e.g. no screen output). The evaluated expression will be converted to standard form using the `canonicalRenameBoundVars()` function before placing it in the dictionary.

5 The `evaluate` Command

The evaluate command will perform substitutions on the parse tree as described in the earlier sections. If `preOrderEvaluate` is set to a non-zero value then the pre-order search will be used to identify the next substitution in the expression, otherwise the post-order search will be used. When performing the evaluations, the number of evaluation steps will limited to `maxEvalSteps`. Finally, if `printLevel` is non-zero, then each step of the evaluation process will print the step number followed by the expression after the step has completed. Examples will be provided of evaluations of various expressions of increasing complexity to provide examples of how the evaluations should proceed.

6 The `dictionary` Command

The dictionary command just prints out the dictionary of combinators that were defined. This command should already be implemented from assignment 2.

7 Implementation Notes

A example `interpreter.h` file is provided with this assignment. This example gives the prototypes for the types of functions that you might implement to complete the assignment. Some example code is given that recursively renames bound variables to unique names. This code will be useful for your goals. Ultimately the functions you will need to create to perform substitutions will also need to be constructed using recursive evaluation with a structure similar to the code provided here. So examine the code in `interpreter.cpp` to get ideas about how to structure your code.

There are several idioms used in this code that will make memory management easier for you. First note that the recursive routines return an `expressionP` type. These functions can be though of as consuming the input `expressionP` and returning a new version. Sometimes the new version is simply the same pointer that was passed in, othertimes it may change. If it changes, then the `updateLeft` and `updateRight` member functions will delete the old version and replace the pointer with the new version. See the example below for a basic template. Note that when the pointer is deleted, the delete method will delete all lower level nodes of the tree, not just the node that the pointer refers. You will find that using this same idiom will be helpful, for example, when replacing an apply node with an updated substituted expression. Simply returning with

the new expression will cause the tree at the appropriate level to be updated. Examine the provided code carefully before starting with the assignment.

```
// Recusively rename variables as long as they are not currently bound by
// slashdot operator
expressionP recursiveRenameVars(expressionP e, const string &name,
     const string &toname) {
  switch(e->Type) {
  case Expression::VARIABLE:
    // If it is a variable and the name matches, do rename
    if(e->name == name)
      e->name = toname ;
  case Expression::COMBINATOR:
    // Ignore combinator variables
    return e ;
  case Expression::SLASHDOT:
    // If operator not reusing name, continue to rename right branch
    if(e->name != name)
      e->updateRight(recursiveRenameVars(e->right,name,toname)) ;
    return e ;
  case Expression::APPLICATION:
    // rename both sides of application
    e->updateLeft(recursiveRenameVars(e->left,name,toname)) ;
    e->updateRight(recursiveRenameVars(e->right,name,toname)) ;
    return e ;
  default:
    throw "Unknown expression type in recursiveRenameVars()" ;
  }
}
```

8 Assignment Evaluation

Two example files and the output from the reference implementation are provided for you to test your program. The first example is `simple.in` which gives some examples of evaluation of substitution expressions of increasing complexity. The example also shows the difference between post-order and pre-order evaluation. The output from the reference implementation is provided in the file `sample.out`. A more comprehensive test input is given in the file `test.in`. A working program should be able to solve in the same number of steps and result in the same final expression (which is printed after using the `canonicalRenameBoundVars()` function. Once your assignment can pass these test inputs with acceptable output, then submit it to the grader for further testing.

Selected Simple Examples:

A two substitution rule evaluation. Substitutions are sequential so pre-order and post-order are the same.

```
evaluate (\x.\y. x (x y)) A B ;
  evaluatePreOrder with expression: (({\x.{\y.(x (x y))}} A) B)
  0--({\y.(A (A y))} B)
```

```
  1--(A (A B))
```

A more complex substitution rule that has a different evaluation flow when executing pre-order or post-order substitution:

```
evaluate (\x.\y. x (x (x y))) (\x. x x) A ;
  evaluatePreOrder with expression: (({\x.{\y.(x (x (x y)))}} {\z.(z z)}) A)
  0--({\y.({\z.(z z)} ({\a.(a a)} ({\b.(b b)} y)))} A)
  1--({\z.(z z)} ({\a.(a a)} ({\b.(b b)} A)))
  2--(({\a.(a a)} ({\b.(b b)} A)) ({\c.(c c)} ({\d.(d d)} A)))
  3--((({\b.(b b)} A) ({\e.(e e)} A)) ({\c.(c c)} ({\d.(d d)} A)))
  4--(((A A) ({\e.(e e)} A)) ({\c.(c c)} ({\d.(d d)} A)))
  5--(((A A) (A A)) ({\c.(c c)} ({\d.(d d)} A)))
  6--(((A A) (A A)) (({\d.(d d)} A) ({\f.(f f)} A)))
  7--(((A A) (A A)) ((A A) ({\f.(f f)} A)))
  8--(((A A) (A A)) ((A A) (A A)))
```

Now the post-order evaluation uses fewer steps, this is because the doubling of A is done at the beginning and then reused in the following steps, whereas in the preorder, extra steps need to be performed for each perform the copied doubling rule.

```
evaluate (\x.\y. x (x (x y))) (\x. x x) A ;
  evaluatePostOrder with expression: (({\x.{\y.(x (x (x y)))}} {\z.(z z)}) A)
  0--({\y.({\z.(z z)} ({\a.(a a)} ({\b.(b b)} y)))} A)
  1--({\y.({\z.(z z)} ({\a.(a a)} (y y)))} A)
  2--({\y.({\z.(z z)} ((y y) (y y)))} A)
  3--({\y.(((y y) (y y)) ((y y) (y y)))} A)
  4--(((A A) (A A)) ((A A) (A A)))
```

Use these simple examples to get the basic code working before attempting to handle the more complex `test.in` test case.