Programming Project 3: Lazy Evaluation and Recursive Definitions

Points: 100

Provided Code

The class solution to Project 2 appears in the support code for Project 3 on GitHub Classroom in a folder named **`Solution to Project 2`**. The folder **`src`** folder contains a collection of files taken from the class solution to **`Project 3`** similar in scope to those provided from Project 2. You only need to write the code for the interpreter file **`Interpreter.java`** and a file called **`Syntax.java`** supporting syntax checking after the program has been parsed but before it is executed. You do not need to create any additional files but you have that option if you think is enhances program organization or clarity. You must support the same testing API required to run the provided Junit4 tests and you need to write additional tests (either by adding tests to **`Assign3Test.java`** or by creating additional test files with names ending in **`Test`**.

Overview

Task Your task is to extend and modify your interpreter from Assignment 2 or one of the solutions for Assignment 2 to:

- perform the context-sensitive checking required to confirm that the input program is a legal program;
- enforce safety in Jam programs by detecting illegal operations before they are executed triggering a Java run-time error. (No meta errors!)
- change the semantics of the **`let`** construct to support recursive bindings ("recursive let"); and
- and support the optional lazy evaluation of the **`cons`** constructor using both the call-by-name and the call-by-need implementation of lazy evaluation.

The details of the each of these extensions is described in the following paragraphs. Some of the work has already been done in the support code. To complete your Project 3 interpreter, you do not need to modify any of the provided support files except **`Interp.java`** and **`Syntax.java`**. You also either need to create a new test file containing your own tests or add the requisite tests to the provided file **`Assign3Test.java.`**

Context-Sensitive Checking The context sensitive constraints on Jam syntax are most easily by performing a separate "checking pass" over the abstract syntax tree right after it has been constructed. There are two context-sensitive conditions that Jam programs must satisfy to be well-formed:

- There are no free variables.
- The same variable name cannot appear twice in the same parameter list (of a **`map`**) or the same collection of let names (introduced in a **`let`** binding).

If you find either of these errors, throw a **`SyntaxException`** with a descriptive error message and abort execution.

The context-sensitive checker should be run before the expression is evaluated. Note that the context-sensitive checker does not care which unary or binary operator was used, or how many arguments were passed to a primitive function. A stub file for writing this checker as a visitor is provided in the file **`Syntax.java`**.

Safety Make your interpreter responsible to catching all Jam run-time errors. Do not rely on the run-time checking performed in executing Java code (meta-errors). All Jam run-time error messages should be of type **`EvalException`** generated directly by your interpreter.

Note: It is not enough to just wrap the entire interpreter in a **`try { ... } catch(Throwable t) { ... }`** block. The exception must be thrown when the Jam run-time error occurs.

Safety checking obviously includes confirming that operators are applied to arguments of the proper type. Hence, the unary operators **`+`** and **`-`** and the binary operators **`+`**, **`-`**, **`/`**, **`*`**, **`<`**, **`<=`**, **`>`** and **`>=`**, may only be applied to integer values. Similarly, the unary operator **`~`** and binary operators **`&`** and **`|`** may only be applied to boolean values. In general, you must check that every language construct is applied to the correct number and type of values including checking that the test value in a Jam **`if`** is a boolean value. Many of these checks were already stipulated as part of the semantics of Jam constructs given in Assignment 2. In no case should your interpreters generate run-time Java errors other than insufficient machine resource errors such as StackOverflow or OutOfMemory (heap overflow) errors, which are *not* classified as meta-errors. Of course, you are welcome to catch the corresponding exceptions and report your own errors.

Lazy cons For all three of your interpreters from Project 2, you will add two lazy **`cons`** evaluation options that defer the evaluation of both arguments of a **`cons`** construction. The data object representing the deferred evaluation is called a *suspension*. Given a **`cons`** node containing suspensions, the **`first`** and **`rest`**

operations evaluate the corresponding argument expressions in the environment where the expressions would have been evaluated without laziness.

Lazy **`cons`** obeys the following equations:

```
first(cons(M,N)) = M
rest(cons(M,N))  = if list?(N) then N else error
```

for all Jam expressions **`M`** and **`N`**, including expressions that diverge or generate run-time errors. Recall that **`list?`** is simply the disjunction of **`null?`** and **`cons?`**.

The lazy versions of **`cons`** postpone evalutaion of argument expressions until they are demanded by a **`first`** or **`rest`** operation. You can use exactly the same suspension mechanism to support call-by-name and call-by-need lazy **`cons`** that you might have used to support call-by-name and call-by-need bindings in the previous assignment. In this case, the suspensions are stored in a **`cons`** structure rather than a binding structure the environment. An embedded suspension is not evaluated until it is "probed" by a **`first`** or **`rest`** operation. Call-by-name lazy evaluation re-evaluates suspensions every time that they are probed. Call-by-need lazy evaluation evaluates a given suspension only once and caches the resulting value in the data constructor (**`cons`** cell in the case of Jam).

Please change the function that converts Jam values to a string representation (**`toString`**() and helpers) so that lists are still displayed as lists in the form **`(1 2 3)`** as before, regardless of the cons mode used. You may want to abort processing lists longer than, say, 1000 elements and have them displayed ending in **`998 999 1000 ...)`**. However, we will not test this behavior. We will just require that lists up to 200 elements are displayed the same way as in the last assignment.

Recursive Let The recursive version of the **`let`** construct introduces a collection of recursive bindings just as we discussed in class. We will not restrict the right hand sides of recursive **`let`** constructions to maps because many useful recursive definitions in Jam with lazy cons violate this restriction. Hence, the implementation of recursive **`let`** closely follows the implementation of the generalized version of **`rec-let`** construct in LC.

The only difference is that Jam recursive **`let`** may introduce several mutually recursive definitions where the corresponding closures all close over the new environment. Since we want invalid forward references in a Jam program to produce a meaningful error message rather than diverging, we will use an explicit list to represent the environment of bindings and destructively update it to create the self-references.

In a call-by-value application, we will initially bind each variable introduced in a **`let`** to a special "undefined" value (e.g., Java **`null`**) that is not a legal Jam value representation and will abort the computation if it is used as the input to any Jam operation. The interpreter evaluates the right-hand sides of the definitions in a recursive **`let`** in left-to-right order and destructively modifies each new binding in the environment as soon as the right-hand side has been determined.

In a call-by-name application, we bind each variable to a suspension (a closure of no arguments or *thunk*) for each right-hand side. Since a suspension wraps the right hand side of each **`let`** binding inside a lambda, the recursive environment can be constructed directly using a recursive binding construction (**`letrec`** or **`define`** in Scheme). No mutation is required in languages like Scheme and ML that support general recursive binding. In languages like Java that do not support general recursive binding, we use mutable binding cells instead.

Note that the validity of definitions in a recursive **`let`** depends on which semantics is used for Jam. Definitions that are valid for call-by-name or call-by-need lazy evaluation may be invalid in other evaluation modes. By this measure, lazy evaluation is better than conventional ("eager") evaluation and call-by-name/call-by-need is better than call-by-value.

Testing You are expected to write unit tests for all of your non-trivial methods or functions *and submit your test code as part of your program*. For more information, refer back to Assignment 1.

In Assignment 2, the Interpreter supported three evaluation methods

```
callByValue()
callByName()
callByNeed()
```

In this assigment, the Interpreter class must support nine evaluation methods:

```
valueValue()
nameValue()
needValue()
valueName()
nameName()
needName()
valueNeed()
nameNeed()
needNeed()
```

where the first word in the method name specifies the policy for evaluating program-defined procedure applications (Jam maps) and the second word specifies the policy for evaluating applications of the data constructor **`cons`**.

Hence the first three methods in the preceding list correspond to the three evaluation methods in Assignment 2.

As in Assignment 2, the **`Interpreter`** class must support two constructors: **`Interpreter(String filename)`** which takes input from the specified file and **`Interpreter(Reader inputStream)`** which takes input from the specified Reader. A given **`Interpreter`** object should support the repeated invocation of any of the nine public evaluator methods. As a result, the same **`Interpreter`** object can be used to test all forms of evaluation (as long as none of the evaluations diverges).

In summary:

```
/** file Interpreter.java **/
class EvalException extends RuntimeException {
    EvalException(String msg) { super(msg); }
}

class SyntaxException extends RuntimeException {
    SyntaxException(String msg) { super(msg); }
}

class Interpreter {
   Interpreter(String fileName) throws IOException;
   Interpreter(Reader reader);

   public JamVal valueValue();
   public JamVal nameValue();
   public JamVal needValue();

   public JamVal valueName();
   public JamVal nameName();
   public JamVal needName();

   public JamVal valueNeed();
   public JamVal nameNeed();
   public JamVal needNeed();
}
```

Lazy List Equality

Extra Credit (10 points)

There are two possible strategies for checking equality of lazy lists that are mathematically sound. The simplest approach is to force evaluation of both input list, which will diverge if either list is infinite, and compare the two finite lists using standard structural equality of lists. This approach is not very attractive because it cannot cope with infinite lists at all. The second possibility is to incrementally perform structural equality checking on the lists, progressively checking deeper elements until there is an inequality between corresponding list prefixes or until both lists finitely terminate (in which case they are equal or disagree on the final tails of both lists. The second option is the one that is generally accepted among the advocates of functional programming. It readily generalizes to lazy trees if we incrementally evaluate the unevaluated fringes of both tree inputs until an inequality is discovered or the trees are determined to be finite and equal.

The code for the second option is essentially the same as the usual structural equality operations on lists and trees. In the case of trees, the ordering of the evaluation of elements on the fringe matters. The natural choice is a breadth first traversal proceeding from left-to-right across each fringe of unevaluated tree nodes. In describing this process are assuming that all of the tree constructors are lazy and that the incremental evaluation of a "node" suspends as soon as it returns some construction.

Consider the following example in which a finite list is compared with an infinite one

```
let xs := cons(0, xs);
    ys := cons(0, cons(0, empty));
in xs = ys
```

The **=** operation first compares the outermost constructors (which may require evaluating code) which are identical, and proceeds to compare the first elements of the two lists which are both **0**, and finally comparing the tails returned by applying the **rest** operation to the two inputs, which yields two conflicting constructions. The outermost constructor of the **rest** of **xs** is **cons** with undetermined arguments while the outermost constructor of **ys** is **empty**. So the **=** operation returns **false**.

If at least one of the input lists is infinite, the only possible results are **false** and divergence. Consider the trivial example

```
let ones := cons(1, ones);
in ones = ones
```

The **=** operation determines that the outermost constructors of both lists are identical and proceeds to compare the **rest** fields of each list which are identical to the original inputs. The operation cannot return true (by noticing that both arguments have identical representations because the operation must behave the same way for all expressions that generate the same infinite list as the definition of **ones**, so it must diverge by performing an infinite recursion on the structure of **ones**. This behavior is easy to understand in the context of domain theory. Every infinite list is the LUB of an ascending chain of finite approximations each ending with the tail element ⊥. Every computable function is continuous. If the **=** returns **true** for two infinite lists x and **y**, then there must be finite approximations to **x** and **y** that yield the same result, but these finite approximations are partial (have a tail that is ⊥) with more than one possible completion so they cannot be equal without violating monotonicity. Comparing infinite lists with the = operator is problematic.

On the other hand, lazily constructed finite lists (that are not partial) behave just like there eager counterparts in applications of the **=** operator as shown the following example:

```
let xs := cons(1, empty);
    ys := cons(1, empty);
in xs = ys
```

returns **true**. This behavior on finite lazy lists is important and is required (rather than extra credit) in all solutions to this assignment.

The support code contains a definition of **equals** that works for all finite instances of **PureList<T>** type. Does it work for the infinite lists that are instances of the lazy extensions of the **JamCons<T>** class? Or do you need to override it? You can earn 10 points extra credit if your interpreter correctly implements **=** operation on lazy lists. The test class **Assign3Test** (as updated on Tuesday, February 15) contains two simple test for the behavior of the **=** operator on infinite lazy lists. One of these tests exhausts the call stack for the main thread, so you may want to use the default stack size when running

This extra credit functionality (correct implementation of = on infinite lists) will not be assumed in any future assignments. Hence, if you choose not to forego the extra credit for implementing **=** correctly on infinite lists, there is no need to worry about its impact on future assignments.

General Implementation Issues

To produce intelligible output for computations involving lazy **cons**, your interpreter must force the complete evaluation of the answer up to a large depth bound (which is a constant in your program) but abandon evaluation after reaching this depth to avoid non-termination. An easy solution is define the **toString()** operation in **JamCons** to generate depth-bounded output. The provided base code (a subset of the class solution) already does this with a depth bound of 1000. So even output of lists in call-by-value interpretation is bounded by this depth limit. None of our test cases involves printing lists longer than this depth bound.

If a computation produces an infinite answer, the large bound on forcing depth prevents the forced computation from running "forever" (until the provided computational resources are exhausted). An interactive user interface that allowed the user to request the expansion of an unevaluated node (to some depth) in an answer would be more satisfactory, but building such an interface is beyond the scope of a routine programming assignment.

Testing and Submitting Your Program

The **`src`** folder in the provided code base includes some Junit4 test classes which are not comprehensive. Make sure that your program passes the sample unit tests in **`Assign1Test.java`** and **`Assign2Test.java`**. You need to add more test cases; you need to test the interpretation of each basic form of program **`AST`**, every inductive **`AST`** construction and some more complex composite programs involving both lists and integers.

Each class and method in your program should be preceded by a short `javadoc` comment stating precisely what it does.

Edit the **`README`** file for your repository so that it

- outlines the organization of your program,
- specifies any compatible additions that you made the provided API (not generally necessary), and
- specifes what testing process you used to confirm the correctness of your program.

Note that the **`README`** file is a GitHub "MarkDown" file, text augmented by a few simple formatting annotations. MarkDown (MD) is much less powerful than HTML but much easier to use. The webpage **markdown-cheat-sheet** is a good summary for learning MarkDown and a useful reference.

Since your program resides in a repository owned by our classroom, you do not need to do anything to submit except push you the latest version of your local repository back to the origin on GitHub.

Your test suite should ideally test every feasible control path in your code. Actual code often contains unreachable error reports. Such control paths obviously cannot be traversed by any test case. We expect you to come very close to ideal coverage.

Please make sure to remove or disable all debugging output generated by program. Excessive use of print statements considerably slows your interpreter and thus testing your project down. If your interpreter generates a timeout error because debug output slows it down, you may not get credit for the test cases that took too long.