Project 2: Three Interpreters for Jam

Points: 100

Files

The support files are provided in the stub directory created when you accept the assignment. Do not accept this assignment before completing Assignment 1.

Overview

Preparation   Examine the files in the repository created for you by GitHub Classroom. The support files are identical to those provided for Project 1 except that the file Parser.java contains the class solution instead of your solution to Project 1. You are free to use either version of Parser.java. If you use your own parser, make sure that the `map` construct accepts an empty parameter list.

Task   Your task is to write three interpreters for the *Jam* language syntax from Project 1: one that performs *call-by-value* evaluation, one that performs *call-by-name* evaluation, and one that performs *call-by-need* evaluation. The input to your interpreter will be a Jam program in the abstract syntax generated by the parser from Project 1. Call-by-need evaluation has the same syntactic semantics as call-by-name but the implementation is "optimized" so we must explain call-by-name evaluation first. The difference between call-by-value evaluation and call-by-name evaluation is straightforward. Consider a syntactic semantics for Jam analogous to the one we gave in Lecture 4 for the language LC. In the subsequent discussion of Jam evaluation, we will call Jam `map`s *procedures* which is consistent with language-independent programming language terminology. In LC, `lambda`-abstractions are procedures. In call-by-value evaluation, the arguments in a procedure application (class `App` in the Java representation of Jam abstract syntax) are reduced to values before the arguments are substituted into the procedure body. In call-by-name evaluation, the argument expressions are substituted into the procedure body *without* being evaluated! These argument expressions are evaluated if and when their values are demanded during the evaluation of the body. This difference is codified by the reduction rules for procedure applications.

In call-by-value,

$$(\texttt{map}\ x_1,\ldots,x_n\ \texttt{to}\ E)\ (V_1,\ldots,V_n) \rightarrow \texttt{E}[x_1 \leftarrow V_1,\ldots,x_n \leftarrow V_n]$$

where $V_1,\ldots,V_n$ are *values*. Hence, the procedure body in an application is not evaluated until the arguments are reduced to values.

In call-by-name,

$$(\texttt{map}\ x_1,\ldots,x_n\ \texttt{to}\ E)\ (M_1,\ldots,M_n) \rightarrow \texttt{E}[x_1 \leftarrow M_1,\ldots,x_n \leftarrow M_n]$$

where $M_1,\ldots,M_n$ are arbitrary *expressions*. This rule performs the substitution immediately rather than reducing $M_1,\ldots,M_n$ to values first.

Since call-by-name defers the evaluation of arguments in procedure applications, some of those arguments may never be evaluated in some cases. As a result, there are Jam programs that converge to an answer for call-by-name but diverge of abort with an error for call-by-value (because an argument in a function application that is never demanded in call-by-name evaluation diverges or aborts in call-by-value evaluation).

For the sake of efficiency, your interpreter must be a "meta-interpreter" that maintains an environment of variable bindings instead of performing explicit substitutions. In the environment, each variable must be bound to a representation that can be evaluated on demand. **Warning**: beware of scoping problems in your call-by-name interpreter. *Make sure that you evaluate each argument expression in the proper environment.* **Hint**: it may be helpful to think of each argument expression as a procedure of no arguments. If you use the correct representation for call-by-name bindings, the implementation of call-by-name requires very little code and the call-by-name and call-by-value interpreters can share most of their code.

We recommend that you use the generic `PureList` class in `PureList.java` to represent an environment.

Call-by-need The call-by-name approach to evaluation is wasteful because it re-evaluates an argument expression *every* time the corresponding parameter is evaluated. By using the list representation for environments and performing destructive operations on the data representation of bindings (name/value pairs) in the environment, it is possible to eliminate this wasteful practice and only evaluate each argument expression once, the first time its value is demanded. This evaluation protocol is called *call-by-need*. For functional languages like Jam, there is no difference in the observable behavior or call-by-name and call-by-need, except that the latter is generally more efficient.

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 Project 1. Since there is no straightforward way to inspect closures or environments (without presuming a particular implementation), the direct output of environments or closures will not be tested. We can still however test aspects of a closure. For example: `arity(map x to x)`.

The visible part (for testing) of your program must include the top-level class: `Interpreter` containing the methods `JamVal callByValue()`, `JamVal callByName()`, and `JamVal callByNeed()`. The interface `JamVal` is defined in the provided parser library files.

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`. Store your source program in a file named `Interpreter.java`. In summary, your should create a source file `Interpreter.java` matching the following template:

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

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

   public JamVal callByValue() { ... };
   public JamVal callByName()  { ... };
   public JamVal callByNeed()  { ... };
}
```

The file `Interp.java` in your repository already contains the preceding code.

Choice of Language Your program must compile and run correctly using Java 8.

Input Language Specification

The following paragraphs define the *Jam* subset that your interpreter should handle and the new parser's input language.

Your interpreter should accept the *Jam* language including:

- the unary operators `+`, `-` and `~`;
- the binary operators `>`, `>=`, `!=`, `+`, `-`, `/`, `*`, `=`, `<`, `<=`, `&` and `|`; and
- the primitive Jam operations `cons?`, `empty?` `number?`, `function?`, `arity`, `list?`, `cons`, `first`, and `rest`.

The supported operators have exactly the same meaning as their Scheme counterparts (where the unary operator ~ is identified with the Scheme `not` function, and the binary operators `&` and `|` are identified with the Scheme `and` and `or` operators) with two exceptions:

- `/` means the integer division,
- = and `!`= mean equality and inequality on arbitrary objects.

The binary operator = behaves exactly like the Scheme `equal?` function, *i.e.* structural equality for objects and identity for closures (functions). For closures, the only way that functions can be equal is if they are actually the same object, i.e. they were created by the same closure allocation.

Some instructive examples of Jam semantics include:

- `5 = 5` evaluates to `true`
- `5 = 6` evaluates to `false`
- `cons(1,cons(2,empty))=cons(1,cons(2,empty))` evaluates to `true`

- `cons(1,cons(2,empty))=cons(1,cons(3,empty))` evaluates to `false`
- `cons(1,cons(2,empty))=cons(1,empty)` evaluates to `false`
- `cons? = cons?` evaluates to `true`
- `cons? = empty?` evaluates to `false`
- `cons? = (map x to x)` evaluates to `false`
- `(map x to x)` = `(map x to x)` evaluates to `false`, because the two maps are distinct objects in memory.
- `let m:=(map x to x); in m = m` evaluates to `true` in call-by-value because the right-hand side of the definition is evaluated once, and from that point on `m` refers to the same object; to `false` in call-by-name, because the right-hand side is evaluated each time `m` is used, so the two objects being compared are not created during the same allocation; and to `true` in call-by-need, because the right-hand side of the definition is evaluated the first time `m` is used and from then on refers to that object.
- The equality operation defined on closures in the two bullets above has implications on the equality of lists: you have to compare two lists item by item using Jam =.

By Scheme, we mean the *intermediate language with lambda* defined in DrRacket used in Comp 210 and Comp 211.

The preceding definition of equality for closures is what is commonly done in languages like Scheme, Java, and C#, but it is unappealing from a mathematical perspective. Many common algebraic transformations break if closures are involved. There are more elegant ways to define equality on closures but they are more expensive. If we "flatten" closure representations so that a closure consists of a lambda-expression (AST map) M and a list of the bindings of the free variables in M, then we can compare the lambda-expressions (as ASTs) and the bindings of the free variables (recursively invoking equality testing of objects). For this assignment, we are using the common, inelegant definition which is easy to implement.

The meanings of the primitive functions are as follows:

- `cons?(p)` returns `true` if `p` was created by `cons()`. Hence, `cons?(empty) = false`.
- `empty?(p)` returns `true` if `p` is `empty` (the empty list).
- `number?(p)` returns `true` if `p` is a number

- `function?(p)` returns `true` if `p` is a function (*i.e.*, a `map` or a primitive function)
- `arity(p)` returns how many parameters the function `p` takes. It returns an evaluation error if `p` is not a function.
- `list?(p)` returns `true` if `p` is a list (anything constructed by `cons()`, and `empty`)
- `cons(p,q)` returns a list consisting of the list `q` with the value `p` added to the front. It returns an evaluation error if `q` is not a list.
- `first(p)` returns the first element of list `p`. It returns an evaluation error if `p` is not a list.
- `rest(p)` returns all but the first element of `p`. It returns an evaluation error if `p` is not a list.

The language constructs `if`, `map`, `let`, and application in JAM have the same meaning as `if`, `lambda`, `let`, and application in Racket/Scheme with two exceptions. First, the call-by-name version of Jam uses call-by-name semantics for function application. Second, when Jam `if` is given a non-Boolean input in the test position, it reports an evaluation error.

Your interpreter should report errors by throwing an `EvalException` containing an appropriate message. All errors abort execution.

The Jam `let` construct is simply an abbreviation for the corresponding lambda application. Hence

```
let v1 := M1;
    ...
    vn := Mn;
in E
```

has exactly the same meaning as

```
(map v1, ..., vn to E)(M1, ..., Mn)
```

Hence, a `let` is a combination of a `map` and an application. Note that the semantics of `let` depends on whether the interpreter is using call-by-value or call-by-name.

Lists are created using `cons` and `empty`, where `empty` is the empty list. The one-element list `(1)` is created using `cons(1,empty).` Since the values returned by `cons(...)` and `empty` are lists, `list?` returns `true` for both of them. The following program evaluates to `cons(true,true)`:

```
cons(list?(cons(5,empty)),cons(list?(empty
),empty))
```

The binary infix operators & ("and") and | ("or") denote the *short-circuit* versions of those operations even in call-by-value Jam. That means these operations look just at their first operand and try to determine the result; only if they cannot determine it from just the first operand do they look at the second at the second operand. For example, `(true | x)` evaluates to `true` regardless of what value `x` has.

The ternary `if-then-else` operation works in a similar way: It first evaluates the test expression appearing between `if` and `then`. If it evaluates to `true`, `if-then-else` evaluates the consequence expression immediately following `then`; if it is `false`, it evaluates the alternative expression immediately following `else`. `if-then-else` never evaluates both the consequence and alternative expressions.

Representation of Program Values

A Jam program manipulates data values that are either numbers, booleans, lists, or function representations (closures). To standardize the visible behavior of Jam interpreters, we stipulate that Jam interpreters *must* observe the following conventions concerning the representation of Jam values in Java. Jam values must be represented using the JamVal interface and subclasses as defined in each of the three parser library files.

Commenting, Testing and Submitting Your Program

The `src` folder in the provided code base includes some Junit4 test classes which are not comprehensive. You need to add more test cases; you should to check every base constant (other than all numbers) and every compound construction as shown in the syntax diagrams.

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 (unnecessary for most submissions), and

- specifes what testing process you used to confirm the correctness of your program.

Each procedure or method in your program should include a short `javadoc` comment stating precisely what it does. Note that the README file is a GitHub “MarkDown” file, which is 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.

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.

All JUnit test classes submitted for grading should be compatible with a generic solution that conforms to the public interface provided by the stub files in the provided support code. In other words, we should be able to compile and run your unit tests against a reference solution used for grading, which has the same public interface as the support code. Your test classes should be defined in files matching the pattern **`"*Assign#Test.java"`**, where **`"#"`** is the current assignment number. For example, **`"MyAssign2Test.java"`** would be a valid name for Assignment 2 unit tests. If you want to include other tests that you do not want graded (e.g., to test your private interfaces), then simply use a different suffix than "Test" for the test class names (e.g., **`"Assign1Test#.java"`**). If you have additional utility classes that are required dependences of your unit test classes, please name those classes using the pattern **`"*Assign#TestSupport.java"`**, which signals that the class is required to support your unit tests, but is not a unit test class that should be independently launched. You may also include test support files with names matching the pattern **`"*AssignAllTestSupport.java"`**, which can contain common test support code that can be reused across multiple assignment submissions without needing to rename the file. Note that our class solutions to the assignments do not rely on such utility classes. We anticipate that most solutions will only include test files of the form **`"*Assign#Test.java`**.

You obviously do not need to “submit” your program, since you are creating it inside a repository on the ownership of the course staff. We can easily access all student repositories.

Implementation Hints

In your call-by-name interpreter, the argument expressions must be evaluated in the environment corresponding to their lexical context in the program. The free variables in argument expressions behave *exactly* like the free variables in functions.

The form of your interpreters should be very similar to the environment-based interpreters presented in class. You should represent ground values by their *Object* equivalents (where lists are represented using the `PureList` class in the parser library file) and function values (closures) as anonymous inner classes (the strategy pattern for representing functions). Implement each of the primitives using the corresponding Java primitive when possible. Of course, you will have to implement the `function?` and `arity` primitives on your own since there are no Java equivalents. Fortunately, this task is straightforward.

Your interpreters should return the Java data values (`JamVal`s or aborting `Exception`s) representing the results of the specified Jam computations.

Note that unary and binary operators are *not* values while primitives are. Unary and binary operators are not legal expressions; hence, they can never be used as values. In addition, note that the binary operators `&` and `|` are special: they do not evaluate their second arguments unless it is necessary. Of course, a Jam programmer can always define `map`s that define the functions corresponding to operators.

In writing your interpreters: follow the definition of the data. Your interpreters should contain a separate visitor `for<class-name>` method for each abstract syntax constructor type.

Remember to "factor out" repeated code patterns which you presumably learned in Comp 215 and/or Comp 310.

Implement a very small subset of the primitive operations first and get everything debugged. Write a call-by-value interpreter first and then call-by-name and call-by-need, sharing code using inheritance wherever possible. Once all three interpreters are working, add support for the operations that you intially left out.