CS 316 Fall 2020
Observe **course policies** in undertaking this project.
All programs must be written in Oracle Standard Edition compliant Java.

PROJECT 3: Emulation of the Runtime Stack
Due: 11/27/20, Friday, 11 PM
Late projects will not be accepted.

The purpose of this project is to acquire better understandings of the runtime stack mechanism by emulating it inside the source program. This project does not depend on Project 2. Project 4 will build on Project 2 and this one.

Create an abstract class `AR` whose objects will be activation records. Actual activation records for function calls will be objects of subclasses of `AR`. Emulation of activation records and the runtime stack is illustrated by the reimplementation of `Parser.assignmentList()`:

```
public static AssignmentList assignmentList()

// <assignment list> --> <assignment> | <assignment> <assignment list>

{
        Assignment assignment = assignment();
        if ( state == State.Id )
        {
                AssignmentList assignmentList = assignmentList();
                return new MultipleAssignment(assignment, assignmentList);
        }
        else
                return assignment;
}
```

The following is the reimplementation using the explicit activation record class `ARassignmentList`:

```
class ARassignmentList extends AR
{
        Assignment assignment; // local variable
        AssignmentList assignmentList; // local variable
        AssignmentList returnVal;

        void assignmentList()

        // <assignment list> --> <assignment> | <assignment> <assignment list>

        {
                ARassignment ar = new ARassignment(); // create a new activation record
                RuntimeStack.push(ar);
                ar.assignment(); // call assignment()
                assignment = ar.returnVal; // extract the return value
                RuntimeStack.pop();

                if ( LexArithArray.state == LexArithArray.State.Id )
                {
                        ARassignmentList ar_ = new ARassignmentList(); // create a new activation record
                        RuntimeStack.push(ar_);
                        ar_.assignmentList(); // call assignmentList()
                        assignmentList = ar_.returnVal; // extract the return value
                        RuntimeStack.pop();

                        returnVal = new MultipleAssignment(assignment, assignmentList);
                }
                else
                        returnVal = assignment;
        }
}
```

`RuntimeStack` is my class implementing various fields/functions to manage the stack of AR objects. The above code closely follows the intermediate-code operational semantics described in Course Notes #8, except that the calls of `assignment()` and `assignmentList()` are made in Java itself, hence "runtime stack emulation".

The following are the general code schemas for the activation record class for function `f` and for reimplementing function call `a = x.f()`, where the type of `x` is `targetType` and the return type is `returnType`. The functions to be reimplemented in this project have no parameters.

```
class ARf extends AR
{
        targetType target;
        field for local variable 1;
        ...
        field for local variable m;
        returnType returnVal;

        void f()
        {
                ...
                here reimplement body code of f()
                ...
        }
}
```

All reimplemented functions will have `void` return type.

The code schema for reimplementing `a = x.f()` is:

```
        ARf ar = new ARf();
        ar.target = value of target variable x;
        RuntimeStack.push(ar);
        ar.f();
        returnType a = ar.returnVal;
        RuntimeStack.pop();
```

You are to reimplement the following functions in the above manner:

- `Parser.assignmentList()`
- `Parser.assignment()`
- `Parser.E()`
- `Parser.term()`
- `Parser.primary()`
- the `M()` functions
- the `Eval()` functions

The other functions remain the same as the originals.

Write classes `ParserStack` and `InterpreterStack` that reimplement, respectively, `Parser` and `Interpreter`. The interpretation results are to be displayed on the terminal as the original `Interpreter` does.

Implement a function to display the following data in legible format in an output file:

- the total number of function calls made so far
- the maximum stack size so far, measured by the number of activation records in the stack
- the ARs of the runtime stack displayed from top to bottom

Call this function every time the reimplemented `primary()` parsing function is about to return and every time the reimplemented `Eval()` for `Primary` objects is about to return.

The runtime stack may be implemented by an array, linked list, or library stack class.

The class templates for this project is provided here.

If a package is used, `ParserStack.main` and `InterpreterStack.main` must be included in the package named `cs316proj3`, all letters in lowercase.

Here are sample outputs of argv[2] files and interpretation results displayed on the terminal obtained by running `InterpreterStack.main`:

- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result
- argv[0] input | argv[2] output | interpretation result

Submission

Email the following materials to keitaro.yukawa@gmail.com with the subject header:

CS 316, Project 3, your full name

- All the classes comprising your source code, including the classes in this page. Make sure to double check no classes are missing.
- A list of all class names showing the inheritance hierarchy, arranged like this page. This may be in text, html, or PDF file.

Do not include any `---.class` files or any test input/output files.

You may email the entire materials in a .zip or .rar compressed file.

The due date is 11/27/20, Friday, 11 PM. No late projects will be accepted. If you haven't been able to complete the project, you may send an incomplete program for partial credit. In this case, include a description of what is and is not working in your program along with what you believe to be the sources of the problems.