# CSE/IT 324 Final Project

Individual Project: Create a LISP interpreter in Java

**Introduction & Goal:**

The goal of this project is to learn about the basic constructs of Java while simultaneously understanding the basic concepts behind creating an interpreter in LISP, a simple functional programming language. This project will bring together many concepts that have been taught in this class over the entire semester.

The final result of this project will be a Java code that executes LISP style commands from a prompt all while within a Java environment. In addition, you must write a report describing the inner workings of your code, and implementation details.

**Code Implementation Requirements:**

Java code MUST successfully parse and execute LISP code as outlined below in the table. The code you implement MUST function like an interactive LISP interpreter and output all intermediate results to a file. The code MUST compile and execute on the CS machines.

An example of execution flow is as follows:

Make command (*make*)

```
Welcome to the fancy new Prompt LISP INTERPRETER, type in LISP commands!
> ( + (+ 2 3) 5)
> 10
> (define b 2)
> b
> (defun ADD (x y) (+ x y))
> ADD
> (ADD (8 3))
> 11
> (quit)
> bye
EXECUTION STOPPED

/**
“results.file”:
10
b
ADD
11
EOF
**/
```

Your interpreter MUST allow for the following expressions:

| Expression | Syntax | Semantics and Example |
|---|---|---|
| Variable reference | *var* | A symbol is interpreted as a variable name; its value is the variable's value.<br>Example: (r) -> 10 |
| Constant literal | *number* | A number evaluates to itself.<br>Example: (12) -> 12 |
| Quotation | (quote *exp*) | Return the *exp* literally, do not evaluate it.<br>Example: (quote (+ 1 2)) -> (+ 1 2) |
| Conditional | (if *test conseq alt*) | Evaluate *test;* if true, evaluate and return *conseq;* otherwise *alt*.<br>Example: (if (> 10 20) (+ 1 1) (+ 3 3)) -> 6 |
| Definition | (define *var exp*) | Define a new variable and give it the value of evaluating the expression *exp*.<br>Examples: (define r 10) |
| Procedure call | (*proc arg…*) | If *proc* is anything other than one of the other symbols (if, define, or quote) then treat it as a procedure. Evaluate *proc* and all the *args,* and then the procedure is applied to the list of *arg* values.<br>Example: (sqrt (* 2 8)) -> 4.0 |
| Assignment | (set! *var exp*) | Evaluate *exp* and assign that value to *var*, which must have been previously defined (with a define or as a parameter to an enclosing procedure).<br>Example: (set! r2 (* r r)) |
| Procedure | (defun (*var…*) *exp*) | Create a procedure with parameter(s) named *var*… and *exp* as the body. Create this using static scoping.<br>Example: (defun (r) (* pi (* r r)))<br>N.B: here *defun* is used to denote procedure declaration. |

**Any design decisions you make must be detailed in the report (such as: which operators you included (+, -, *, etc.), what pre-defined procedures you included (sqrt, exp, etc.), and other various things). Individual implementation is entirely up to you, but the above 8 criteria MUST exist as described above.**

A makefile MUST be provided that works on CS machines. If your files do not compile on the CS machines you will receive a 0 for the code section of the project!

A file has uploaded containing test cases.

**Report Requirements:**
The report must be in the most recent standard ACM format, a template of which can be found at https://www.acm.org/publications/proceedings-template.

The report must contain

1. Descriptions of how each expression in the above table were implemented. Most descriptions should include the types of Java constructs you used for your implementation and other various information pertaining to the specific expression that is being described. (Example: The implementation of definition involved creating a Java HashMap that…)
2. Descriptions of class topics that apply to each expression implementation. (Example: In class we learned about static scoping and how information on the stack is represented. This idea is applied when using different environments for various procedures…)
3. Descriptions of implementation details that are important/pertain to the completion of the project but don't necessarily fall into the two previous categories. (Example: In order to implement local/global environments in the usage of procedures I did the following…; I defined various lisp symbols (like >,<,=,-,+,etc.) using…)
4. Anything else you think worth mentioning (why your solution is exceptionally good, areas that your implementation lack in, if you couldn't get one of the expressions to work)
5. A final section that describes your concluding thoughts on this project and if you thought it was fun/interesting.
6. A list of references (website, tutorial, code source, or otherwise) used when implementing this project. You are free to use any source you find on the Internet, just make sure to reference it!

No example document will be posted on the website, document sectioning is up to you, the student.

**Submission Requirements:**
Your assignment must be submitted before the scheduled time and date on the canvas website. Your report should be submitted as a PDF file and your code should be submitted as a tarball. The PDF must conform to the naming convention "lastname_project2.pdf". The tarball must conform to the naming convention "lastname_firstname.tar.gz". In both of these cases, obviously replace "lastname" with YOUR last name and "firstname" with YOUR first name.

In your code tarball there must be a MAKEFILE that compiles your code and creates the executable that, when run, will come up with a prompt similar to the above example. This prompt should be able to execute LISP style code matching the format of the expressions in the table defined above. Also include a README that contains specific details on the running environment and how to use your code if you cannot get it to conform to the requirements. If your code works exactly as requested you can omit the README, but if it doesn't work as intended you will lose points.

**Grading:**

Each submission will be graded as follows:

75% Code Implementation
- 10% make file
- 5% style & commenting
- 10% for outputting to a file
- 10% for having a prompt that evaluates expressions
- 40% for each expression's correct implementation, around 5% each

25% Report
- 5% Grammar
- 5% Formatting and Citation
- 15% Analyses and Descriptions of required content described above

35% Extra credit

Implement a dynamic scoping mechanism. This dynamic scoping mechanism will ignore the statically scoped environments and instead looks at the Caller's AR to determine what the value of a defined variable will be. You will only get credit for an implementation of this mechanism if you also describe, implement, and include a minimum of 5 test cases that showcase your dynamic scoping mechanism in direct comparison to a statically scoped version. Your examples should show differing results when using the statically scoped method versus the dynamically scoped method and should be appended to the end of your report as an appendix. In addition to the implementation and test data you must also include a separate section in your report that contains descriptions of your implementation, drawings of the run-time stacks for each of your examples, and a hand-written tracing of the results from your examples to show that they match the output of your application (this will be similar to question 3. a) from assignment 2).

Additional Related Assignment 15 points-- Project Update

15 pts of the final grade of 115 points will be in a project update. In this update you must upload **java code and jar file** that attempts an implementation of the first **6** expressions listed in the table above. It does not need to be 100% correct, but it does need to make a valid and decent attempt at solving the problem. This submission also requires a **makefile** that compiles the code used in your implementation. This update needs to be submitted by the deadline posted on Canvas in the form of a tarball.

Have fun and good luck!