A Lisp interpreter for our custom LISP dialect, built in Java and running on the JVM. Final project for Mr. Brown's APCS class, second semester. The dialect will include a few unusual features, such as the ability to include other files and define macros as preprocessor directives. The initial implementation will be minimalist and based upon John McCarthy's classic paper "A Micro-Manual for LISP - Not the whole truth," a copy of which has been provided in this repository. We will then slowly add features on an experimental branch and push to the master branch when they are stable. Our implementation of LISP will utilize some of the datastructures and algorithms we have learned throughout the semester, namely queues and trees. More will be added here in the next few weeks.

Implementation of Term2 Concepts

Queue: The first phase of the interpreter (the Lexer) will receive the file to interpret and return a queue of Token objects. Whitespace will be first added around parentheses. Each of these Tokens will be enqueued via a scanner, and each Token will be

separated on whitespace (except in the case of Strings). When all of them are enqueued, they will be dequeued by the main program, and passed to the parser.

Tree: The parser will construct an abstract syntax tree that will be fed to the evaluator. This tree will be formed as Tokens are sent from the Lexer's queue to the Parser. Each time a Token is passed to the Parser, the parser will add a node to the tree. The type of this node will be dependent on the type of symbol. The tree will abstract away unnecessary syntax included in the tokens, such as parentheses, as the structure of the program will be conveyed by the structure of the tree.

Functionality

Data Types

- `Number`- initially these will be integers, but floating points will be added eventually
- `String`- stored in java's native `String` type
- `Boolean`- with `T` for true and `NIL` for false

Data Structures

- `List`- constructed using cons, stored as a `LinkedList` in Java
- `atom`- stores one primitive datatype

Functions

Initially, the following predefined functions will be available to users of Zarodenk Lisp. These definitions are paraphrased from the John McCarthy paper mentioned above.

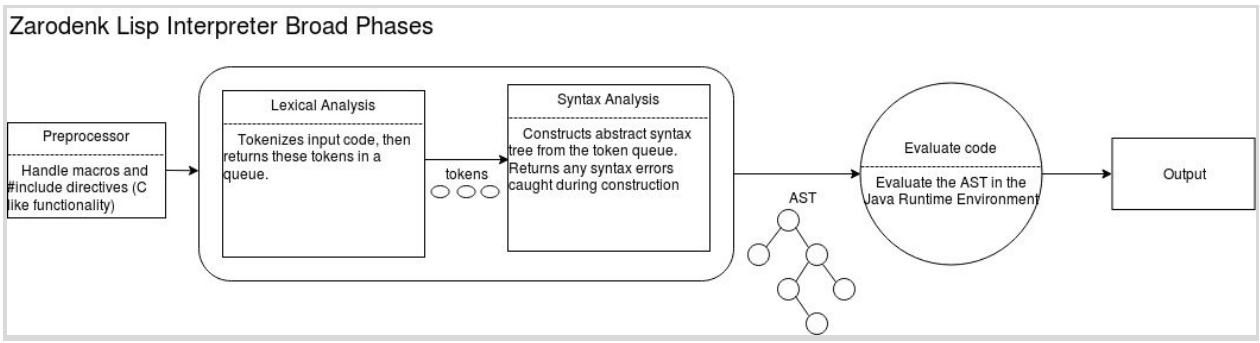
- `(quote e)` evaluates to `e`.
- `(car l)` evaluates to the first element in non-empty list `l`.
- `(cdr l)` evaluates to the list `l` without its first element.
- `(cons l1 l2)` evaluates to the list that is the result of prefixing the lists `l1` and `l2` together.
- `(equal e1 e2)` evaluates to `T` if expression `e1` is equal to expression `e2`. The expression evaluates to `NIL` otherwise.
- `(atom e)` evaluates to `T` if the expression.
- `(cond (pi ei) .. (pn en))` evaluates to value ei where pi is the first of the p's whose value is NOT `NIL` (or `T`).

- `((lambda (vi ... vn) e) ei ... en)` evaluates to `e`, where the variables `vi ... vn` take the values of expressions `ei ... en`.
- `((label f (lamda (vi ... vn) e)) ei ... en) evaluates to the same as`((lambda (vi ... vn) e) ei ... en)`, except that when`(f ai ... an)` is invoked, it is replaced with (label f (lambda (vi ... vn) e)). This allows for recursive functionality.
- `(print v)` will print the value `v` to the console.
- `(defun f (vi ... vn)) e)` will be used to permanently defines function `f` so it can be reused.
- `(def s v)` will define the variable `s` as value `v`.
- `(list ei ... en)` evaluates to `(cons ei(cons ... (cons en nil)))`.
- `(and p q)` evaluates to `T` if `p` and `q` both evaluate to `T`, otherwise evaluates to `NIL`.
- `(or p q)` evaluates to `T` if either `p` or `q` are true, otherwise evaluates to `NIL`.
- `(not p)` evaluates to `T` if `p` is `NIL`, and `NIL` if `p` is `T`.
- `(operation ei ... en)` evaluates to (in infix) `ei operation ... operation en`, where the e's are `numbers`, and `operation` is an arithmetic operation (+, -, *, /, %).
- `(comparator ei ... en)` evaluates to (in infix) `ei comparator ... comparator en`, where the 's are `numbers`, and `comparator` is a numerical comparator (=, >, <, >=, <=, !=).

Misc.

- As with most LISP dialects, Zarodenk Lisp is dynamically typed
- Recursion will be added after the other core functionality is implemented, as it's significantly more complex to implement.
- "Short circuit" functionality will be availible with `and` statements
- `NIL` is the name of the empty list `()`

Interpreter


Zarodenk Lisp Interpreter Broad Phases
Preprocessor
Handle macros and #include directives (C like functionality)
Lexical Analysis
Tokenizes input code, then returns these tokens in a queue.
tokens
Syntax Analysis
Constructs abstract syntax tree from the token queue. Returns any syntax errors caught during construction
AST
Evaluate code
Evaluate the AST in the Java Runtime Environment
Output