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 data structures 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 data type

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.

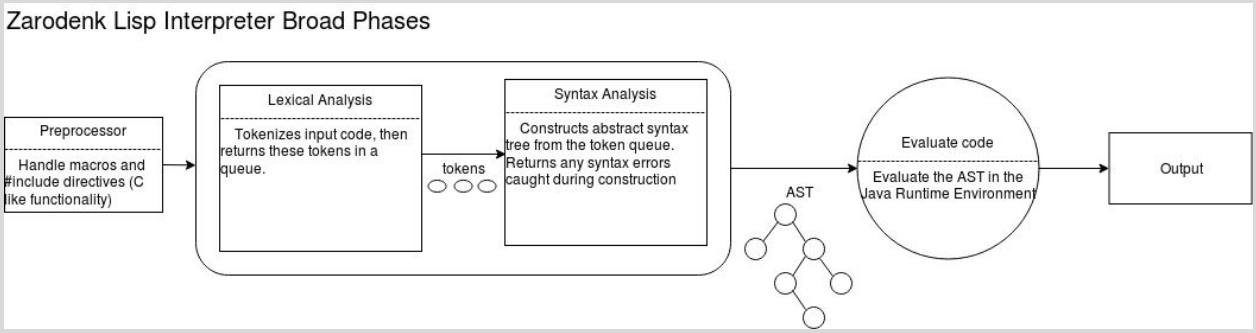
- `(quote e)` evaluates to `e`.
- `(car l)` evaluates to the first element in non-empty list `l`.
- `(cdr l)` evaluates to the list `l` without its first element.
- `(cons` l_1 l_2`)` evaluates to the list that is the result of prefixing the lists l_1 and l_2 together.
- `(equal e1 e2)` evaluates to `T` if expression e_1 is equal to expression e_2. The expression evaluates to `NIL` otherwise.
- `(atom e)` evaluates to `T` if the expression.
- `(cond (`p_i e_i`) .. (`p_n e_n`))` evaluates to value ei where pi is the first of the p's whose value is NOT `NIL` (or `T`).
- `((lambda (`v_i `...` v_n`) e)` e_i `...` e_n`)` evaluates to `e`, where the variables v_i `...` v_n take the values of expressions e_i `...` e_n.

- `((label f (lambda (v_i ... v_n) e)) e_i ... e_n)` evaluates to the same as `((lambda (vi ... vn) e) ei ... en)`, except that when `(f a_i ... a_n)` is invoked, it is replaced with `(label f (lambda (v_i ... v_n) e))`. This allows for recursive functionality.
- `(print v)` will print the value `v` to the console.
- `(defun f (v_i ... v_n)) 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 e_i ... e_n)` evaluates to `(cons e_i(cons ... (cons e_n 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 e_i ... e_n)` evaluates to (in infix) `e_i operation ... operation e_n`, where the e's are `numbers`, and `operation` is an arithmetic operation (+, -, *, /, %).
- `(comparator e_i ... e_n)` evaluates to (in infix) `e_i comparator ... comparator e_n`, 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 available with `and` statements
- `NIL` is the name of the empty list `()`

Interpreter



Lexer (Lexical Analysis)

The lexer will take the raw LISP code and turn it into a queue of tokens to be read by the parser. It will accomplish this via the following steps:

1. Read file as a String
2. add whitespace around (and)
3. Use scanner to split resulting string on whitespace (except in the case of Strings, which should be kept together), and pass all non-whitespace Strings to Tokens
4. Tokens will determine type by using regular expressions (RegEx), and be added in order to a queue of tokens. A list of Token types has been listed below.
    - LPAREN- (
    - RPAREN-)
    - SYM- a (any none primitive atom)
    - STR- "a"
    - NUM- 1 (floating points to be added later)

The queue of tokens will be returned and subsequently fed to the parser.

Ex:

Code

```
(def a 3)
(cond
  ((> a 2)
   (print "Big num"))
  (T
   (print "Small num")))
(print "Done!")
```

Token queue returned

```
Token(LPAREN,"(")
Token(SYM,"def")
Token(SYM,"a")
Token(NUM,"3")
Token(RPAREN,")")
Token(LPAREN,"(")
Token(SYM,"cond")
Token(LPAREN,"(")
Token(LPAREN,"(")
Token(SYM,">")
Token(SYM,"a")
Token(NUM,"2")
```

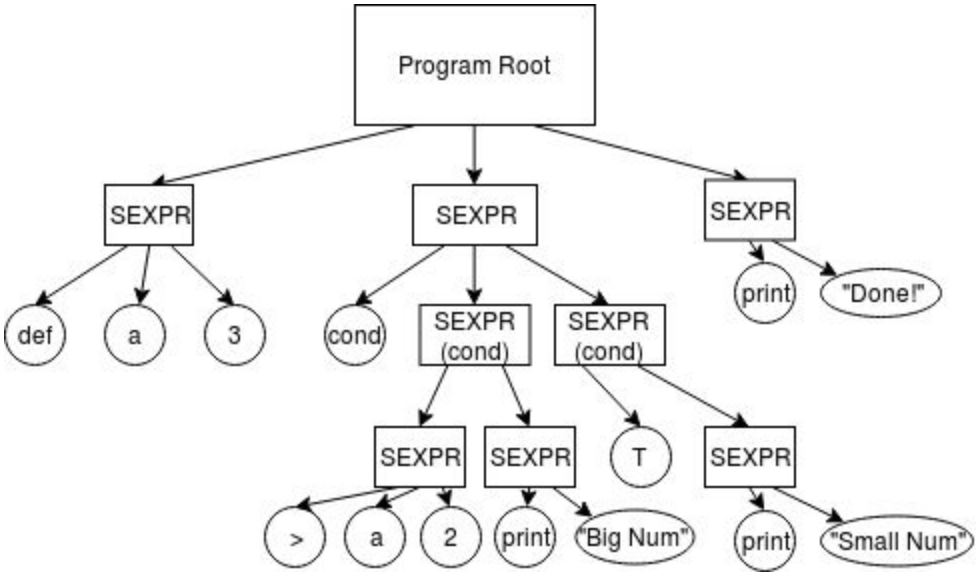
```
Token(RPAREN,")")
Token(LPAREN,"(")
Token(SYM,"print")
Token(STR,""Big num"")
Token(RPAREN,")")
Token(RPAREN,")")
Token(LPAREN,"(")
Token(SYM,"T")
Token(LPAREN,"(")
Token(SYM,"print")
Token(STR,""Small num"")
Token(RPAREN,")")
Token(RPAREN,")")
Token(RPAREN,")")
Token(LPAREN,"(")
Token(SYM,"print")
Token(STR,""Done!"")
Token(RPAREN,")")
```

Parser (Syntactical and Semantic Analysis)

Turns token queue into an abstract syntax tree (AST) to be passed to the eval machine. This will also check for syntactical errors, such as too few or too many parens, or malformed s expressions. The following steps will be taken to construct this AST (recursively):

1. Form the root node, which will have each individual s-expression in order (left to right).
2. Add each token as a node to the current tree
    - for each non `PAREN` token, add it as a leaf node
    - for an `LPAREN`, create a new subtree node.
    - for an RPAREN, go back up a level.
3. repeat step 2 recursively for all Tokens. Parens should not be included, as the tree structure will represent the delineations they represented.

AST constructed from token queue returned by previous Lexer example



Evaler

The evaler evaluates the abstract syntax tree produced by the parser, and performs the actions asked by it. To evaluate the tree, a post order traversal will be used to run through the tree and evaluate each subtree. Except when it is a special case (cond sub expressions, for example), the symbol directly at the start of a subtree will be assumed to be a function. When a definition (def or defunc) is called, the symbol being assigned a value will be saved in the namespace for the scope where the call is made. Generally the scope will be global, except in functions, where it will be in the namespace of that function. The “value” of a function will be defined as the AST described in that function definition. Then, when evaluating the function when it is called later, this AST will be substituted in for the function definition, along with the arguments specified in it’s argument list. This substitution will only be done as needed, using a “lazy” strategy similar to Haskell. This will enable recursion to be carried out efficiently and easily.