Implementation document

Structure

First we read a text file. Then the contents of that text file are passed to the Tokenizer, which produces Tokens from the text. From these tokens an Abstract Syntax Tree is parsed by the Parser and the various subparsers recursively. The tree is executed recursively, as each node has a method that unwinds the resulting value. There is a garbage collector that tries to track the life of each resulting node, and free them if they are not needed.

Performance

The tokenization process handles each letter only once, so initially the splitting is done in `O(n)`. However the tokens need to be identified as to being of some type, and this process is `O(xy)` where `x` is the amount of possible token types and `y` is the length of the string that is being handled. `x` is actually static so it could be marked as a multiplier of 29. `y` comes from the assumption that when regexps are compiled they can match the string in `O(n)`, but I have no clue about the implementation. So fully it would be `O(nxy)`, but since `x=29` and `y` tends to be quite small (average token length is only a few characters), it actually resembles `O(n)` mostly. The space requirements also grow linearly.

The parsing process handles each token only once, `O(n)`. Space requirements also grow linearly.

Map is a usual implementation of a hash table, amortized `O(1)` for most operations. The space requirement is linear `O(n)`.

Vector is a usual implementation of a dynamically growing array. Amortized `O(1)` for inserting and deleting from the end, `O(n)` for other places. Linear space requirement.

Possible problems

The garbage collector is broken, I don't have motivation to fix it.

Sources

- Previous experience
- (http://en.wikipedia.org/wiki/Tracing_garbage_collection)