Scanner

quarta-feira, 14 de maio de 2025 18:55

- Crafting interpreters
    - Take in ra source code
        - as a series of characters
        - group in into a series of chunks called tokens
    - Switch with a delusions of grandeur
    - Define how to run the code
        - Receive a source file and process it
    - Good first stage is to read the tokens and print it.
    - Not error handling but sav file some information to find the line latter on
        - linefile
        - file name
    - Separate the code
        - The ones that identify the error and the one that reports them
    - Lexical analysis
        - scan throug the list of characters and group them together into the smallest sequence that still represent something
        - Each blob of characters is called a lexeme
        - when we group the lexeme and boundle it together with the other data, the result is a token
        - Token
            - Keywords are parte of the shae of a language grammar
                - We would like to know if the next workd si a while for xample
            - Parser could categorize tokens from the raw lexeme by comparing the strings (slow and ugly)
            - At the point we recoginze a lexeme, we also remember which king of lexeme it represents (we have a diferent type for each ey word, operator, bit, type ...)
                - Defined in a enum
            - Literal Value (numbers and string)
            - Location information
                - Define a class token with all the information we need (linefile for example. Type, lexeme, leteral and line)
                    - Also good to make a to_String() method
            - Regular language and epressions
                - The core of a scanner s a loop
                - Starting at the first character of the source code
                    - figures it out which lexeme it belongs
                    - consume it and any following characters that are part of the lexeme
                    - When reach the end of the lexeme, emits a token
                    - Keeps doing that, eating characters and occasionally excreting tokens until reach the end of the source code
                - May consider to define a regexp for each kind of lexeme
                    - The rules that determine how a particular language groups characters into lexeme are called lexical grammar
                    - Tools like LEX are designed expressly to let you throw a handful of regexps and they give a comple scanner back
                    - (wont delegating it) but we can....
        - Scanner class
            - Store the raw source code into a string
            - define a list to be ready to fill with tokens (not yet generated)
            - Loop each character in the string source code
            - When it run out of characters, append one final "end of file" token
                - not needed but makes it cleaner
                - Depends on a couple of fields,
                    - start -> points to the first character of the lexeme
                    - current -> points to the character currently been considered
                        - offsets that index into the string to be looped
                    - Line -> tracks what source line currenis on (can improve this... linefile)
                        - Produce tokens that know their location
                    - Helper to see if we reached the end
                        - current >= source.length()
        - Recognizing lexemes
            - heart of the scanner
            - each turn of the loop we scan a single token
            - Starting with the single characters
                - consume the next character and pick a token type for it
                    - Why consume the next?
                        - don't think we need... maybe java starts list position at 1 and not 0
                        - maybe we need... yeah... we need to get the next token... but this may change
                        - ok.. we do need because this is the move foward of the while
                - add token, substring from the start to the current character
                - add the token to the list
        - Lexical errors
            - If not handled, will be silent discarted
                - Probably what i'm going to do
            - handled as a default case, or not supported
        - Operators

            - Some of them requires 2 characters
                - In this case we match the first one and compare the next one... is it < followe by a =? then BANG_eQUAL, otherwise just BANG
        - Longer lexemes
            - Comments \\
                - When we find it... jump to the next line... IMPORTANT
                - Comments usually goes to the end of the line... so if finding it... create a comment token untill the end of the line
                - I'll need to implement this
            - This is our general strategy for handling longer lexemes. After we detect the beginning of one, we shunt over to some lexeme-specific code that keeps eating characters until it sees the end.
            - We can dein peek one or two ahead to match a lexeme... usually one is faster
            - Comments aren't meaningful lexemes and the parser doesn't want to deal with them (don't call add Token() for comments)
                - Whenn start looking for the next lexeme, start member get's reset and the comment's lexeme disappears
                - sip over other meaningless characters (new lines and white spaces) (' ')(\r \t n)
                    - Just remember to increment a new line when skippng lines
                    - For white spaces, simple go back to the beginning of the scan loop to start a new lexeme
        - String literals
            - Always begins with a specific character "
                - create a case for that
                - If doesn't find a maching end... generate an error
                - Consume characters until we hit the final "
                - Consider multiple line strings
                - When creating the token, we also produce the string value, that will be used latter by the interpreter
        - Number letarails
            - A series of digits optionally followed by a "." and more digits
            - To recognize a number, we look for any digit > "0" and < "9"
            - Separate method to consume the rest of the literal in case of number
            - consume as many digits as we find
                - For the integer and if a "." for the float part as well
                - Here we need a seond lookahead, because we don't want to consume a "." if no digit after that
            - convert the lexeme to the numberic value
        - Reserved words and identifiers
            - we can't use a peek one ahead because the user can type a lot of things... like orNot and this is not an or logical operator
            - Use the maximal munch principle
                - when two lexical grammar rules can both match a chunk of code that the scanner is looking at, whichever one matches the most characters wins
                - it means we can't easily detect a reserved word until we've reached the end of what might instead be na identifier.
                - Reserved words are identifiers claimed by the laguage
            - we assume any lexeme starting with a letter or underscore is na identifier
                - if digit, number()
                - else if apha() identifier()
                    - is lpha >='a' <='z' || >='A' && <='Z' || == '_'
                - else unexpeted character error
            - To handle keywords, we see if the identifier's lexeme is one of the reserver words
                - In this case, reserved word class
                - Define a set of reserved keywords for that in a map
                - so, after we scan na identifier, we check to see if it matches anything in the map
                - If meches, use the reserved keywork as token, otherwise it's just a user-defined identifier
    - Type some expected and unexpected code and see if it produces the expected tokens

- Engineering a compiler
    - Scanner is the only pass in the compiler to touch everyy character of the input program
    - Needs to be highly effcient
    - First stage of a 3 part process that compilers uses to understand the input program
    - Scanner or lexical analyzer
        - reads a stream of characers and produces a stream of words
        - if the word is valid, the compiler assigns a syntatic category
    - There are some tools for that but implement it is simple and can be fast and robust
        - Most textbooks and courses uses the generated scanners
        - most comercial compilers and open-source compilers use hand crafted scanners
            - which can be faster because the implementation can optimize it as needed
    - 3 diferent ways
        - table-driven scanner
        - direct-coded scanner
        - hand-coded aproach
    - microsyntax
        - specifies how to group characters into words and how to separate words that run together
        - Some times keywords and reserved words match the rule for na identifier but have special meanings
        - To recognize keywords, the scanner can either use dictionary lookup or encode the keywords directly into its microsyntax rule

        - The compiler writer starts from a specification of the language microsyntax
        - Impemented to requires just O(1) time per character
            - So they run in time proportional to the number of characters in the input stream
    - Reconizing words
        - Let's identify the word new
            - Assuming a routine NextChar -> that returns the next character
        - The code test for n followed by e followed by w
        - Th code fragmentperforms one test per character