Syntax highlighter

Ricardo Alfredo Calvo Pérez

20/5/2024

TC2037 - Computers methods implementation

Index

Summary

In this project, I'm to create a code processor capable of reading Elixir code files and converting them into HTML files. The purpose of this transformation is to provide a syntax-highlighted representation of the Elixir code for better readability and presentation in a web browser based on different types of reserved words and tokens.

Lexical categories

Based on the elixir language the used categories are:

- Reserved words
  - defmodule, defp, def, do, end, false, true, cond, case, if, else, nil.
  - `~r/^(defmodule|defp|def|do|end|false|true|cond|case|when|if|else|nil)(?=\s)/`
- Functions
  - every word started in lowercase followed by a opening parenthesis.
  - `~r/^[a-z]\w*(\!+)?(?=\()/`
- Unused variables
  - this variables are the ones that exists in a function but is not called
  - `~r/^\_[a-z]\w*(\d)*?(\:+)?/`
- Variables
  - every word started in lowercase
  - `~r/^[a-z]\w*(\d)*?(\:+)?/`
- Comments
  - everything after a "#" symbol
  - `~r/^\#.*/`
- Modules
  - every word started with uppercase

    - `~r/^[A-Z]\w*(\d)*?/`
- Attributes
    - This are based on the attributes of a module
    - `~r/^\@\w*/`
- String
    - everything inside quote symbols
    - `~r/^\".*\"/`
- Numbers
    - every number
    - `~r/^\d+(\.\d+)?/`
- Operators
    - +, -, *, /, =, ==, ===, !=, ., ,, |>, ->, &, <>, <, and >.
    - `~r/^(\+|\-|\*|\/|\=|\==|\===|\!=|\.|\,|\|>|\->|\&|\<>|\<|\>)/`
- Atoms
    - every *single* word after a ":"
    - `~r/^\:\w+(\d)*?/`
- Containers
    - (,), {, }, [,]
    - `~r/^[\(\)\{\}\[\]]/`
- Regular expression
    - based on the regular expression syntax wrote between "r~/" and "/"
    - `~r/^\~r.+\//`
- Spaces
    - captures single spaces and tabs
    - `~r/^\s/`

Code reasoning

To use this code correctly you'll need to move the your elixir file to the same direction. For organization purposes you'll need to move you to read file to the folder called "ToReadElixirFiles", but when calling the function DO NOT write the folder direction, the program all ready knows here to find your file, call the function "convert_file" this way `Project.convert_file("file_name.ex")` or `Project.convert_file("file_name.exs")`. If you followed the previews steps you'll find in your terminal an ":ok" message, and you can find your new html file in the "HTML_Results" folder.

What this does it that it will create a new html file based on the name of the file your are sending, then it will write the opening structure of HTML architecture, then it'll read line by line the code elixir code you sent.

On the other hand, we have `Project.convert_folder_parallel("ToReadElixirFiles")`. This function reads all the files inside this directory using threads to speed up the process. There is also `Project.convert_folder("ToReadElixirFiles")`, which serves the same purpose but does not use threads, processing the files in the normal way

To achieve our program to read line by line the file provided by the user, I used the `File.stream!()` function. Followed by a `Enum.map()` function that helps the program to implement the token rules to each line. This map will call the menu function `find_coincidences()` which leads as its name says, to find the coincidences with the regular expression rules.

`find_coincidences()` first call is to find if the first word of the sentence is a reserved word, if not it will skip to the next function which searches if the first word is a function, if not it will skip to the next function, and so on with the other categories mentioned before.

In case of having a match depending on the rule, it'll save the hole coincidence in a list of lists called *result* in this way `[["token","match"],["token","match]]`. Each function has each's own token depending on what we match with. After saving the match will divide the sentence by the length of the coincidence, so the start of the sentence updates an we can do the process again to find the coincidences with the rest.

After gathering all coincidences, the program will write the results in the html template with the function `write_results()` . The template this function use is `match. In case that the token is a space it will just write a space and not the hole template.

Reflexion

This situation is crucial to understand and implement the use of regular expression comparisons, using this algorithm in which each rule is divided by names of what it do makes it easy to read and understand to third parties. However this can make the BigO complexity a little high, but this will depends on the size of the line and file, and how many functions it need to hop in to find the coincidence, this can make waiting time longer, but will detail more of the BigO complexity later on.

But in short words, execution time is not slow, thanks to the elixir functions such as `Enum.map()`

BigO complexity

Understanding how the code works we can say that depending on how many lines the file has and how long the lines in the script is, therefore the BigO complexity is approximately \$O_{(C \times L)}\$ where \$C\$ is the number of coincidences and \$L\$ is the number of lines in the file.

Ethical implications

The development of advanced code processing tools offers significant benefits in terms of efficiency and accuracy, but it also comes with a set of ethical responsibilities. It is crucial for developers to carefully consider these implications and work to mitigate potential risks, ensuring that their technology is used safely, fairly, and responsibly for the benefit of society as a whole.

Some unethical usage could be using this technology to find sensitive information like names, directions, date births, phone numbers, mails or passwords of people who hasn't given you their concent.