Group Meeting - (10/03/2023)

We designed subgroups within our team to experiment with approaches proposed last week. The subgroups are organized as follows, and these subgroups can always change depending on the progress and findings we will accomplish.

Subgroup Progress

- **Subgroup 1** (Maximo & Shahir):

We have been looking into the code snippet that was provided for us. We realized that these examples are front-end code that creates a form/operative interface. Our approach is based on the **Back-End** approach. We saw that 84 Lumber uses services from Unixware 7. Is this a service the company's programmers use themselves? What does Unixware 7 really do? Does it provide a web API, does it store the company's data? I'm guessing that it also hosts the basic programs and runs them through the webserver that it has. We would like to see a live demonstration of how this basic program would work, where it connects to, where does the data comes from (a database/datafiles) and a general picture of the structure of how the system is hosted and operated.

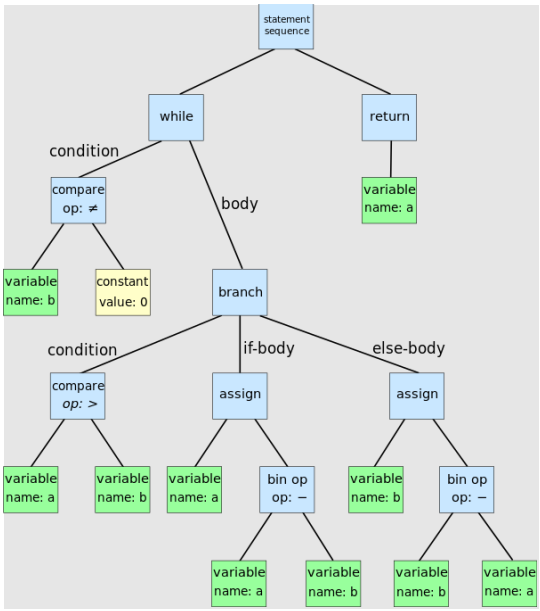
- **Subgroup 2** (John & Pranav & Youssif):

Takeaway of the progress: We aim to establish a pipeline to generate documentation based on the C codebase. To accomplish this goal, we propose three steps.

Step 1: The usage of ANTLR (Another Tool for Language Recognition):

- We aim to create a parsing grammar to break down various components of the C codebase into their respective roles such as functions, parameters, variables, etc.
- Through this step, we want to create AST (abstract syntax tree) to efficiently store this information for lexical analysis. This data structure is typically used to learn HTML and its structure.

Example of AST (Reference: https://en.wikipedia.org/wiki/Abstract_syntax_tree).



Challenges: We are struggling with installing a massive number of dependencies to compile programming languages from C. Due to this challenge, we are also open to explore other approaches to accomplish the same goal.

Step 2: Intermediate Streamlining process

- We are currently thinking of which tools and software to use to generate comments in **Doxygen** format. This step was specifically suggested by @Mishra, Pranav, and @Chu, John and @Goda, Youssif also agreed. We have considered traversing AST using the DFS algorithm and extracting its information. However, we are on the way to producing options to implement this idea. We are open to hearing any type of suggestions.

Step 3: Generate documentation that explains C codebases.

- Using Doxygen, we aim to create user-friendly documentation similarly to Oracle 7 Java Documentation. Below, I demonstrate a documentation that is generated using Doxygen. As a trivial example, @Chu, John created an **input.c** file that has one function called ***add(int a, int b)***. This function simply produces the value that adds two parameters.

Commented [CJ1]: @Mishra, Pranav

Commented [CJ2]: @Chu, John

Commented [CJ3]: @Goda, Youssif

Commented [CJ4]: @Chu, John

Figure 2. Doxygen documentation.

Goal for the next week: Implement the AST from this trivial example.

Resources

Detailed Explanations about ANTLR

1. Involves the creation of a parsing grammar used to break down different C statements into their respective roles
2. Parsing grammar will operate on the code of a particular language. One thing to note is that the complexity of a parsing model increases with the sophistication of a given program.
3. As an example, let us see what an ANTLR4 model does on a simple “Hello World” C Program:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

4. An ANTLR4 grammar for this program would look like this (turn over to the next page)

```antlr
// Define lexer rules (similar to the previous example)

// Define parser rules
parser grammar CParser;

program: includeDirective functionDeclaration;

includeDirective: '#include' '<' ID '>' ';';

functionDeclaration: INT ID '(' ')' compoundStatement;

compoundStatement: '{' statementList '}';

statementList: statement*;

statement: 'printf' '(' STRING ')' ';'
         | 'return' NUM ';';

// Other grammar rules for declarations, expressions, etc. would be required
```

5. The result of an ANTLR4 parsing is an abstract syntax tree (in other words, a “boilerplate” of pseudocode whose characteristics are determined by the creator of the model and the program)

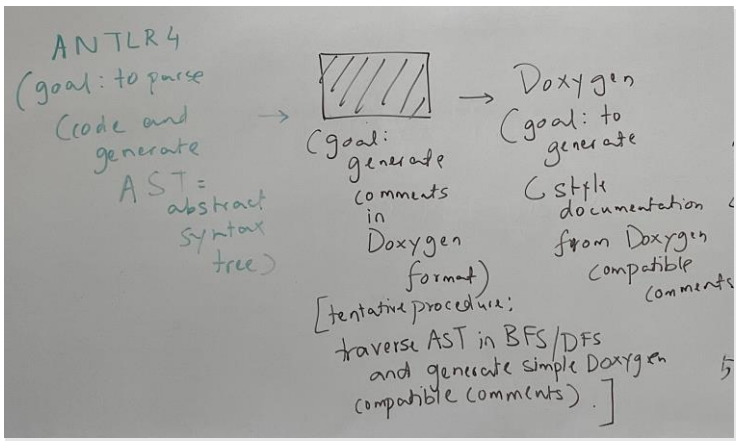
```scss
(program
  (includeDirective
    (#include
      <stdio.h>))
  (functionDeclaration
    (INT int)
    (ID main)
    ()
    (compoundStatement
      ({)
      (statementList
        (statement
          (printf
            (()
            ("Hello, World!\n")
            ()))
          (;))
        (statement
          (return
            (NUM 0))
          (;)))
      (}))
  (EOF))
```

Copy code

Reger

A possible pipeline for documentation generation, if the C Basic code has no comments (Suggested by @Mishra, Pranav).

Commented [CJ5]: @Mishra, Pranav



The black box in the middle is the tricky step, as it involves making changes to the original file by adding comments if they are not there already.

Possible Black Box Solution: spaCy

SpaCy is a python library used for generating docstrings and comments based on analyzing components of a code and the relationship between them. SpaCy's task would be simplified by running on the AST since it is a structured data structure that has already parsed the info in the program, making it easier for spaCy to analyze the relationship between components and generating more accurate comments, which can further enhance the accuracy of the documentation produced by doxygen.

Foreseaable problem: spaCy generates human-readable text, not Doxygen compatible text. Doxygen tags will have to be added to the comments generated by spaCy.

Thus, it will be necessary to analyze the spaCy generated comments, add the appropriate tags, format the comments and add them at the appropriate locations in the original code.

Step 3: Doxygen to generate documentation for C codebase.

1. Download the Doxygen at the https://www.doxygen.nl/download.html. If you scroll down, you will be able to see “Sources and Binaries” section. Download doxygen that corresponds to your OS. After downloading it, add its executable file to your System $PATH variable.
2. Clone the following GitHub repository: git clone https://github.com/doxygen/doxygen.git using the Visual Studio terminal.
3. Use the command ```cd doxygen ``` to go into a doxygen folder.
4. Use ``` mkdir build ``` command to create a build folder.
5. Use the command ``` cd build ``` to go into build folder.
6. Use the following command: ***cmake -G "Visual Studio 17 2022" ..***
7. Download flex here: Releases · lexxmark/winflexbison (github.com) and update the $PATH environment variable.
8. Now, create a Doxygen configuration file anywhere you want within the ```build ``` directory. In my case, I created ``` example codes ```folder inside ``` build/src```. Use the command ```doxygen –g```. If this command does not work, use the following command: "C:\Program Files\doxygen\bin\doxygen.exe" -g
9. Configure **Doxygen**. You can open the text editor of this file and change some settings. I will send you my Doxygen configuration file if you want to be at the same page with me.
10. Prepare an example C code and annotate it with the doxygen-style formatted comments.
11. Run the Doxygen using the following command: & "<PATH of your doxygen.exe file>" Doxyfile. Run this in the directory where your Doxyfile configuration is located at.

Future Work:

Thinking about what the “black-box” would comprise.

Please turn over

This is a pipeline that @Mishra, Pranav came up with, and @Chu, John agreed with this.

Commented [CJ6]: @Chu, John