Project Journal: Weeks 11-16 (1/28 - 3/9)

BLARG! Systems - Ryan Kispert

Weekly Goals:

Week 11:

- Continue working on lexer.
- Evaluate the status of the current lexer.

Week 12:

- Add literal recognition.
- Add keyword recognition to lexer.
- Begin researching Abstract Syntax Trees.

Week 13:

- Test all parts of lexer (unit testing).
- Add identifier recognition.

Week 14:

- Continue researching Abstract Syntax Trees.
- Sketch methods for AST structure.

Week 15:

- Begin implementing AST structs.
- Continue implementing AST structs.

Week 16:

- Start researching methods for AST generation.
- Finalize AST structs.

Research:

As I have been finishing my lexer (primarily just adding more recognition methods), I shifted to focus on the Syntax Analysis phase of the BLARG! compiler. Due to time constraints, I have already recognized that I will be unable to complete an entire compiler and will only be including the 3 essential steps for the compiler. A normal compiler will usually include 6 primary steps, those being Lexical Analysis (which I have just finished), Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Optimization, and Code Generation.[2] The Lexical Analysis stage is simply converting the input file into tokens the compiler can manipulate more easily. Syntax Analysis

checks to ensure the generated tokens meet the grammar rules for the language, and usually output the tokens in a new format referred to as an Abstract Syntax Tree (AST). An AST is a way of organizing the code into logical steps (called nodes) and may additionally make code generation easier. Normal compilers then perform Semantic Analysis, which essentially verifies that the code makes “sense”. The compiler will then generate a sort of “nearly-compiled” code in the Intermediate Code Generation step. This compiled code is sent through several optimization techniques, and then finally compiled into the final output code, usually machine code (assembly/binaries) or a low-level language. However, it is notable that the middle 3, (Semantic Analysis, Intermediate Generation, and Optimization) are primarily helper steps. These steps serve an important role in creating production-level output code, however, they are not critical for the primary function of a compiler: converting one type of code to another. Since these steps are more complicated than some of the others, and I have extremely limited time, I have elected to remove them from my compiler for now. This still means my next step is Syntax Analysis, and in turn producing an Abstract Syntax Tree, which is luckily not an overly daunting task. Again, I have turned to the *minic* repository by NikRadi for some ideas on how to proceed with this. Similar to my thoughts on the lexer they had created, the AstNode header[1] file seemed to be a bit redundant on certain parts, and I decided I could accomplish the same thing in one struct, rather than several. I intend to create a linked-list[2] structure for my nodes similar to *minic*, with an enum describing what kind of Node it is.

Accomplishments:

- Fixed a frustrating bug in reading a file thanks to my advisor’s notes

```
char buffer[BUFSIZE * sizeof(char)];
char* output = NULL;
int total = 0;

while (!feof(fp)) {  // Until the end of the file
        int read = fread(buffer, sizeof(char), BUFSIZE, fp);  // Read the next BUFSIZE characters from the file
        output = (char*)realloc(output, total + read);   // Note:  skipping error handling from realloc!
        memcpy(output + total, buffer, read);  // Copy what was read to the end of the output buffer
        total += read;  // Keep track of the total read
}
fclose(fp);

output = (char*)realloc(output, total + 1);   // Note:  skipping error handling from realloc!
output[total] = '\0';   // Null-terminate the string
filelen = strlen(output);

return output;
```

[1] A file containing definitions and declarations for C languages
[2] A data type containing nodes where each node contains data and a reference (link) to the next node in the sequence. This allows for dynamic memory allocation and efficient insertion and deletion operations compared to arrays. (https://www.geeksforgeeks.org/data-structures/linked-list/)

- Lexer is complete and tested

```
struct Lexer lexer;
lexer.source = buffer;
lexer.source_len = filelen;
lexer.index = 0;
lexer.line = 1;
lexer.line_index = 0;
lexer.token_cnt = 0;
lexer.tokens = (Token*)malloc((filelen) * sizeof(Token));

LexSource(&lexer);

free(lexer.source);

int i = 0;
for (int i = 0; i < lexer.token_cnt; ++i) {
        printf("%d | ", lexer.tokens[i].type);
}

free(lexer.tokens); // TEMPORARILY ensure token memory is released
```

- Started outlining what the Syntax Analysis phase will include
- Outlined Abstract Syntax Tree structure

Reflection:

I have made some major progress these weeks, and I am in turn much more on track to have a solid presentation by the end of the year. However, I still am concerned about my ability to have a fully functioning compiler by the end of the year due to my schedule constraints with college visits, leadership projects, and other classes. I do believe that, as I stated in my initial presentation, even without a complete prototype I will have learned enough to present an interesting and unique project.

Bibliography:

1. *NikRadi. (n.d.). Nikradi/minic: A C-compiler written in C. GitHub. https://github.com/NikRadi/minic/tree/master*
2. *GfG. (2023, November 8). Phases of a compiler. GeeksforGeeks. https://www.geeksforgeeks.org/phases-of-a-compiler/*