Compiler Construction Mini - Project

By

136768 Nyaga Phil

128693 Monari Maria

135358 Sheldon Ngetich

135904 Tricia Nyoike

130002 Esther Gacheru

131128 Sharon Wendoh

134676 Beryl Mase

Introduction to HTML Lexer Mini-Project

This mini-project focuses on the development of a lexer for parsing HTML content, implemented in C. It reads through input text—in this case, HTML—and breaks it down into meaningful units called tokens.

1. **Tokenization**: The core functionality of this project is to scan HTML content and tokenize it. Tokens here represent various elements of HTML, such as start tags, end tags, text content, comments, self-closing tags, doctype declarations, and more.

2. **Token Types**: The lexer classifies these tokens into distinct categories, each representing different aspects of HTML syntax. This categorization is crucial for the subsequent parsing stage, where the syntactic structure of the HTML is analyzed.

3. **Buffer Management**: Efficient handling of text data is achieved through dynamic buffer management. The buffer dynamically adjusts its size to accommodate the text being read, ensuring memory efficiency and scalability.

4. **File Handling**: The project includes file I/O operations to read HTML content from a file, demonstrating practical utility in real-world scenarios where HTML content often resides in files.

5. **Modularity and Extensibility**: The code structure is modular, making it easier to understand, maintain, and extend. For instance, the lexer can be easily adapted to include more complex features or to handle different types of input..

IMPLEMENTATION

Here's how the simplification might look in a basic lexer:

1. **Define Token Types**: You would have fewer token types.

c
```c
typedef enum {
    TOKEN_START_TAG,
    TOKEN_END_TAG,
    TOKEN_TEXT,
    TOKEN_COMMENT,
    TOKEN_SELF_CLOSING_TAG,
    TOKEN_DOCTYPE,
    TOKEN_EOF,  // End of file token
    TOKEN_UNKNOWN  // Any unrecognized token
} TokenType;
```

2. **Define Token Structure**: The token structure remains the same.

c
```c
typedef struct {
    TokenType type;
    char* value;
} Token;
```

3. **Reading Input**: The function to read the next character remains unchanged.

c
```c
int next_char(FILE *input) {
    return fgetc(input);
}
```

4. **Tokenize Input**: The tokenization function now ignores anything related to attributes.

c
```c
Token next_token(FILE *input) {
    Token token;
```

```
    int ch;
    init_buffer();

    while ((ch = next_char(input)) != EOF) {
        switch (ch) {
            case '<':
                // Look ahead to determine if it's a start tag, end tag, comment, or doctype
                break;
            case '>':
                // Finalize the current tag token
                token.type = TOKEN_START_TAG; // or TOKEN_END_TAG based on context
                token.value = strdup(buffer);
                return token;
            default:
                // Handle text content outside of tags
                append_to_buffer(ch);
                break;
        }
    }

    // Handle the case where the last token is text
    if (buffer_size > 0) {
        token.type = TOKEN_TEXT;
        token.value = strdup(buffer);
        return token;
    }

    // Signal end of file
    token.type = TOKEN_EOF;
    token.value = NULL;
    return token;
}
```

5. **Buffer Management**: Buffer management can be simplified as well since you're not handling attributes.

c

```
char* buffer;
int buffer_size;
int buffer_capacity;
```

```
// Buffer functions remain unchanged from previous example
```

6. **Main Loop**: The main loop to process tokens also remains unchanged.

c
```
int main(int argc, char *argv[]) {
    FILE *input = fopen("input.html", "r");
    if (input == NULL) {
        // Handle file opening failure
        return 1;
    }

    init_buffer();
    Token token;
    do {
        token = next_token(input);
        // Process the token, e.g., print it or convert it to Markdown
        if (token.value != NULL) {
            free(token.value); // Remember to free the token value to avoid memory leaks
        }
    } while (token.type != TOKEN_EOF);

    free(buffer);
    fclose(input);
    return 0;
}
```

Connection Between Solution and Concepts Learned in Class

1. **Lexical Analysis:**
    - In the context of the HTML-to-Markdown solution, lexical analysis involves breaking down the input HTML document into a sequence of tokens or lexical units. These tokens represent fundamental elements of the HTML language, such as tags, attributes, text content, and comments.
2. **Token Types:**
    - You can define token types that correspond to the different elements and constructs in HTML, such as START_TAG, END_TAG, TEXT_CONTENT, EOF, and COMMENT. These token types align with the lexical analysis phase.

3. **Parsing:**
    - Recursive descent parsing, a top-down parsing technique, can be used to parse HTML. It involves writing parsing functions for each non-terminal symbol in the grammar, such as <html>, <body>, <p>, etc. Recursive descent parsing aligns with the parsing phase.
4. **Semantic Analysis:**
    - While HTML is not a strongly typed language like programming languages, you may perform a form of semantic analysis to ensure the logical correctness of the HTML structure. For example, checking that opening and closing tags match appropriately.