50.051 Programming Language Concepts

W10-S1 Tokenization (Part 2)

Matthieu De Mari

Where are we right now?

We have functions that:

- Scan a source.c file for code and puts it into a string,

- Splits the string using whitespaces and \n symbols to produce lexemes,

- Checks those lexemes to recognize, using RegEx, if each lexeme matches a vast collection of keywords, operators, punctuation signs, identifiers or literals, or is an unknown type of lexeme,

- In case of a match, create a token which assembles the TOKEN_TYPE that has been recognized and the lexeme,

- And, if no type is recognized, assign a default TOKEN_UNKNOWN type.

Where are we right now?

On this lecture,

- Relaxing the hypothesis that all elements in code are nicely separated with whitespaces or \n symbols.

- Conflict resolution in the case of ambiguous tokens that could be classified as more than one type.

- Error handling in the case of incorrect/unknown lexemes.

- Recognizing and dropping comments.

- (If time allows, pre-processing)

- Closing on the topic of Tokenization.

Important hypothesis we made

Hypothesis: The source code will have all the substrings that will become our future lexemes, clearly separated using whitespaces and \n symbols.

This will allow for a very simple string splitting to produce lexemes.

- In practice, almost never true, and will have to go.

  → **Left-to-right scan, character by character, to produce lexemes.**

```c
#include <stdio.h>
int main()
{
    int num = 1;
    while (num <= 10)
    {
        printf("The number is: %d\n", num);
        num++;
    }
    return 0;
}
```

An intuition for an updated splitting procedure

- If this hypothesis goes out the window...
- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "i"
Token types it could be: identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window...

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "in"
Token types it could be: identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "int"
Token types it could be: keyword, identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "int " (with a white space!)
Token types it could be: **nothing!**

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "int" (**backtrack** to last lexeme that had at least one valid type)
Token types it could be: keyword, identifier

Addressing lexical ambiguities

The lexeme "int", could technically be of **two possible types**:

- It could be the **keyword** int, used to declare an integer type of variable,

- It could be a variable/function/class/... name, or in other words, an **identifier** of some sort.

In general, most programming languages will forbid using keywords as identifiers. This is easily fixed by **enforcing a token type priority**!

Keywords, Operators, Punctuation > Identifiers, Literals

An intuition for an updated splitting procedure

- If this hypothesis goes out the window...

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

"int" lexeme produced and recognized as type KEYWORD_INT.

Current lexeme = "int" (**backtrack** to last lexeme that had at least one valid type)
Token types it could be: **keyword**, identifier (**resolved by priority rule**)

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "" (**lexeme has been categorised**, **discard** it, and **continue**)
Token types it could be: nothing

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = " " (single whitespace?)
Token types it could be: Whitespace type if Python? Nothing in C?

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

$$int\ fort= 17;$$

Current lexeme = "" (irrelevant in the case of C, **discard**)
Token types it could be: Whitespace type if Python? Nothing in C?

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "f"
Token types it could be: identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "fo"
Token types it could be: identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "for"
Token types it could be: keyword, identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "for" (**shall we call that a lexeme and categorise it now?**)
Token types it could be: keyword, identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window...
- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "for" (shall we call that a lexeme and categorise it now? **No, continue**)
Token types it could be: keyword, identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "fort"
Token types it could be: identifier

An intuition for an updated splitting procedure

- If this hypothesis goes out the window…
- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

Current lexeme = "fort="
Token types it could be: **nothing**

An intuition for an updated splitting procedure

- If this hypothesis goes out the window...

- No other choice but to scan source code string, character by character and build our lexemes on-the-fly!

int fort= 17;

"fort" lexeme produced and recognized as type IDENTIFIER.

Current lexeme = "fort" (**backtrack** to last lexeme that had at least one valid type)
Token types it could be: identifier

Maximal Munch Algorithm

Maximal Munch idea: scan one character at a time, from left to right, always try to produce the longest lexeme as possible.

- **As long as one token type can be matched to the current lexeme, keep scanning.**

- **When you produce a lexeme that cannot fit any token type, backtrack to the last valid one and assign a token type.**

- **If the last valid lexeme could be classified as multiple token types, use our priority rule from earlier on token types.**

- **Ignore and discard any lexeme that consists of a single irrelevant character (e.g. whitespace or \n character).**

Maximal Munch Algorithm

This basically means two things

1. **We will have to reuse our functions that were checking if a given lexeme could fit a given token type, using RegEx or *strcmp()*.**

2. **We will no longer treat the string splitting task (which produces lexemes to be categorised) and the lexeme classification as two different tasks to be done sequentially.**

→ **Instead, the lexeme classification will happen at the same time as the splitting of the code string, on-the-fly.**

*(**Note:** Ultimately, this means that we will have to scan many more lexemes than before. In turn, it will make the tokenizer slower.*
But that is the price to pay to relax the whitespace hypothesis!)

Error handling for invalid tokens

In our previous lecture, we mentioned, as a challenge, a Tokenizer v1.2, which had a **line-tracking feature for handling errors**.

In short, this feature requires,

- To **add an extra attribute to our Token objects**, more specifically, the **line at which said lexeme appears** in the code.

- During the left-to-right scan, we **keep track of how many \n symbols** we have encountered and **use a counter *line_number*** to do so.

- If a Token is not recognized (e.g. a weird character like "@" appears in a C code and produces a TOKEN_UNKNOWN type), an **error message will show the problematic token, lexeme and line it appears at**.

In Code files/1.
Implement a left-to-right scan, character by character, with maximal munch

Step1: Prepare variables for tracking.
Step2: Implement a loop going through characters.

```c
267    // Prepare maximal munch tracking variables
268    size_t source_code_length = strlen(source_code);
269    size_t current_position = 0;
270    int current_line_number = 1;
271
272    // A possible maximal munch implementation
273    while (current_position < source_code_length) {
274        // Current lexeme candidate to be matched
275        TokenType token_type = TOKEN_UNKNOWN;
276        size_t longest_match = 0;
277        char matched_lexeme[256] = {0};
278
279        // Left to right scan producing candidate lexemes and testing them
280        for (size_t i = current_position; i < source_code_length; ++i) {
281            char candidate_lexeme[256] = {0};
282            memcpy(candidate_lexeme, source_code + current_position, i - current_position + 1);
283            TokenType candidate_token_type = TOKEN_UNKNOWN;
284            size_t candidate_match_length = i - current_position + 1;
285
```

Note: we use the short-circuit evaluation property of the OR statements in the if() to only compute the first non-unknown token and assign it to candidate_token_type.

```c
// If at least one match on one of the possible token types,
// candidate_token_type will be the first token match found.
// (This is the reason why we have ordered our enum a certain way)
// Additional note: In the given if statement, we will evaluate each condition
// one by one using short-circuit evaluation.
// As soon as one condition evaluates to true (i.e., the candidate_token_type is not TOKEN_UNKNOWN),
// the remaining conditions will not be evaluated, and the if block will execute.
// In this case, the candidate_token_type variable will have the value of the first
// "non-TOKEN_UNKNOWN" token type found during the evaluation of the conditions.
// The other conditions will not execute after that, and the code inside the if block will be executed
// using the found token_type and the corresponding matched_lexeme.
if ((candidate_token_type = is_keyword_strcmp(candidate_lexeme)) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_operator_strcmp(candidate_lexeme)) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_punctuation_strcmp(candidate_lexeme)) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_identifier_regex(candidate_lexeme) ? TOKEN_IDENTIFIER : TOKEN_UNKNOWN) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_integer_literal_regex(candidate_lexeme) ? TOKEN_LITERAL_INT : TOKEN_UNKNOWN) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_float_literal_regex(candidate_lexeme) ? TOKEN_LITERAL_FLOAT : TOKEN_UNKNOWN) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_char_literal_regex(candidate_lexeme) ? TOKEN_LITERAL_CHAR : TOKEN_UNKNOWN) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_string_literal_regex(candidate_lexeme) ? TOKEN_LITERAL_STRING : TOKEN_UNKNOWN) != TOKEN_UNKNOWN) {
        token_type = candidate_token_type;
        longest_match = candidate_match_length;
        strcpy(matched_lexeme, candidate_lexeme);

}
```

(There are obviously other ways to implement this that are a bit more readable, but meh.)

```c
// If at least one match on one of the possible token types,
// candidate_token_type will be the first token match found.
// (This is the reason why we have ordered our enum a certain way)
// Additional note: In the given if statement, we will evaluate each condition
// one by one using short-circuit evaluation.
// As soon as one condition evaluates to true (i.e., the candidate_token_type is not TOKEN_UNKNOWN),
// the remaining conditions will not be evaluated, and the if block will execute.
// In this case, the candidate_token_type variable will have the value of the first
// "non-TOKEN_UNKNOWN" token type found during the evaluation of the conditions.
// The other conditions will not execute after that, and the code inside the if block will be executed
// using the found token_type and the corresponding matched_lexeme.
if ((candidate_token_type = is_keyword_strcmp(candidate_lexeme)) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_operator_strcmp(candidate_lexeme)) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_punctuation_strcmp(candidate_lexeme)) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_identifier_regex(candidate_lexeme) ? TOKEN_IDENTIFIER : TOKEN_UNKNOWN) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_integer_literal_regex(candidate_lexeme) ? TOKEN_LITERAL_INT : TOKEN_UNKNOWN) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_float_literal_regex(candidate_lexeme) ? TOKEN_LITERAL_FLOAT : TOKEN_UNKNOWN) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_char_literal_regex(candidate_lexeme) ? TOKEN_LITERAL_CHAR : TOKEN_UNKNOWN) != TOKEN_UNKNOWN ||
    (candidate_token_type = is_string_literal_regex(candidate_lexeme) ? TOKEN_LITERAL_STRING : TOKEN_UNKNOWN) != TOKEN_UNKNOWN) {
        token_type = candidate_token_type;
        longest_match = candidate_match_length;
        strcpy(matched_lexeme, candidate_lexeme);
}
```

The for loop stops when a lexeme has been found, either
- **a maximal length one has been recognized,**
- **or a problematic one has been recognized,**
- **or a single white space character of some sort has been recognized.**

Either way, something is supposed to happen with this lexeme.

```c
// After we have found our candidate lexeme, create token
if (token_type != TOKEN_UNKNOWN) {
Token *token = create_token(token_type, matched_lexeme, current_line_number);
    token_stream = (Token **)realloc(token_stream, (token_count + 1) * sizeof(Token *));
    token_stream[token_count] = token;
    token_count++;
    printf("Token { type: %d, lexeme: '%s', line: '%d'}\n", token->type, token->lexeme, token->line_number);
    current_position += longest_match;
} else {
    // Update the line number when encountering a newline character
    if (source_code[current_position] == '\n') {
      current_line_number++;
    }
    // Show error message, unless we have a whitespace character or some sort
    if (source_code[current_position] != ' ' && source_code[current_position] != '\n' &&
        source_code[current_position] != '\t' && source_code[current_position] != '\r') {
      printf("Error: Unrecognized character '%c' at line %zu\n", source_code[current_position], current_position);
    }
    ++current_position;
}
```

Adjusting our Token object to include a line_number attribute.
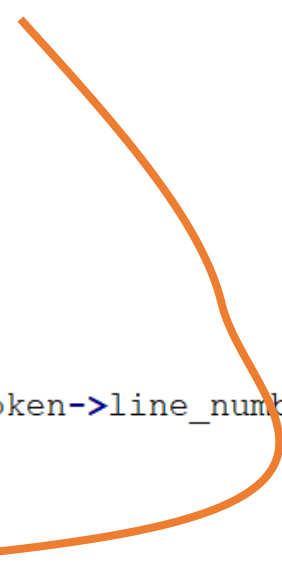
```c
40    // Our Token object
41    typedef struct {
42        TokenType type;
43        char *lexeme;
44        int line_number;
45    } Token;
46
47
48    // Constructor for the Token struct
49    Token *create_token(TokenType type, const char *lexeme, int line_number) {
50        Token *token = (Token *)malloc(sizeof(Token));
51        token->type = type;
52        token->lexeme = strdup(lexeme);
53        token->line_number = line_number;
54        return token;
55    }
56
57
58    // Destructor for the Token struct
59    void free_token(Token *token) {
60        free(token->lexeme);
61        free(token);
62    }
63
```

Keep track of a line_number.

```c
int main() {
    // Load source code from file
    // For testing, will put code string in source_code string directly (simpler).
    // You can uncomment/comment things as needed, if you want to play with a source.c file instead.
    /*
    const char *filename = "source.c";
    char *source_code = read_source_code(filename);
    if (source_code == NULL) {
        fprintf(stderr, "Error reading source code from '%s'\n", filename);
        return 1;
    }
    */
    char *source_code = strdup("int x =7 float 2.5e-1 char 'a' char* \"hello\"; \n"
                              "while( for) if== return 1t0 +- @ * / & &&;");

    // Create tokens stream
    Token **token_stream = NULL;
    size_t token_count = 0;

    // Prepare maximal munch tracking variables
    size_t source_code_length = strlen(source_code);
    size_t current_position = 0;
    int current_line_number = 1;
```

In maximal munch procedure, increase line number when \n is encountered

```c
// After we have found our candidate lexeme, create token
if (token_type != TOKEN_UNKNOWN) {
Token *token = create_token(token_type, matched_lexeme, current_line_number);
    token_stream = (Token **)realloc(token_stream, (token_count + 1) * sizeof(Token *));
    token_stream[token_count] = token;
    token_count++;
    printf("Token { type: %d, lexeme: '%s', line: '%d'}\n", token->type, token->lexeme, token->line_number);
    current_position += longest_match;
} else {
    // Update the line number when encountering a newline character
    if (source_code[current_position] == '\n') {
        current_line_number++;
    }
    // Show error message, unless we have a whitespace character or some sort
    if (source_code[current_position] != ' ' && source_code[current_position] != '\n' &&
        source_code[current_position] != '\t' && source_code[current_position] != '\r') {
        printf("Error: Unrecognized character '%c' at line %zu\n", source_code[current_position], current_position);
    }
    ++current_position;
}
```
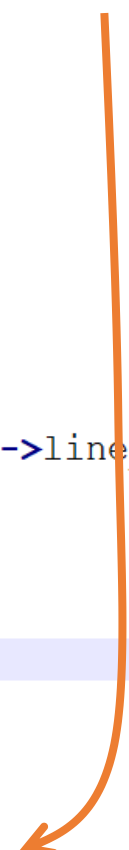
Upon creating a token, use current line number.

```c
// After we have found our candidate lexeme, create token
if (token_type != TOKEN_UNKNOWN) {
Token *token = create_token(token_type, matched_lexeme, current_line_number);
    token_stream = (Token **)realloc(token_stream, (token_count + 1) * sizeof(Token *));
    token_stream[token_count] = token;
    token_count++;
    printf("Token { type: %d, lexeme: '%s', line: '%d'}\n", token->type, token->lexeme, token->line_number);
    current_position += longest_match;
} else {
    // Update the line number when encountering a newline character
    if (source_code[current_position] == '\n') {
        current_line_number++;
    }
    // Show error message, unless we have a whitespace character or some sort
    if (source_code[current_position] != ' ' && source_code[current_position] != '\n' &&
        source_code[current_position] != '\t' && source_code[current_position] != '\r') {
        printf("Error: Unrecognized character '%c' at line %zu\n", source_code[current_position], current_position);
    }
    ++current_position;
}
```

Error message shows problematic token along with its line number

```c
// After we have found our candidate lexeme, create token
if (token_type != TOKEN_UNKNOWN) {
Token *token = create_token(token_type, matched_lexeme, current_line_number);
    token_stream = (Token **)realloc(token_stream, (token_count + 1) * sizeof(Token *));
    token_stream[token_count] = token;
    token_count++;
    printf("Token { type: %d, lexeme: '%s', line: '%d'}\n", token->type, token->lexeme, token->line_number);
    current_position += longest_match;
} else {
    // Update the line number when encountering a newline character
    if (source_code[current_position] == '\n') {
        current_line_number++;
    }
    // Show error message, unless we have a whitespace character or some sort
    if (source_code[current_position] != ' ' && source_code[current_position] != '\n' &&
        source_code[current_position] != '\t' && source_code[current_position] != '\r') {
        printf("Error: Unrecognized character '%c' at line %zu\n", source_code[current_position], current_position);
    }
    ++current_position;
}
```

```
char *source_code = strdup
("int x =7 float 2.5e-1 char 'a' char* \"hello\"; \n"
"while( for) if== return 1t0 +- @ * / & &&;");
```

Our Tokenizer v1.3 with
- **Line tracking,**
- **More types of tokens,**
- **Left-to-right scan,**
- **Maximal Munch,**
- **Error messages for unknown tokens,**

Works!

```
Token { type: 0, lexeme: 'int', line: '1'}
Token { type: 19, lexeme: 'x', line: '1'}
Token { type: 12, lexeme: '=', line: '1'}
Token { type: 20, lexeme: '7', line: '1'}
Token { type: 1, lexeme: 'float', line: '2'}
Token { type: 21, lexeme: '2.5e-1', line: '2'}
Token { type: 2, lexeme: 'char', line: '2'}
Token { type: 22, lexeme: ''a'', line: '2'}
Token { type: 3, lexeme: 'char*', line: '2'}
Token { type: 23, lexeme: '"hello"', line: '2'}
Token { type: 16, lexeme: ';', line: '2'}
Token { type: 4, lexeme: 'while', line: '4'}
Token { type: 14, lexeme: '(', line: '4'}
Token { type: 5, lexeme: 'for', line: '4'}
Token { type: 15, lexeme: ')', line: '4'}
Token { type: 6, lexeme: 'if', line: '4'}
Token { type: 13, lexeme: '==', line: '4'}
Token { type: 7, lexeme: 'return', line: '4'}
Token { type: 20, lexeme: '1', line: '4'}
Token { type: 19, lexeme: 't0', line: '4'}
Token { type: 8, lexeme: '+', line: '4'}
Token { type: 9, lexeme: '-', line: '4'}
Error: Unrecognized character '@' at line 102
Token { type: 10, lexeme: '*', line: '4'}
Token { type: 11, lexeme: '/', line: '4'}
Token { type: 17, lexeme: '&', line: '4'}
Token { type: 18, lexeme: '&&', line: '4'}
Token { type: 16, lexeme: ';', line: '4'}
```

Limitations to the current code

Our current maximal munch will detect invalid characters, such as @, but will not necessarily detect invalid literals.
Instead, it will split identifiers with invalid names.

- For instance, it will split the lexeme "1t00" into LITERAL(1) and IDENTIFIER(t00), instead of a single UNKNOWN(1t00).

- Per se, it is not that bad, as **it will be caught by PARSER** on the next step: After all, literals cannot be followed by identifiers.

- So it is up to us to decide who should raise the error: the tokenizer or the parser.

Recognizing typical errors

We could, however, recognize some typical errors, by using RegEx to flag them.

For instance, we could create a token type for lexemes that start with at least one digit and are then followed by at least one non-digit character.

^[0-9]+[^0-9]+(.)$*

When recognized, raise errors, as shown earlier.

The tokenizer could also recognize that a semicolon is missing at the end of a given statement and add it to the code.

```
int main() {
    int x = 10
    int y = 20;
    return x + y;
}
```

Advanced Error Handling in Tokenization

Some advanced tokenizers might even attempt to fix typical errors!

But they are usually part of a larger system, such as an IDE or a code editor with syntax-checking features.

- These advanced tokenizers work in conjunction with other components like parsers and error recovery mechanisms to detect and correct common errors in source code.

- When the tokenizer encounters an invalid lexeme, it might try to guess the intended token based on the surrounding context or by applying heuristics.

- In difficult cases, it might even flag the error for further processing, leaving it to the parser to figure it out.

Recognizing and discarding comments

At the moment, our tokenizer is **not correctly recognizing comments.**

- **Comments should be discarded:** we shall not create tokens for comments.

- Instead, we should recognize when a line comment appears, by recognizing two // in succession

- After that, we should then discard any scanned character until a \n is seen, indicating the end of the comment.

- *(To some extent, we would do the same thing with block comments using /* and */ as beginning and end of comment symbols).*

**In Code files/2.
We have written a function
that will scan the source
code and replace comments
with white spaces.**

```cpp
// Will replace all line comments with whitespaces
// Every time a // character is recognized, keep replacing
// characters with whitespaces, until a \n character is seen,
// indicating the end of the comment.
// Note: Does not cover block comments.
void remove_line_comments(char *source_code) {
    size_t source_code_length = strlen(source_code);
    bool in_comment = false;

    for (size_t i = 0; i < source_code_length; ++i) {
        if (source_code[i] == '/' && source_code[i + 1] == '/') {
            in_comment = true;
        }


        if (in_comment && source_code[i] != '\n') {
            source_code[i] = ' '; // Replace comment characters with white spaces.
        } else {
            in_comment = false;
        }
    }
}
```

It is then used in our main, after the code has been retrieved from the source file.

```c
// Tokenizer v1.1
// Almost same main() as before
int main() {
    // Load source code from file
    // For testing, will put code string in source_code string directly (simpler).
    // You can uncomment/comment things as needed, if you want to play with a source.c file instead.
    /*
    const char *filename = "source.c";
    char *source_code = read_source_code(filename);
    if (source_code == NULL) {
        fprintf(stderr, "Error reading source code from '%s'\n", filename);
        return 1;
    }
    */
    char *source_code = strdup("int x =7 // comment\nfloat 2.5e-1 char 'a' char* \"hello\"; \n"
                              "// another \n while( for) if== return 1t0 +- @ * / & &&;");

    // Discard comments
    remove_line_comments(source_code);

    // Create tokens stream
    Token **token_stream = NULL;
    size_t token_count = 0;
```

Pre-processing commands

Pre-processing commands (or pre-processor directives) are instructions for the C pre-processor that are evaluated **before the actual compilation process**.

During tokenization, we have a few options for handling pre-processor directives:

- **Simply ignore them:** If your goal is to tokenize the C source code without considering the effects of pre-processing directives, you can simply ignore them during tokenization.
  You can remove pre-processor directives by replacing their characters with spaces, similar to how we handled comments.

Pre-processing commands

Pre-processing commands (or pre-processor directives) are instructions for the C pre-processor that are evaluated **before the actual compilation process**.

During tokenization, we have a few options for handling pre-processor directives:

- **Tokenize them:** If you want to recognize pre-processing directives and include them as tokens in the token stream, you can extend the tokenizer to create tokens for pre-processing directives.

Pre-processing commands

Pre-processing commands (or pre-processor directives) are instructions for the C pre-processor that are evaluated **before the actual compilation process**.

During tokenization, we have a few options for handling pre-processor directives:

- **Fully process them:** If you want to build a complete C compiler, you should pre-process the source code **before** tokenization. In this case, you would probably implement a separate pre-processor module that handles pre-processor directives as specified by the C standard.

This is a more complex approach and requires to study what is the C pre-processor doing, which is <u>out-of-scope</u>.

How about function imports?

And, in the case of function imports, what would happen?

- The tokenizer's main job is to break the source code into a stream of tokens.

- If you plan on building a more complete compiler or analyser tool, you will need to handle header files and their contents.

- In such cases, you should consider implementing a separate pre-processor module, which processes the #include directives and other pre-processing commands before tokenization (again, this is <u>out-of-scope</u>!)

How about function imports?

To handle imports from other files, the pre-processor will have to:

- Detect #include directives in the source code.

- Resolve the paths of the included files (either local or from standard libraries in a different folder), and raise errors if invalid paths or files not found.

- Read the contents of the included files and replace the #include directives with the contents of the included files.

How about function imports?

After the pre-processing of the source code, we will have resolved all #include directives.

- The resulting pre-processed source code will have replaced all its include/function imports in the source file with the definition of said functions it has retrieved from additional files.

- This new file combining everything is then passed to the tokenizer.

- This way, the tokenizer will process the entire code as a single file, and it will generate a token stream for the entire program, as before.

So, what happens when #include <stdio.h>?

- The C pre-processor searches for the stdio.h header file. The search is usually performed in predefined locations, such as system directories and/or any directories specified by the -I option when compiling.

- Once the pre-processor finds the stdio.h file, it reads its contents.

- The pre-processor replaces the #include <stdio.h> directive in our source code with the contents of the stdio.h file. This process effectively merges the declarations and definitions from the stdio.h header into our source code.

- The modified source code (with stdio.h contents) is passed to the compiler, which compiles the program.

- The compiler uses the declarations from stdio.h to ensure correct usage of functions and data structures provided by the standard library.

Practice 1: Removing block comments

Following the logic of the code shown in class, how would you remove block comments from the source code now?

- We provide a template in Code files/3.

- The function to modify starts on line 110 and is called in the main(), after the line comments have been removed.

- *(Solution is provided in Code files/4.)*

Challenge 1: Would you raise an error message if a block comment starts /* but does not end, i.e. a */ is not found after a /*?
How about if a */ is found before a /*?

Challenge 2: Linking additional files

If the source code in source.c suggests importing a function from a source2.c file, how would you

- Read the code in source.c,

- Read the code in source2.c,

- And eventually replace the include command in the source.c file, with contents from the source2.c file?

Could conflicts happen between functions defined in both files?
If so, how would you handle them?

(Good luck coding it, if you feel like!)