

50.051 Programming Language Concepts

W9-S3 Tokenization (Part 1)

Matthieu De Mari



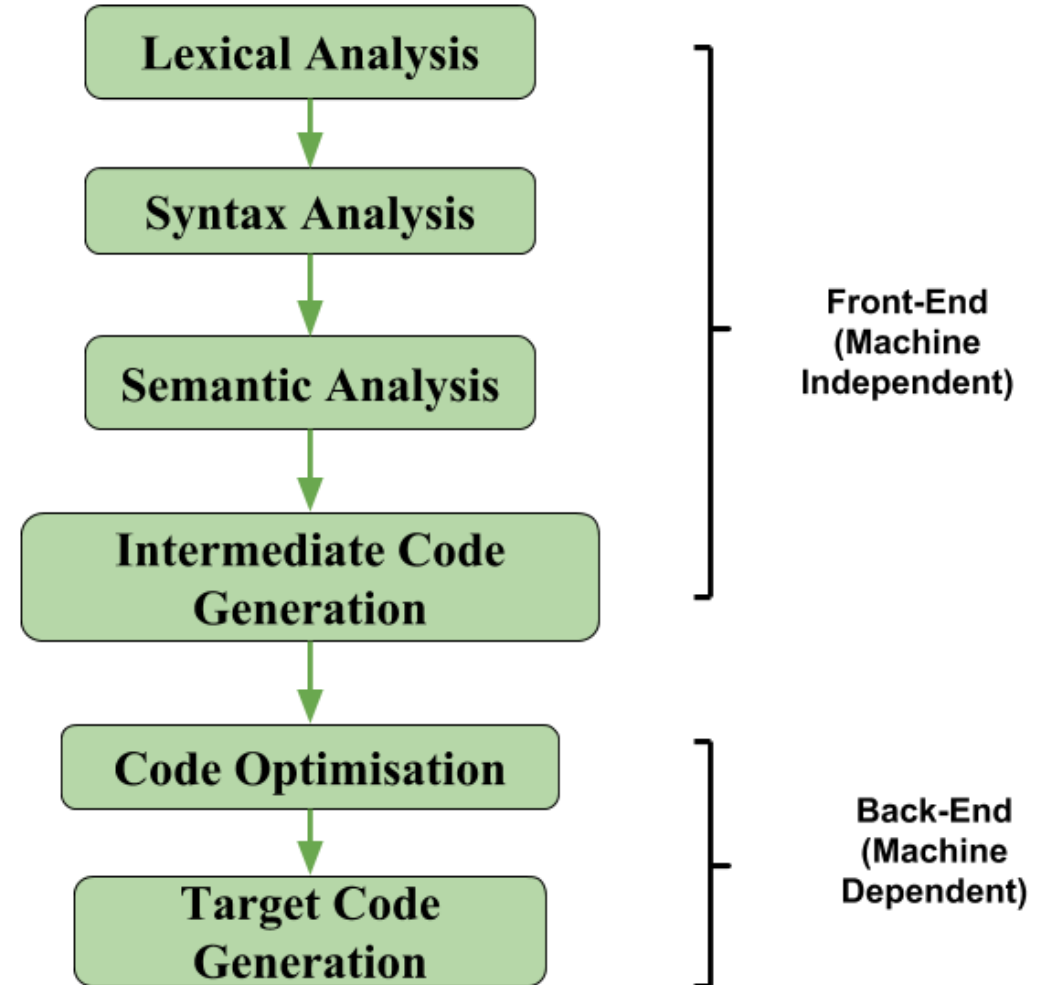
The front-end of a compiler

Definition (The front-end part of a compiler):

The **front-end of a compiler** is responsible for analysing the source code, and converting it into a form that can be used by the rest of the compiler.

It involves tasks, such as:

- **Lexical analysis,**
- **Syntax analysis,**
- and **Semantic analysis.**



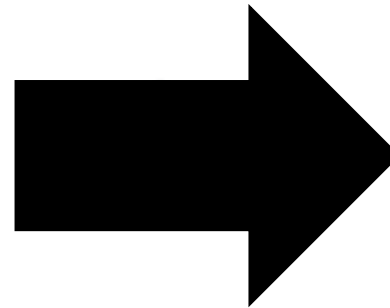
Lexical Analysis

Definition (**Lexical Analysis**):

During **lexical analysis**, the source code is broken down into **tokens**, which represent the individual components of the language.

It is sometimes referred to as **scanning** or **tokenization**.

```
int min(int firstNumber, int secondNumber)
{
    if (firstNumber > secondNumber) {
        return secondNumber;
    }
    else (
        return firstNumber;
    )
}
```



KEYWORD (int)
IDENTIFIER (min)
PUNCTUATION (open_par)
KEYWORD (int)
VARIABLE (firstNumber)
...

Lexical Analysis

Definition (**Tokens**):

Tokens are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Keyword:** A keyword is a **reserved word** in a programming language that **has a special meaning** and cannot be used as an identifier.

Examples of keywords in the C programming language:

- int, double, long, ...
- if, else, while, ...
- return, ...
- etc.

Lexical Analysis

Definition (**Tokens**):

Tokens are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Identifier:** An identifier is a **name given to a variable, function, or other entity** in a program. Will follow rules, such as starting with a letter or underscore and consisting of letters, digits, and underscores.

Examples of identifiers:

- x, counter, variable_1,
- myFunction,
- etc.

Lexical Analysis

Definition (**Tokens**):

Tokens are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Literal:** A **value** of some sort to be **assigned to a variable**.

Examples of literals in the C programming language:

- 12542
- 12654165.52
- “hello”
- Etc.

Lexical Analysis

Definition (**Tokens**):

Tokens are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Operator:** An operator is a **symbol that performs a specific operation** on one or more values.

Examples of operators in the C programming language:

- +, -, *, /,
- =,
- &&,
- etc.

Lexical Analysis

Definition (**Tokens**):

Tokens are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Punctuation symbol:** A punctuation symbol is a **symbol used to separate or group different parts of a program**

Examples of punctuation symbols in the C programming language

- Braces {} and parentheses ()
- Commas, semicolons,
- Quotation marks,
- Etc.

Steps for Lexical Analysis

Definition (Steps for **Lexical Analysis**):

During lexical analysis, the compiler **reads the source code character by character** and identifies each token.

- It does so, based on its **position** and **context** within the code.
- The compiler also uses **a set of rules or patterns** called **regular expressions** (something for Week 9), **to recognize different types of tokens**, such as keywords, identifiers, operators, and punctuation symbols.

Once the compiler has identified the tokens, it **assigns each token a specific type/category**, based on its role and meaning in the program.

First assumptions for our tokenizer

To keep things simple, we will make basic assumptions about the source code to be tokenized and how it has been written.

Some of them might be unrealistic, but we will relax them later on.

- White spaces between all tokens, so that we can use a simple split operation using whitespaces and `\n` symbols as separators, to produce all the substrings to be analysed and used as tokens. We will eventually relax this constraint later on.
- Start with basic tokens, e.g. keywords only
Later on, we will add new token types, starting with simple characters/operators (+ * - / ;), and then identifiers/literals.
- No comments or include/pre-processing operations for now.

Scanning the source code, step by step

In this example, we demonstrate how to read a source code string from a source file named "source.c". For instance, the code could simply consist of

```
int x = 1023 ;
```

Or

```
int while for if return intwhile
```

We use simple file I/O operations, as described on the right.

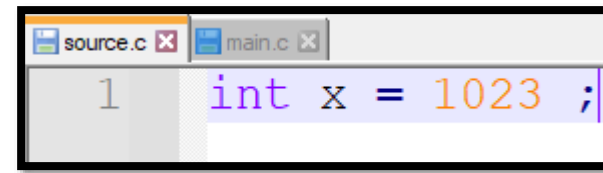
- Open the file with `fopen()`.
- Find the file size by seeking to the end and getting the current position with `ftell()`.
- Rewind the file position to the beginning with `rewind()`.
- Allocate memory for the source code string.
- Read the content of the file into the string with `fread()`.
- Null-terminate the string.
- Close the file with `fclose()`.

From Code files/1.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  char* read_source_code(const char *filename) {
5      // Open the file in filename, in read mode
6      // Check for opening errors
7      FILE *file = fopen(filename, "r");
8      if (file == NULL) {
9          fprintf(stderr, "Error opening file: %s\n", filename);
10         exit(EXIT_FAILURE);
11     }
12
13     // Move the file position indicator to the end of the file
14     // And get the current file position (which is the size of the file)
15     fseek(file, 0, SEEK_END);
16     long file_size = ftell(file);
17
18     // Now that we know the length of the file,
19     // Rewind the file position indicator back to the beginning of the file
20     rewind(file);
```

```
21
22 // Allocate memory for the source code string
23 // (file_size + 1 for the null-terminator)
24 char *source_code = (char *)malloc((file_size + 1) * sizeof(char));
25 if (source_code == NULL) {
26     fprintf(stderr, "Error allocating memory for source code\n");
27     exit(EXIT_FAILURE);
28 }
29
30 // Read the content of the file into the source_code string
31 size_t read_size = fread(source_code, sizeof(char), file_size, file);
32
33 // Add a null-terminator at the end of the source_code string
34 source_code[read_size] = '\0';
35
36 // Close the file
37 fclose(file);
38
39 // Return the source_code string as output
40 return source_code;
41 }
```

```
42
43 int main() {
44     // Specify the filename where the source code is
45     const char *filename = "source.c";
46
47     // Call the read_source_code() function to read the content of the file into a string
48     char *source_code = read_source_code(filename);
49
50     // Print the content of the source_code string
51     printf("Source code content:\n%s\n", source_code);
52
53     // Free the memory allocated for the source_code string
54     free(source_code);
55
56     // ENd
57     return 0;
58 }
```



Defining a token object

So far,


- We have a function that reads source code from an external file and stores the code in a string. Later on, we will split this string of code using whitespaces and `\n` symbols as separators.
- It will decompose the source code string into **lexemes**, i.e. **substrings of the source code corresponding to the different elements of the source code**.
- These lexemes then need to be **classified as keywords, identifiers, literals, operators or punctuation**.

`int x = 1023 ;` \longrightarrow `"int", "x", "=", "1023", ";"`

Defining a token object

Speaking of, we need to define Token Types (keyword, operators, etc.)

- For simplicity, use an enumeration for different token types and list them with explicit names, in order.
- Start simple, with KEYWORD, IDENTIFIER, NUMBER, OPERATOR and UNKNOWN Token Types.

"int", "x", "=", "1023", ";"  *TOKEN(KEYWORD_INT, "int"),
TOKEN(IDENTIFIER, "x"),
TOKEN(OPERATOR_ASSIGN, "="),
TOKEN(LITERAL_INT, "1023"),
TOKEN(END_OF_LINE, ";")*

From Code files/2.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5
6  // Let us list the different types of tokens as an enum first
7  // TOKEN_KEYWORD = 0, for keywords, e.g. int, while, for, if, etc.
8  // TOKEN_IDENTIFIER = 1, for variables and functions names, e.g. x, my_function, etc.
9  // TOKEN_NUMBER = 2, for numerical values to be assigned to int variables, e.g. 0, 1, 1023.
10 // TOKEN_OPERATOR = 3, for operators, e.g. +, *, =, etc.
11 // TOKEN_UNKNOWN = 4, for any lexemes that does not seem to fall in any category, e.g. ???, 0ab54df, etc.
12 typedef enum {
13     TOKEN_KEYWORD,
14     TOKEN_IDENTIFIER,
15     TOKEN_NUMBER,
16     TOKEN_OPERATOR,
17     TOKEN_UNKNOWN
18 } TokenType;
```

Defining a token object

Then, we need to define a Token object.

- For simplicity, let us define a struct Token.
- It will hold the token type and the lexeme for each of the substrings we have defined after the split.
- For good practice we will also define a constructor function *create_token()* and a destructor function *free_token()* for these TokenType variables.

```
20
21 // Our Token object, consisting of a TokenType and a lexeme
22 typedef struct {
23     TokenType type;
24     char *lexeme;
25 } Token;
26
27
28 // Constructor for the Token struct
29 Token *create_token(TokenType type, const char *lexeme) {
30     Token *token = (Token *)malloc(sizeof(Token));
31     token->type = type;
32     token->lexeme = strdup(lexeme);
33     return token;
34 }
35
36
37 // Destructor for the Token struct
38 void free_token(Token *token) {
39     free(token->lexeme);
40     free(token);
41 }
42
```

Using the Token object

In the main() function,

- We start by defining an array of sample lexemes for each token type,
- We then iterate through the TokenType enumeration,
- Create tokens using the create_token() function,
- And print the token information.
- After printing the information, we free the memory allocated for the tokens using the free_token() function.

Important: at the moment, we are not yet able to recognize the Token types correctly! (That will be our next step!)

```

44 // Main, demonstrating how Token will look like
45 int main() {
46     // Create an array of sample lexemes for each token type
47     const char *sample_lexemes[] = {
48         "int",           // TOKEN_KEYWORD
49         "variable_name", // TOKEN_IDENTIFIER
50         "42",           // TOKEN_NUMBER
51         "+",            // TOKEN_OPERATOR
52         "",             // TOKEN_END_OF_FILE
53         "???",          // TOKEN_UNKNOWN
54     };
55
56     // Iterate through the TokenType enum and create tokens using the sample lexemes
57     for (int i = 0; i <= TOKEN_UNKNOWN; i++) {
58         // Fetch token type and lexeme
59         TokenType type = (TokenType)i;
60         const char *lexeme = sample_lexemes[i];
61
62         // Create token object
63         Token *token = create_token(type, lexeme);
64
65         // Display it
66         printf("Token: Type = %d, Lexeme = '%s'\n", token->type, token->lexeme);
67
68         // Free token object
69         free_token(token);
70     }
71
72     // End
73     return 0;
74 }

```

Token: Type = 0, Lexeme = 'int'

Token: Type = 1, Lexeme = 'variable_name'

Token: Type = 2, Lexeme = '42'

Token: Type = 3, Lexeme = '+'

Token: Type = 4, Lexeme = ''

Recognizing Token Types for given Lexemes

Our next step is then to logically write functions that will **recognize the Token Type of a given lexeme**.

- We need a function that receives a lexeme as input,
- Checks if the lexeme matches the pattern of a given category,
- And if it does, creates a Token entry with the TokenType that has been recognized and the given lexeme.

Recognizing Token Types for given Lexemes

Let us begin with the simplest Token Type to recognize: **Keywords**.

- **A lexeme x will be recognized as a Keyword Token Type if and only if there is an exact match between the lexeme x and one of the possible keywords of C (“int”, “while”, “if”, “for”, “return”, etc.).**
- This means that we need to know about the full list of keywords to recognize in the C language.
- Also, as a side note, this explains why keywords are reserved and cannot be used for variable names/functions in many programming languages.

Recognizing Token Types for given Lexemes

As we have seen earlier in a Practice activity,

- This could be done with a simple **FSM**, which considers as **acceptable inputs x the exact string “SUTD”, and nothing else.**
- As seen in the previous lecture, this FSM is exactly what happens behind the scenes for to the **RegEx “^SUTD\$”**.

We start by implementing a Keyword recognition function, based on RegEx, and consider some of the possible keywords to demonstrate.

- **Our RegEx: “^(int|while|for|if|return)\$”**
- (Normally, we should list waaay more keywords, but let us keep it simple.)


```

1  #include <stdio.h>
2  #include <string.h>
3  #include <regex.h>
4  #include <stdbool.h>
5
6
7  // Recognizing keywords using RegEx
8  bool is_keyword_regex(const char *lexeme) {
9      // Define a regular expression pattern for the predefined keywords
10     // Using the | operator to list keywords here, adding the start ^ and $
11     // to enforce the fact that there should be no other characters.
12     const char *pattern = "^(int|while|for|if|return)$";
13
14     // Define and compile the regular expression
15     regex_t regex;
16     int result = regcomp(&regex, pattern, REG_EXTENDED | REG_NOSUB);
17
18     // If there is an error compiling the regular expression, return false
19     if (result != 0) {
20         return false;
21     }
22
23     // Match the lexeme against the regular expression
24     result = regexec(&regex, lexeme, 0, NULL, 0);
25
26     // Free the memory allocated for the regular expression
27     regfree(&regex);
28
29     // If the lexeme matches the regular expression, return true
30     if (result == 0) {
31         return true;
32     }
33
34     // Otherwise, return false
35     return false;
36 }

```

From Code files/3.

Recognizing keywords using strcmp

Since we are comparing strings and looking for exact matches here, we could also consider an alternate implementation,

- Using ***strcmp()***,
- To compare our lexeme *x*
- To all the possible keywords of the language that we would have previously assembled in a list.

(But to be honest, these strcmp() operations would be roughly similar to an FSM of some sort anyway, so not a big difference...)

```
38
39 // Recognizing keywords using strcmp
40 bool is_keyword_strcmp(const char *lexeme) {
41     // List of predefined keywords
42     const char *keywords[] = {"int", "while", "for", "if", "return", NULL};
43
44     // Iterate through the keywords
45     for (int i = 0; keywords[i] != NULL; i++) {
46         // Compare the lexeme with the current keyword using strcmp()
47         if (strcmp(lexeme, keywords[i]) == 0) {
48             // If the lexeme matches the keyword, return true
49             return true;
50         }
51     }
52
53     // If no keyword matches the lexeme, return false
54     return false;
55 }
56
57
```

Strcmp vs. RegEx

In practice, we should prefer the `strcmp()` implementation over the RegEx, here.

- Using regular expressions for this specific “exact match” case might be less efficient than using a *strcmp()* implementation.
- Especially considering that the number of keywords is small and they have a simple structure.
- **Keep in mind, however, that regular expressions will be required for more complex token recognition tasks, such as recognizing identifiers or literals with specific patterns.**
- Either way, let us assemble and try to recognize keywords lexemes.

```
57
58 // Testing both functions
59 int main() {
60     // Array of lexemes to test
61     const char *lexemes[] = {"int", "while", "for", "if", "return", "intwhile", NULL};
62
63     // Iterate through the lexemes and test both is_keyword_strcmp() and is_keyword_regex() functions
64     for (int i = 0; lexemes[i] != NULL; i++) {
65         const char *lexeme = lexemes[i];
66
67         // Check if the lexeme is a keyword using is_keyword_strcmp()
68         bool is_keyword_with_strcmp = is_keyword_strcmp(lexeme);
69
70         // Check if the lexeme is a keyword using is_keyword_regex()
71         bool is_keyword_with_regex = is_keyword_regex(lexeme);
72
73         // Print the results
74         printf("Lexeme '%s':\n", lexeme);
75         printf("  is_keyword_strcmp: %s\n", is_keyword_with_strcmp ? "true" : "false");
76         printf("  is_keyword_regex:   %s\n", is_keyword_with_regex ? "true" : "false");
77     }
78
79     return 0;
80 }
81
```

```

57
58 // Testing both functions
59 int main() {
60     // Array of lexemes to test
61     const char *lexemes[] = {"int", "return", "intwhile", NULL};
62
63     // Iterate through the lexemes and test them with the strcmp() and is_keyword_regex() functions
64     for (int i = 0; lexemes[i] != NULL; i++) {
65         const char *lexeme = lexemes[i];
66
67         // Check if the lexeme is a keyword using strcmp()
68         bool is_keyword_with_strcmp = strcmp(lexeme, "int") == 0 || strcmp(lexeme, "return") == 0 || strcmp(lexeme, "intwhile") == 0;
69
70         // Check if the lexeme is a keyword using regex
71         bool is_keyword_with_regex = is_keyword_regex(lexeme);
72
73         // Print the results
74         printf("Lexeme '%s':\n", lexeme);
75         printf("    is_keyword_strcmp: %s\n", is_keyword_with_strcmp ? "true" : "false");
76         printf("    is_keyword_regex: %s\n", is_keyword_with_regex ? "true" : "false");
77     }
78
79     return 0;
80 }
81

```

```

Lexeme 'int':
    is_keyword_strcmp: true
    is_keyword_regex: true
Lexeme 'while':
    is_keyword_strcmp: true
    is_keyword_regex: true
Lexeme 'for':
    is_keyword_strcmp: false
    is_keyword_regex: false
Lexeme 'if':
    is_keyword_strcmp: false
    is_keyword_regex: false
Lexeme 'return':
    is_keyword_strcmp: true
    is_keyword_regex: true
Lexeme 'intwhile':
    is_keyword_strcmp: false
    is_keyword_regex: false

```

Our first Tokenizer

Let us assemble everything we have so far

- A code that reads the code in a source.c file and puts it in a big string,
- A code that splits the big string into lexemes, using whitespaces and \n symbols as separators.
- An enum structure for storing the different TokenType,
- A Token struct to store each lexeme and TokenType corresponding to the lexemes,
- A code that can recognize if a lexeme is a KEYWORD or if it should be considered an UNKNOWN Token Type instead,
- A code that repeats this operation for each lexemes and creates Tokens accordingly

Our first Tokenizer

Let us assemble everything we have so far

- A code that reads the code in a source.c file and puts it in a big string,
- A code that splits the big string into lexemes, using whitespaces and \n symbols as separators.
- An enum structure for storing the different TokenType,
- A Token struct to store each lexeme and TokenType corresponding to lexemes,
- A code that can recognize if a lexeme is a KEYWORD or if it should be considered an UNKNOWN Token Type instead,
- A code that repeats this operation for each lexemes and creates Tokens accordingly

Let us assemble everything now!

From Code files/4.

```
72 int main() {
73     // Read the source code from the file
74     const char *filename = "source.c";
75     char *source_code = read_source_code(filename);
76     if (source_code == NULL) {
77         fprintf(stderr, "Error reading source code from '%s'\n", filename);
78         return 1;
79     }
80
81     // Tokenize the source code into lexemes
82     char *lexeme = strtok(source_code, " \t\n\r");
83     while (lexeme != NULL) {
84         // Check if the lexeme is a keyword using is_keyword_strcmp()
85         bool is_keyword = is_keyword_strcmp(lexeme);
86
87         // Create a token for the lexeme
88         TokenType token_type = is_keyword ? TOKEN_KEYWORD : TOKEN_UNKNOWN;
89         Token *token = create_token(token_type, lexeme);
90
91         // Print the token information
92         printf("Token { type: %d, lexeme: '%s' }\n", token->type, token->lexeme);
93
94         // Free the memory allocated for the token
95         free_token(token);
96
97         // Move to the next lexeme
98         lexeme = strtok(NULL, " \t\n\r");
99     }
100
101     // Free the memory allocated for the source code
102     free(source_code);
103
104     return 0;
105 }
```

Recognizing Punctuation

Our next step will be to recognize punctuation, for instance the “;” used at the end of each line.

- Technically, this is identical to recognizing keywords, except that we look for a match with the string “;” instead of – say – “int”.

This can be easily implemented by

- Adjusting the Token Types to include a TOKEN_END_OF_LINE type,

```
8 // Update the TokenType enumeration to include TOKEN_END_OF_LINE
9 typedef enum {
10     TOKEN_KEYWORD,
11     TOKEN_END_OF_LINE,
12     TOKEN_UNKNOWN
13 } TokenType;
14
```

Recognizing Punctuation

Our next step will be to recognize punctuation, for instance the “;” used at the end of each line.

- Technically, this is identical to recognizing keywords, except that we look for a match with the string “;” instead of – say – “int”.

This can be easily implemented by

- Adjusting the Token Types to include a `TOKEN_END_OF_LINE` type,
- We again use a `strcmp()` operation to check if the lexeme exactly matches “;” ,
- And if so, we will create a Token accordingly,
- Otherwise check if it is a `KEYWORD` or `UNKNOWN` Token like before.

```
50 // Almost same main() as before
51 int main() {
52     char *source_code = (char*) malloc(100*sizeof(char));
53     if(source_code == NULL)
54     {
55         printf("Memory allocation failed!");
56         return 1;
57     }
58     strcpy(source_code, "int while;\nfor if;\nintwhile;");
59
60     char *lexeme = strtok(source_code, " \t\n\r;");
61     while (lexeme != NULL) {
62         TokenType token_type;
63         // Let us now recognize ; characters!
64         if (strcmp(lexeme, ";") == 0) {
65             token_type = TOKEN_END_OF_LINE;
66         } else if (is_keyword_strcmp(lexeme)) {
67             token_type = TOKEN_KEYWORD;
68         } else {
69             token_type = TOKEN_UNKNOWN;
70         }
71
72         Token *token = create_token(token_type, lexeme);
73         printf("Token { type: %d, lexeme: '%s' }\n", token->type, token->lexeme);
74         free_token(token);
75         lexeme = strtok(NULL, " \t\n\r;");
76     }
77
78     free(source_code)
79
80     return 0;
81 }
```

Quick question, why do we need ; anyway?

In C, the semicolon (;) typically serves as a statement delimiter, indicating the end of a statement or line.

- It must be used and cannot be omitted, or the compiler will crash.
- At the moment, we suspect that this will help the compiler to understand the structure of the code and determine where each statement begins and ends.
- If C did not require semicolons, it would still be possible for a compiler to process the code, but the language would need to be designed differently, and additional rules would have to be introduced to determine the end of a statement (e.g. Python relies on indentation).

Quick question, why do we need ; anyway?

To be honest, the semicolon symbol does not significantly affect the tokenization process.

- The sole purpose of the tokenizer is to break the input source code into individual lexemes,
- Recognize the types of these lexemes and create Tokens with the appropriate types accordingly,
- And the tokenizer does so, regardless of whether the syntax makes sense or not,
- *(I mean “int for while if return intwhile” seriously?)*
- And whether each line correctly finishes with semicolons or not.

Quick question, why do we need ; anyway?

- The semicolon, however, serves as a statement delimiter, which will help the **PARSER** (our next step after tokenization) to correctly identify and group statements in the code.
- This will typically allow the compiler to understand the structure of the code, distinguish between different statements, and later identify any syntax errors.
- The semicolon symbol (;) then plays a crucial role in the **PARSING** phase by helping the compiler to correctly understand and represent the structure of the code.
- But this **SYNTAX** analysis comes later, and it is not the job of the **LEXICAL** analysis (or **TOKENIZATION**) part.

Recognizing additional punctuation/operators

As with the “;” symbol, most

- punctuation signs (curly braces, parentheses, etc.)
- or operators (e.g. “+”, “-”, “*”, “/”, etc.)

will consist of a single special character

We can then create Token Types for each of these operators, as before.

```
8 | // Update the TokenType enumeration
9 | #typedef enum {
10 |     TOKEN_KEYWORD,
11 |     TOKEN_ADD,           // The '+' operator
12 |     TOKEN_SUBTRACT,      // The '-' operator
13 |     TOKEN_MULTIPLY,      // The '*' operator
14 |     TOKEN_DIVIDE,        // The '/' operator
15 |     TOKEN_LEFT_PAREN,    // The '(' character
16 |     TOKEN_RIGHT_PAREN,   // The ')' character
17 |     TOKEN_END_OF_LINE,   // The ';' character
18 |     TOKEN_UNKNOWN
19 | } TokenType;
```


Quick question

Shall we use a single Token Type `TOKEN_OPERATOR` for all operators; or should we make more categories of Token Types (`TOKEN_ADD`, `MULT`, `SUB`, `DIV`, etc.)? Same thing for `KEYWORDS` and `PUNCTUATION`.

- Some compilers may indeed use separate token types for each operator, as shown in the previous example.
- However, other compilers might choose to use a single `TOKEN_OPERATOR` type instead, which will simplify the tokenization process, but may require additional work during the parsing to determine the specific type of operator.
- This approach can be more efficient and flexible, as it reduces the number of distinct token types and allows the parser to handle various operators based on context.

Fair enough, what does GCC do then?

- GCC uses a the most generic approach of the two and tends to categorize operators under a single token type `TOKEN_OPERATOR`.
- It will later use the lexeme information to differentiate between them during the parsing phase.
- For simplicity however, we will assume that different types of tokens will be implemented.
- *(This means, we should technically go back and define more Token Types for keywords, such as `TOKEN_KEYWORD_INT`, `TOKEN_KEYWORD_WHILE`, etc.)*

Back to our additional punctuation/operators

As with the “;” symbol, most

- punctuation signs (curly braces, parentheses, etc.)
- or operators (e.g. “+”, “-”, “*”, “/”, etc.)

will consist of a single special character

We can then create Token Types for each of these operators, as before.

```
8 | // Update the TokenType enumeration
9 | #typedef enum {
10 |     TOKEN_KEYWORD,
11 |     TOKEN_ADD,           // The '+' operator
12 |     TOKEN_SUBTRACT,      // The '-' operator
13 |     TOKEN_MULTIPLY,      // The '*' operator
14 |     TOKEN_DIVIDE,        // The '/' operator
15 |     TOKEN_LEFT_PAREN,    // The '(' character
16 |     TOKEN_RIGHT_PAREN,   // The ')' character
17 |     TOKEN_END_OF_LINE,   // The ';' character
18 |     TOKEN_UNKNOWN
19 | } TokenType;
```

Back to our additional punctuation/operators

Another quick note:

- Checking that each opened parenthesis gets closed is NOT the job of the TOKENIZER.
- Just like before with “;”, this falls under the SYNTAX analysis category.
- And will therefore be the job of the PARSING phase.

```
8 | // Update the TokenType enumeration
9 | typedef enum {
10 |     TOKEN_KEYWORD,
11 |     TOKEN_ADD,           // The '+' operator
12 |     TOKEN_SUBTRACT,      // The '-' operator
13 |     TOKEN_MULTIPLY,      // The '*' operator
14 |     TOKEN_DIVIDE,        // The '/' operator
15 |     TOKEN_LEFT_PAREN,    // The '(' character
16 |     TOKEN_RIGHT_PAREN,   // The ')' character
17 |     TOKEN_END_OF_LINE,   // The ';' character
18 |     TOKEN_UNKNOWN
19 | } TokenType;
```

Back to our additional punctuation/operators

Another quick note:

- Checking that each opened parenthesis gets closed is NOT the job of the TOKENIZER.

Question: BTW, is there any RegEx that could check that any opened parenthesis in the string *x* has been closed?

```
8 | // Update the TokenType enumeration
9 | typedef enum {
10 |     TOKEN_KEYWORD,
11 |     TOKEN_ADD,           // The '+' operator
12 |     TOKEN_SUBTRACT,      // The '-' operator
13 |     TOKEN_MULTIPLY,      // The '*' operator
14 |     TOKEN_DIVIDE,        // The '/' operator
15 |     TOKEN_LEFT_PAREN,    // The '(' character
16 |     TOKEN_RIGHT_PAREN,   // The ')' character
17 |     TOKEN_END_OF_LINE,   // The ';' character
18 |     TOKEN_UNKNOWN
19 | } TokenType;
```

```
// Now covering lots of possible token types!
// But to be honest there are many many more!
if (is_keyword_strcmp(lexeme)) {
    token_type = TOKEN_KEYWORD;
} else if (strcmp(lexeme, "+") == 0) {
    token_type = TOKEN_ADD;
} else if (strcmp(lexeme, "-") == 0) {
    token_type = TOKEN_SUBTRACT;
} else if (strcmp(lexeme, "*") == 0) {
    token_type = TOKEN_MULTIPLY;
} else if (strcmp(lexeme, "/") == 0) {
    token_type = TOKEN_DIVIDE;
} else if (strcmp(lexeme, "(") == 0) {
    token_type = TOKEN_LEFT_PAREN;
} else if (strcmp(lexeme, ")") == 0) {
    token_type = TOKEN_RIGHT_PAREN;
} else if (strcmp(lexeme, ";") == 0) {
    token_type = TOKEN_END_OF_LINE;
} else if (strcmp(lexeme, "&") == 0) {
    token_type = TOKEN_SINGLE_AND;
} else if (strcmp(lexeme, "&&") == 0) {
    token_type = TOKEN_DOUBLE_AND;
} else else {
    token_type = TOKEN_UNKNOWN;
}
```

**Found in main()
function to match
operators and
additional
punctuation
symbols.**

From Code files/6.

Recognizing identifiers

Identifier: An identifier is a **name given to a variable, function, or other entity** in a program.

In C, a valid identifier name should follow rules, such as

- starting with a letter or underscore,
- and consisting of letters, digits, and underscores.

Examples of valid identifiers:

- x, counter, variable_1,
- myFunction,
- etc.

Recognizing identifiers

Is there a finite list of possible identifiers like for keywords, operators and punctuation signs?

No. And because of that reason, we cannot use a list of possible strings to match and a *strcmp()* method like before. We have no other choice, but to rely on RegEx to transcribe the grammar rules for identifiers.

- Start with a letter or underscore,
- and consist of letters, digits, and underscores (or what we called word characters earlier `\w`).

Recognizing identifiers

Is there a finite list of possible identifiers like for keywords, operators and punctuation signs?

No. And because of that reason, we cannot use a list of possible strings to match and a *strcmp()* method like before. We have no other choice, but to rely on RegEx to transcribe the grammar rules for identifiers.

- Start with a letter or underscore,
- and consist of letters, digits, and underscores (or what we called word characters earlier `\w`).

Possible RegEx for identifiers:

`“^[a-zA-Z][a-zA-Z0-9_]*$”` or `“^[a-zA-Z_]\w*$”`

Quick note on \w and others in RegEx

The C RegEx library expects two escape characters to be used \\w (or any other similar notation like \\d, \\s, etc.).

This means that the \\w in the RegEx below is not a typo.

“^[a-zA-Z_]\\w\$”*

In some other RegEx engines, a simple \w might do (so be careful!).

“^[a-zA-Z_]\w\$”*

```
7 // Recognizing identifiers using RegEx
8 bool is_identifier_regex(const char *lexeme) {
9     // Define a regular expression pattern for identifiers
10    // Identifiers start with a letter or an underscore, followed by any
11    // combination of letters, digits, or underscores.
12    const char *pattern = "[a-zA-Z_]\\w*";
13
14    // Define and compile the regular expression
15    regex_t regex;
16    int result = regcomp(&regex, pattern, REG_EXTENDED | REG_NOSUB);
17
18    // If there is an error compiling the regular expression, return false
19    if (result != 0) {
20        return false;
21    }
22
23    // Match the lexeme against the regular expression
24    result = regexec(&regex, lexeme, 0, NULL, 0);
25
26    // Free the memory allocated for the regular expression
27    regfree(&regex);
28
29    // If the lexeme matches the regular expression, return true
30    if (result == 0) {
31        return true;
32    }
33
34    // Otherwise, return false
35    return false;
36 }
37
```

From Code files/7.

```
37
38 // Testing the function
39 int main() {
40     // Array of lexemes to test
41     const char *lexemes[] = {"_identifier", "variable", "Int", "123invalid", "valid123", "with_underscore", "a0?", NULL};
42
43     // Iterate through the lexemes and test the is_identifier_regex() function
44     for (int i = 0; lexemes[i] != NULL; i++) {
45         const char *lexeme = lexemes[i];
46
47         // Check if the lexeme is an identifier using is_identifier_regex()
48         bool is_identifier_with_regex = is_identifier_regex(lexeme);
49
50         // Print the results
51         printf("Lexeme '%s':\n", lexeme);
52         printf("  is_identifier_regex:  %s\n", is_identifier_with_regex ? "true" : "false");
53     }
54
55     return 0;
56 }
57
```

```

37
38 // Testing the function
39 int main() {
40     // Array of lexemes to test
41     const char *lexemes[] = {"_identifier", "variable", "Int", "123invalid", "valid123", "with_underscore", "a0?", NULL};
42
43     // Iterate through the lexemes and test the is_identifier_regex() function
44     for (int i = 0; lexemes[i] != NULL; i++) {
45         const char *lexeme = lexemes[i];
46
47         // Check if the lexeme is an identifier using is_identifier_regex()
48         bool is_identifier_with_regex = is_identifier_regex(lexeme);
49
50         // Print the results
51         printf("Lexeme '%s':\n", lexeme);
52         printf("    is_identifier_regex: %s\n", is_identifier_with_regex ? "true" : "false");
53     }
54
55     return 0;
56 }
57

```

```

Lexeme '_identifier':
    is_identifier_regex: true
Lexeme 'variable':
    is_identifier_regex: true
Lexeme 'Int':
    is_identifier_regex: true
Lexeme '123invalid':
    is_identifier_regex: false
Lexeme 'valid123':
    is_identifier_regex: true
Lexeme 'with_underscore':
    is_identifier_regex: true
Lexeme 'a0?':
    is_identifier_regex: false

```

Recognizing literals

Literal: A **value** of some sort to be **assigned to a variable**.

Examples of literals in the C programming language:

- 12542
- 12654165.52
- “hello”
- Etc.

Recognizing integer literals

(Unsigned *Integer*) Literal: A **numerical value** of some sort to be assigned to an **integer variable**.

Examples of integer literals in the C programming language:

- 0, 42, 856841, Etc.

Examples of invalid integer literals in the C programming language:

- -7
- 42.0
- 00123
- 0123followedbyletters

Recognizing unsigned integer literals

To recognize integer literals, we will have to rely on regular expression patterns to match

- either a zero,
- or a non-zero digit followed by a sequence of zero or more digits.

Possible RegEx for unsigned integer literals:

`“^(0|[1-9][0-9]*)$”` or `“^(0|[1-9]\\d*)$”`


```
6 // Recognizing integer literals using RegEx
7 bool is_integer_literal_regex(const char *lexeme) {
8     // Define a regular expression pattern for integer literals
9     // Integer literals are a sequence of digits without leading zeros.
10    const char *pattern = "(0|[1-9][0-9]*)$";
11
12    // Define and compile the regular expression
13    regex_t regex;
14    int result = regcomp(&regex, pattern, REG_EXTENDED | REG_NOSUB);
15
16    // If there is an error compiling the regular expression, return false
17    if (result != 0) {
18        return false;
19    }
20
21    // Match the lexeme against the regular expression
22    result = regexec(&regex, lexeme, 0, NULL, 0);
23
24    // Free the memory allocated for the regular expression
25    regfree(&regex);
26
27    // If the lexeme matches the regular expression, return true
28    if (result == 0) {
29        return true;
30    }
31
32    // Otherwise, return false
33    return false;
34 }
```

From Code files/8.

```
36 // Testing the function
37 int main() {
38     // Array of lexemes to test
39     const char *lexemes[] = {"42", "12345", "00123", "0", "1notinteger", NULL};
40
41     // Iterate through the lexemes and test the is_integer_literal_regex() function
42     for (int i = 0; lexemes[i] != NULL; i++) {
43         const char *lexeme = lexemes[i];
44
45         // Check if the lexeme is an integer literal using is_integer_literal_regex()
46         bool is_integer = is_integer_literal_regex(lexeme);
47
48         // Print the results
49         printf("Lexeme '%s':\n", lexeme);
50         printf("    is_integer_literal_regex:  %s\n", is_integer ? "true" : "false");
51     }
52
53     return 0;
54 }
55
```

```

36 // Testing the function
37 int main() {
38     // Array of lexemes to test
39     const char *lexemes[] = {"42", "12345", "00123", "0", "1notinteger", NULL};
40
41     // Iterate through the lexemes and test the is_integer_literal_regex() function
42     for (int i = 0; lexemes[i] != NULL; i++) {
43         const char *lexeme = lexemes[i];
44
45         // Check if the lexeme is an integer literal
46         bool is_integer = is_integer_literal_regex(lexeme);
47
48         // Print the results
49         printf("Lexeme '%s':\n", lexeme);
50         printf("    is_integer_literal_regex: %s\n", is_integer ? "true" : "false");
51     }
52
53     return 0;
54 }
55

```

```

Lexeme '42':
    is_integer_literal_regex: true
Lexeme '12345':
    is_integer_literal_regex: true
Lexeme '00123':
    is_integer_literal_regex: false
Lexeme '0':
    is_integer_literal_regex: true
Lexeme '1notinteger':
    is_integer_literal_regex: false

```

Practice: Recognizing more types of literals

Possible RegEx for unsigned integer literals:

`“^(0|[1-9][0-9]*)$”` or `“^(0|[1-9]\\d*)$”`

What would the RegEx look like if we wanted to recognize a signed float literal then?

Practice: Recognizing more types of literals

Possible RegEx for unsigned integer literals:

`“^(0|[1-9][0-9]*)$”` or `“^(0|[1-9]\\d*)$”`

What would the RegEx look like if we wanted to recognize a signed float literal then?

Valid: 0, 0.0, 7, -7, -4.25, +7.32, 147.687000, etc.

Invalid: 1.2.3, 03.14, 34t.023, 2.4f7, etc.

Practice: Recognizing more types of literals

Possible RegEx for unsigned integer literals:

`“^(0|[1-9][0-9]*)$”` or `“^(0|[1-9]\\d*)$”`

What would the RegEx look like if we wanted to recognize a signed float literal then?

`“^[-+]?((0|[1-9]\\d*)(\\.\\d*)?)$”`

Practice: Recognizing more types of literals

Possible RegEx for unsigned integer literals:

`“^(0|[1-9][0-9]*)$”` or `“^(0|[1-9]\\d*)$”`

What would the RegEx look like if we wanted to recognize a signed float literal then?

→ But what about exponential notations for signed float literals, e.g. 1e+5, 0.25e-17, etc.?

(Answer not provided, leaving it as a challenge!)

Additional information in Tokens

On top of the Token Type and lexeme, additional attributes could be added to the Token object, for instance:

- The line number (which can be tracked by counting the number of `\n` that have been scanned during the splitting)
- A positional index (indicating that the lexeme starts on position 7 of its line for instance).

In [1]:

```
1 x = 10
2 y = x + 10followedbyletters
```

Cell In[1], line 2

```
y = x + 10followedbyletters
          ^
```

SyntaxError: invalid decimal literal

Additional information in Tokens, and Errors!

- This additional information could typically be used when an UNKNOWN token is recognized to produce an error message.
- Python does that well, indicates the line at which the error occurred and – on the latest Python versions – might even indicate the location of the error using a ^ symbol.

In [1]:

```
1 x = 10
2 y = x + 10followedbyletters
```

Cell In[1], line 2

```
y = x + 10followedbyletters
          ^
```

SyntaxError: invalid decimal literal

Additional information in Tokens, and Errors!

- These Error messages may in turn require more RegEx to recognize typical errors.
- E.g. here, we are trying to add a decimal variable (x), with a literal reading as 10followedbyletters.
- This 10followedbyletters lexeme violates the RegEx defined for int and float literals and gets caught.
- The error message then displays that we have an “invalid literal”.

In [1]:

```
1 x = 10
2 y = x + 10followedbyletters
```

Cell In[1], line 2

```
y = x + 10followedbyletters
          ^
```

SyntaxError: invalid decimal literal

Additional information in Tokens, and Errors!

- In some cases, it might even be worth writing a RegEx for recognizing a typical scenario of an invalid int literal!
- For instance, RegEx for an invalid decimal literal that starts with at least one digit and is then followed by at least one non-digit character:

“^[0-9]+[^0-9]+”

In [1]:

```
1 x = 10
2 y = x + 10followedbyletters
```

Cell In[1], line 2

```
y = x + 10followedbyletters
          ^
```

SyntaxError: invalid decimal literal

Additional information in Tokens, and Errors!

Note: In some simple cases, this type of errors can be caught during tokenization.

And error messages can be implemented during Tokenization.

But most of the time, they will be caught by the **Parser** instead, during the Syntax Analysis step of the compilation (more on this later).

In [1]:

```
1 x = 10
2 y = x + 10followedbyletters
```

Cell In[1], line 2

```
y = x + 10followedbyletters
          ^
```

SyntaxError: invalid decimal literal

Conclusion

Let us call it a day for now. On the next lecture,

- Conflict resolution in the case of ambiguous tokens that could be classified as more than one type.
- Relaxing the hypothesis that all elements in code are nicely separated with whitespaces or `\n` symbols.
- Error handling in the case of incorrect/unknown lexemes.
- Recognizing and dropping comments
- And more!

Quiz time!

What is the primary purpose of tokenization in compilers?

- A. Parsing source code
- B. Converting source code to machine code
- C. Breaking source code into meaningful elements
- D. Optimizing source code for performance

Quiz time!

What is the primary purpose of tokenization in compilers?

- A. Parsing source code
- B. Converting source code to machine code
- C. Breaking source code into meaningful elements**
- D. Optimizing source code for performance

Quiz time!

Which of the following is NOT a typical step in tokenization?

- A. Reading source code from a file
- B. Identifying lexemes in the source code
- C. Classifying tokens by type
- D. Checking that each opening parenthesis is matching a closing parenthesis

Quiz time!

Which of the following is NOT a typical step in tokenization?

- A. Reading source code from a file
- B. Identifying lexemes in the source code
- C. Classifying tokens by type
- D. Checking that each opening parenthesis is matching a closing parenthesis (that would be the job of the SYNTAX analysis/PARSER)**

Quiz time!

Which of the following is NOT a common token type in programming languages?

- A. Keyword
- B. Punctuation
- C. Operator
- D. Comment
- E. Whitespace

Quiz time!

Which of the following is NOT a common token type in programming languages?

- A. Keyword
- B. Punctuation
- C. Operator
- D. Comment (We will not even bother writing tokens for code that has been commented!)**
- E. Whitespaces (?)**

Quiz time!

Which of the following is NOT a common token type in programming languages?

- A. Keyword
- B. Punctuation
- C. Operator
- D. Comment (We will not even bother writing tokens for code that has been commented!)**
- E. Whitespace (We usually discard them, unless the language is Python, as they could use for indentation?)**