50.051 Programming Language Concepts

W9-S1 Lexical Analysis and Tokenization (Part 1)

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

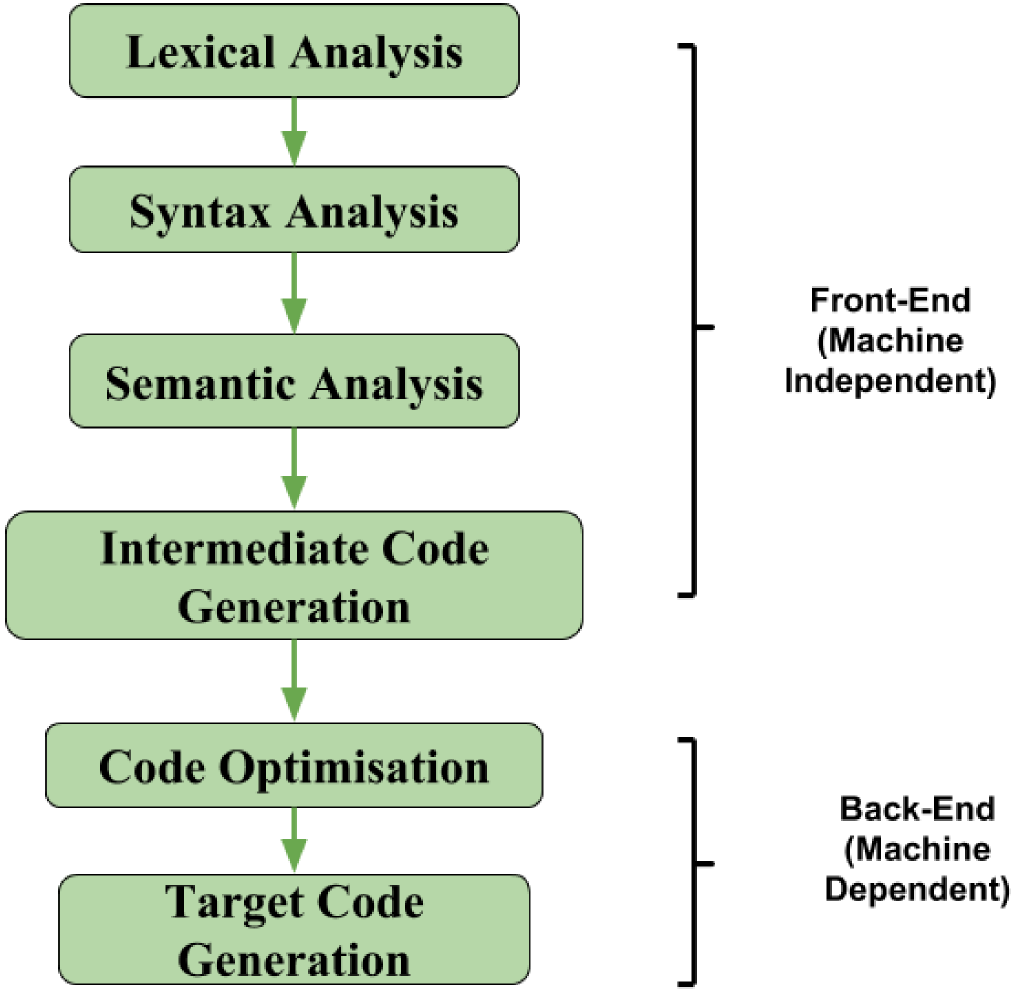
The front-end of a compiler

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** (as seen in a previous week).
- These can be used **to recognize different types of tokens**, such as keywords, identifiers, operators, and punctuation symbols, etc.

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 punctuations/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.

```
#include <stdio.h>
#include <stdlib.h>

char* read_source_code(const char *filename) {
    // Open the file in filename, in read mode
    // Check for opening errors
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        fprintf(stderr, "Error opening file: %s\n", filename);
        exit(EXIT_FAILURE);
    }

    // Move the file position indicator to the end of the file
    // And get the current file position (which is the size of the file)
    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);

    // Now that we know the length of the file,
    // Rewind the file position indicator back to the beginning of the file
    rewind(file);
```

```c
    // Allocate memory for the source code string
    // (file_size + 1 for the null-terminator)
    char *source_code = (char *)malloc((file_size + 1) * sizeof(char));
    if (source_code == NULL) {
        fprintf(stderr, "Error allocating memory for source code\n");
        exit(EXIT_FAILURE);
    }

    // Read the content of the file into the source_code string
    size_t read_size = fread(source_code, sizeof(char), file_size, file);

    // Add a null-terminator at the end of the source_code string
    source_code[read_size] = '\0';

    // Close the file
    fclose(file);

    // Return the source_code string as output
    return source_code;
}
```

```c
int main() {
    // Specify the filename where the source code is
    const char *filename = "source.c";

    // Call the read_source_code() function to read the content of the file into a string
    char *source_code = read_source_code(filename);

    // Print the content of the source_code string
    printf("Source code content:\n%s\n", source_code);

    // Free the memory allocated for the source_code string
    free(source_code);

    // ENd
    return 0;
}
```

source.c main.c

```c
int x = 1023 ;
```

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 ; → *"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 KEWORD, 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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


// Let us list the different types of tokens as an enum first
// TOKEN_KEYWORD = 0, for keywords, e.g. int, while, for, if, etc.
// TOKEN_IDENTIFIER = 1, for variables and functions names, e.g. x, my_function, etc.
// TOKEN_NUMBER = 2, for numerical values to be assigned to int variables, e.g. 0, 1, 1023.
// TOKEN_OPERATOR = 3, for operators, e.g. +, *, =, etc.
// TOKEN_UNKNOWN = 4, for any lexemes that does not seem to fall in any category, e.g. ???, 0ab54df, etc.
typedef enum {
    TOKEN_KEYWORD,
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_OPERATOR,
    TOKEN_UNKNOWN
} 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.

```c
// Our Token object, consisting of a TokenType and a lexeme
typedef struct {
    TokenType type;
    char *lexeme;
} Token;


// Constructor for the Token struct
Token *create_token(TokenType type, const char *lexeme) {
    Token *token = (Token *)malloc(sizeof(Token));
    token->type = type;
    token->lexeme = strdup(lexeme);
    return token;
}


// Desctructor for the Token struct
void free_token(Token *token) {
    free(token->lexeme);
    free(token);
}
```

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!)*

```c
// Main, demonstrating how Token will look like
int main() {
    // Create an array of sample lexemes for each token type
    const char *sample_lexemes[] = {
        "int",           // TOKEN_KEYWORD
        "variable_name", // TOKEN_IDENTIFIER
        "42",            // TOKEN_NUMBER
        "+",             // TOKEN_OPERATOR
        ";"              // TOKEN_END_OF_FILE
    };

    // Iterate through the TokenType enum and create tokens using the sample lexemes
    for (int i = 0; i <= TOKEN_UNKNOWN; i++) {
        // Fetch token type and lexeme
        TokenType type = (TokenType)i;
        const char *lexeme = sample_lexemes[i];

        // Create token object
        Token *token = create_token(type, lexeme);

        // Display it
        printf("Token: Type = %d, Lexeme = '%s'\n", token->type, token->lexeme);

        // Free token object
        free_token(token);
    }

    // End
    return 0;
}
```

```
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 way more keywords, but let us keep it simple.)

From Code files/3.

```c
#include <stdio.h>
#include <string.h>
#include <regex.h>
#include <stdbool.h>


// Recognizing keywords using RegEx
bool is_keyword_regex(const char *lexeme) {
    // Define a regular expression pattern for the predefined keywords
    // Using the | operator to list keywords here, adding the start ^ and $
    // to enforce the fact that there should be no other characters.
    const char *pattern = "^(int|while|for|if|return)$";

    // Define and compile the regular expression
    regex_t regex;
    int result = regcomp(&regex, pattern, REG_EXTENDED | REG_NOSUB);

    // If there is an error compiling the regular expression, return false
    if (result != 0) {
        return false;
    }

    // Match the lexeme against the regular expression
    result = regexec(&regex, lexeme, 0, NULL, 0);

    // Free the memory allocated for the regular expression
    regfree(&regex);

    // If the lexeme matches the regular expression, return true
    if (result == 0) {
        return true;
    }

    // Otherwise, return false
    return false;
}
```

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…)

```c
// Recognizing keywords using strcmp
bool is_keyword_strcmp(const char *lexeme) {
    // List of predefined keywords
    const char *keywords[] = {"int", "while", "for", "if", "return", NULL};

    // Iterate through the keywords
    for (int i = 0; keywords[i] != NULL; i++) {
        // Compare the lexeme with the current keyword using strcmp()
        if (strcmp(lexeme, keywords[i]) == 0) {
            // If the lexeme matches the keyword, return true
            return true;
        }
    }

    // If no keyword matches the lexeme, return false
    return false;
}
```

Strcmp vs. RegEx

In practice, should we prefer the strcmp() implementation over the RegEx one, if the number of keywords to recognize is small?

- 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.

```c
// Testing both functions
int main() {
    // Array of lexemes to test
    const char *lexemes[] = {"int", "while", "for", "if", "return", "intwhile", NULL};

    // Iterate through the lexemes and test both is_keyword_strcmp() and is_keyword_regex() functions
    for (int i = 0; lexemes[i] != NULL; i++) {
        const char *lexeme = lexemes[i];

        // Check if the lexeme is a keyword using is_keyword_strcmp()
        bool is_keyword_with_strcmp = is_keyword_strcmp(lexeme);

        // Check if the lexeme is a keyword using is_keyword_regex()
        bool is_keyword_with_regex = is_keyword_regex(lexeme);

        // Print the results
        printf("Lexeme '%s':\n", lexeme);
        printf("  is_keyword_strcmp: %s\n", is_keyword_with_strcmp ? "true" : "false");
        printf("  is_keyword_regex:  %s\n", is_keyword_with_regex ? "true" : "false");
    }

    return 0;
}
```

```c
// Testing both functions
int main() {
    // Array of lexemes to test
    const char *lexemes[] = {"int", "                         turn", "intwhile", NULL};

    // Iterate through the lexemes ar                         trcmp() and is_keyword_regex() functions
    for (int i = 0; lexemes[i] != NU
        const char *lexeme = lexemes

        // Check if the lexeme is a                         strcmp()
        bool is_keyword_with_strcmp =                         e);

        // Check if the lexeme is a                         regex()
        bool is_keyword_with_regex =                          ;

        // Print the results
        printf("Lexeme '%s':\n", lexe
        printf("  is_keyword_strcmp:                         trcmp ? "true" : "false");
        printf("  is_keyword_regex:                          egex ? "true" : "false");
    }

    return 0;
}
```

```
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: true
  is_keyword_regex:  true
Lexeme 'if':
  is_keyword_strcmp: true
  is_keyword_regex:  true
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 TokenTypes,
- 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 lexeme 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 TokenTypes,
- 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

Let us assemble everything now!

From Code files/4.

```c
int main() {
    // Read the source code from the file
    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;
    }

    // Tokenize the source code into lexemes
    char *lexeme = strtok(source_code, " \t\n\r");
    while (lexeme != NULL) {
        // Check if the lexeme is a keyword using is_keyword_strcmp()
        bool is_keyword = is_keyword_strcmp(lexeme);

        // Create a token for the lexeme
        TokenType token_type = is_keyword ? TOKEN_KEYWORD : TOKEN_UNKNOWN;
        Token *token = create_token(token_type, lexeme);

        // Print the token information
        printf("Token { type: %d, lexeme: '%s' }\n", token->type, token->lexeme);

        // Free the memory allocated for the token
        free_token(token);

        // Move to the next lexeme
        lexeme = strtok(NULL, " \t\n\r");
    }

    // Free the memory allocated for the source code
    free(source_code);

    return 0;
}
```

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,

```c
// Update the TokenType enumeration to include TOKEN_END_OF_LINE
typedef enum {
    TOKEN_KEYWORD,
    TOKEN_END_OF_LINE,
    TOKEN_UNKNOWN
} TokenType;
```

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,
- To do so, we again use a strcmp() operation to check if the lexeme exactly matches the punctuation “;” ,
- And if so, we will create a Token accordingly,
- Otherwise check if it is a KEYWORD or UNKNOWN Token like before.

```c
// Almost same main() as before
int main() {
    char *source_code = (char*) malloc(100*sizeof(char));
    if(source_code == NULL)
    {
        printf("Memory allocation failed!");
        return 1;
    }
    strcpy(source_code, "int while;\nfor if;\nintwhile;");

    char *lexeme = strtok(source_code, " \t\n\r;");
    while (lexeme != NULL) {
        TokenType token_type;
        // Let us now recognize ; characters!
        if (strcmp(lexeme, ";") == 0) {
            token_type = TOKEN_END_OF_LINE;
        } else if (is_keyword_strcmp(lexeme)) {
            token_type = TOKEN_KEYWORD;
        } else {
            token_type = TOKEN_UNKNOWN;
        }

        Token *token = create_token(token_type, lexeme);
        printf("Token { type: %d, lexeme: '%s' }\n", token->type, token->lexeme);
        free_token(token);
        lexeme = strtok(NULL, " \t\n\r;");
    }

    free(source_code)

    return 0;
}
```

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.
  (as a counterexample, 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?).*
- For the same reason, it will not bother checking that 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 tokens corresponding to statements in the code.
- This will typically allow the compiler to identify syntax errors, or sequences of tokens that are incorrect.
- 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 <u>not the job</u> 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.

```
// Update the TokenType enumeration
typedef enum {
    TOKEN_KEYWORD,
    TOKEN_ADD,           // The '+' operator
    TOKEN_SUBTRACT,      // The '-' operator
    TOKEN_MULTIPLY,      // The '*' operator
    TOKEN_DIVIDE,        // The '/' operator
    TOKEN_LEFT_PAREN,    // The '(' character
    TOKEN_RIGHT_PAREN,   // The ')' character
    TOKEN_END_OF_LINE,   // The ';' character
    TOKEN_UNKNOWN
} 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.

```c
// Update the TokenType enumeration
typedef enum {
    TOKEN_KEYWORD,
    TOKEN_ADD,          // The '+' operator
    TOKEN_SUBTRACT,     // The '-' operator
    TOKEN_MULTIPLY,     // The '*' operator
    TOKEN_DIVIDE,       // The '/' operator
    TOKEN_LEFT_PAREN,   // The '(' character
    TOKEN_RIGHT_PAREN,  // The ')' character
    TOKEN_END_OF_LINE,  // The ';' character
    TOKEN_UNKNOWN
} 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 found in the string *x* has been closed?

```
// Update the TokenType enumeration
typedef enum {
    TOKEN_KEYWORD,
    TOKEN_ADD,            // The '+' operator
    TOKEN_SUBTRACT,       // The '-' operator
    TOKEN_MULTIPLY,       // The '*' operator
    TOKEN_DIVIDE,         // The '/' operator
    TOKEN_LEFT_PAREN,     // The '(' character
    TOKEN_RIGHT_PAREN,    // The ')' character
    TOKEN_END_OF_LINE,    // The ';' character
    TOKEN_UNKNOWN
} 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

Question: 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 use the shortcut for word characters \w).

Restricted

Recognizing identifiers

Question: 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 use the shortcut for word characters \w).

Possible RegEx for identifiers:

"^[a-zA-Z_][a-zA-Z0-9_]*$"*** *or equivalently* ***"^[a-zA-Z_]\\w*$"

Quick note on \\w and \w in RegEx

On some machine, the C RegEx library sometimes expects two escape characters to be used. That is, you should use \\w (or any other similar notation like \\d, \\s, etc.) instead of just \w...

Not sure why though.

Anyway, 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 (try it and figure it out for your machine?).

“^[a-zA-Z_]\w*$”

From Code files/7.

```c
// Recognizing identifiers using RegEx
bool is_identifier_regex(const char *lexeme) {
    // Define a regular expression pattern for identifiers
    // Identifiers start with a letter or an underscore, followed by any
    // combination of letters, digits, or underscores.
    const char *pattern = "^[a-zA-Z_]\\w*$";

    // Define and compile the regular expression
    regex_t regex;
    int result = regcomp(&regex, pattern, REG_EXTENDED | REG_NOSUB);

    // If there is an error compiling the regular expression, return false
    if (result != 0) {
        return false;
    }

    // Match the lexeme against the regular expression
    result = regexec(&regex, lexeme, 0, NULL, 0);

    // Free the memory allocated for the regular expression
    regfree(&regex);

    // If the lexeme matches the regular expression, return true
    if (result == 0) {
        return true;
    }

    // Otherwise, return false
    return false;
}
```

```c
// Testing the function
int main() {
    // Array of lexemes to test
    const char *lexemes[] = {"_identifier", "variable", "Int", "123invalid", "valid123", "with_underscore", "a0?", NULL};

    // Iterate through the lexemes and test the is_identifier_regex() function
    for (int i = 0; lexemes[i] != NULL; i++) {
        const char *lexeme = lexemes[i];

        // Check if the lexeme is an identifier using is_identifier_regex()
        bool is_identifier_with_regex = is_identifier_regex(lexeme);

        // Print the results
        printf("Lexeme '%s':\n", lexeme);
        printf("  is_identifier_regex:  %s\n", is_identifier_with_regex ? "true" : "false");
    }

    return 0;
}
```

```c
// Testing the function
int main() {
    // Array of lexemes to test
    const char *lexemes[] = {"_identifier", "variable", "Int", "123invalid", "valid123", "with_underscore", "a0?", NULL};

    // Iterate through the lexemes and test the is_identifier_regex() function
    for (int i = 0; lexemes[i] != NULL; i++) {
        const char *lexeme = lexemes[i];

        // Check if the lexeme is an identifier using is_identifier_regex()
        bool is_identifier_with_regex = is_identifier_regex(lexeme);

        // Print the results
        printf("Lexeme '%s':\n", lexeme);
        printf("  is_identifier_regex:  %s\n", is_identifier_with_regex ? "tru
    }

    return 0;
}
```

```
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 unsigned integer variable**.

Examples of integer literals in the C programming language:

- 0, 42, 856841, Etc.

Examples of invalid unsigned integer literals in the C 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])$”* *or* *“^(0|[1-9]\\d*)$”*

From Code files/8.

```c
// Recognizing integer literals using RegEx
bool is_integer_literal_regex(const char *lexeme) {
    // Define a regular expression pattern for integer literals
    // Integer literals are a sequence of digits without leading zeros.
    const char *pattern = "^(0|[1-9][0-9]*)$";

    // Define and compile the regular expression
    regex_t regex;
    int result = regcomp(&regex, pattern, REG_EXTENDED | REG_NOSUB);

    // If there is an error compiling the regular expression, return false
    if (result != 0) {
        return false;
    }

    // Match the lexeme against the regular expression
    result = regexec(&regex, lexeme, 0, NULL, 0);

    // Free the memory allocated for the regular expression
    regfree(&regex);

    // If the lexeme matches the regular expression, return true
    if (result == 0) {
        return true;
    }

    // Otherwise, return false
    return false;
}
```

```c
// Testing the function
int main() {
    // Array of lexemes to test
    const char *lexemes[] = {"42", "12345", "00123", "0", "1notinteger", NULL};

    // Iterate through the lexemes and test the is_integer_literal_regex() function
    for (int i = 0; lexemes[i] != NULL; i++) {
        const char *lexeme = lexemes[i];

        // Check if the lexeme is an integer literal using is_integer_literal_regex()
        bool is_integer = is_integer_literal_regex(lexeme);

        // Print the results
        printf("Lexeme '%s':\n", lexeme);
        printf("  is_integer_literal_regex:  %s\n", is_integer ? "true" : "false");
    }

    return 0;
}
```

```c
// Testing the function
int main() {
    // Array of lexemes to test
    const char *lexemes[] = {"42", "12345", "00123", "0", "1notinteger", NULL};

    // Iterate through the lexemes and test the is_integer_literal_regex() function
    for (int i = 0; lexemes[i] != NULL; i++) {
        const char *lexeme = lexemes[i];

        // Check if the lexeme is an integer liter                                  ex()
        bool is_integer = is_integer_literal_regex

        // Print the results
        printf("Lexeme '%s':\n", lexeme);
        printf("  is_integer_literal_regex:  %s\n"                                  );
    }

    return 0;
}
```

```
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 'notinteger':
  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?

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 line 2, 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 a certain type of problematic token (e.g. invalid decimal literals) is recognized. Use it to produce a more specific 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 an explicit message instead of a generic one.

```
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!

- 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 is defined as:

  “^[0-9]+[^0-9]+”

```
In [1]:
x = 10
y = x + 10followedbyletters
```

```
  Cell In[1], line 2
    y = x + 10followedbyletters
             ^
SyntaxError: invalid decimal literal
```

- Extra: How about one that can catch “00123” as an invalid int?

Restricted

Additional information in Tokens, and Errors!

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

And error messages can then be implemented during Tokenization.

(Not shown, leaving it as a challenge for students to come up with RegEx capable of catching common errors!)

```
In [1]:
x = 10
y = x + 10followedbyletters
```

```
  Cell In[1], line 2
    y = x + 10followedbyletters
             ^
SyntaxError: invalid decimal literal
```

Restricted

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/lexemes in the code are nicely separated with whitespaces or \n symbols.
- Error handling in the case of incorrect and unknown lexemes.
- Recognizing and discarding comments from the source code.
- 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 be used for indentation?)**

If time allows, let us define our Tokenizer v1.0

Assemble all concepts from earlier to produce a Tokenizer v1.0, which:

- Will receive a string “source.c” to indicate the name of the file to be tokenized.
- Will read the code from the file and split it into lexemes assuming whitespaces and \n are used to separate all lexemes in code.
- Will classify the lexemes, one at a time, and will create a Token struct for each lexeme, which will contain:
  - The TokenType,
  - The lexeme.
- Eventually, will assemble all Token structs in an array of structs, which we are going to call our **Tokens Stream**, and is the output of the Tokenization task.

Then: Tokenizer 1.0 → Tokenizer v1.1

Additional features of Tokenizer v1.1 (for extra challenge):

1. The Tokenizer should now use more token types.
   a. Instead of having a single TOKEN_KEYWORD, it should have more Token types for keywords, such as TOKEN_KEYWORD_WHILE, TOKEN_KEYWORD_IF, etc.
   b. Instead of a single TOKEN_OPERATOR, it should have TOKEN_OPERATOR_ADD, TOKEN_OPERATOR_MUL, etc.
   c. Instead of a single TOKEN_PUNCTUATION, it should have TOKEN_PUNCT_OPENPAR, TOKEN_PUNCT_SEMICOLON, etc.
   d. Instead of a single TOKEN_LITERAL, it should have several possible types of literals such as TOKEN_LITERAL_INT, TOKEN_LITERAL_FLOAT, TOKEN_LITERAL_CHAR, TOKEN_LITERAL_STRING.
   e. This last part will probably require to come up with additional functions and RegEx to recognize new types such as floats, chars and strings.

Challenge: Tokenizer 1.1 → Tokenizer v1.2

Additional features of Tokenizer v1.2 (for extra challenge):

1. The Token struct should now include a third attribute, of type int, corresponding to the line index on which the lexeme was found in the source code.
   a. This will require to rework the string splitting part a bit and using a counter to keep track of the line index at which each lexeme was found.
   b. Later on, we will also rework the Token struct definition as well as the constructor and destructor functions to include the line_number attribute.
2. Along with the stream of tokens the Tokenizer 1.0 should return a bool with value True if no Tokens of types UNKNOWN appear in the Tokens Stream; and False otherwise.
3. If a Token of type UNKNOWN appears, there should be an error message showing the problematic lexeme and the line index at which it appears in the source code.