# Tokenizer and Parser

## Lab

# Task for this week

- Implement Tokenizer
- Implement Parser

- The code structure/skeleton is available on Wattle
- This lab contains assessable items!

- Submission Guidelines
  - The last slide contains information about the submission
  - Read it carefully to avoid losing marks!

# Task 1 – Tokenizer (0.5 marks)

The main objective of this task is to implement a **simple tokenizer**. The **input of the tokenizer is a string.** Your goal is to break down the character stream and **return a sequence of tokens**.

First, take a look at `Token.java` and understand the types of tokens we want to extract from a string. There are 7 types of tokens:

**INT**: a decimal 32-bit unsigned integer (e.g. 4869)
**ADD**: a single add character "+"
**SUB**: a single minus character "-"
**MUL**: a single star character "*"
**DIV**: a single slash character "/"
**LBRA**: a single left round bracket "("
**RBRA**: a single right round bracket ")"

```
public class Token {

    public enum Type {UNKNOWN, INT, ADD, SUB, MUL, DIV, LBRA, RBRA};
    private String _token = "";
    private Type _type = Type.UNKNOWN;

    public Token(String token, Type type) {
        _token = token;
        _type = type;
    }

    public String token() {
        return _token;
    }

    public Type type() {
        return _type;
    }
}
```

# Task 1 – Tokenizer

The main tokenizer class, **`MyTokenizer.java`**, inherits abstract class **`Tokenizer.java`** which has three methods:

**hasNext()** – check if there are still other tokens in the buffer

**current()** – return the current token

**next()** – find and extract a new token from buffer and store it in the current token

**`MyTokenizer.java`** implements **`Tokenizer.java`** and defines two additional private fields: **_buffer** and **currentToken**. **_buffer** keeps the string that we want to tokenize, and **currentToken** keeps a Token instance which is extracted by **next()** method.

Read the comments in the code to understand the details of how these functions must be implemented.

Check the constructor of **MyTokenizer**, which will initially save the input string to _buffer and extract the first token using next(). After, check the implementation of **hasNext**() and **current**() in **MyTokenizer**.

# Task 1 – Tokenizer

You job is to implement the missing part of the **next()** method. Do not change any other part of the code.

We implemented the (+) and (-) tokens for you. **The other methods should be implemented in a similar way** (the integer tokenizer may be a bit different). White space is also removed for you.

Modify the **next()** implementation such that it deals with the tokens "*", "/", "(", ")" and unsigned integer.

```
public void next() {
    _buffer = _buffer.trim(); // remove whitespace

     if(_buffer.isEmpty()) {
        currentToken = null;    // if there's no string left, set currentT
        return;
     }

    char firstChar = _buffer.charAt(0);
    if(firstChar == '+')
        currentToken = new Token("+", Token.Type.ADD);
    if(firstChar == '-')
        currentToken = new Token("-", Token.Type.SUB);

    // TODO: Implement multiplication and division tokenising
    // TODO: Implement left round bracket and right round bracket
    // TODO: Implement integer literal tokenising
    // HINT: Character.isDigit() may be useful
    // ########## YOUR CODE STARTS HERE ##########



    // ########## YOUR CODE ENDS HERE ##########

    // Remove the extracted token from buffer
    int tokenLen = currentToken.token().length();
    _buffer = _buffer.substring(tokenLen);
}
```

# Task 1 – Tokenizer

Before implementing the actual method, please go and check the test cases in **`TokenizerTest.java`**.

Test cases are one of the best ways to define the requirements of the code (see below one example).

```
@Test(timeout=1000)
public void testAddToken() {
    tokenizer = new MyTokenizer(passCase);

    //check the type of the first token
    assertEquals("wrong token type", Token.Type.ADD, tokenizer.current().type());

    //check the actual token value"
    assertEquals("wrong token value", "+", tokenizer.current().token());
}
```

# Task 1 – Tokenizer

Do not forget to implement your code within the block indicated by the following comments: 'YOUR CODE STARTS HERE' and 'YOUR CODE ENDS HERE'.

How to test your code?

The `TokenizerTest.java` file has 4 test cases, check each of them.

To assess your code, we will use 5 different test cases, each worth 0.5/5. We will test each method with empty tokens, and evaluate different expressions.

Again: check the submission guidelines to avoid losing marks (you may get zero marks)!

# Task 2 – Parser (1.5 marks)

The main objective of this part is to **implement a simple parser**. It is simple, but not easy, you need to dedicate some time to understand the code and start implementing it. Besides that, to complete this task, **you first need to complete Task 1**.

After you implement the tokeniser, copy the files `Token.java`, `MyTokenizer.java`, and `Tokenizer.java` to your Parser project.

Now the goal of this task is to implement a parser for the following grammar:

```
<exp> ::= <term> | <term> + <exp> | <term> - <exp>
<term> ::= <factor> | <factor> * <term> | <factor> / <term>
<factor> ::= <unsigned integer> | ( <exp> )
```

You can try to recreate it using this online parser:
https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/

# Task 2 – Parser

Try to understand the grammar and implement the parser.

Go through `Exp.java`, `AddExp.java`, `IntExp.java`, `SubExp.java`, `DivExp.java`, `MultExp.java` files:

Check the implementations of **show()** and **evaluate()** methods:

- **show()** method is designed to return the content of parsed expression
- **evaluate()** method evaluates and executes the expression and return the result.

Go through **`ParserTest.java`**:

**ParserTest** is a JUnit test class, which defines the proper behaviour of the parser through a set of examples.

Read the code and try to understand what the requirements of the parser are.

Implement the following missing parts of **`Parser.java`**:

- parseExp() method
- parseTerm() method
- parseFactor() method

*It can be difficult to start, but I will give you some hints in the next slide! If you do not understand it, ask for help, your tutor is there to help you understand it.

# Task 2 – Parser

Note that here I give some tips to start implementing **parseExp()**.

Try to understand all classes and check the **`ParserTest.java`**. It may help you understand even more the code.

Ask your tutor if you need help. It is important that you understand how to implement a parser!

**It is time to code!**

```
/*
<exp>    ::= <term> | <term> + <exp> | <term> - <exp>
<term>   ::=  <factor> | <factor> * <term> | <factor> / <term>
<factor> ::= <unsigned integer> | ( <exp> )
 */
public Exp parseExp() {
    // TODO: Implement parse function for <exp>
    // ########## YOUR CODE STARTS HERE ##########

    //You must parse <exp> here, note that it can be a <term> OR <term> + <exp> OR <term> - <exp>:
    //<exp> ::= <term> | <term> + <exp> | <term> - <exp>

    //START reading from LEFT to RIGHT

    //DECLARE AND READ term

    //After reading the term, check for the next token
    //hasNext?
        //true
        //Is the token + or - ? check for both

        //read the token, and the next variable
        //return the corresponding exp
        //  ...

    //if there is only one term (only <term>)
    //return term

    // ########## YOUR CODE ENDS HERE ##########
}
```

# Task 2 – Parser

Do not forget to implement your code within the block indicated by the following comments: 'YOUR CODE STARTS HERE' and 'YOUR CODE ENDS HERE'.

How to test your code?

The `ParserTest.java` file has 7 test cases, check each of them.

To assess your code, we will use 13 different test cases, each worth 1.5/13. We will test each method and evaluate different expressions.

Again: check the submission guidelines to avoid losing marks (you may get zero marks)!

# Task 2 – Parser

Note that we evaluate the correctness of the code based on the methods: **show()** and **evaluate()**.

```
@Test(timeout=1000)
public void testSimpleCase(){
    tokenizer = new MyTokenizer(SIMPLECASE);
    try{
        Exp exp = new Parser(tokenizer).parseExp();
        assertEquals("incorrect display format", "(1 + 2)", exp.show());
        assertEquals("incorrect evaluate value", 3, exp.evaluate());
    }catch (Exception e){
        fail(e.getMessage());
    }
}
```

# Submission Guidelines

- **Assignment deadline: see the deadline on Wattle (always!)**
- Submission mode: via Wattle (Lab Tokenizer and Parsing)

Submission format (IMPORTANT):

- Upload **only** your final version of `MyTokenizer.java` (for task 1) and `Parser.java` (for task 2) to Wattle
- Each test case must **run for at most 1000ms**, otherwise it will fail (zero marks).
- **Do not** change the file names
- **Do not** upload any other files (only the specified files are needed)
- **Do not** upload a folder (your submission should be only **two java files**).
- The answers will be marked by an automated marker.
  - **Do not** change the structure of the source code including class name, package structure, etc.
  - You are only allowed to edit the designated code segment indicated in the comments.
- **Do not** import packages outside of the standard java SE package. The list of available packages can be found here: https://docs.oracle.com/en/java/javase/12/docs/api/index.html
- Any violation of the submission format will result in zero marks
- Reference: see lecture slides / https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/