Compiler Manual

This manual should give insight into how you can use the compiler and how it works in the background. If there are any questions left, you can contact the team members.

How to use the compiler

Prebuild binaries can be found on the release page.

Usage on Linux

Start the compiler with the following command:

```
./compiler_linux [OPTIONS]
```

Usage on Windows

To run the compiler, open up a command line of your choice and start the compiler with the following command:

```
./compiler_windows.exe [OPTIONS]
```

Run from Source Code

If you intend to run the compiler directly from the source code instead of using a prebuild binary, you must install Python 3.6 or greater and pip.
After installing python and pip, install ply with the following command:

```
pip3 install ply
```

After cloning the repository, simply start the compiler with the following command:

```
python compiler.py [OPTIONS]
```

Option	Description
-h	Show the help menu with all options
-f [FILENAME]	Compile a file
-v	Verbose, print the decoded syntax for every code section
-t	Output the syntax tree corresponding to the onto the command line
-i	Output the syntax tree corresponding to the code as an image in browser

Syntax

Code that follows the syntax rules in this paragraph can be decoded successfully by this compiler. After each instruction(for example assigning a variable or printing information) you need to put a semicolon. Excluded from this rule are only the instructions if, while and for.
If there is an input that we can not decode you will get an error message that should help you to correct the error.

String:

A string can exist of the letters a-z and A-Z. The capitalization of letters gets decoded correctly and therefore matters. Mutated vowels and the sharp s are currently not working and won't be implemented. The keywords for the code can not be used as normal strings as well.

Numbers:

The syntax can work with two types of numbers: Integer and floating numbers.
Integer numbers consist of the algebraic sign at the first digit and as many digits with the numbers 0-9 as you want.
Floating Numbers consist of a normal integer number at the start, a point to split the integer and the decimal part and as many digits with the numbers 0-9 for the decimal places as you want.

Variables:

To initialize a variable of your liking please use this syntax:

type name = input;

To choose the variable you need to replace the placeholder "type" with the keyword for the variable you desire:

Type	**Identifier**
Integer	int
Float	float
String	string
Boolean	bool

By replacing the placeholder "name" you can set the name of your new variable. For the name of your liking you can use a normal string.
As an input for the variables float and int you can just write the number you wish. The input for the Boolean variables can be set to the keyword true or false. To save a string you need to put your input into quotation marks.
To update a variable you can use the syntax:

```
name = input;
```

Arithmetic Operations:

Arithmetic operations can be performed with numbers and variables. You can also use multiple operations in one big operation. To decide which operation we perform first we use the common rules for Algebra, which were specified in the book "La Géométrie" written by the author René Descartes. You can use the following operations:

Arithmetic Operation	**Syntax**
Addition	pl1 + pl2
Subtraction	pl1 - pl2
Multiplication	pl1 * pl2
Division	pl1 / pl2
Division with no remainder	pl1 // pl2
Modulo	pl1 % pl2

The placeholders pl1 and pl2 need to be changed to either your number or your variable. The spaces can be filled in, but are not needed for correct decoding.
The operations Division, Division with no remainder and Modulo do not accept the value zero for the placeholder pl2.(If you wonder why, Maths does not allow this either).
The operation Addition can also be used to attach to strings or variables of the type string together. To attach strings you need to put your input into quotation marks.

Logic Operations:

To compare two values with each other you can use the following logic operations:

Logic Operation	**Syntax**
or	pl1 || pl2
and	pl1 && pl2
equal	pl1 == pl2

not equal	pl1 != pl2
right greater	pl1 < pl2
left greater	pl1 > pl2

To connect multiple rules you can use brackets around the priorized operation. The operations OR and AND can only be performed to values of the type boolean.

Operations	**Syntax**
right greater	pl1 < pl2
right greater or equal	pl1 <= pl2
left greater	pl1 > pl2
left greater or equal	pl1 >= pl2

These operations can only be performed on numbers.

comment:

With the hashtag symbol at the beginning of the line, the whole line can be used as a comment and will not be decoded by the compiler.

IF:

To execute code only when certain conditions get fulfilled, you can use the if instruction. The syntax for the if instruction goes by the following:

```
if(condition){statement1}else{statement2}
```

The part “else{statement2}” can be left out if not needed. To use the if syntax in your code you need to replace the placeholder condition, statement1 and statement2 with codebits according to your use case. To work properly the code in place of the placeholder condition needs to be equal to one of both boolean values. Is the condition equal to true the code in place of statement1 gets executed. If the condition is equal to false the code in place of the placeholder statement2 gets executed.

While:

To execute the same code as long as a certain variable is equal to the boolean value true, you can use the while instruction. The syntax for this instruction goes by the following:

```
while(condition){statement}
```

To use this instruction in your code you need to replace the placeholder condition with your condition and the placeholder statement with the codebit that should be executed as long as the condition is equal to true.

For:

To execute the same code more than one time, you can use the for instruction. Because there are different implementations of for-loops in the different coding-languages, we decided to implement three different variants.

Code	Condition	Description
`for variable in (iterations){statement}`	- variable must be a string and is the name for the control variable - iterations must be a number or variable of type int - statement can be replaced by any standard code section	The code in place of the placeholder statement gets executed as many times as specified by the number in place of iterations. For every iteration the value of the control variable gets updated.
`for variable in (start; end){statement}`	- variable must be a string and is the name for the control variable - start & end must be a number or variable of type int - statement can be replaced by any standard code section	The code in place of the placeholder statement gets executed as many times as specified by the difference between start and end. For every iteration the value of the control variable gets updated.
`for variable in (start; end; step){statement}`	- variable must be a string and is the name for the control variable - start, end & step must be a number or variable of type int or float - statement can be replaced by any standard code section	The code in place of the placeholder statement gets executed as many times as the step section fits between the start and end value. For every iteration the value of the control variable gets updated.

predefined functions

For a bigger functionality of the compiler we decided to implement predefined functions:

- If you want to print a piece of information during running the code, you can use the syntax

```
print(pl1);
```

  For your usagecase you can replace the placeholder pl1 with either Strings, Numbers or even Variables.

Developer Documentation

This part of the documentation contains the information needed to understand the source code for the compiler and to implement new features.

General

The compiler is entirely written in Python.
The compiler makes use of the library Ply, which is a “pure-Python implementation of the compiler construction tools lex and yacc”.
It provides the same functionality as their respective tools, but without the limitations of c and the features of python, which in general simplifies the whole source code, but also eliminates the need for memory management as it would be needed in c.

Tokens

The tokens, which then will be used in the lexical and syntactic analysis, are stored in tokens/tokens.py.
The tokens are stored in a list, which then gets used by the lexical and syntactic analysis.
In the same file the reserved keywords are defined in a dictionary, with their token and their corresponding value for matching.

Lexical analysis

The lexical analysis for the corresponding token is defined in lexical/lexical.py.
To define a regex for a specific token, the following syntax is used:

```
t_<TOKEN_NAME> = <MATCHING_REGEX>
# Example for the AND-Token
t_AND = r'&&'
```

If additional actions are needed after matching the token, for example changing the type of

the value of the token, a function should be defined for the specific token.
The function gets a Token-Object as a parameter, which then can be used to change the attributes of that token and should be returned at the end of the function.

```
def t_<TOKEN_NAME>(t):
      <MATCHING_REGEX>
      <CODE_TO_EXECUTE>
# Example for the NUMBER-Token
def t_NUMBER(t):
      r'\d+\.?\d*'
      t.value = float(t.value)
      return t
```

With `t_ignore = <MATCHING_REGEX>` specific characters can be ignored.

For improved error output, newlines should be counted, to include the line number in case of an error.
To count the line number, the following function is used.

```
def t_newline(t):
      r'\n'
      t.lexer.lineno += len(t.value)
```

In case of an error, Ply uses a standard error-function.
In our source code, the error message includes the illegal character and the corresponding line number. After printing the corresponding error message, the line simply gets ignored..

```
def t_error(t):
      print("Illegal character " + t.value[0] + " on line " +
str(t.lexer.lineno))
      t.lexer.skip(1)
```

Syntactical Analysis

The syntactic analysis is splitted into multiple rules, to improve the readability of the source code.
All syntax rules can be found in syntax/rules/.
The “entry” rule for each analysis is the ‘statements’-rule which can be found in syntax/rules/expressions.py.
To define a grammatical rule, a function with a corresponding grammar specification and their corresponding actions must be defined. The defined function must include one parameter, so that an object with the parsed values can be given to the function.

```
def p_<RULE_NAME>(p):
      <GRAMMAR_SPECIFICATION>
      <CODE_TO_EXECUTE>
# Example for matching the print statement (print("Test"))
def p_print(p):
      'expression : PRINT ROUND_START expression ROUND_END'
      p[0] = ['PRINT', p[3]]
```

Each grammar rule should return an array, which includes instructions for the interpreter and the values needed for the execution of the instruction, as obtained from further syntactic parsing.

Interpreter

The interpreter itself can be found in executor.py.
It loops through the arrays returned from the syntactical analysis, and executes the according statement with the according values.

The additional values itself can, depending on the instruction, contain another array with further instructions. These come in place, if f.e. a loop should execute a code block multiple times. The instructions inside the sub-array correspond to the corresponding code block defined inside the curly brackets, and simply gets executed.

```
def execute_tree(syntax_tree):
      for statement in syntax_tree:
            result = execute(statement)
            print(result)

def execute(tree):
      if not isinstance(tree,list):
            return tree
      #Additional if-statements
if tree[0] == 'FOR':
      loopVar = tree[1]
      loop_reptitions = int(execute(tree[2]))

      if verbose: print('FOR-LOOP-START (' + str(loop_repetitions) +
')')
      for x in range(loop_repetitions):
            savedVariable.update({loopVar : x})
            execute_tree(tree[3])
      return 'FOR-LOOP-END'

if tree[0] == 'PRINT':
      print(execute(tree[1]))
```

```
      return
#Additional if-statements

#Example interpreting
#Initial syntax tree:
#[['FOR', 3.0, [['PRINT', 'Hello World']]]]
#Output:
# FOR-LOOP-START (3)
# Hello World
# Hello World
# Hello World
# FOR-LOOP-END
```