***Group 31**: BERNAD Thomas (50221636) & GUICHARD Lucas (50221623)*
***Deadline**: June 10, 2023 (Spring Semester)*
***Professor**: KIM Hyosu*

TERM-PROJECT COMPILER

01: CODE → VDECL CODE | FDECL CODE | CDECL CODE| ε
02: VDECL → vtype id semi | vtype ASSIGN semi
03: ASSIGN → id assign RHS
04: RHS → EXPR | literal | character | boolstr
05: EXPR → EXPR addsub EXPR | EXPR multdiv EXPR
06: EXPR → lparen EXPR rparen | id | num
07: FDECL → vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
08: ARG → vtype id MOREARGS | ε
09: MOREARGS → comma vtype id MOREARGS | ε
10: BLOCK → STMT BLOCK | ε
11: STMT → VDECL | ASSIGN semi
12: STMT → if lparen COND rparen lbrace BLOCK rbrace ELSE
13: STMT → while lparen COND rparen lbrace BLOCK rbrace
14: COND → COND comp COND | boolstr
15: ELSE → else lbrace BLOCK rbrace | ε
16: RETURN → return RHS semi
17: CDECL → class id lbrace ODECL rbrace
18: ODECL → VDECL ODECL | FDECL ODECL | ϵ

1) <u>Discard an ambiguity in the CFG:</u>

In production 01 (CODE → VDECL CODE | FDECL CODE | CDECL CODE | ε), the non-terminal CODE can derive a sequence of VDECL, FDECL, and CDECL non-terminals followed by another CODE non-terminal. This allows for multiple derivations.

In production 05 (EXPR → EXPR addsub EXPR | EXPR multdiv EXPR), the non-terminal EXPR on both sides allows for ambiguity in expressions involving addition/subtraction and multiplication/division.

In production 11 (STMT → VDECL | ASSIGN semi), the non-terminal STMT can derive either a VDECL or an ASSIGN statement followed by a semi-colon.

To remove it we can rewrite the production like so:
01: CODE -> DECLS
02: DECLS -> DECL DECLS | epsilon
03: DECL -> VDECL | FDECL | CDECL
04: VDECL -> vtypeid semi | vtype ASSIGN semi
05: ASSIGN -> id assign RHS
06: RHS -> EXPR | literal | character | boolstr

07: EXPR -> EXPR addsub TERM | TERM
08: TERM -> TERM multdiv FACTOR | FACTOR
09: FACTOR -> lparen EXPR rparen | id | num
10: FDECL -> vtype id lparen ARGS rparen lbrace BLOCK RETURN rbrace
11: ARGS -> vtype id MOREARGS | epsilon
12: MOREARGS -> comma vtype id MOREARGS | epsilon
13: BLOCK -> STMT BLOCK | epsilon
14: STMT -> VDECL | ASSIGN semi | IFSTMT | WHILESTMT
15: IFSTMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE
16: ELSE -> else lbrace BLOCK rbrace | epsilon
17: WHILESTMT -> while lparen COND rparen lbrace BLOCK rbrace
18: COND -> EXPR comp EXPR | boolstr
19: RETURN -> return RHS semi
20: CDECL -> class id lbrace ODECL rbrace
21: ODECL -> VDECL ODECL | FDECL ODECL | epsilon

2) Construct a SLR parsing table for the non-ambiguous CFG:

You can find in *SLRParserGenerator.pdf* file, the constructed SLR parsing table for the non-ambiguous CFG. To create it, we used the tool specified in the subject : https://jsmachines.sourceforge.net/machines/slr.html website.

3) Implement a SLR parsing program for the simplified Java programming language by using the constructed table:

We use python3 for this project. Then, to use our program you just need to execute this command in your Linux terminal:

python3 syntax_analyzer.py examples/*your_desired_file*.java

As you will see, we provide in total 3 input files in examples/ folder that you can use freely:

- Adder.java
- OctalDecimal.java
- LeastCommonMultiple.java

The output of our program is in two steps. First, we display the tokens' value and type obtained from the input file and secondly, the parsing tree is displayed.

```
----------TOKENS----------
class   (L1)    --->    class
Adder   (L1)    --->    id
{       (L1)    --->    lbrace
public  (L3)    --->    id
static  (L3)    --->    id
void    (L3)    --->    vtype
main    (L3)    --->    id
(       (L3)    --->    lparen
String  (L3)    --->    vtype
args    (L3)    --->    id
)       (L3)    --->    rparen
{       (L3)    --->    lbrace
int     (L5)    --->    vtype
first   (L5)    --->    id
=       (L5)    --->    assign
```

```
----------PARSE TREE----------
CODE
  CDECL
    class
    Adder
    ODECL
      VDECL
        int
        first
        ASSIGN
          =
          EXPR
            TERM
              RHS
                10
      ODECL
        VDECL
```

4) Bonus:

- As you can see above, we can give in input a real java file and not just a list of tokens as suggested in the subject. This feature allows us to create a more complete syntax analyser, just like in the real world.
- Also this pre-parsing step is able to avoid java comment lines (“// Comment”).
- All our code is inline documented. You can read it and understand it step by step.
- Also, when we display an error, the line when it occurs is specified.

```
Exception: Parsing error (L3): Expected rbrace, found id
```