Syntax of this year's new version of the *Recursive Student-Programming-Language*, **RecSPL 2024**

Note: Every **block of text highlighted by a background colour** can be regarded as ONE TOKEN coming from the Lexer

// some comments are provided, too, such that you can better understand the idea behind the syntax

---

```
PROG      ::=  main GLOBVARS ALGO FUNCTIONS

GLOBVARS  ::=  // nullable
GLOBVARS  ::=  VTYP VNAME , GLOBVARS // there can be as many glob-vars as we like

VTYP      ::=  num
VTYP      ::=  text

VNAME     ::=  a token of Token-Class V from the Lexer  // see the Appendix below

ALGO      ::=  begin INSTRUC end

INSTRUC   ::=  // nullable
INSTRUC   ::=  COMMAND ; INSTRUC

COMMAND   ::=  skip  // an empty algorithmic step in which nothing happens
COMMAND   ::=  halt
COMMAND   ::=  print ATOMIC
COMMAND   ::=  ASSIGN
COMMAND   ::=  CALL // call to a void-function that only updates global variables
COMMAND   ::=  BRANCH
                    // Note: no LOOP, because we use functional recursions instead of loops

ATOMIC    ::=  VNAME
ATOMIC    ::=  CONST

CONST     ::=  a token of Token-Class N from the Lexer  // see the Appendix below
CONST     ::=  a token of Token-Class T from the Lexer  // see the Appendix below

ASSIGN    ::=  VNAME < input   // from the user during run-time
ASSIGN    ::=  VNAME = TERM // Deep nesting of assignment terms is allowed: see below

CALL      ::=  FNAME( ATOMIC , ATOMIC , ATOMIC )   // we only allow un-nested params
                                                   // such that our Project will not get too complicated

BRANCH    ::=  if COND then ALGO else ALGO // also our Conditions will be quite simple

TERM      ::=  ATOMIC
TERM      ::=  CALL // call to a result-function that emits a return-value
TERM      ::=  OP    // in general, operations in assigments can be deeply nested: see below

OP        ::=  UNOP( ARG )        // for simplicity we do not allow function-calls as args
OP        ::=  BINOP( ARG , ARG )
```

ARG ::= ATOMIC
ARG ::= OP // *this recursive rule permits the deep-nesting of operations*

COND ::= SIMPLE // *for simplicity we do not allow very deeply nested Conditions;*
COND ::= COMPOSIT // *we permit only one level of nesting Conditions in this project*

SIMPLE ::= BINOP**(** ATOMIC **,** ATOMIC **)**
COMPOSIT ::= BINOP**(** SIMPLE **,** SIMPLE **)**
COMPOSIT ::= UNOP **(** SIMPLE **)**

UNOP ::= **not**
UNOP ::= **sqrt** // *the square root of real numbers*

BINOP ::= **or**
BINOP ::= **and**
BINOP ::= **eq**
BINOP ::= **grt** // *greater than >*
BINOP ::= **add**
BINOP ::= **sub**
BINOP ::= **mul**
BINOP ::= **div**

FNAME ::= a token of **Token-Class F** from the Lexer // *see the Appendix below*

FUNCTIONS ::= // *nullable*
FUNCTIONS ::= DECL FUNCTIONS

DECL ::= HEADER BODY

HEADER ::= FTYP FNAME**(** VNAME **,** VNAME **,** VNAME **)**
// *for simplicity, all our functions have 3 "incoming" parameters*

FTYP ::= **num**
FTYP ::= **void**

BODY ::= PROLOG LOCVARS ALGO EPILOG SUBFUNCS **end**

PROLOG ::= **{** // *the prolog is an important concept, as you will see later in chapter 9*
EPILOG ::= **}** // *the epilog is an important concept, as you will see later in chapter 9*

LOCVARS ::= VTYP VNAME **,** VTYP VNAME **,** VTYP VNAME **,**
// *for simplicity, all our functions have 3 local variables*
// *in addition to their three "incoming" parameters*

SUBFUNCS ::= FUNCTIONS // *we allow functions to have their own local sub-functions*

APPENDIX: Lexical Categories, presented as Regular Expressions

Token-Class V for user-defined variable names: **V_**[**a**–**z**]([**a**–**z**]|[**0**–**9**])*

// *Note: the prefix* **V_** *makes it easy for you to distinguish variables from the reserved keywords*

Token-Class F for user-defined function names: **F_**[**a**–**z**]([**a**–**z**]|[**0**–**9**])*

// Note: the prefix **F_** *makes it easy for you to distinguish functions from the reserved keywords*

Token-Class T for short snippets of text (strings):

"[A-Z][a-z][a-z][a-z][a-z][a-z][a-z][a-z]" |
"[A-Z][a-z][a-z][a-z][a-z][a-z][a-z]" |
"[A-Z][a-z][a-z][a-z][a-z][a-z]" |
"[A-Z][a-z][a-z][a-z][a-z]" |
"[A-Z][a-z][a-z][a-z]" |
"[A-Z][a-z][a-z]" |
"[A-Z][a-z]" |
"[A-Z]"

// Note: For the sake of simplicity our short strings contain ***up to*** *eight characters*
// between quotation marks, whereby the string's first character is Capitaliszed.

Token-Class N for numbers (which are composed of digits and may possibly include a decimal dot):

0 |
0.([**0**–**9**])* [**1**–**9**] |
-0.([**0**–**9**])* [**1**–**9**] |
[**1**–**9**]([**0**–**9**])* |
-[**1**–**9**]([**0**–**9**])* |
[**1**–**9**]([**0**–**9**])***.** ([**0**–**9**])* [**1**–**9**] |
-[**1**–**9**]([**0**–**9**])***.** ([**0**–**9**])* [**1**–**9**]

// Note: in our programming language we do not distinguish between INT and REAL
// The ***dot*** *here belongs to the language itself, not to the meta language of Regular Expressions!*
// The yellow Minus- belongs to the language itself, for the composition of Negaive Numbers.
// The white Dash– belongs to the meta language of Regular Expressions!

Token-Class Reserved Keyword:

Everything that is **green** in the context-free grammar of above belongs to this general class of 'fixed' tokens which are *not* user-defined. For the sake of project-simplicity these tokes have been defined in such a manner that they **can be *very easily* distinguished** by the Lexer from each other as well as also from the user-defined tokens. In this way *you can simply avoid most of the complications* that we had discussed in the context of Chapter 1 (Section 1.8) and in Homework #2.
Moreover, such as in Part **d)** of our Homework #2 we will also use **blank spaces** (and/or line break) in order to "help" our Lexer with its decision-making about when to accept a token and to re-set the DFA to its start-state for the next token's identification.
