Syntactic analysis

For the *parser*, we have decided to implement our LL-grammar design as recursive descent. The main advantages are significant decomposition, transparency of the code, as same as simplicity of extension of the functionality etc. It requires a bit more abstract point of view on the designed model during the implementation, nevertheless the draft is still clear to see.

The *parser* gains tokens from the *scanner* and calls the functions to process the tokens (syntactically, call the semantic check in *pedant*, generate code in *generator*). The functions expect the exact sequence of tokens of the exact type, however if it will not acquire the right ones, the error is raised. This process is similar to the designed one, it does not include the direct implementation of the stack for the pushdown automaton though.

When the numeric expression is expected (arguments in function call, right side of assignment), the separate predictive parser (*ExpressionParse*) is called. The result is a stack, saved in semantic analyzer (*pedant*).

Logical expression (in the condition, or cycle) is understood as two expressions separated with a comparison sign.

The scanner labels the identifiers of functions and variables as general symbols. It is then retyped, based on the context in the *parser*.

Semantic analysis

After the predictive parser processes the expression, it is saved in the pedant as pre-fixed stack. At the end of the creation, predictive parser turns the stack upside down, and checks the type compatibility. If it is needed, the special tokens are inserted after the operand which needs to be converted. In case of incompatibility, it raises error. It is then left behind in the *pedant*.

Afterwards, the *parser* is accepted to proceed and it sends the target datatype, expression has to be casted to. The stack is then generated, the possible conversion afterwards, and the target to assign to.

For the logical comparison, the last type has to be memorized. When the second stack will come, the typecast is send to convert the first, or the second and typecast to the second one, or none of it.

Generator

*Generator* possesses a state stack (*GState stack*) and a label stack. When it handles stack from *pedant* or single token from *pedant* or *parser*, the top of the state stack indicates, what to do with it.

The state stack is operated directly from the *parser*, which calls functions *G\_\**, that pushes the state onto the stack, or directly generates some code, or both. The result is, that the *generator* knows what to do, when it receives a stack or a token is send, and generates correctly.

When a block (function, condition, loop) appears, the comparison and jumps are generated and for that, unique name generator is essential. Our compiler, generating label names 7 characters long, is limited to 56,8 billion of jumps, names are in format $aaaaaa, which appears as sufficient. The label name is also pushed to the special *label stack*.

Later, when the block ends, the additional instructions, such as labels and jumps are needed, so the function *G\_EndBlock()* is called. The *generator* then finds what to end in the *GState stack*, and gains the right data from *label stack* to insert the label name to the instruction.

Printing is, due to absence of instruction, that would print the top of the stack, also provided by generating of new temporary frame (to avoid the duplicity of the name when printing in cycle), creating a variable to it, and print the variable.

When a declared, but not defined function is called, the parameter names are not known to assign the arguments to in the function call. It is replaced with \*<order> and moved in the function body. It also sets the limit to the 999 parameters in single function.