LAB 1a

p1

```
declare int a, b, c, max.  read int a, b, c.
max :=
a.

if a < b then
{
        max := b.
        if b < c then max := c.
}
otherwise  if a < c then max   :=
        c.   print max.
```

p2

```
declare int nr.     declare boolean ok :=  true.
read int nr.

if nr < 2 then ok := false.

for i: <2, nr / 2, 1>  if nr % i = 0 then ok := false.      print
ok.
```

p3

```
declare int n, sum := 0.  declare list  int
numbers[n].    read int n.

for i: <0, n - 1, 1>  read numbers[i].
        sum := sum + numbers[i].
print sum.
```

p1err

```
declare str “mystring .   ~didn’t close quotation marks  declare int
n3.   ~identifiers can contain only letters
```

LAB 1b

Lexic

a. Special symbols, representing:

- Operators + - * / % < <= = \= => > := and or not
- Separators [ ] { } , . space
- Reserved words: declare read int string boolean list for while function if otherwise then print
- Comments ~

b. Identifiers: a sequence of letters with no digits

- identifier ::= letter { letter }
- letter ::= “A” | “B” | … | “Z” | “a” | “b” | … | “z”

c. Constants:

- Integer:

intconst ::= [“+” | “-“] nzDigit { “0” | nzDigit }

digit ::= “0” | “1” | … | “9”

nzDigit ::= “1” | … | “9”

- String:

stringconst ::= “ \” “ { letter | digit} ” \” “

- Boolean:

booleanconst ::= “true” | “false”

Tokens

int string boolean declare read int string boolean list for while function if otherwise then print ~ + - * / % < <= = \= => > [ ] { } , .

Syntax

Program ::= Stmt { Stmt }

Stmt ::= (DeclareStmt | AssignStmt | PrintStmt | IfStmt | WhileStmt | ForStmt | ReadStmt) "."
DeclareStmt ::= "declare" type (identifier ["[" intconst "]"] | AssignStmt)

Type ::= "int" | "string" | "boolean"

AssignStmt ::= identifier ":=" term

Term ::= factor [operator (factor | term)]

Operator ::= "+" | "-" | "*" | "/" | "%"

Factor ::= identifier | intconst | stringconst | booleanconst

IfStmt ::= "if" Condition "then" Stmt { Stmt } ["otherwise" Stmt { Stmt }]

Condition ::= term relOperator term

RelOperator ::= "<" | "> " | "=" | "<=" | "=>" | "\="

WhileStmt ::= "while" Condition "execute" Stmt { Stmt }

ForStmt ::= “for” identifier “:<” (intconst | term) “,” (intconst | term) “,” intconst “>” Stmt { Stmt }

PrintStmt ::= “print” term

ReadStmt ::= “read” type identifier

LAB 2 & 3 https://github.com/sekilm/University/tree/main/FLCD

The scanner implementation uses the following approach:

- We initialize the symbol table as a hash table. The has table uses a simple chaining approach where nodes will be used to chain items that land on the same position. The hash function consists of computing the sum of the identifiers’ characters ascii values. The insert function adds the identifier based on the hash function and the find function returns the position (a.k.a. the ascii sum) of the searched identifier. To the symbol table we add the tokens which are identifiers or constants.
- We initialize the PIF as a list to which we add every pair of (token, symbol table position).
- The tokens will be read and saved in a list, from a text file using the get_tokens function.
- We read the text file that contains our problem (which we can swap by changing the filename variable at the beginning), splitting every line by every symbol in our language. Reserved words, operators and separators will be added to PIF, but not to the symbol table, while identifiers and constants will be added to both. If any token is not part of any of the above categories, a lexical error will be printed in the console.
- The symbol table and the PIF are both printed in their own files.