The minilanguage can be a restricted form of a known programming language, and should contain the following:

- 2 simple data types and a user-defined type:

```
// store whole numbers
integer

// store text (strings)
text

/*
  a structure, just like in C/C++, used to
  store multiple data types, useful when
  sending packs of data
*/
inventory: structure {
 product_quantity: dictionary;
 bills: list;
 money: integer;
};
```

- statements:
 - assignment:
  - a = 100000;
 - input/output:
  - a = input();
  - output(a);
 - conditional:
  - if(true) {}
  - match('W') { case 'A': break; default: break; }
 - loop:
  - for(start = 0; start < 100; start++) {}
  - while(true) {}

- some conditions will be imposed on the way the identifiers and constants can be formed:

- i) Identifiers: no more than 256 characters

- ii) constants: corresponding to your types

Example: the minilanguage specification should include lexical and syntactical details:
Specification (file Lexic.txt)

Alphabet:
- a. Upper (A-Z) and lower case letters (a-z) of the English alphabet
- b. Underline character '_';
- c. Decimal digits (0-9);

Lexic:
- a. Special tokens, representing:
 - operators:
  - addition: '+'
  - substraction: '-'
  - multiplication: '*'

- division: ‘/‘
- remainder of the division: ‘%‘
- to the power of: ‘**‘
- assignment: ‘=‘
- add and assign: ‘+=‘
- substract and assign: ‘-=‘
- divide and assign: ‘/=‘
- multiply and assign: ‘*=‘
- to the power of and assign: ‘**=‘
- less than: ‘<‘
- less or equal than: ‘<=‘
- greater than: ‘>‘
- greater or equal than: ‘>=‘
- equal: ‘==‘
- not equal: ‘!=‘
- logical and: ‘and‘ | ‘&&‘
- logical or: ‘or‘ | ‘||‘
- increment: ‘++‘
- decrement: ‘--‘
- return type: ‘->‘
- selector: ‘.‘

- separators:
- tokens: ‘,‘
- statements: ‘;‘
- blocks: ‘{}‘
- types | labels: ‘:‘
- size: ‘[]‘
- callables: ‘()‘
- words: ‘ ‘

- reserved words:
- ‘nothing‘
- ‘integer‘
- ‘text‘
- ‘character‘
- ‘boolean‘
- ‘true‘
- ‘false‘
- ‘list‘
- ‘dictionary‘
- ‘structure‘
- ‘if‘
- ‘else‘
- ‘match‘
- ‘case‘
- ‘while‘
- ‘for‘
- ‘fun‘
- ‘return‘
- ‘fixed‘
- ‘try‘
- ‘catch‘
- ‘throw‘
- ‘go‘

- b. identifier:
 - ‘variable‘
 - ‘_variable‘
 - ‘variable1‘
 - ‘_variable1‘
 - ‘_1‘
 - ‘_1st_variable_‘

- a sequence of letters and digits, such that the first character is a letter (or underscore); the rule is:
 - identifier = {‘_‘}letter{‘_‘} | {‘_‘}digit{‘_‘} | {‘_‘}letter{‘_‘}{letter}{‘_‘}{digit}{‘_‘} |
{‘_‘}digit{‘_‘}{letter}{‘_‘}{digit}{‘_‘}
 - letter = "a" | ... | "z" | "A" | ... | "Z"
 - digit = "0" | ... | "9"

- c. constants: ‘fixed‘ ‘IDENTIFIER‘ ‘:‘ ‘TYPE‘ | arraydecl

- 1. integer:
 - noconst := ‘+‘ number | ‘-‘ number | number

 - number := nonzerodigit{digit} | digit

- 2. character:
 - character := ’letter’ | ’digit’

- 3. string:
 - constchar := "string"

 - string := char{string}

 - char := ’letter’ | ’digit’

Syntax:
- Sintactical rules: (file Syntax.in)

- program := ‘fun‘ ‘main‘ ‘(‘ PARAMETERS ‘)‘ ‘->‘ ‘nothing‘

- decllist := declaration | declaration ‘,‘ decllist

- declaration := IDENTIFIER ‘:‘ type

- type_value := ‘nothing‘ | ‘boolean‘ | ‘character‘ | ‘integer‘ | ‘text‘ | ‘list‘ | ‘dictionary‘ | ‘structure‘

- arraydecl := type_value ‘[‘ SIZE ‘]‘

// example: an array of 100 integers with the name "numbers"
numbers: integer[100]

- type := type_value | arraydecl

- cmpdstmt := ‘{‘ statement1 ‘;‘ ... statementN ‘;‘ ‘}‘

- stmtlist := stmt | stmt ‘;‘ stmtlist

- simplstmt := assignstmt | iostmt

- stmt := simplstmt | structstmt

- assignstmt := IDENTIFIER ‘=‘ expression

- expression := expression ‘+‘ term | term

- term := term ‘*‘ factor | factor

- factor := ‘(‘ expression ‘)‘ | IDENTIFIER

- iostmt := ‘input(‘ TEXT ‘)‘ | ‘output‘ ‘(‘ IDENTIFIER ‘)‘

- structstmt := cmpdstmt | ifstmt | whilestmt

- ifstmt:

```
// if statement
if(condition) {
 statement;
} else {
 statement;
}
```

- matchstmt:

```
// match statement
match(expression) {
 case CONST:
  statement;
  break;

 default:
  break;
}
```

- whilestmt:

```
// while statemen
while(condition) {
 statement;
}
```

- condition = ‘(‘ expression RELATION expression ‘)‘

- RELATION = ‘<‘ | ‘<=‘ | ‘=‘ | ‘!=‘ | ‘>=‘ | ‘>‘