MASTER
INFORMATIQUE

Calculability

GUI for RAM programs with parser and interpreter

Authors:
Schmied Margaux
Galbiati Federica
Fissore Davide
Venturelli Antoine

Professor:
Enrico Formenti

2021/2022

Contents

1 Introduction

For the calculability course, we had to realize a project. This project is a RAM language interpreter. The RAM language is a pseudo code discussed in class with 4 instructions. This project is divided in 3 parts: the parser, the interpreter and the graphics.

2 About RAM program

A *RAM* program is a program made by :

- a potentially infinity memory with three main registers
  - the *memory counter (R_C)* which allows to know at each moment what is the current instruction to be executed. If R_C is 0 or a integer bigger then the number of instructions of the code, the program execution will stop
  - R_0 the first register of the memory representing the input of the program
  - R_1 the second register which can be seen as the case of the memory containing the output of our program
- 4 instructions :
  - $R_k = R_k + 1$ which increases the value stock in the register k by one
  - $R_k = R_k \dot{-} 1$, same as previous instruction, but the value is decreased by one (NB : if the value in R_k is equal to 0, we have that $0 \dot{-} 1 = 0$ since RAM machines work in $\mathbb{N}$ number set
  - $IF\ R_k \neq 0\ THEN\ GOTOB\ n$ means that if the register k does not equal 0 then $R_C = R_C \dot{-} n$ (here, as before the subtraction is done as follow $A \dot{-} B = max(0, A - B)$
  - $IF\ R_k \neq 0\ THEN\ GOTOF\ n$ which makes the following operation : $R_C = R_C + n$

With a RAM program, as demonstrated in course, it is possible to code all the programs we usually code with any other programming language such as *Python, C, Java, etc.*

Another important aspect of RAM program we should consider is that, thanks to its infinite registers and the fact that efficiency time of programs is not taken, this program model can be reused and adapted to every new programming language.

Finally working only in $\mathbb{N}$ does not restrict RAM programs, since in fact $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ can be seen as an extensions of $\mathbb{N}$.

We can also introduce some predefined and useful macros, such as, *rp* and *lp* (resp. for *right_part* and *left_part*), allowing to treat the value of a register as a *Cantor's couple* (see 4.4), this let you pass a variable number of parameters.
For example, if you have a program that calculate the sum of x and y then you will put n that is $cantor(x, y)$ in R_0.
Other two macros are *push* and *pop* which respectively push and pop the value from a register k to a very big register that normally should not be used by the program (we have chosen the 2^{64} register).

3 Design choices

3.1 Choice of tools

- Technologies : **Python3.9.9**
  To realize this project we used the following libraries:
  - ply==3.11 (For the parser)
  - tkhtmlview==0.1.0 (For the help)

3.2 Installation

You can use the file **install.sh**

```
$ chmod u+x install.sh
$ ./install.sh
```

Or

```
$ python3 -m pip install -r requirements.txt
```

4 Integers as instructions or programs

4.1 Integer representation

When we deal with RAM program, as said in Section 2, we do not treat subtraction as if we were in $\mathbb{Z}$ or other set of number including $\mathbb{Z}$, we are supposed to be in the natural number world.

In a RAM program subtraction is represented by the $\dot{-}$ symbol giving us an expression of the type $A \dot{-} B = max(0, A - B)$ therefore the we created the *Int* class that extends the *int* class and overrides the __*sub*__ function as shown in Listing 1 .

Listing 1: The Int class

```
class Int(int):
    def __sub__(self, x: int):
        return Int(max(int(self) - x, 0))
    ...
```

4.2 Gödelisation function

As seen in course every instruction can be represented as an integer, thanks to Gödel bijective function (look at listing 2).

Listing 2: Gödel encoding function

```
    $Rk = Rk + 1 \rightsquigarrow 3 \times k$
    $Rk = Rk - 1 \rightsquigarrow 3 \times k + 1$
    $IF\ Rk \neq 0\ THEN\ GOTOB\ n \rightsquigarrow 3 \times [k, [1, [n, 0]]] + 2$
    $IF\ Rk \neq 0\ THEN\ GOTOF\ n \rightsquigarrow 3 \times [k, [0, [n, 0]]] + 2$
 Where :
    - k is the register number
    - is the number of jump to do inside a if
```

4.3 Gödel inverse function

When we are given an integer and we know that it is an instruction, we can easily decode it with modulo operations (look at listing 3).

Listing 3: Gödel decoding function

```
if $X \equiv 0 \mod 3 \rightsquigarrow$ we have a $Rk = Rk + 1$ instruction
if $X \equiv 1 \mod 3 \rightsquigarrow$ we have a $Rk = Rk - 1$ instruction
  in both cases, k is equal to $\lfloor X/3 \rfloor$
if $X \equiv 2 \mod 3 \rightsquigarrow$ we have a jump instruction
  where $X' = \lfloor X/3 \rfloor$ and we can decode $X$ in $[k, [j, [n, 0]]]$
  with Cantor inverse function (see next paragraph)
```

Decoding an instruction like can be done by the *decode_ int_ instr* in *decode_ int.py* file.

4.4 Cantor pairing function

Now, we know how to code an instruction, we can so code a program, that is a list of instruction, thanks to Cantor function (look at listing 4)

Listing 4: Cantor encoding function

```
    $[x, y] = \frac{(x+y)\times(x+y+1)}{2} + y + 1 = n$
    if b is a couple, we encode it at first,
    and that recursively until the last tuple
```

The encoding of two integer to a Cantor's int is made by the *cantor* method of the *Int* class.

4.5 Cantor inverse function

The Cantor inverse function is not as simple as computing the Gödel inverse one. The Cantor Pairing function is a *Diophantine equation* (voir definition at ...) and finding back the two variables x and y from n ask to iterate from 0 the n to find an intermediate n'.

Since n may be very big, (it is the case if we encode multiple instructions) this task may become very slow, therefore, to speed it, we make a *binary search* to have an answer in a logarithmic time instead of a linear one.

The decoding function of a Cantor's int to a couple of the form $(x_1, (x_2, (..., (x_n, 0))))$ is made by the *Int* class with the *int_ to_ couple* method.

Listing 5: Cantor decoding function

```
class Int(int):
    ...
    def cantor_inv(self):
        tmp = self.aux()
        r = self - ((tmp - 1) * tmp / 2 + 1)
        l = tmp - r - 1
        return l, r

    def int_to_couple(s):
        res = Int(s).cantor_inv()
        return res if res[1] == 0
               else (res[0], Int(res[1]).int_to_couple())
    ...
```

5 Pre-processing

5.1 Include

You have the possibility to include a file. Like the C language, just write *#include filename* where '*filename*' is the path of the file you want embedded. At the pre-processing phase, each line which contain such this instruction will be replace by the content of the precise filename. If the path doesn't exist, an error (that will not interrupt the execution) will be raised.

5.2 Define

This interpreter also offered the possibility of using the *#define name thing* instruction. Therefore, all '*name*' will be replaced by '*thing*' in the file

5.3 Some precisions

When you go to execute the file, you will not see your file modified by the preprocessing instructions, they will be done internally. However, you always have the option to do it manually to see it in **Tools** -> **Apply Preprocessing**. The current file will then be modified.

6 Parser

The first step of our RAM interpreter is to parse the instructions. At first we tried to use **ANTLR4**, it's a parser generator for reading, processing, executing, or translating structured text. Margaux having already used it, it seemed to be a good choice to make the parsing, but once most of the grammar was done, we had a problem.

When we wanted to add the name of the macros and each instruction was detected as a string caused by a priority action problem. Therefore, we started to seek for a solution and we found a Python library named **PLY** that was easier to use and seemed to solve our problem.

PLY is a Python implementation of the lex and yacc tools commonly used to write parsers and compilers. The parsing is based on the same **LALR** algorithm used by many yacc tools. **LALR** parser is a simplified version of an **LR** parser, for parsing text according to a set of production rules specified by a formal grammar for a computer language. **LR** is left-to-right derivation.

Our parser detects 10 main kind of grammar rules.

Listing 6: Detected instruction

```
0. Rk = Rk + 1
1. Rk = Rk - 1
2. if Rk != 0 then gotob n
3. if Rk != 0 then gotof n
4. push Rk
5. pop Rk
6. rp(Rk, Rl)
7. lp(Rk, Rl)
8. begin macro <name> (Rk, Rl, ..., Rz)<code> end macro;
9. <name> (Rk, Rl, ..., Rz)
```

6.1 Lexical analysis

The first step of our parser is to make the lexical analyse which verifies if the used vocabulary is correct. We have decided to allow only lower case letters except for the **R** in registers which can only be capitalized.

So we detected all the words that would be useful to us in addition to the strings for the names of the macros and the numbers for the registers.

6.2 Syntactic analysis

The creation of grammar A took longer than expected. It syntactic analysis verifies that the arrangement of words is coherent. We had to create a redundant grammar because the instructions to be parsed in the macros and outside are very similar. For example, in macros, we can have registers with a not known value (a register with name R_x is authorized inside a macro definition but not outside).

6.3 Semantic analysis

The semantic analysis verifies that the instructions have a meaning else we raise an error.

In listing 7 the number associated with the instructions is the same as in listing 6.

Listing 7: Semantic rules

```
1. The instructions 0 and 1 must add and subtract 1.
2. The instructions 2 and 3 must compare Rk with 0.
3. The register list of instruction 8 must contain only
   "register variables"[1].
4. The instruction 9 must have the same number of
   arguments as this statement.
5. All instructions of a macro using "register variables
   " must be those declared in the arguments of the
   macro.

[1] The variable registers are in Rstring format.
```

If one of the lexical, syntactic or semantic rules is not respected an error is raised indicating the line of the problem.

6.4 Creation of the chained list

All parsed instructions are translated into **Instruction** objects except for macro declarations which create **Macro** objects. **Macro** are a sub type of instruction that have the particularity to contain Instructions. When parsing the text we create a chained list with our instructions and we do the same for the **Instructions** in the **Macro**.

The Figure 1 shows the tree created from the code of Listing 8. The left branch represent the next Instruction and the right Instruction in the Macro.

Listing 8: Example of code with macro

```
begin macro clear(Rx)
    Rx = Rx - 1
    if Rx != 0 then gotob 1
end macro;
R0 = R0 + 1
clear(R0)
```

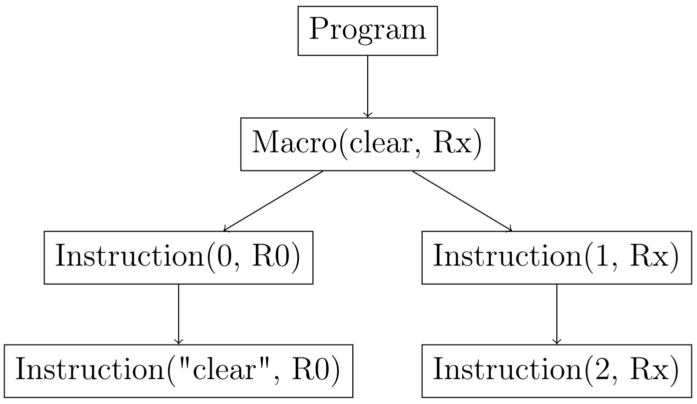


Figure 1: Tree of the example code

7 Interpreter

The phase of interpretation is made after the parsing. An interpreter is made of an (hypothetical) infinite memory, a set of macros definitions and a list of instruction and three special registers:

- $R_C \rightsquigarrow$ the register counter that indicates the current instruction
- $R_0 \rightsquigarrow$ the input register
- $R_1 \rightsquigarrow$ the output register

7.1 Internal representation of the memory

The memory of the interpreter is represented by the *RAM* class witch a class which extends the *dict* class of Python. This class redefines :

- the *getitem* method : if we try to access a not existing key of the *dict*, it is firstly created with the 0 value and then 0 is returned.
- the setitem method : which casts the value associated to the key as an *Int*

7.2 How the interpreter treats macros

When the interpreter receives a list of instructions to be interpreted, it tries at first to remove every instruction that is a macro call. For example if we have a code like the following :

Listing 9: RAM code with macro representing the $\times 2$ operation

```
begin macro times_two(Rx, Ry)
Ry = Ry + 1
Ry = Ry + 1
Rx = Rx - 1
end macro;
R0 = R0 + 1
times_two(R0, R1)
if R0 ≠ 0 then gotob 1
R1 = R1 - 1
R1 = R1 - 1
```

The interpreter will receive :

- The macro name *times_ two* which put $R0 \times 2$ in $R1$
- The list of 5 instruction :
  - R0 = R0 + 1
  - add_reg(R0, R1)
  - if R0 ≠ 0 then gotob 1
  - R1 = R1 - 1
  - R1 = R1 - 1

At first, the interpreter reads every instruction and when it detects the

times_ two(R0, R1)

macro call, it replaces that line by the body of the macro definition, where parameters of body of the the macro has been replaced by the macro call arguments.

It is important to note, that if we only perform this substitution, our code will be broken, since the jump inside *IF* statements will not work as expected.

In fact, when we make a macro-call replacement, we should always keep attention on the number of step to do and actually modify them.

The code of Listing 9 will be so replaced by the following one :

Listing 10: RAM code without macro representing the $\times 2$ operation

```
R0 = R0 + 1
R1 = R1 + 1
R1 = R1 + 1
R0 = R0 − 1
if R0 ≠ 0 then gotob 3
R1 = R1 − 1
R1 = R1 − 1
```

7.3 A bug we still have in jump

The code is able to correct the jump in *IF* statements in the case where the *IF*

- is not inside a macro
- is inside a macro but it does not *go out* from the macro body

Let’s look Listing 11.

Listing 11: RAM code causing a bug

```
begin macro macro_little_jump(Rx)
if Rx != 0 then gotob 1
end macro;
macro_with_1000_lines(R10)
macro_little_jump(R0)
```

In this example, we should notice that the jump inside the macro *macro_little_jump* should be replaced by an *IF* with a jump of 1000 + 1 lines (the number of lines of *macro_with_1000_lines* + the original jump = 1), but in reality the jump is not modified.

The macro with an *IF* statement, in fact, only consider its own body definition.

Notwithstanding this, a code like in Listing 12, will work, since the *IF* jump remains inside the macro scope and it will be corrected to a jump of 1001 steps.

Listing 12: RAM code causing a bug

```
begin macro macro_little_jump(Rx)
macro_with_1000_lines(R10)
if Rx != 0 then gotob 1
end macro;
macro_little_jump(R0)
```

7.4 Interpretation of instructions

Once we have ended the macro substitution process, we can finally compute our instructions one by one or the whole program till

$$R_C = 0 \ \vee \ R_C > |instructions|$$

via the *execute* method on every *Instruction*.

We are allowed to get the state of the memory of our program at every moment via the *memory* attribute of the *Interpreter* or get the output (that is R_1) thanks the *get_output* method.

7.5 Predefined macros

The interpreter knows some predefined macros :

- *push* R_k which pushes the the content of R_k in a very big register (2^{64})
- *pop* R_k which takes the value from register 2^{64} and puts it in R_k
- *rp(*R_x*,* R_y*)* which takes the *right part* of R_x and put it in R_y
- *lp(*R_x*,* R_y*)* which take the *left part* of R_x and put it in R_y

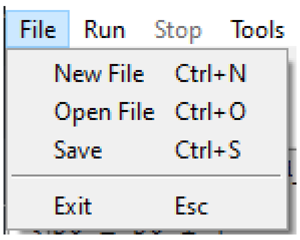
7.6 Parsed object to Interpreter one

In Sections 6 we have said that parser needs chained lists to create step by step the object that will be returned after the parsing. On the other hand, for reason of time complexity, the interpreter must work with normal list since accessing the *i-th* element should be as fast as possible. We know that getting the last element of a chained list takes a linear time, whereas in a *Python* list the time needed should be almost constant.

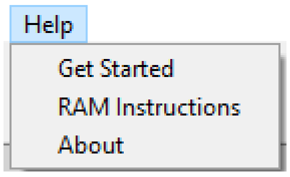
Therefore, after the parsing phase, it is necessary to operate this transformation so that the interpreter can work.

The *parser_instr_to_interp_list* function in *pars_to_interp.py* file does exactly this task.

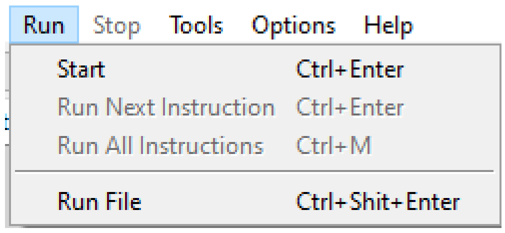
We create a *list* L of instruction and a *dictionary* D of macros and then we go through the chained list. So, if we find a *macro* definition we add it to the D and if we find an *instruction* then we will append it into L.



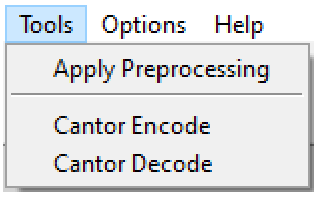
(a) Menu file



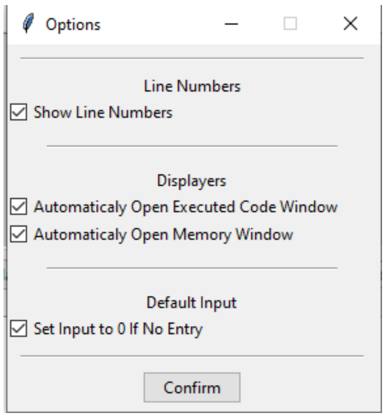
(b) Menu help



(c) Menu run



(d) Menu tools


Options

Line Numbers

☑ Show Line Numbers

Displayers

☑ Automaticaly Open Executed Code Window

☑ Automaticaly Open Memory Window

Default Input

☑ Set Input to 0 If No Entry

Confirm

(e) Menu option

Figure 2: Menu previews

8 Graphic interface

8.1 Menu bar

On the menu bar we can find :

- **File**:
  - *New File* : Open a new blank file into a new tab (with an ephemeral default name, until you save it).
  - *Open File* : Choose and open an existing file with your OS file browser and load it.
  - *Save* : Save the current file on the disk. If it is a new created file, the name and save location will be asked.
  - *Exit* : Quit the application. Be careful to previously save your unsaved files before performing this action.

- **Run**:
  - *Start* : Begin the sequential execution of the current file. This means that you start executing the first instruction, then you manually execute each of the following instructions. The program stops as soon as the last instruction has been executed.
    Remark: this command will be disabled if the sequential execution has already been started.
  - *Run Next Instruction* : Execute the next instruction of the program.
    Remark: this command is disabled if the sequential execution hasn't already been started.
  - *Run All Instructions* : Execute all the next instructions of the program. The delay between two execution is 0.1 sec.
    Remark: this command is disabled if the sequential execution hasn't already been started.
  - *Run File* : Execute the whole file in one go.
- **Stop**: Interrupt the current sequential execution. Memory will be reset.
  Remark: this command is disabled if the sequential execution hasn't already been started.
- **Tools**:
  - *Apply Preprocessing* : Modify the current file by executing only the preprocessing instructions (#define and #include). It is useful when we got an error to see the involved line or to see the final program which will be parsed.
  - *Cantor Encode* : Open a utility window in which you can convert two integers to one with the Cantor's function.
  - *Cantor Decode* : Like the previous command, but the reverse function.
- **Options**: Open the settings window in which we can check/uncheck:
  - Show the lines numbers of the text editor
  - Automatically open the executed code's window
  - Automatically open the memory's window
  - Put the default value 0 if there is no R0 entry

  You have to confirm to save and apply your choices. These will be saved on an JSON file.
- **Help**:
  - *Get Started*: Open an window in which the main command are explain. This is a user manual.
  - *RAM instructions*: Explanation of the four different RAM instructions.
  - *About*: Developers of the application.

8.2 Icons bar

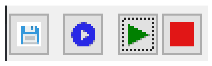


Figure 3: Icon bar

The icon bar (Figure 3) is composed of four icons. The first one is a shortcut to save the current file. The second run the whole file, the third start the sequential execution and the last is the Stop shortcut.

8.3 Text editor

This widget is where you can write and edit your code. This is also here the file you open will be loaded. Line numbers are displayed to the left of the text entry field.

8.4 Entry

You can set the value of the R0 register when you run the program. If no value is entered, an error message will be displayed, except if you set the adequate option.

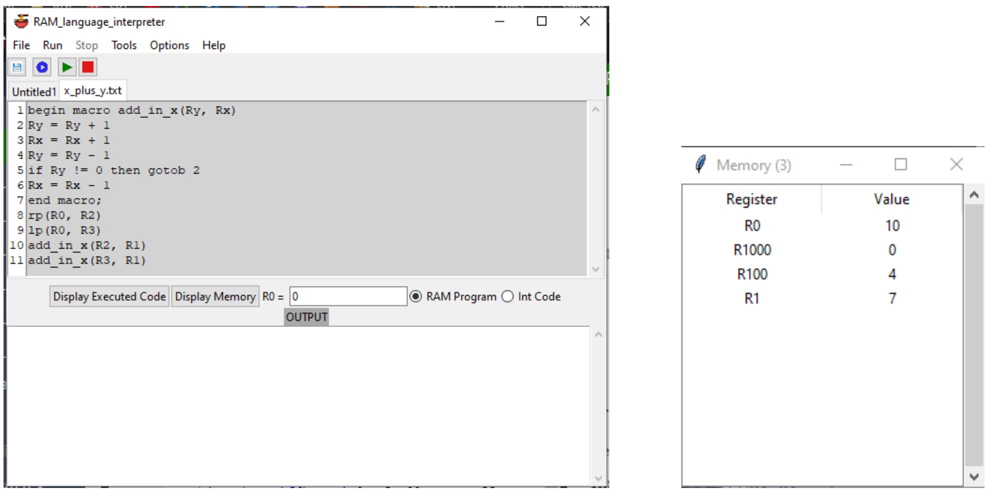
8.5 Displayers

Two windows are available to see and debug your RAM program. The first (*Display Executed Code*) displays all the instructions after the preprocessing and parsing phases. If you execute a code sequentially, the last executed instruction will be highlighted. Since then, you can easily see and debug the progress of the program.

The second one (*Display Memory*) will display the state of the memory (registers). Equally, the memory will be actualize to each iterations. Each window is assigned to a file.

8.6 Output console

It is in this terminal that you will see the file execution's result. Error messages will also be displayed there. If the program contains a grammatical error or an unrecognized instruction, the file will not be interpreted and the number of the line on which the said error appears will be displayed. Attention, this is the line corresponding to the preprocessed file. You can adjust the space taken by it easily by hovering the mouse just below the widget where the code is inserted, as you would with a window.



```
begin macro add_in_x(Ry, Rx)
Ry = Ry + 1
Rx = Rx + 1
Ry = Ry - 1
if Ry != 0 then gotob 2
Rx = Rx - 1
end macro;
rp(R0, R2)
lp(R0, R3)
add_in_x(R2, R1)
add_in_x(R3, R1)
```

Register	Value
R0	10
R1000	0
R100	4
R1	7

(a) Main panel (b) Memory panel

Figure 4: Example of our graphic interface

8.7 Shortcuts

Several shortcut allow you to perform action quickly.

Inside the text editor:

- ***Ctrl+Shit+Enter*** : Run whole file.
- ***Ctrl+Enter*** : Start sequential execution.
- ***Ctrl+M*** : Run all instructions sequentially.
- ***Right-Click*** : Show contextual menu.

Depending on whether you are in RAM program or Int code mode, the right-click contextual menu changes.
In **RAM Program mode**, you can mark a line as the end of file when you run it. You can remove the marker. You also have the possibility of executing the file directly from this contextual menu.
In **Int Code mode**, you will be offered to convert the file that contains the int to a RAM program in a new tab, or you can directly execute the int.

On the main window:

- ***Ctrl+S*** : Save the current file.
- ***Ctrl+O*** : Open file.
- ***Ctrl+N*** : Create new file.
- ***Esc*** : Exit

9 Distribution of tasks

The distribution of the tasks was done relatively naturally according to our respective affinity. Antoine made the graphic part, Davide the interpreter, Margaux the parser and Federica the help panel.

10 Summary

Finally, we have a version of our application in which most of the requirements of the specifications have been implemented. You can interpret a RAM program or directly an integer. If the program is not correct, the user will be warned. It is quite easy to use the software.

Although this application is complete, we could consider some graphical improvements like open a folder and display it in the tree explorer on the left, as in the traditional text editor. More ambitious would be create an editor that able to do automatic completions and suggest syntactic corrections. From a back-end point of view, we could improve our interpreter by taking into account a jump which exceeds the context of a macro.

This project allowed us to finally understand what is the basis of any computer and any program. it was very formative and gave us a lot of perspective.

Appendices

A Grammar

```
program ::= code

code ::=
    expression code
    | expression

expression ::=
    "R" number "=" "R" number "+" number
    | "R" number "=" "R" number "-" number
    | "if" "R" number "!=" number "then" "gotob" number
    | "if" "R" number "!=" number "then" "gotof" number
    | "push" "R" number
    | "pop" "R" number
    | "rp" "(" listRegister ")"
    | "lp" "(" listRegister ")"
    | name "(" listRegister ")"
    | "begin" "macro" name "(" macroListRegister ")" macroCode "end" "macro"";"

listRegister ::=
    "R" NUMBER "," listRegister
    | "R" NUMBER
    |

macroListRegister ::=
    "R" name "," macroListRegister
    | "R" name
    |

macroCode ::=
    macroExpression macroCode
    | macroExpression

macroExpression ::=
    macroId "=" macroId "+" number
    | macroId "=" macroId "-" number
    | "if" macroId "!=" number "then" "gotob" number
    | "if" macroId "!=" number "then" "gotof" number
    | "push" macroId
    | "pop" macroId
    | "rp" "(" listMacroId ")"
    | "lp" "(" listMacroId ")"
    | name "(" listMacroId ")"
    | "begin" "macro" name "(" macroListRegister ")" macroCode "end" "macro"";"

macroid ::=
    "R" name
    | "R" NUMBER

listMacroId ::=
    macroId "," listMacroId
    | macroId
    |

name ::=
    STRING ( NUMBER | STING )*
```