Licenciatura em Ciências da Computação

January 16, 2022

1 Processamento de Linguagens 2021/2022

1.1 Trabalho Prático nº 2: Gramáticas/Compiladores

1.1.1 Grupo 3

Alexandre Rodrigues Balde (A70373)
Marco Alexandre Félix de Lima (A86030)

Revisão git:

[ ]:
```
!git log --show-signature
```

```
commit 2d6f564be826573dff68fc44da1c8d26d316ac27 (HEAD ->
master)
gpg: Signature made Sun Jan 16 20:40:31 2022 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Sun Jan 16 20:40:13 2022 +0000

    Complete project, report :confetti:

commit aa5e1fd4f1dddbc86c258f08b6de32fe804f6df6
gpg: Signature made Wed Dec 29 22:28:28 2021 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Wed Dec 29 22:28:12 2021 +0000

    Refactor error code, test cases
```

```
commit 4e68f7c23230b9ce03a44a29b369d4b12fdebdda
gpg: Signature made Wed Dec 29 00:57:29 2021 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Wed Dec 29 00:55:19 2021 +0000

    Implement 2D arrays

    Also, refactor bytecode generated by while cycle.

commit c50d0d6399936691f0b570c608dd6b6a3d018b70
gpg: Signature made Tue Dec 28 02:44:23 2021 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Tue Dec 28 02:43:48 2021 +0000

    Implement arrays, and write test cases for presentation

    Also, miscellaneous fixes and improvements to the grammar, as well
    as some additional production rules.

commit e409cadb9c8693a2c14183bbc644955b623b27a3
gpg: Signature made Fri Dec 24 02:42:39 2021 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Fri Dec 24 02:42:26 2021 +0000

    Implement 'if-then-else' and 'while-do' commands

commit 607a8681038294bf4f2139920be32434c4cf0bc1
gpg: Signature made Thu Dec 23 03:12:49 2021 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
```

```
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Thu Dec 23 03:12:29 2021 +0000

    Complete IO commands, add undeclared/wrong variable type checks

commit 44d1c1f31576398da68b33c41c987d904b0b6817
gpg: Signature made Wed Dec 22 02:52:35 2021 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Wed Dec 22 02:52:10 2021 +0000

    "Add parsing and code generation for integer assignments"

commit 7c42e6398e5320df6597e2bac1039a25c35064e0
gpg: Signature made Sun Dec 19 02:32:58 2021 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Sun Dec 19 02:32:58 2021 +0000

    Update language generation with translation grammar (phase 1)

commit 10845923a6059015c53e6865b9260c04f125782e
gpg: Signature made Sat Dec 18 16:47:37 2021 WET
gpg:                using RSA key
F0091B21E65C67761EF62A163144D3169524929E
gpg: Good signature from "Alexandre Baldé (WSL Win11 Pro)
<alexandrer_b@outlook.com>" [ultimate]
Author: Alexandre Baldé (WSL Win11 Pro) <alexandrer_b@outlook.com>
Date:   Sat Dec 18 16:47:37 2021 +0000

    Init repo
```

1.2 Implementação e Relatório

De seguida está o código para o analisador léxico da linguagem criada.

[ ]:
```
import ply.lex as lex
```

[ ]:
```
t_PLUS       = r'\+'
t_MINUS      = r'\-'
t_TIMES      = r'\*'
t_DIVIDE     = r'\/'
t_MOD        = r'\%'
t_ASSIGN     = r'\<\-'
t_Decl       = r'\='
t_LPAREN     = r'\('
t_RPAREN     = r'\)'
t_LBRACKET   = r'\{'
t_RBRACKET   = r'\}'
t_COMMA      = r'\,'
t_LSQBRACKET = r'\['
t_RSQBRACKET = r'\]'

t_GT         = r'\>'
t_GE         = r'\>\='
t_LT         = r'\<'
t_LE         = r'\<\='
t_EQ         = r'\=\='
t_NEQ        = r'\!\='

t_AND        = r'\&\&'
t_OR         = r'\|\|'
t_NOT        = r'\~'

reserved = {
    'DeclBegin' : 'DeclBegin',
    'DeclEnd' : 'DeclEnd',
    'int' : 'IntDecl',
    'bool' : 'BoolDecl',
    'string' : 'StringDecl',
    'read' : 'ReadString',
    'write' : 'WriteString',
    'if' : 'If',
    'then' : 'Then',
    'else' : 'Else',
    'while' : "While",
    "error" : 'Err'
}

def t_Bool(t):
```

```
    r'True|False'
    import distutils.util
    t.value = bool(distutils.util.strtobool(t.value))
    return t

def t_String(t):
    r'\"[^\"]*\"'
    return t

def t_Name(t):
    r'[a-zA-Z][a-zA-Z0-9]*'
    # Necessário para captar palavras reservadas.
    t.type = reserved.get(t.value, 'Name')
    return t

def t_Integer(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_comment(t):
    r'(/\*(.|\n)*?\*/)|(//.*)'
    pass

t_ignore = ' \r\n\t'

def t_error(t):
    print('Illegal character: ' + t.value[0])
    return

tokens = list(reserved.values()) + [
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'MOD',
    'ASSIGN',
    'Decl',
    'LPAREN',
    'RPAREN',
    'LBRACKET',
    'RBRACKET',
    'COMMA',
    'LSQBRACKET',
    'RSQBRACKET',

    "GT",
```

```
    "GE",
    "LT",
    "LE",
    "EQ",
    "NEQ",
    'AND',
    'OR',
    'NOT',

    'Name',
    'Integer',
    'String',
    'Bool'
]
```

[ ]:
```
len(tokens)
```

[ ]: 39

[ ]:
```
from lang_lex import lexer
```

1.3 Gramática da linguagem

As regras da gramática são:

<Program> ::= <DeclBlock> <CommandBlock>

<DeclBlock> ::= <DeclBegin> <Vars> <DeclEnd>

<CommandBlock> ::= <CommandBlock> <Command>
| <Command>

<Command> ::= Assign
| <WriteString> LPAREN <PrintableList> RPAREN
| <Err> LPAREN String RPAREN
| <IfThenElse>
| <IfThen>
| <WhileDo>

<Vars> ::= <Vars> <Var>
| <Var>

```
<Var>   ::=  <IntVar>
           |  <BoolVar>
           |  <StringVar>
           |  <ArrVar>
           |  <Arr2Var>
```

1.3.1 Sobre expressões numéricas e booleanas

Numa primeira instância definiram-se as produções para expressões numéricas da seguinte forma:

```
<ExpressionI>   ::=  <ExpressionI>  PLUS  <TermI>
                   |  <ExpressionI>  MINUS  <TermI>
                   |  <TermI>

<TermI>   ::=  <TermI>  TIMES  <FactorI>
             |  <TermI>  DIVIDE  <FactorI>
             |  <TermI>  MOD  <FactorI>
             |  <FactorI>
```

```
FactorI   ::=  Integer
             |  Name
             |  Name  LSQBRACKET  <ExpressionI>  RSQBRACKET
             |  Name  LSQBRACKET  <ExpressionI>  RSQBRACKET  LSQBRACKET  <ExpressionI>
             |  <LPAREN>  <ExpressionI>  <RPAREN>
             |  MINUS  <LPAREN>  <ExpressionI>  <RPAREN>
             |  MINUS  Integer

<ExpressionB>   ::=  Bool
                   |  <ExpressionB>  AND  <ExpressionB>
                   |  <ExpressionB>  OR  <ExpressionB>
                   |  NOT  LPAREN  <ExpressionB>  RPAREN
                   |  LPAREN  <ExpressionB>  RPAREN
                   |  <ExpressionI>  LT  <ExpressionI>
                   |  <ExpressionI>  LE  <ExpressionI>
                   |  <ExpressionI>  GT  <ExpressionI>
                   |  <ExpressionI>  GE  <ExpressionI>
                   |  <ExpressionI>  EQ  <ExpressionI>
                   |  <ExpressionI>  NEQ  <ExpressionI>
```

Para tratar da precedência dos operadores, usou-se a funcionalidade `precedence` do PLY:

```
precedence = (
     ('nonassoc', 'LT', 'GT', 'LE', 'GE', 'EQ', 'NEQ'),
     ('left', 'PLUS', 'MINUS'),
     ('left', 'MOD', 'TIMES', 'DIVIDE'),
     ('left', 'OR'),
     ('left', 'AND')
)
```

No entanto, por se achar que esta solução não era tão fléxivel como captar a precedência dos operadores através de níveis adicionais nas regras de produção, reescreveram-se a produções tomando como inspiração a gramática ANSI C:

<ExpressionB> ::= <ExpressionB> OR <AndExpressionB>
<ExpressionB> ::= <AndExpressionB>

<AndExpressionB> ::= <AndExpressionB> AND <EqExpressionB>
<AndExpressionB> ::= <EqExpressionB>

<EqExpressionB> ::= <EqExpressionB> EQ <RelExpressionB>
<EqExpressionB> ::= <EqExpressionB> NEQ <RelExpressionB>
<EqExpressionB> ::= <RelExpressionB>

<RelExpressionB> ::= <RelExpressionB> LT <ExpressionI>
<RelExpressionB> ::= <RelExpressionB> LE <ExpressionI>
<RelExpressionB> ::= <RelExpressionB> GT <ExpressionI>
<RelExpressionB> ::= <RelExpressionB> GE <ExpressionI>
<RelExpressionB> ::= <ExpressionI>

<ExpressionI> ::= <ExpressionI> PLUS <TermI>
| <ExpressionI> MINUS <TermI>
| <TermI>

<TermI> ::= <TermI> TIMES <FactorI>
| <TermI> DIVIDE <FactorI>
| <TermI> MOD <FactorI>
| <FactorI>

<FactorI> ::= Integer
| Name
| Name LSQBRACKET <ExpressionI> RSQBRACKET
| Name LSQBRACKET <ExpressionI> RSQBRACKET LSQBRACKET <ExpressionI
| <LPAREN> <ExpressionI> <RPAREN>
| MINUS <LPAREN> <ExpressionI> <RPAREN>
| MINUS Integer
| <UnaryExpressionB>

<UnaryExpressionB> ::= NOT <FinalExpressionB>
<UnaryExpressionB> ::= MINUS <FinalExpressionB>
<UnaryExpressionB> ::= <FinalExpressionB>

<FinalExpressionB> ::= Integer
<FinalExpressionB> ::= Name
<FinalExpressionB> ::= Bool
<FinalExpressionB> ::= LPAREN <ExpressionB> RPAREN

1.3.2 Produções restantes

<IntVar> ::= IntDecl Name Decl <ExpressionI>
| IntDeclName
<BoolVar> ::= BoolDecl Name Decl <ExpressionB>
| BoolDeclName
<StringVar> ::= StringDecl Name Decl String
| StringDeclName

Neste trabalho escolheu-se suportar arrays de uma e duas dimensões, cujas produções se encontram abaixo.

<ArrVar> ::= IntDecl LSQBRACKET Integer RSQBRACKET Name
<Arr2Var> ::= IntDecl LSQBRACKET Integer RSQBRACKET LSQBRACKET Integer RSQBRA

Note-se que se considera o acesso a um *array* num dado índice uma expressão numérica, pelo que fazem parte das produções para ExpressionI.

Comandos para IO, *assignments* e controlo de fluxo.

Assign ::= Name ASSIGN <ExpressionI>
| Name LSQBRACKET <ExpressionI> RSQBRACKET ASSIGN <ExpressionI>
| Name LSQBRACKET <ExpressionI> RSQBRACKET LSQBRACKET <ExpressionI> RS
| Name ASSIGN ExpressionB
| Name ASSIGN String
| Name ASSIGN <ReadString> LPAREN RPAREN
| Name LSQBRACKET <ExpressionI> RSQBRACKET ASSIGN <ReadString> LPAREN
| Name LSQBRACKET <ExpressionI> RSQBRACKET LSQBRACKET <ExpressionI> RS

Em relação a impressão de itens em `stdout`, usa-se uma sintaxe similar à do Python e.g.

```
write(x, ";\nabc", 1, 2, True, "y é: \n", y)
```

<PrintableElem> ::= <ExpressionI>
| <ExpressionB>
| String
| <PrintableList> COMMA <PrintableElem>
| <PrintableElem>

<IfThenElse> ::= If <ExpressionB> Then LBRACKET <CommandBlock> RBRACKET Else L
<IfThen> ::= If <ExpressionB> Then LBRACKET <CommandBlock> RBRACKET

<WhileDo> ::= While <ExpressionB> LBRACKET <CommandBlock> RBRACKET

1.4 Geração de instruções para a VM

[ ]:
```
import ply.yacc as yacc
import os

import sys

from lang_yacc import parser_Notebook
```

As funções abaixo lidam com dois error que são possíveis com frequência:

- redeclarar uma variavel duas vezes no bloco inicial de declarações
- utilizar uma variável que não foi declarada

Como aparecem em dezenas de produções, foi mais produtivo passar o tratamento de erros para uma função à parte em vez de replicar o código em cada lugar que é preciso.

A contrapartida é que como o tratamento de erros é mais genérico, não se pode lidar com situações diferentes de forma diferente:

- usar uma variável que não existe numa declaração é diferente de fazê-lo numa expressão numérica que serve de condição a um IfThenElse.

[ ]:
```
def redeclaration_error(p, prod_index):
    if p[prod_index] in parser_Notebook.var_types.keys() or p[prod_index] in␣
↪parser_Notebook.var_stack_loc.keys():
        line   = p.lineno(prod_index)
        index  = p.lexpos(prod_index)
        print("Error: redeclaring variable;")
        print(f"Redeclaration in line number {line}, character {index}: \n(..)␣
↪{' '.join(p[1:(prod_index+1)])} (..)")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
```

[ ]:
```
def undeclared_error(p, production_index):
    if p[production_index] not in parser_Notebook.var_types.keys() or␣
↪p[production_index] not in parser_Notebook.var_stack_loc.keys():
        line   = p.lineno(production_index)
        index  = p.lexpos(production_index)
        print("Error: use of undeclared variable;")
        print(f"Undeclared variable in line number {line}, character {index}:␣
↪\n(..) {p[production_index]} (..)")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
```

[ ]:
```
def p_lang_grammar(p):
    "Program : DeclBlock CommandBlock"
    p[0] = p[1]
    p[0].append("start\n")
    p[0] += p[2]
    p[0].append("stop")
    print(p[0])
    if parser_Notebook.exito:
        parser_Notebook.program = ''.join(p[0])
```

[ ]:
```
def p_lang_decls(p):
    "DeclBlock : DeclBegin Vars DeclEnd"
    p[0] = p[2]
```

[ ]:
```
def p_lang_vars_1(p):
    "Vars : Vars Var"
    p[0] = p[1]
    p[0] += p[2]
```

```
def p_lang_vars_2(p):
    "Vars : Var"
    p[0] = p[1]

def p_lang_var_int(p):
    "Var : IntVar"
    p[0] = p[1]

def p_lang_var_bool(p):
    "Var : BoolVar"
    p[0] = p[1]

def p_lang_var_str(p):
    "Var : StringVar"
    p[0] = p[1]

def p_lang_var_arr(p):
    "Var : ArrVar"
    p[0] = p[1]

def p_lang_var_arr2(p):
    "Var : Arr2Var"
    p[0] = p[1]

def p_lang_IntVar(p):
    "IntVar : IntDecl Name Decl ExpressionI"
    redeclaration_error(p, 2)
    parser_Notebook.var_types[p[2]] = int
    parser_Notebook.var_stack_loc[p[2]] = parser_Notebook.nextvar_addr
    parser_Notebook.nextvar_addr += 1
    p[0] = p[4]

def p_lang_IntVar_default(p):
    "IntVar : IntDecl Name"
    redeclaration_error(p, 2)
    parser_Notebook.var_types[p[2]] = int
    parser_Notebook.var_stack_loc[p[2]] = parser_Notebook.nextvar_addr
    parser_Notebook.nextvar_addr += 1
    p[0] = [f"\tpushi {0}\n"]

def p_lang_BoolVar(p):
    "BoolVar : BoolDecl Name Decl ExpressionB"
    redeclaration_error(p, 2)
    parser_Notebook.var_types[p[2]] = bool
    parser_Notebook.var_stack_loc[p[2]] = parser_Notebook.nextvar_addr
    parser_Notebook.nextvar_addr += 1
    p[0] = [f"\tpushi {int(p[4])}\n"]
```

```
def p_lang_BoolVar_default(p):
    "BoolVar : BoolDecl Name"
    redeclaration_error(p, 2)
    parser_Notebook.var_types[p[2]] = bool
    parser_Notebook.var_stack_loc[p[2]] = parser_Notebook.nextvar_addr
    parser_Notebook.nextvar_addr += 1
    p[0] = [f"\tpushi {int(False)}\n"]

def p_lang_StringVar(p):
    "StringVar : StringDecl Name Decl String"
    redeclaration_error(p, 2)
    parser_Notebook.var_types[p[2]] = str
    parser_Notebook.var_stack_loc[p[2]] = parser_Notebook.nextvar_addr
    parser_Notebook.nextvar_addr += 1
    p[0] = [f"\tpushs {p[4]}\n"]

def p_lang_StringVar_default(p):
    "StringVar : StringDecl Name"
    redeclaration_error(p, 2)
    parser_Notebook.var_types[p[2]] = str
    parser_Notebook.var_stack_loc[p[2]] = parser_Notebook.nextvar_addr
    parser_Notebook.nextvar_addr += 1
    empty = r'""'
    p[0] = [f"\tpushs {empty}\n"]

def p_lang_ArrVar_default(p):
    "ArrVar : IntDecl LSQBRACKET Integer RSQBRACKET Name"
    line   = p.lineno(1)
    index  = p.lexpos(1)
    redeclaration_error(p, 5)
    if p[3] < 0:
        print("Declaring array with negative length.")
        print(f"Array with invalid length in line number {line}, character␣
↪{index}: \n{' '.join(map(str, p[1:6]))}.")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    parser_Notebook.var_types[p[5]] = list
    parser_Notebook.var_stack_loc[p[5]] = parser_Notebook.nextvar_addr
    parser_Notebook.nextvar_addr += p[3]
    # by default, just creates a pointer to array with specified length, with␣
↪all
    # positions set to 0.
    p[0] = [f"\tpushn {p[3]}\n"]

def p_lang_Arr2Var_default(p):
```

```
    "Arr2Var : IntDecl LSQBRACKET Integer RSQBRACKET LSQBRACKET Integer␣
↪RSQBRACKET Name"
    line   = p.lineno(1)
    index  = p.lexpos(1)
    redeclaration_error(p, 8)
    if p[3] < 0 or p[6] < 0:
        print("Declaring two-dimensional array with negative lengths.")
        print(f"Array with invalid length in line number {line}, character␣
↪{index}: \n{' '.join(map(str, p[1:9]))}.")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    parser_Notebook.var_types[p[8]] = (list, list)
    parser_Notebook.var_stack_loc[p[8]] = parser_Notebook.nextvar_addr
    parser_Notebook.nextvar_addr += p[3] * p[6]
    parser_Notebook.arr2_row_len[p[8]] = p[3]
    # by default, just creates a pointer to array with specified length, with␣
↪all
    # positions set to 0.
    p[0] = [f"\tpushn {p[3] * p[6]}\n"]
```

[ ]:
```
def p_lang_commandblock_1(p):
    "CommandBlock : CommandBlock Command"
    p[0] = p[1]
    p[0] += p[2]

def p_lang_commandblock_2(p):
    "CommandBlock : Command"
    p[0] = p[1]

def p_lang_command_1(p):
    "Command : Assign"
    p[0] = p[1]
```

[ ]:
```
def p_lang_command_assign_arr_position(p):
    "Assign : Name LSQBRACKET ExpressionI RSQBRACKET ASSIGN ExpressionI"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] not in [list]:
        line   = p.lineno(1)
        index  = p.lexpos(1)
        print("Error: assigning integer expression to non-array variable;")
        print(f"Wrong types in variable assignment in line number {line},␣
↪character {index}: \n{' '.join(map(str, p[1:6]))}")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
```

```
        p[0] = ["\tpushgp\n"]
        p[0].append(f'\tpushi {parser_Notebook.var_stack_loc[p[1]]}\n')
        p[0].append("\tpadd\n")
        p[0] += p[3]
        p[0] += p[6]
        p[0].append("\tstoren\n")

def p_lang_command_assign_arr2_position(p):
    "Assign : Name LSQBRACKET ExpressionI RSQBRACKET LSQBRACKET ExpressionI␣
 ↪RSQBRACKET ASSIGN ExpressionI"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] not in [(list, list)]:
        line   = p.lineno(1)
        index  = p.lexpos(1)
        print("Error: assigning integer expression to non-array variable;")
        print(f"Wrong types in variable assignment in line number {line},␣
 ↪character {index}: \n{p[1]}[..][..] <- (..)")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
        p[0] = ["\tpushgp\n"]
        p[0].append(f'\tpushi {parser_Notebook.var_stack_loc[p[1]]}\n')
        p[0].append("\tpadd\n")
        p[0] += p[3]
        p[0].append(f"\tpushi {parser_Notebook.arr2_row_len[p[1]]}\n")
        p[0].append("\tmul\n")
        p[0] += p[6]
        p[0].append("\tadd\n")
        p[0].append("\tstoren\n")

def p_lang_command_assign_2(p):
    "Assign : Name ASSIGN ExpressionB"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] not in [int, bool]:
        line   = p.lineno(1)
        index  = p.lexpos(1)
        print("Error: assigning boolean expression to non-integer variable;")
        s = "{...}"
        print(f"Wrong types in variable assignment in line number {line},␣
 ↪character {index}: \n{' '.join(map(str, p[1:4]))} {s}.")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
        p[0] = p[3]
        p[0].append(f'\tstoreg {parser_Notebook.var_stack_loc[p[1]]}\n')
```

```
def p_lang_command_assign_3(p):
    "Assign : Name ASSIGN String"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] not in [str]:
        line   = p.lineno(1)
        index  = p.lexpos(1)
        print("Error: assigning string to non-string variable;")
        print(f"Wrong types in variable assignment in line number {line},␣
↪character {index}: \n{' '.join(map(str, p[1:4]))}")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
        p[0] = [f"\tpushs {p[4]}\n"]
        p[0].append(f'\tstoreg {parser_Notebook.var_stack_loc[p[1]]}\n')

def p_lang_command_assign_4(p):
    "Assign : Name ASSIGN ReadString LPAREN RPAREN"
    undeclared_error(p, 1)
    p[0] = ["\tread\n"]
    if parser_Notebook.var_types[p[1]] in [int, bool]:
        p[0].append("\tatoi\n")
    if parser_Notebook.var_types[p[1]] is bool:
        p[0].append("\tnot\n")
        p[0].append("\tnot\n")
    p[0].append(f"\tstoreg {parser_Notebook.var_stack_loc[p[1]]}\n")

def p_lang_command_assign_arr_read(p):
    "Assign : Name LSQBRACKET ExpressionI RSQBRACKET ASSIGN ReadString LPAREN␣
↪RPAREN"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] not in [list]:
        line   = p.lineno(1)
        index  = p.lexpos(1)
        print("Error: using indexes on non-array variable;")
        print(f"Wrong types in variable assignment in line number {line},␣
↪character {index}: \n{' '.join(map(str, p[1:6]))}")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
        p[0] = ["\tpushgp\n"]
        p[0].append(f'\tpushi {parser_Notebook.var_stack_loc[p[1]]}\n')
        p[0].append("\tpadd\n")
        p[0] += p[3]
        p[0].append("\tread\n")
```

```
        p[0].append("\tatoi\n")
        p[0].append("\tstoren\n")

def p_lang_command_assign_arr2_read(p):
    "Assign : Name LSQBRACKET ExpressionI RSQBRACKET LSQBRACKET ExpressionI␣
 ↪RSQBRACKET ASSIGN ReadString LPAREN RPAREN"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] not in [(list, list)]:
        line   = p.lineno(1)
        index  = p.lexpos(1)
        print("Error: using indexes on non-array variable;")
        print(f"Wrong types in variable assignment in line number {line},␣
 ↪character {index}: \n{p[1]}[..][..] <- (..)")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
        p[0] = ["\tpushgp\n"]
        p[0].append(f'\tpushi {parser_Notebook.var_stack_loc[p[1]]}\n')
        p[0].append("\tpadd\n")
        p[0] += p[3]
        p[0].append(f"\tpushi {parser_Notebook.arr2_row_len[p[1]]}\n")
        p[0].append("\tmul\n")
        p[0] += p[6]
        p[0].append("\tadd\n")
        p[0].append("\tread\n")
        p[0].append("\tatoi\n")
        p[0].append("\tstoren\n")
```

[ ]:
```
def p_lang_comma_sep_name(p):
    "PrintableElem : Name"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] is str:
        p[0] = [f"\tpushg {parser_Notebook.var_stack_loc[p[1]]}\n"]
        p[0].append("\twrites\n")
    if parser_Notebook.var_types[p[1]] in [int, bool]:
        p[0] = [f'\tpushg {parser_Notebook.var_stack_loc[p[1]]}\n']
        p[0].append("\twritei\n")
    if parser_Notebook.var_types[p[1]] in [list, (list, list)]:
        p[0] = [f"\tpushs \"\n\"\n"]
        p[0].append(f"\tpushi {parser_Notebook.var_stack_loc[p[1]]}\n")
        p[0].append("\tpushs \"Array com endereço inicial \"\n")
        p[0].append("\twrites\n")
        p[0].append("\twritei\n")
        p[0].append("\twrites\n")

def p_lang_comma_sep_expressionB(p):
```

```
    "PrintableElem : ExpressionB"
    p[0] = p[1]
    p[0].append("\twritei\n")

def p_lang_comma_sep_string(p):
    "PrintableElem : String"
    p[0] = [f"\tpushs {p[1]}\n"]
    p[0].append("\twrites\n")

def p_lang_comma_sep_list_not_empty(p):
    "PrintableList : PrintableList COMMA PrintableElem"
    p[0] = p[1]
    p[0] += p[3]

def p_lang_comma_sep_list_empty(p):
    "PrintableList : PrintableElem"
    p[0] = p[1]

def p_lang_command_4(p):
    "Command : WriteString LPAREN PrintableList RPAREN"
    p[0] = p[3]

def p_lang_command_error(p):
    "Command : Err LPAREN String RPAREN"
    p[0] = [f"\terr {p[3]}\n"]
```

A disjunção lógica entre x e y corresponde a $x+y-x\cdot y$ em álgebra booleana, ou seja, $x+y+(-1)\cdot x\cdot y$.

[ ]:
```
def p_lang_expressionB_or(p):
    "ExpressionB    : ExpressionB OR AndExpressionB"
    p[0] = p[1]
    p[0].append("\tnot\n")
    p[0].append("\tnot\n")
    p[0] += p[3]
    p[0].append("\tnot\n")
    p[0].append("\tnot\n")
    p[0].append("\tdup 2\n")
    p[0].append("\tmul\n")
    p[0].append("\tpushi -1\n")
    p[0].append("\tmul\n")
    p[0].append("\tadd\n")
    p[0].append("\tadd\n")

def p_lang_expressionB_and(p):
    "ExpressionB    : AndExpressionB"
    p[0] = p[1]
```

```
def p_lang_andExpressionB_and(p):
    "AndExpressionB : AndExpressionB AND EqExpressionB"
    # Conjunção lógica corresponde a multiplicação em álgebra grupo booleana.
    p[0] = p[1]
    p[0].append("\tnot\n")
    p[0].append("\tnot\n")
    p[0] += p[3]
    p[0].append("\tnot\n")
    p[0].append("\tnot\n")
    p[0].append("\tmul\n")

def p_lang_andExpressionB_eq(p):
    "AndExpressionB : EqExpressionB"
    p[0] = p[1]

def p_lang_eqExpressionB_eq(p):
    "EqExpressionB  : EqExpressionB EQ RelExpressionB"
    p[0] = p[1]
    p[0] += p [3]
    p[0].append("\tequal\n")

def p_lang_eqExpressionB_neq(p):
    "EqExpressionB  : EqExpressionB NEQ RelExpressionB"
    p[0] = p[1]
    p[0] += p[3]
    p[0].append("\tequal\n")
    p[0].append("\tnot\n")

def p_lang_eqExpressionB_rel(p):
    "EqExpressionB  : RelExpressionB"
    p[0] = p[1]

def p_lang_relExpressionB_lt(p):
    "RelExpressionB : RelExpressionB LT ExpressionI"
    p[0] = p[1]
    p[0] += p [3]
    p[0].append("\tinf\n")

def p_lang_relExpressionB_le(p):
    "RelExpressionB : RelExpressionB LE ExpressionI"
    p[0] = p[1]
    p[0] += p [3]
    p[0].append("\tinfeq\n")

def p_lang_relExpressionB_gt(p):
    "RelExpressionB : RelExpressionB GT ExpressionI"
    p[0] = p[1]
```

```
    p[0] += p [3]
    p[0].append("\tsup\n")

def p_lang_relExpressionB_ge(p):
    "RelExpressionB : RelExpressionB GE ExpressionI"
    p[0] = p[1]
    p[0] += p [3]
    p[0].append("\tsupeq\n")

def p_lang_relExpressionB_not(p):
    "RelExpressionB : ExpressionI"
    p[0] = p[1]

def p_lang_expressionI_plus(p):
    "ExpressionI : ExpressionI PLUS TermI"
    p[0] = p[1]
    p[0] += p[3]
    p[0].append("\tadd\n")

def p_lang_expressionI_minus(p):
    "ExpressionI : ExpressionI MINUS TermI"
    p[0] = p[1]
    p[0] += p[3]
    p[0].append("\tsub\n")

def p_lang_expressionI_termI(p):
    "ExpressionI : TermI"
    p[0] = p[1]

def p_lang_termI_mul(p):
    "TermI       : TermI TIMES FactorI"
    p[0] = p[1]
    p[0] += p[3]
    p[0].append("\tmul\n")

def p_lang_termI_div(p):
    "TermI       : TermI DIVIDE FactorI"
    p[0] = p[1]
    p[0] += p[3]
    p[0].append("\tdiv\n")

def p_lang_termI_mod(p):
    "TermI       : TermI MOD FactorI"
    p[0] = p[1]
    p[0] += p[3]
    p[0].append("\tmod\n")
```

```
def p_lang_termI(p):
    "TermI       : FactorI"
    p[0] = p[1]

def p_lang_factorI_ArrVar(p):
    "FactorI     : Name LSQBRACKET ExpressionI RSQBRACKET"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] not in [list]:
        line   = p.lineno(1)        # line number of the ASSIGN token
        index  = p.lexpos(1)        # Position of the ASSIGN token
        print("Error: array expression with non-array variable;")
        s = "{...}"
        print(f"Expression in line number {line}, character {index}: \n{s}␣
 ↪{p[1]} {s}.")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
        p[0] = ["\tpushgp\n"]
        p[0].append(f'\tpushi {parser_Notebook.var_stack_loc[p[1]]}\n')
        p[0].append("\tpadd\n")
        p[0] += p[3]
        p[0].append("\tloadn\n")

def p_lang_factorI_Arr2Var(p):
    "FactorI     : Name LSQBRACKET ExpressionI RSQBRACKET LSQBRACKET␣
 ↪ExpressionI RSQBRACKET"
    undeclared_error(p ,1)
    if parser_Notebook.var_types[p[1]] not in [(list, list)]:
        line   = p.lineno(1)        # line number of the ASSIGN token
        index  = p.lexpos(1)        # Position of the ASSIGN token
        print("Error: array expression with non-array variable;")
        s = "{...}"
        print(f"Expression in line number {line}, character {index}: \n{s}␣
 ↪{p[1]} {s}.")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
        p[0] = ["\tpushgp\n"]
        p[0].append(f'\tpushi {parser_Notebook.var_stack_loc[p[1]]}\n')
        p[0].append("\tpadd\n")
        p[0] += p[3]
        p[0].append(f"\tpushi {parser_Notebook.arr2_row_len[p[1]]}\n")
        p[0].append("\tmul\n")
        p[0] += p[6]
        p[0].append("\tadd\n")
```

```
        p[0].append("\tloadn\n")

def p_lang_factorI_not(p):
    "FactorI : UnaryExpressionB"
    p[0] = p[1]

def p_lang_unaryExpressionB_not(p):
    "UnaryExpressionB : NOT FinalExpressionB"
    p[0] = p[2]
    p[0].append("\tnot\n")

def p_lang_factorI_minus_expressionI(p):
    "UnaryExpressionB  : MINUS FinalExpressionB"
    p[0] = ["\tpushi 0\n"]
    p[0] += p[2]
    p[0].append("\tsub\n")

def p_lang_unaryExpressionB_expB(p):
    "UnaryExpressionB : FinalExpressionB"
    p[0] = p[1]

def p_lang_finalExpressionB_int(p):
    "FinalExpressionB     : Integer"
    p[0] = [f"\tpushi {p[1]}\n"]

def p_lang_finalExpressionB_var(p):
    "FinalExpressionB     : Name"
    undeclared_error(p, 1)
    if parser_Notebook.var_types[p[1]] not in [int, bool]:
        line   = p.lineno(1)        # line number of the ASSIGN token
        index  = p.lexpos(1)        # Position of the ASSIGN token
        print("Error: integer expression with non-integer variable;")
        s = "{...}"
        print(f"Expression in line number {line}, character {index}: \n{s}␣
↪{p[1]} {s}.")
        parser_Notebook.exito = False
        p[0] = []
        raise SyntaxError
    else:
        p[0] = [f'\tpushg {parser_Notebook.var_stack_loc[p[1]]}\n']

def p_lang_finalExpressionB_bool(p):
    "FinalExpressionB : Bool"
    if p[1]:
        p[0] = ["\tpushi 1\n"]
    else:
        p[0] = ["\tpushi 0\n"]
```

```
def p_lang_expressionB_name(p):
    "FinalExpressionB : LPAREN ExpressionB RPAREN"
    p[0] = p[2]

def p_error(p):
    parser_Notebook.exito = False
    if p:
        print("Syntax error on token: ", p.type)
        # Just discard the token and tell the parser_Notebook it's okay.
        parser_Notebook.errok()
    else:
        print("Unexpected EOF! Does the program have commands?")
```

[ ]:
```
def p_lang_command_ifthenelse(p):
    "Command : IfThenElse"
    p[0] = p[1]

def p_lang_command_ifthen(p):
    "Command : IfThen"
    p[0] = p[1]

def p_lang_ifthenelse(p):
    "IfThenElse : If ExpressionB Then LBRACKET CommandBlock RBRACKET Else␣
 ↪LBRACKET CommandBlock RBRACKET"
    p[0] = p[2]
    p[0].append(f"\tjz iffalse{parser_Notebook.if_count}\n")
    p[0] += p[5]
    p[0].append(f"\tjump ifrest{parser_Notebook.if_count}\n")
    p[0].append(f"iffalse{parser_Notebook.if_count}: ")
    p[0] += p[9]
    p[0].append(f"ifrest{parser_Notebook.if_count}: ")
    parser_Notebook.if_count += 1

def p_lang_ifthen(p):
    "IfThen : If ExpressionB Then LBRACKET CommandBlock RBRACKET"
    p[0] = p[2]
    p[0].append(f"\tjz ifrest{parser_Notebook.if_count}\n")
    p[0] += p[5]
    p[0].append(f"ifrest{parser_Notebook.if_count}: ")
    parser_Notebook.if_count += 1
```

[ ]:
```
def p_lang_command_while(p):
    "Command : WhileDo"
    p[0] = p[1]

def p_lang_while(p):
```

```
    "WhileDo : While ExpressionB LBRACKET CommandBlock RBRACKET"
    cnt = parser_Notebook.while_count
    parser_Notebook.while_count += 1
    p[0] = [f"while{cnt}: "]
    p[0] += p[2]
    p[0].append(f"\tjz whilerest{cnt}\n")
    p[0] += p[4]
    p[0].append(f"\tjump while{cnt}\n")
    p[0].append(f"whilerest{cnt}: ")
```

O `parser_Notebook` tem como variáveis internas:

- um dicionário que faz corresponder ao nome de cada variável no programa o tipo com que foi declarada
- o próximo endereço livre na pilha que será usado para a próxima variável a ser declarada, se existir, ou o valor do apontador `sp` aquando do início do programa (instrução `START`)
- um dicionário que a cada variável faz corresponder o seu endereço, relativo à parte da pilha que contém variáveis globais
- um dicionário que armazena, para cada *array* de duas dimensões declarado, o comprimento das suas linhas. Isto é necessário para poder fazer indexação.
- contadores para o número de construções `if` e `while`, um para cada tipo de instrução. Estes números são necessários para identificarem - univocamente - as labelas no código máquina geradas por cada uma destas instruções, já que são incrementados por cada instrução do seu género encontrada.
- uma *flag* `exito`, para saber se o *parsing* foi efetuado com sucesso. Se for encontrado um erro semântico e não sintático e.g. variável redeclarada ou tipos incorretos numa expressão, a *flag* serve para poder sinalizar ao programa que pode descartar o código máquina que construiu até ao momento.

[ ]:
```
parser_Notebook.var_types = {}
parser_Notebook.nextvar_addr = 0
parser_Notebook.var_stack_loc = {}
# Used to keep track of each 2d array's row length.
# Needed for indexing and assignment.
parser_Notebook.arr2_row_len = {}

# Used to tag if-then-else labels to make them unique.
parser_Notebook.if_count = 0
parser_Notebook.while_count = 0

parser_Notebook.exito = True
```

1.5 Exemplos de programas na linguagem, e código máquina produzido

1.5.1 Programa 1

ler 4 números e dizer se podem ser os lados de um quadrado.

```
DeclBegin
int lado1
int lado2
int lado3
int lado4
DeclEnd
lado1 <- read()
lado2 <- read()
lado3 <- read()
lado4 <- read()
if lado1 == lado2 && lado2 == lado3 && lado3 == lado4 then {
    write("Os números inseridos formam os 4 lados de um quadrado, com medida: ", lado1, ".\n")
} else {
    write("Os números inseridos não formam os 4 lados de um quadrado.\n")
}
```

Código produzido

```
llvm pushi 0 pushi 0 pushi 0 pushi 0 start read atoi storeg 0 read atoi storeg 1 read atoi storeg 2 read atoi storeg 3 pushg 0 pushg 1 equal not not pushg 1 pushg 2 equal not not mul not not pushg 2 pushg 3 equal not not mul jz iffalse0 pushs "Os números inseridos formam os 4 lados de um quadrado, com medida: " writes pushg 0 writei pushs ".\n" writes jump ifrest0 iffalse0: pushs "Os números inseridos não formam os 4 lados de um quadrado.\n" writes ifrest0: stop
```

1.5.2 Programa 2

ler um inteiro N, depois ler N números e escrever o menor deles.

```
DeclBegin
/* Maior inteiro positivo representável em 32 bits */
int min = 2147483647
int num = -1
int input
DeclEnd
while num < 0 {
    write("Escreva quantos números quer ler.\n")
    num <- read()
    if num < 0 then {
        write("Insira um número não-negativo!\n")
    }
}
while num > 0 {
    write("Insira um número. (Faltam ", num, " números.)\n")
    input <- read()
```

```
    if input < min then {
        min <- input
    }
    num <- num - 1
}
write("O menor número é: ", min, ".\n")
```

Código produzido

```
        pushi 2147483647
        pushi 0
        pushi 1
        sub
        pushi 0
start
while0:         pushg 1
        pushi 0
        inf
        jz whilerest0
        pushs "Escreva quantos números quer ler.\n"
        writes
        read
        atoi
        storeg 1
        pushg 1
        pushi 0
        inf
        jz ifrest0
        pushs "Insira um número não-negativo!\n"
        writes
ifrest0:        jump while0
whilerest0: while1:     pushg 1
        pushi 0
        sup
        jz whilerest1
        pushs "Insira um número. (Faltam "
        writes
        pushg 1
        writei
        pushs " números.)\n"
        writes
        read
        atoi
        storeg 2
        pushg 2
        pushg 0
        inf
        jz ifrest1
```

```
        pushg 2
        storeg 0
ifrest1:        pushg 1
        pushi 1
        sub
        storeg 1
        jump while1
whilerest1:     pushs "O menor número é: "
        writes
        pushg 0
        writei
        pushs ".\n"
        writes
stop
```

1.5.3 Programa 3

ler N (constante do programa) números e calcular e imprimir o seu produtório.

```
DeclBegin
int prod = 1
int num = -1
int input
DeclEnd
while num < 0 {
    write("Escreva quantos números quer ler.\n")
    num <- read()
    if num < 0 then {
        write("Insira um número não-negativo!\n")
    }
}
while num > 0 {
    write("Insira um número. (Faltam ", num, " números.)\n")
    input <- read()
    prod <- prod * input
    num <- num - 1
}
write("O produto dos números lidos é: ", prod, ".\n")
```

1.5.4 Programa 4

contar e imprimir os números impares de uma sequência de números naturais.

```
DeclBegin
int yesNo = 1
int total
int impares
int input
DeclEnd
```

```
while yesNo != 0 {
    write("Insira um número.\n")
    input <- read()
    total <- total + 1
    if input % 2 == 1 then {
        impares <- impares + 1
    }
    write("Foram lidos ", total, " números, dos quais ", impares, " são ímpares.\n")
    write("Continuar? Insira qualquer número não nulo para sim, 0 para não.\n")
    yesNo <- read()
}
```

Código produzido

```
        pushi 1
        pushi 0
        pushi 0
        pushi 0
start
while0:         pushg 0
        pushi 0
        equal
        not
        jz whilerest0
        pushs "Insira um número.\n"
        writes
        read
        atoi
        storeg 3
        pushg 1
        pushi 1
        add
        storeg 1
        pushg 3
        pushi 2
        mod
        pushi 1
        equal
        jz ifrest0
        pushg 2
        pushi 1
        add
        storeg 2
ifrest0:        pushs "Foram lidos "
        writes
        pushg 1
        writei
        pushs " números, dos quais "
```

```
        writes
        pushg 2
        writei
        pushs " são ímpares.\n"
        writes
        pushs "Continuar? Insira qualquer número não nulo para sim, 0 para não.\n"
        writes
        read
        atoi
        storeg 0
        jump while0
whilerest0: stop
```

1.5.5 Programa 5

ler e armazenar N números num array; imprimir os valores por ordem inversa.

```
DeclBegin
int len = 10
int ix
int[10] arr
DeclEnd
write("Insira ", len, " números.\n")
while ix < len {
    arr[ix] <- read()
    ix <- ix + 1
}
write("Os números lidos, por ordem inversa, são:\n")
ix <- len - 1
while ix >= 0 {
    if ix < 0 || ix > 10 then {
        error ("Índice de array fora de limites.")
    }
    write("Índice ", ix, " é: ", arr[ix], ".\n")
    ix <- ix - 1
}
```

Código produzido

```
        pushi 10
        pushi 0
        pushn 10
start
        pushs "Insira "
        writes
        pushg 0
        writei
        pushs " números.\n"
        writes
```

```
while0:         pushg 1
        pushg 0
        inf
        jz whilerest0
        pushgp
        pushi 2
        padd
        pushg 1
        read
        atoi
        storen
        pushg 1
        pushi 1
        add
        storeg 1
        jump while0
whilerest0:     pushs "Os números lidos, por ordem inversa, são:\n"
        writes
        pushg 0
        pushi 1
        sub
        storeg 1
while1:         pushg 1
        pushi 0
        supeq
        jz whilerest1
        pushg 1
        pushi 0
        inf
        not
        not
        pushg 1
        pushi 10
        sup
        not
        not
        dup 2
        mul
        pushi -1
        mul
        add
        add
        jz ifrest0
        err "Índice de array fora de limites."
ifrest0:        pushs "Índice "
        writes
        pushg 1
        writei
```

```
        pushs " é: "
        writes
        pushgp
        pushi 2
        padd
        pushg 1
        loadn
        writei
        pushs ".\n"
        writes
        pushg 1
        pushi 1
        sub
        storeg 1
        jump while1
whilerest1: stop
```

1.5.6 Extra : Programa 6

ler e armazenar N^2 números num array $N \times N$; imprimir os valores por ordem inversa.

```
DeclBegin
int len = 3
int l
int c
int[3][3] arr
DeclEnd
write("Insira ", len*len, " números.\n")
while l < len*len {
    while c < len*len {
        arr[l][c] <- read()
        c <- c + 1
    }
    l <- l + 1
}
write("Os números lidos, por ordem inversa, são:\n")
l <- len - 1
c <- len - 1
while l >= 0 {
    c <- len - 1
    while c >= 0 {
        write("Índice (", l, ", ", c, ")", " é: ", arr[l][c], ".\n")
        c <- c - 1
    }
    l <- l - 1
}
```

Código produzido

```
        pushi 3
        pushi 0
        pushi 0
        pushn 9
start
        pushs "Insira "
        writes
        pushg 0
        pushg 0
        mul
        writei
        pushs " números.\n"
        writes
while1:         pushg 1
        pushg 0
        pushg 0
        mul
        inf
        jz whilerest1
while0:         pushg 2
        pushg 0
        pushg 0
        mul
        inf
        jz whilerest0
        pushgp
        pushi 3
        padd
        pushg 1
        pushi 3
        mul
        pushg 2
        add
        read
        atoi
        storen
        pushg 2
        pushi 1
        add
        storeg 2
        jump while0
whilerest0:     pushg 1
        pushi 1
        add
        storeg 1
        jump while1
whilerest1:     pushs "Os números lidos, por ordem inversa, são:\n"
        writes
```

```
        pushg 0
        pushi 1
        sub
        storeg 1
        pushg 0
        pushi 1
        sub
        storeg 2
while3:         pushg 1
        pushi 0
        supeq
        jz whilerest3
        pushg 0
        pushi 1
        sub
        storeg 2
while2:         pushg 2
        pushi 0
        supeq
        jz whilerest2
        pushs "Índice ("
        writes
        pushg 1
        writei
        pushs ", "
        writes
        pushg 2
        writei
        pushs ")"
        writes
        pushs " é: "
        writes
        pushgp
        pushi 3
        padd
        pushg 1
        pushi 3
        mul
        pushg 2
        add
        loadn
        writei
        pushs ".\n"
        writes
        pushg 2
        pushi 1
        sub
        storeg 2
```

```
        jump while2
whilerest2:     pushg 1
        pushi 1
        sub
        storeg 1
        jump while3
whilerest3: stop
```

[ ]:
```
from subprocess import run
run(["./vms", f"../test/nivel4.vm"], cwd=os.path.abspath('../vms'))
```

```
Insira um número.
Foram lidos 1 números, dos quais 0 são ímpares.
Continuar? Insira qualquer número não nulo para sim, 0 para não.
```

[ ]:
```
CompletedProcess(args=['./vms', '../test/nivel4.vm'], returncode=0)
```