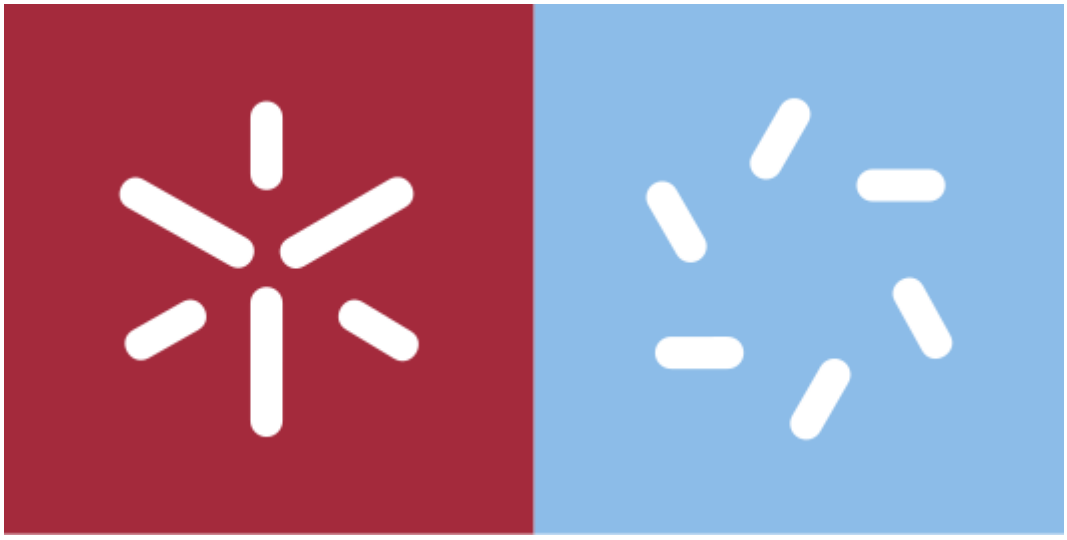

CIA, um compilador que gera código para uma VM

Instituição: Universidade do Minho

Unidade Curricular: Processamento de Linguagens e Compiladores (2021/2022)

Identificação: plc21TP2gr03

Equipa: Alef Keuffer (A91383); Catarina Quintas (A91650); Ivo Lima (A90214)

Introdução

No âmbito da Unidade Curricular de Processamento de Linguagens e Compiladores foi-nos proposto a construção de um compilador de uma linguagem de programação, de forma a consolidar os conhecimentos adquiridos nas aulas sobre parsing, yacc e gramáticas independentes de contexto. O objetivo deste relatório é aumentar a nossa capacidade de escrever gramáticas, sendo estas independentes de contexto(GIC) ou tradutoras(GT) e através delas, desenvolver processadores de linguagens segundo o método da tradução.

Gramática de *Parsing*

Nesta secção, iremos definir a linguagem dada como solução do problema, analisando todos os constituintes de uma gramática. De acordo com o estudo ao longo do semestre, definimos uma gramática para a representação de uma linguagem imperativa.

Expressões:

```
<expr> : ID | STR | INT | REAL | <list>
```

Listas:

```
<list> : '(' <seq> ')'
```

Sequência de Produções:

```
<seq>  : ε | <expr> <seq>
```

Tokens:

```
<REAL> : r"[+-]?[0-9]*\.[0-9]+"
```

```
<INT>  : r"[+-]?[0-9]+"
```

```
<ID>   : r"[-+?`,*/!#$@%^&=.a-zA-Z0-9_]+"
```

```
<STR>  : r'"[^"]*"'
```

```
<comment> : r'⟨[^⟩]⟩'
```

Declaração de variáveis:

```
(do
    (decl (x int) (y float) (z str)) `tipos simples`
    (decl (arr array 5)) `array`
    (mdecl mat 2 3) `matriz`
)
```

Definição de funções:

Sintaxe: `defun <nome> <n_args> <n_outs> (<body>)`

```
(do
  (defun readInt 0 1 ((let ($1 (atoi (read))))))
)
```

Estratégias

Ciclo

Para um ciclo simples ou incondicional:

```
l0:
<body>
jump l0
```

While

Para um ciclo condicional temos:

```
l0:
<cond>
jz l1
<body>
jump l0
l1:
```

If

Para `if ⟨cond⟩ then ⟨c1⟩ else ⟨c2⟩` basta fazer o seguinte:

```
⟨cond⟩
jz l0:
⟨c1⟩
jump l1
l0:
⟨c2⟩
l1:
```

Case (Super If)

$$
\begin{array}{l}
(\text{case } v \\
\qquad\qquad (v_1 \quad\quad c_1) \\
\qquad\qquad (v_2 \quad\quad c_2) \\
\qquad\qquad\quad \vdots \\
\qquad\qquad (v_n \quad\quad c_n) \\
)
\end{array}
\quad\equiv\quad
\begin{array}{l}
\text{if } v = v_1 : \\
\qquad \text{eval } c_1 \\
\text{elif } v = v_2 : \\
\qquad \text{eval } c_2 \\
\qquad\quad \vdots \\
\text{elif } v = v_n : \\
\qquad \text{eval } c_n
\end{array}
$$

Note que "push" não é um comando.

```
push v

dup 1
```

```
push v_1
equal

jz e_2
c_1
jump e_f

e_2:
dup 1
push v_2
equal

jz e_3
c_2
jump e_f
.
.
.
e_n:
dup 1
push v_n
equal

jz e_f
c_n

e_f:
c_f
```

Funções:

Para as funções pensamos da seguinte forma `f :: p1,p2,...,pn → r1,r2,...,rk`, ou seja:

```
start
pushn k
push pn
push pn-1
...
push pn

pusha f
call
pop n // remove arguments from stack
// k top args are the result of f
```

```
stop

f:
pushl -n        // p1
pushl -n+1      // p2
...
pushl -1        // pn
storel (-n-k)   // r1
storel (-n-k+1) // r2
...
storel (-n-1)   // rk

return
```

Organização da Máquina

Elementos da Máquina

1. Pilhas

Constituição de 2 pilhas:

- Pilha de Execução (valores: Inteiros, Reais, Endereços)
- Pilha de Chamadas: contém pares de apontadores (i, f). O endereço i guarda o registo de instrução pc e f o registo fp .

2. Heaps

Constituição de 2 heaps (são referenciadas por endereços):

- String Heap
- Structured Block Heap
    - Contem um certo número de valores (do mesmo tipo dos valores que se podem encontrar na pilha).

3. Registos

Constituição de 4 registos:

- sp (stack pointer): aponta para o topo corrente da pilha. Ele aponta para a primeira célula livre da pilha.
- fp (frame pointer): aponta para o endereço de base das variáveis locais.
- gp (global pointer): contem o endereço de base das variáveis globais.
- pc (program counter): aponta para a instrução corrente (da zona de código) por executar.

4. Auxiliares

- 1x Zona de Código

Conceitos

Endereço

- Pode apontar para 4 tipos de informação:
    1. Código
    2. Pilha
    3. Bloco estruturado
    4. String

Instruções

Aceitam 1 ou 2 parâmetros. Que podem ser:

- constantes inteiras
- constantes reais
- cadeias de caracteres delimitadas por aspas.
    - Estas cadeias de caracteres seguem as mesmas regras de formatação que as cadeias da linguagem C (em particular no que diz respeito aos caracteres especiais como `\"`, `\n` ou `\\`),
- uma etiqueta simbólica designando uma zona no código.

Convenção:

- Empilhar um valor `x` $\equiv$ `P[sp]:= x;sp++`
- Empilhar n vezes um valor $x \equiv$ iterar n vezes a operação anterior.
- Retirar, ou tirar da pilha n valor consiste em decrementar de n o valor de sp.
- O topo da pilha representa o último valor colocado na pilha, ou seja $P[sp-1]$, o valor anterior representa o penúltimo valor, o sub-topo colocado na pilha, ou seja $P[sp-2]$.
- Se x designar um endereço na pilha então $x[n]$ designa um endereço situada n células por cima.

Comparação:

$$\text{False} \equiv 0$$

O resultado duma operação de comparação é um inteiro que vale 0 ou um número diferente de 0.

Operações:

- $\text{INF} \equiv P[sp-2] < P[sp-1]$
- $\text{JUMP label} \equiv pc := \text{addr(label)}$
- $\text{JZ label} \equiv \text{IF } !P[sp-1] : pc = \text{label} \approx \text{IF } !P[sp-1] : \text{JUMP addr(label)}$

- $\text{PUSHA label} \equiv \text{PUSHA addr(label)}$
- $\text{ADD} \equiv P[sp-2] + P[sp-1]$
- $\text{STOREN} \equiv P[sp-3][P[sp-2]] := P[sp-1]$ ou seja, addr, pos, value.
- $\text{PADD} \equiv P[sp-2] + P[sp-1]$ onde $P[sp-2] : \text{addr}$ e $P[sp-1] : \text{INT}$

Exemplos de funcionamento

A seguir, apresenta-se algumas resoluções referentes às questões colocadas pelo docente da disciplina.

1. Efetuar a leitura de 4 números decidindo se estes podem ser os lados de um quadrado.

```
(do
    (decl (a int)
          (b int)
          (c int)
          (d int)
    )

    (let (a (atoi (read)))
         (b (atoi (read)))
         (c (atoi (read)))
         (d (atoi (read)))
    )

    (case (mul (equal a b) (mul (equal a c) (equal a d)))
      (1 ((writes "Pode ser")))
    )
)
```

Código $Assembly$ gerado:

```
START
PUSHN 1
PUSHN 1
PUSHN 1
PUSHN 1
READ
ATOI
STOREL 0
READ
ATOI
STOREL 1
```

```
READ
ATOI
STOREL 2
READ
ATOI
STOREL 3
PUSHL 0
PUSHL 1
EQUAL
PUSHL 0
PUSHL 2
EQUAL
PUSHL 0
PUSHL 3
EQUAL
MUL
MUL
DUP 1
PUSHI 1
EQUAL
JZ l1
PUSHS "Pode ser"
WRITES
JUMP l1
l1:
STOP
```

2. Efetuar a leitura de um inteiro N, seguido por uma nova leitura de N números e devolução do menor valor.

```
(do
    (decl
        (n int)
        (num int)
        (min int)
        (i int)
    )

    (let
        (min 0)
        (i 0)
        (n (atoi (read)))
    )
```

```
    (while (inf i n)
      (
          (let
            (num (atoi (read)))
          )
          (case (equal i 0)
            (1 ((let
                 (min num)
               ))
            )
          )
        (case (sup min num)
          (1 ((let
                (min num)
              ))
           )
        )
        (let
          (i (add i 1))
        )
      )
    )
    (writei min)
)
```

Código $Assembly$ gerado:

```
START
PUSHN 1
PUSHN 1
PUSHN 1
PUSHN 1
PUSHI 0
STOREL 2
PUSHI 0
STOREL 3
READ
ATOI
STOREL 0
la0:
PUSHL 3
PUSHL 0
INF
JZ la1
READ
ATOI
```

```
STOREL 1
PUSHL 3
PUSHI 0
EQUAL
DUP 1
PUSHI 1
EQUAL
JZ l1
PUSHL 1
STOREL 2
JUMP l1
l1:
PUSHL 2
PUSHL 1
SUP
DUP 1
PUSHI 1
EQUAL
JZ l3
PUSHL 1
STOREL 2
JUMP l3
l3:
PUSHL 3
PUSHI 1
ADD
STOREL 3
JUMP la0
la1:
PUSHL 2
WRITEI
STOP
```

3. Interpretar N (que representará a constante do programa) números, calculando e imprindo o seu produtório.

```
(do
  (defun readInt 0 1 ((let ($1 (atoi (read))))))
  (decl
    (n int)
    (p int)
    (i int)
    (num int)
  )
```

```
  (let
    (p 1)
    (n (call readInt))
  )

  (let (i 0))
  (while (inf i n)
    ((let (num (call readInt)))
    (let (p (mul p num)))
    (let (i (add i 1))))
  )

  (writei p)

)
```

Código $Assembly$ gerado:

```
START
PUSHN 1
PUSHN 1
PUSHN 1
PUSHN 1
PUSHI 1
STOREL 1
PUSHN 1
PUSHA la0
CALL
POP 0
STOREL 0
PUSHI 0
STOREL 2
la1:
PUSHL 2
PUSHL 0
INF
JZ la2
PUSHN 1
PUSHA la0
CALL
POP 0
STOREL 3
PUSHL 1
PUSHL 3
MUL
```

```
STOREL 1
PUSHL 2
PUSHI 1
ADD
STOREL 2
JUMP la1
la2:
PUSHL 1
WRITEI
STOP
la0:
READ
ATOI
STOREL -1
RETURN
```

4. Formalizar a contagem e impressão dos números ímpares de uma sequência de N números naturais.

```
(do
  (decl (i int) (size int) (num2 array 10))
  (let (i 0) (size 10))

  (while (inf i size) (
        (aset num2 i (atoi (read)))
        (let (i (add i 1)))
    )
  )

  (let (i (sub i 1)))

  (while (supeq i 0) (
        (case (mod (aref num2 i) 2)
            (1 ((writei (aref num2 i))))
        )
        (let (i (sub i 1)))
    )
  )
)
```

Código $Assembly$ gerado:

```
START
PUSHN 1
```

```
PUSHN 1
PUSHN 10
PUSHI 0
STOREL 0
PUSHI 10
STOREL 1
la0:
PUSHL 0
PUSHL 1
INF
JZ la1
PUSHGP
PUSHI 2
PADD
PUSHL 0
READ
ATOI
STOREN
PUSHL 0
PUSHI 1
ADD
STOREL 0
JUMP la0
la1:
PUSHL 0
PUSHI 1
SUB
STOREL 0
la2:
PUSHL 0
PUSHI 0
SUPEQ
JZ la3
PUSHGP
PUSHI 2
PADD
PUSHL 0
LOADN
PUSHI 2
MOD
DUP 1
PUSHI 1
EQUAL
JZ l1
PUSHGP
PUSHI 2
```

```
PADD
PUSHL 0
LOADN
WRITEI
JUMP l1
l1:
PUSHL 0
PUSHI 1
SUB
STOREL 0
JUMP la2
la3:
STOP
```

5. Realizar a leitura e armazenamento de N números num array e fazer a impressão dos valores pela ordem inversa.

```
(do
  (decl (i int) (size int) (num2 array 10))
  (let (i 0) (size 10))
  (defun readInt 0 1 ((let ($1 (atoi (read))))))

  (while (inf i size) (
        (aset num2 i (call readInt))
        (let (i (add i 1)))
    )
  )

  (let (i (sub i 1)))

  (while (supeq i 0) (
        (writei (aref num2 i))
        (let (i (sub i 1)))
    )
  )
)
```

Código $Assembly$ gerado:

```
START
PUSHN 1
PUSHN 1
PUSHN 10
PUSHI 0
```

```
STOREL 0
PUSHI 10
STOREL 1
la1:
PUSHL 0
PUSHL 1
INF
JZ la2
PUSHGP
PUSHI 2
PADD
PUSHL 0
PUSHN 1
PUSHA la0
CALL
POP 0
STOREN
PUSHL 0
PUSHI 1
ADD
STOREL 0
JUMP la1
la2:
PUSHL 0
PUSHI 1
SUB
STOREL 0
la3:
PUSHL 0
PUSHI 0
SUPEQ
JZ la4
PUSHGP
PUSHI 2
PADD
PUSHL 0
LOADN
WRITEI
PUSHL 0
PUSHI 1
SUB
STOREL 0
JUMP la3
la4:
STOP
la0:
```

```
READ
ATOI
STOREL -1
RETURN
```

6. Invocação e utilização de um programa `potencia()`, que começa por ler do input a base B e o expoente E devolvendo o valor B^E.

```
(do
    (defun pow 2 1 (
        (let (#1 (atoi (read))) (#2 (atoi (read))) ($1 1))
        (while (sup #2 0) (
            (let ($1 (mul $1 #1)) (#2 (sub #2 1)))
        ))
    ))

    (writei (call pow))
)
```

Código $Assembly$ gerado:

```
START
PUSHN 3
PUSHA la0
CALL
POP 2
WRITEI
STOP
la0:
READ
ATOI
STOREL -2
READ
ATOI
STOREL -1
PUSHI 1
STOREL -3
la1:
PUSHL -1
PUSHI 0
SUP
JZ la2
PUSHL -3
PUSHL -2
```

```
MUL
STOREL -3
PUSHL -1
PUSHI 1
SUB
STOREL -1
JUMP la1
la2:
RETURN
```

Conclusão

Em suma, este projeto permitiu-nos aprofundar os nossos conhecimentos em relação aos conteúdos lecionados nas aulas teóricas e práticas, tendo como principal foco a ferramenta de processamento de texto `FLEX`. Ao longo da elaboração do trabalho, reparamos em 2 pequenas falhas, ou seja, o nosso programa não permite a declaração de arrays dinâmicos e também não consegue fazer o tratamento de erros semânticos, pois consideramos que o utilizador sabe codificar a nossa linguagem.

Apêndice A

Código do Programa:

Analisador Léxico

```
import ply.lex as lex
import sys

tokens = ["REAL", "INT", "ID", "STR" ,"LP", "RP",'comment']

# declaração das Palavras-Reservadas e dos Simbolos de Classe (variáveis)
t_LP  = r"\("
t_RP  = r"\)"

def t_comment_tag(t):
    r'⟨[^⟩]⟩'
    return t

def t_REAL(t):
    r"[+-]?[0-9]*\.[0-9]+"
    return t

def t_INT(t):
    r"[+-]?[0-9]+"
    return t
```

```
def t_ID(t):
    r"[-+?`,*/!#$@%^&=.a-zA-Z0-9_]+"
    return t

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

# declaração dos Carateres que podem aparecer no texto de entrada e que devem ser ignorado
t_ignore = " \n\t"

# declaração da ação a fazer relativa aos Carateres que NÃO podem aparecer no texto de ent
def t_error(t):
    print("Carater ilegal: ", t.value)


lexer = lex.lex()

if __name__ == "__main__":
    for line in sys.stdin:
        lexer.input(line)
        tok = lexer.token()
        while tok:
            print(tok)
            tok = lexer.token()
```

Parser

```
from __future__ import division
import ply.yacc as yacc
import sys
from lex7 import tokens
import re

import math
import operator as op


"""
expr : ID | STR | INT | REAL | list
list : ( seq )
seq  : ε | expr seq
"""

def get_order():
    order = parser.global_vars
    parser.global_vars += 1
    return order

def get_label1():
    lab = f'la{parser.label_count1}'
    parser.label_count1 += 1
```

```
    return lab

class Var:
    def __init__(self, name, order, var_type):
        self.name = name
        self.order = order
        self.var_type = var_type
        self.value = None

class FUNC:
    def __init__(self,name,nargs,nouts,pos):
        self.name = name
        self.nargs = nargs
        self.nouts = nouts
        self.pos = pos
    def aref(self,argnum):
        return -self.nargs + int(argnum) - 1
    def rref(self,resnum):
        return -self.nargs - self.nouts + int(resnum) - 1

def is_id(p):
    return type(p) == str and p[0] != '"'

def is_str(p):
    return type(p) == str and p[0] == '"'

def is_var(p):
    return type(p) != list and p in parser.ids and type(parser.ids[p]) == Var

################################################################################################
def push_rec(p,commands,parse_state):
    parg = push(p)
    if parg:
        commands.append(parg)
    else:
        g_eval_expr(p,commands,None,parse_state)

def push(p):
    t = type(p)
    if t == list:
        return False
    if t == int:
        return f'PUSHI {p}'
    elif t == float:
        return f'PUSHF {p}'
    elif t == FUNC:
        return f'PUSHA {p.pos}'
    elif is_str(p):
        return f'PUSHS {p}'
    elif is_var(p):
        return push_l(parser.ids[p].order)
    elif p[0] == '#':
        return push_l(parser.funcstack[-1].aref(p[1:]))
    elif p[0] == '$':
        return push_l(parser.funcstack[-1].rref(p[1:]))
```

```
    else:
        return False

def push_l(p):
    return f'PUSHL {p}'

def write(p):
    if p == 'int':
        return f'WRITEI'
    elif p == 'float':
        return f'WRITEF'
    elif p == 'str':
        return f'WRITES'

def repeat(n,stack,parse_state):
    push_rec(n,stack,parse_state,parse_state)
    push_rec(1,stack,parse_state,parse_state)
    push_rec(SUB,stack,parse_state,parse_state)
    rstart = get_label1()
    rend = get_label1()
    stack.append(rstart + ':')
    push_rec(n,stack,parse_state,parse_state)
    stack.append(jz(rend))
    stack.append(push(0))
    push_rec(n,stack,parse_state,parse_state)
    stack.append(push(-1))
    stack.append(ADD)
    stack.append(store_l(parser.ids['n'].order))
    stack.append(jump(rstart))
    stack.append(rend + ':')

def dup(p):
    return f'DUP {p}'


def push_l(p):
    return f'PUSHL {p}'


def push_g(p):
    return f'PUSHL {p}'


def push_n(p):
    return f'PUSHN {int(p)}'


def push_a(p):
    return f'PUSHA {p}'


def pop(n):
    return f'POP {n}'
```

```
def store_l(p):
    return f'STOREL {p}'


def jump(label):
    return f'JUMP {label}'


def jz(label):
    return f'JZ {label}'

def vms(instructions,opt_write_ins=''):
    import os
    if 'STOP' not in instructions:
        instructions = ['START'] + instructions + ['STOP']
    stop_index = instructions.index('STOP')
    instructions = instructions[:stop_index] + ['WRITE' + opt_write_ins.upper()] + instruc

    VMS_PROG = '../vms/vms'
    CODE_DIR = '.vms_codes'
    if not os.path.exists(CODE_DIR):
        os.mkdir(CODE_DIR)
    code = '\n'.join(instructions)
    i = 0
    while os.path.exists(f"{CODE_DIR}/.vms_code{i}.vm"):
        i += 1
    fh = open(f"{CODE_DIR}/.vms_code{i}.vm", "w")
    fh.write(code)
    fh.close()
    stream = os.popen(f'{VMS_PROG} {f"{CODE_DIR}/.vms_code{i}.vm"}')
    return stream.read()

CALL = 'CALL'
RETURN = 'RETURN'
MULT = 'MUL'
EQUAL = 'EQUAL'
START = 'START'
STOP = 'STOP'
ADD = 'ADD'
SUB = 'SUB'
PADD = 'PADD'
PUSHGP = 'PUSHGP'
STOREN = 'STOREN'
LOADN = 'LOADN'
##############################################################################################

def g_eval_expr(p,commands,astop=None,parse_state=None):
    """
    3 relevant cases:
        - subexpression
        - variable
        - primitive value
    """
    if parse_state is None:
        parse_state = []
```

```
    match p[0]:
        case 'decl':
            for pair in p[1:]:
                var_name = pair[0]
                var_type = pair[1]
                parser.ids[var_name] = Var(
                    var_name,
                    parser.global_vars,
                    var_type)
                if var_type == 'array':
                    if len(pair) > 2:
                        parser.ids[var_name].end = parser.ids[var_name].order + \
                            pair[2] - 1
                        parser.global_vars += int(pair[2])
                        commands.append(push_n(pair[2]))
                    else:
                        parser.ids[var_name].order = None
                else:
                    parser.global_vars += 1
                    commands.append(push_n(1))
        case 'setsize':
            if parser.ids[p[1]].var_type == 'array' and parser.ids[p[1]].order is None:
                instructions = parse_state['commands_so_far'] + g_eval_expr(p[2],[])
                arr_size = int(vms(instructions,'i'))
                commands.append(push_n(arr_size))
                parser.ids[p[1]].order = parser.global_vars
                parser.ids[p[1]].end = parser.ids[p[1]].order + arr_size - 1
                parser.global_vars += arr_size
        case 'let':
            for pair in p[1:]:
                var_name = pair[0]
                var_value = pair[1]
                push_rec(var_value,commands,parse_state)
                if var_name[0] == '#':
                    ref = parser.funcstack[-1].aref(var_name[1:])
                elif var_name[0] == '$':
                    print('ref -------',var_name)
                    ref = parser.funcstack[-1].rref(var_name[1:])
                else:
                    ref = parser.ids[var_name].order
                commands.append(store_l(ref))
        case 'mirror':
            commands.append(r'⊕')
            commands.append(push(p[1])[6:])
            commands.append(r'⊕')
        case 'mdecl':
            parser.ids[p[1]] = Var(p[1],parser.global_vars,'matrix')
            parser.ids[p[1]].dim1 = p[2]
            parser.ids[p[1]].dim2 = p[3]
            commands.append(push_n(p[2]*p[3]))
            parser.global_vars += p[2]*p[3]
        case 'mref':
            # (mref array_name row col)
            commands.append(push_l(parser.ids[p[1]].order + p[2]*parser.ids[p[1]].dim2 + p
        case 'mset':
```

```
            commands.extend([push(p[4]), store_1(
                parser.ids[p[1]].order + p[2]*parser.ids[p[1]].dim2+p[3])])
        case 'aref':
            # (aref array_name index)
            if type(p[1]) != list and p[1] in parser.ids and type(parser.ids[p[1]]) == Var
                commands.append(PUSHGP)
                push_rec(parser.ids[p[1]].order,commands,parse_state)
                commands.append(PADD)
                push_rec(p[2],commands,parse_state)
                commands.append(LOADN)
        case 'aset':
            # (aset array_name index value)
            commands.append(PUSHGP)
            push_rec(parser.ids[p[1]].order,commands,parse_state)
            commands.append(PADD)
            push_rec(p[2],commands,parse_state)
            push_rec(p[3],commands,parse_state)
            commands.append(STOREN)
        case 'while':
            begin_while = get_label1()
            end_while = get_label1()
            commands.append(begin_while + ':')
            push_rec(p[1],commands,parse_state)
            commands.append(jz(end_while))
            for body_part in p[2]:
                push_rec(body_part,commands,parse_state)
            commands.append(jump(begin_while))
            commands.append(end_while + ':')
        case ('mul' | 'add' | 'sub' | 'div' |
              'fmul' | 'fadd' | 'fsub' | 'fdiv' | 'mod' |
              'inf' | 'infeq' | 'sup' | 'supeq' |
              'finf' | 'finfeq' | 'fsup' | 'fsupeq'| 'equal'):
            for arg in p[1:]:
                push_rec(arg,commands,parse_state)
            commands.append(p[0].upper())
        case 'writei' | 'writef' | 'writes' | 'atoi' | 'atof' | 'itof' | 'ftoi' | 'stri' |
            push_rec(p[1],commands,parse_state)
            commands.append(p[0].upper())
        case 'read':
            commands.append(p[0].upper())
        case 'goto':
            commands.append('goto' + jump(p[1]))
        case 'label':
            commands.append('goto' + p[1])
        case 'call':
            parser.funcstack.append(parser.ids[p[1]])
            commands.append(push_n(parser.ids[p[1]].nouts+parser.ids[p[1]].nargs))
            for arg in p[2:]:
                push_rec(arg,commands,parse_state)
            push_rec(p[1],commands,parse_state)
            commands.extend([push(parser.ids[p[1]]),CALL,pop(parser.ids[p[1]].nargs)])
        case 'defun':
            if astop is not None:
                astop.append(True)
            parser.ids[p[1]] = FUNC(p[1],p[2],p[3],get_label1())
```

```
            parser.funcstack.append(parser.ids[p[1]])
            commands.append(parser.ids[p[1]].pos + ":")
            for body_part in p[4]:
                push_rec(body_part,commands,parse_state)
            commands.append(RETURN)
        case 'case':
            push_rec(p[1],commands,parse_state)
            commands.append(dup(1))
            end_case = 'l' + str(parser.label_count + len(p[2:]))
            parser.label_count += 1
            for i,case0 in enumerate(p[2:]):
                expr1,expr2 = case0[0],case0[1]
                if i == 0:
                    push_rec(expr1,commands,parse_state)
                    commands.append(EQUAL)
                    next_case_label = 'l' + str(parser.label_count)
                    parser.label_count += 1
                    commands.append(jz(next_case_label))
                    for body_part in expr2:
                        push_rec(body_part,commands,parse_state)
                    commands.append(jump(end_case))
                else:
                    commands.append(next_case_label + ":")
                    commands.append(dup(1))
                    push_rec(expr1,commands,parse_state)
                    commands.append(EQUAL)
                    next_case_label = 'l' + str(parser.label_count)
                    parser.label_count += 1
                    commands.append(jz(next_case_label))
                    push_rec(expr2,commands,parse_state)
                    if i != len(p[2:]) - 1:
                        commands.append(jump(end_case))
            commands.append(next_case_label + ':')
    return commands

##############################################################################################

def p_expr_ID(p):
    """expr : ID"""
    p[0] = p[1]
    print('p_expr_ID =',p[0])

def p_expr_STR(p):
    """expr : STR"""
    p[0] = p[1]
    print('p_expr_STR =',p[0])

def p_expr_INT(p):
    """expr : INT"""
    p[0] = int(p[1])
    print('p_expr_INT =',p[0])

def p_expr_REAL(p):
    """expr : REAL"""
    p[0] = float(p[1])
```

```
    print('p_expr_REAL =',p[0])

#############################################################################################

commands31 = []
def p_expr_list(p): #quando programa termina
    """expr : list"""
    p[0] = p[1]
    print('p_expr_list =', p[0])
    parse_state = dict.fromkeys(['commands_so_far', 'astop', 'expr'])
    if p[0][0] == 'do':
        res1 = []
        res2 = []
        parse_state['commands_so_far'] = []
        for expr in p[0][1:]:
            astop = []
            res = g_eval_expr(expr,[],astop,parse_state)
            if astop:
                res2.extend(res)
            else:
                res1.extend(res)
            parse_state['commands_so_far'] = [START] + res1 + [STOP] + res2
        commands = parse_state['commands_so_far']
        vms(commands)
        pre_mirror = '\n'.join(commands)
        post_mirror = re.sub(r'\n⊕\n([^⊕]+)⊕\n',r' \1',pre_mirror)
        print(post_mirror)

# fim da expressao
def p_list(p):
    """list : LP seq RP"""
    p[0] = p[2]
    print('p_list =', p [2])

#############################################################################################

def p_seq(p):
    """seq : expr seq """
    p[0] = [p[1]] + p[2]
    print('p_seq =', [p[1]],'+',p[2])

def p_seq_empty(p):
    """seq : """
    p[0] = []

#############################################################################################

def p_error(p):
    print("Syntax error", p)
    parser.exito = False

#############################################################################################

parser = yacc.yacc()
parser.exito = True
```

```
##############################################################################################
class Object(object):
    pass

parser.funcs = {}
parser.states = Object()
parser.states.decl = False
# where identifiers will be stored
parser.types = {'int', 'real', 'string'}
parser.functions = {'decl', 'while', 'let', 'defprim', 'defun'}
parser.funcstack = []
parser.ids = dict.fromkeys(parser.functions.union(parser.types))
parser.exito = True
parser.label_count1 = 0
parser.global_vars = 0
parser.label_count = 0
parser.label_count1 = 0
parser.label_count2 = 0
parser.local_vars = {}
parser.decls = []
parser.whilestack = []
parser.casestack = []
##############################################################################################

if __name__ == "__main__":
    text = ''
    for line in sys.stdin:
        match line:
            case '_@_\n':
                parser.parse(text)
                tok = parser.token()
                while tok:
                    print(tok)
                    tok = parser.token()
                text = ''
            case '!\n':
                print(vms(commands31[-1]))
                text = ''
            case _:
                text += line

if parser.exito:
    print("Parsing finished successfully!")
```