Universidade do Minho
Ciências da Computação
Processamento de Linguagens e Compiladores

TP2: Gramáticas e Compiladores

Relatório de Desenvolvimento

Grupo 8
Ana Beatriz Ribeiro e Silva (A91678)
Beatriz Fernandes Oliveira (A91640)

Conteúdo

1 Introdução

No seguinte relatório encontra-se a exposição do processo de resolução do trabalho prático proposto no âmbito da cadeira Processamento de Linguagens e Compiladores, que aborda a construção de um compilador de linguagens imperativas simples para a *Assembly da Virtual Machine*.

É importante saber que um compilador é definido como um programa que, a parir de um código fonte, cria um programa semanticamente equivalente, no entanto, escrito numa linguagem diferente, para que a instrução possa ser lida pelo computador.

Neste trabalho vamos então apresentar a linguagem imperativa que desenvolvemos, e também o programa criado para a leitura desta, com o auxilio da linguagem de programação Python e do módulo **.ply**.

2 Descrição do Problema

O trabalho tem vários propósitos, sendo que um deles é fornecer uma melhor compreensão da matéria lecionada durante o semestre. Com este projeto é esperado que a nossa aptitude de escrever gramáticas, sejam elas independentes de contexto (**GIC**), ou tradutoras (**GT**), se torne melhor e, da mesma forma, encoraja o progresso das nossas capacidades de escrever gramáticas independentes do contexto que satisfazem a condição LR() usando BNF puro. Entre os objetivos a cumprir com a resolução do trabalho proposto encontram-se:

- o desenvolvimento de processadores de linguagens segundo o método da *tradução dirigida pela sintaxe*, a partir de uma GT;
- o desenvolvimento de um **compilador** que gera código para a **VM (Virtual Machine**);
- a utilização de geradores de compiladores baseados em GT, em concreto o **Yacc**,e também pelo gerador de analisadores léxicos **Lex**, ambos na versão **PLY** do **Pyhton**.

A linguagem criada tem, ainda, de suportar as seguintes funcionalidades:

- declarar variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas;
- efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis;
- ler do standard input e escrever no standard output;
- efetuar instruções condicionais para controlo do fluxo de execução;
- efetuar instruções cíclicas para controlo do fluxo de execução, permitindo o seu aninhamento;
- declarar e manusear variáveis estruturadas do tipo array de inteiros, em relação aos quais é apenas permitida a operação de indexação;

É de realçar que as variáveis só poderão ser declaradas no início do programa, sem opção para re-declarações das mesmas, ou até mesmo utilizações destas sem a sua declaração prévia. E, ainda, se não for explicitado o valor da variável, considera-se que o valor é zero.

3 Decisões de Implementação

De forma a cumprir muitos dos requisitos pedidos, consideramos essencial que a nossa gramática, para além de aceitar int e float, aceita-se também strings e, ainda, arrays de variadas dimensões. Para além disso, consideramos também extremamente necessário a existência de regras de produção na gramática que permitissem a utilização de comandos como o *if (expression) then commands*, o *if (expression) then commands else commands* e *while (expression) do commands*. De uma forma muito simples, conseguimos, ainda, que a nossa linguagem permitisse a utilização de funções cujos argumentos podem ser ou nulos, ou unitários, ou, então, uma sequência de argumentos. Assim sendo, apresentamos de seguida a nossa gramática.

3.1 Gramática

```
<prog> : <commands> ';'
<commands> : <commands> ';' <command>
           | <command>

<command> : <declaration>
          | <assignment>
          | <flux_control>
          | <prim_func_call>

<stype> : int | float | str

<sconst> : float_const
         | int_const
         | str_const

<declaration> : id_const : <stype>
              | id_const : <stype> <declaration_array_subscript_seq>

<declaration_array_subscript> : [ int_const ]
<declaration_array_subscript_seq> : <declaration_array_subscript>
                                  | <declaration_array_subscript_seq> <array_subscript>

<ref> : <id_ref>
      | <array_ele_ref>

<id_ref> : id_const

<array_ele_ref> : array_1d_ref
                | array_2d_ref

<array_subscript> : [ <expression> ]
<array_1d_subscript> : <array_subscript>
<array_2d_subscript> : <array_1d_subscript> <array_subscript>

<array_1d_ref_value> : <id_const> <array_1d_subscript>
<array_2d_ref_value> : <id_const> <array_2d_subscript>

<assignment> : <id_ref> <assignment_operator> <expression>
             | <array_ele_ref> <assignment_operator> <expression>

<expression> : <sconst>
             | <prim_func_call>
             | <ref_value>
             | <arithmetic_exp_p0>

<relational_operator> : '=' | <' | '>' | <diferent_operator> | <GEQ_operator> | <LEQ_operator>

<arithmetic_exp_p0> : <arithmetic_exp_p1>
                    | <arithmetic_exp_p0> <relational_operator> <arithmetic_exp_p1>
```

```
<arithmetic_exp_p1> : <arithmetic_exp_p2>
                    | <arithmetic_exp_p1> '+' <arithmetic_exp_p2>
                    | <arithmetic_exp_p1> '-' <arithmetic_exp_p2>

<arithmetic_exp_p2> : <arithmetic_exp_p3>
                    | <arithmetic_exp_p2> '*' <arithmetic_exp_p3>
                    | <arithmetic_exp_p2> '/' <arithmetic_exp_p3>
                    | <arithmetic_exp_p2> '%' <arithmetic_exp_p3>

<arithmetic_exp_p3> : <arithmetic_exp_p4>
                    | ! <arithmetic_exp_p4>

<arithmetic_exp_p4> : <ref_value>
                    | <sconst>
                    | <prim_func_call>
                    | '(' <expression> ')'

<ref_value> : <array_ele_ref>
            | <id_ref>

<flux_control> : <if_command>
               | <while_command>

<if_command> : <if_then>
             | <if_then_else>

<if_then> : if <expression> then <commands> fi

<if_then_else> : if <expression> then <commands> else <commands> fi

<while_command> : while <expression> do <commands> od

<prim_func_call> : <primitive_id_const> '(' <func_args> ')'

<func_args> : <func_arg>
            | <func_args> ',' <func_arg>
            | <expression>
            | E

<comment> : r'`[^`]*`'

<float_const> : r'[0-9]*\.[0-9]+'
<int_const> : r'\d+'
<str_const> : r'".*[^\\]"'
<id_const> :  r'[a-zA-Z_][a-zA-Z_0-9]*'

<assignment_operator> : r':='
<diferent_operator> : r'/='
<GEQ_operator> : r'>='
<LEQ_operator> : r'<='
```

4 Exemplos de utilização

- ler 4 números e ver se são lados de um quadrado;

```
a : int;
b : int;
c : int;
d : int;

_start();
a := _atoi(_read());
b := _atoi(_read());
c := _atoi(_read());
d := _atoi(_read());

if (a = b) * (a = c) * (a = d) then
    _writes("Sao lados do quadrado")
else
    _writes("Nao sao lados do quadrado")
fi;
_stop();
```

Código assembly gerado:

```
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
PUSHG 0
PUSHG 2
EQUAL
MUL
PUSHG 0
PUSHG 3
EQUAL
MUL
JZ l0
PUSHS "Sao lados do quadrado"
WRITES
JUMP l1
PUSHS "Nao sao lados do quadrado"
WRITES
l1:
STOP
```

- ler um inteiro N, depois ler N números e escrever o menor destes;

```
N : int;
min : int;
n : int;
i : int;

_start();
N := _atoi(_read());
min := _atoi(_read());
i := 1;

while i < N do
    n := _atoi(_read());
    if n < min then
       min := n
    fi;
    i := i + 1
 od;
 _writei(min);
 _stop();
```

Código assembly gerado:

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
PUSHI 1
STOREG 3
l1:
PUSHG 3
PUSHG 0
INF
JZ l2
READ
ATOI
STOREG 2
PUSHG 2
PUSHG 1
INF
JZ l0
PUSHG 2
STOREG 1
l0:
PUSHG 3
PUSHI 1
ADD
STOREG 3
JUMP l1
l2:
PUSHG 1
WRITEI
STOP
```

- ler N números, sendo N uma constante do programa, e calcular e imprimir o seu produtório;

```
N : int;
prod : int;
i : int;
n : int;

_start();
N := _atoi(_read());
prod := 1;
i := 0;

while i < N do
     n := _atoi(_read());
     prod := prod * n;
     i := i + 1
od;
_writei(prod);
_stop();
```

Código assembly gerado:

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
PUSHI 1
STOREG 1
PUSHI 0
STOREG 2
l0:
PUSHG 2
PUSHG 0
INF
JZ l1
READ
ATOI
STOREG 3
PUSHG 1
PUSHG 3
MUL
STOREG 1
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP l0
l1:
PUSHG 1
WRITEI
STOP
```

- contar e imprimir os números ímpares de uma sequência de números naturais;

```
i : int;
N : int;
a : int[10];
```

```
_start();
i := 0;
N := 10;

while i < N do
    a[i] := _atoi(_read());
    i := i + 1
od;

i := 0;

 while i < N do
    if a[i] % 2 then
        _writei(a[i])
    fi;
    i := i + 1
 od;
_stop();
```

Código assembly gerado:

```
PUSHI 0
PUSHI 0
PUSHN 10
START
PUSHI 0
STOREG 0
PUSHI 10
STOREG 1
l0:
PUSHG 0
PUSHG 1
INF
JZ l1
PUSHGP
PUSHI 2
PADD
PUSHG 0
READ
ATOI
STOREN
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP l0
l1:
PUSHI 0
STOREG 0
l3:
PUSHG 0
PUSHG 1
INF
JZ l4
PUSHGP
PUSHI 2
PADD
PUSHG 0
LOADN
PUSHI 2
```

```
MOD
JZ l2
PUSHGP
PUSHI 2
PADD
PUSHG 0
LOADN
WRITEI
l2:
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP l3
l4:
STOP
```

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

```
i : int;
N : int;
a : int[10];

_start();
i := 0;
N := 10;

while i < N do
    a[i] := _atoi(_read());
    i := i + 1
od;

i:= i - 1;

while i >= 0 do
    _writei(a[i]);
    i := i - 1
od;
_stop();
```

Código assembly gerado:

```
PUSHI 0
PUSHI 0
PUSHN 10
START
PUSHI 10
STOREG 1
PUSHI 0
STOREG 0
l0:
PUSHG 0
PUSHG 1
INF
JZ l1
PUSHGP
PUSHI 2
PADD
PUSHG 0
READ
ATOI
```

```
STOREN
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP l0
l1:
PUSHG 0
PUSHI 1
SUB
STOREG 0
l2:
PUSHG 0
PUSHI 0
SUPEQ
JZ l3
PUSHGP
PUSHI 2
PADD
PUSHG 0
LOADN
WRITEI
PUSHG 0
PUSHI 1
SUB
STOREG 0
JUMP l2
l3:
STOP
```

5 Conclusão

A resolução deste trabalho esteve repleta de desafios, no entanto, consideramos possível afirmar, com confiança, que nos ajudou a aprofundar e a cimentar os conhecimentos adquiridos durante as aulas de Processamento de Linguagens e Compiladores. Em particular, permitiu alargar o nosso espectro do conhecimento relativo ao processo de criação e ao funcionamento de um compilador. Adicionalmente, deu-nos a oportunidade de relembrar o conhecimento já adquirido sobre o funcionamento de stacks.

Fazendo uma apreciação crítica da nossa solução, consideramos, ainda que existe uma limitação. Que corresponde à inexistência de uma configuração que permita a invocação de funções ou, até mesmo, a sua definição. Apesar de termos criado a *prim_func_call*, esta apenas permite-nos utilizar funções primitivas, tornando-se um pouco redundante, uma vez que essas funções acabam por se transformar nos respetivos comandos em Assembly.

Gostaríamos de ter implementado mais funcionalidades para além das que foram requisitadas, mas devido a uma questão de tempo, não foi possível, no entanto, diríamos que aquilo que está presente foi feito com cuidado para garantir os melhores resultados possíveis. Podemos ver ainda, no apêndice abaixo, todo o código escrito para a solução do trabalho proposto.

A Código do Programa

A.1 Analisador Léxico

```
import ply.lex as lex
import sys

tokens = ["id_const","float_const", "int_const", "str_const",'int','str','float',
'assignment_operator','diferent_operator','GEQ_operator','LEQ_operator',
'if','then','else', 'do','od','while','fi','primitive_id_const','comment']

literals = ['(',')','[',']','=','+','-','*','/','%','<','>',':','!',';','','']

# declaração das Palavras-Reservadas e dos Simbolos de Classe (variáveis)

def t_if(t):
    r'\bif\b'
    return t

def t_fi(t):
    r'\bfi\b'
    return t

def t_then(t):
    r'\bthen\b'
    return t

def t_else(t):
    r'\belse\b'
    return t

def t_do(t):
    r'\bdo\b'
    return t

def t_od(t):
    r'\bod\b'
    return t

def t_while(t):
    r'\bwhile\b'
    return t

def t_diferent_operator(t):
    r'/='
    return t

def t_GEQ_operator(t):
    r'>='
    return t

def t_LEQ_operator(t):
    r'<='
    return t

def t_assignment_operator(t):
    r':='
    return t

def t_int(t):
    r'\bint\b'
```

```
    return t

def t_float(t):
    r'\bfloat\b'
    return t

def t_str(t):
    r'\bstr\b'
    return t

def t_comment(t):
    r'`[^`]*`'
    pass

def t_primitive_id_const(t):
    r'_[a-zA-Z_][a-zA-Z_0-9]*'
    return t

def t_id_const(t):
    r'[a-zA-Z][a-zA-Z_0-9]*'
    return t

def t_float_const(t):
    r"[0-9]*\.[0-9]+"
    t.value = float(t.value)
    return t

def t_int_const(t):
    r"[0-9]+"
    t.value = int(t.value)
    return t

def t_str_const(t):
    r'"[^"]*"'
    # t.value = t.value[1:-1]
    return t

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

# declaração da ação a fazer relativa aos carateres que não podem aparecer no texto de entrada
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()
```

A.2 Yacc

```
import sys
import ply.yacc as yacc
from lex import tokens

class GlobalEnv(dict):
    def __init__(self):
        self.naddr = 0
        self.nlabels = 0

    @property
    def label(self):
        label = self.nlabels
        self.nlabels += 1
        return f'l{label}'

    @property
    def addr(self):
        addr = self.naddr
        self.naddr += 1
        return addr

    def decl(self,var_name,var_type,subtype=None,dims=None):
        if subtype is None:
            self.update({var_name : Var(self,var_type)})
        else:
            self.update({var_name : Array(self,var_type,subtype,dims)})

class Var:
    def __init__(self,env,var_type):
        self.addr = env.addr
        self.type = var_type
        self.size = 1

class Array(Var):
    def __init__(self,env,var_type,subtype,dims):
        from math import prod
        super().__init__(env,var_type)
        self.subtype = subtype
        self.dims = [int(dim) for dim in dims]
        self.size = prod(self.dims)
        env.naddr += self.size - 1

    def get(self,n):
        """Only makes sense for arrays. @n is an array of indexes."""
        if len(n) == 1:
            return n[0]
        if self.size >= 1:
            from math import prod
            N = self.dims
            d = len(self.dims)
            return sum(prod(N[j] for j in range(i,d)) * n[i] for i in range(d))//d

global_env = GlobalEnv()
global_debug = True

def p_prog(p):
    """
    prog : commands ';'
```

```
    """
    p[0] = p[1]
    if global_debug:
        print('p_prog')
    print(p[0])

def p_commands_2(p):
    """
    commands : commands ';' command
    """
    p[0] = p[1] + p[3]

    if global_debug:
        print('p_commands_2')

def p_commands_1(p):
    """
    commands : command
    """
    p[0] = p[1]
    if global_debug:
        print('p_commands_1')

def p_command(p):
    """
    command : declaration
            | assignment
            | flux_control
            | prim_func_call
    """
    p[0] = p[1]
    if global_debug:
        print('p_command')

def p_stype(p):
    """
    stype : int
          | float
          | str
    """
    p[0] = p[1]

def p_sconst_float(p):
    """
    sconst : float_const
    """
    p[0] = f'PUSHF {p[1]}\n'
    if global_debug:
        print('p_sconst_float')

def p_sconst_int(p):
    """
    sconst : int_const
    """
    p[0] = f'PUSHI {p[1]}\n'
    if global_debug:
        print('p_sconst_int')

def p_sconst_str(p):
    """
    sconst : str_const
```

```
    """
    p[0] = f'PUSHS {p[1]}\n'
    if global_debug:
        print('p_sconst_str')

def p_declaration_simple(p):
    """
    declaration : id_const ':' stype
    """
    global_env.decl(p[1],p[3])
    p[0] = f'PUSHI 0\n'
    if global_debug:
        print('p_declaration_simple')

def p_declaration_array(p):
    """
    declaration : id_const ':' stype declaration_array_subscript_seq
    """
    global_env.decl(p[1],'array',p[3],p[4])
    p[0] = f'PUSHN {global_env[p[1]].size}\n'
    if global_debug:
        print('p_declaration_array')

def p_ref(p):
    """
    ref : id_ref
        | array_ele_ref
    """
    p[0] = p[1]
    if global_debug:
        print('p_ref')

def p_id_ref(p):
    """
    id_ref : id_const
    """
    # p[0] = f'PUSHG {global_env[p[1]].addr}\n'
    p[0] = global_env[p[1]].addr
    if global_debug:
        print('p_id_ref')

def p_array_subscript(p):
    """
    array_subscript : '[' expression ']'
    """
    p[0] = p[2]
    if global_debug:
        print('p_array_subscript')

def p_array_1d_subscript(p):
    """
    array_1d_subscript : array_subscript
    """
    p[0] = p[1]
    if global_debug:
        print('p_array_1d_subscript')

def p_array_2d_subscript(p):
    """
    array_2d_subscript : array_1d_subscript array_subscript
    """
```

```
    p[0] = [p[1],p[2]]
    if global_debug:
        print('p_array_2d_subscript')

def p_array_1d_ref(p):
    """
    array_1d_ref : id_const array_1d_subscript
    """
    p[0] = f'PUSHGP\nPUSHI {global_env[p[1]].addr}\nPADD\n{p[2]}'
    if global_debug:
        print('p_array_1d_ref')

def p_array_2d_ref(p):
    """
    array_2d_ref : id_const array_2d_subscript
    """
    p[0] = f'PUSGHP\nPUSHG {global_env[p[1]].addr}\nPADD\nPUSHI{global_env[p[1]].dims[0]}\n{p[2][0]}\nMUL\
    if global_debug:
        print('p_array_2d_ref')


def p_array_ele_ref(p):
    """
    array_ele_ref : array_1d_ref
                  | array_2d_ref
    """
    p[0] = p[1]
    if global_debug:
        print('p_array_ele_ref')

def p_declaration_array_subscript(p):
    """
    declaration_array_subscript : '[' int_const ']'
    """
    p[0] = p[2]
    if global_debug:
        print('p_declaration_array_subscript')

def p_declaration_array_subscript_seq_1(p):
    """
    declaration_array_subscript_seq : declaration_array_subscript
    """
    p[0] = [p[1]]
    if global_debug:
        print('p_declaration_array_subscript_seq_1')

def p_declaration_array_subscript_seq_2(p):
    """
    declaration_array_subscript_seq : declaration_array_subscript_seq declaration_array_subscript
    """
    p[0] = p[1] + [p[2]]
    if global_debug:
        print('p_declaration_array_subscript_seq_2')

def p_assignment(p):
    """
    assignment : id_ref assignment_operator expression
    """
    p[0] = p[3] + f'STOREG {p[1]}\n'
    if global_debug:
        print('p_assignment')
```

```
def p_assignment_array(p):
    """
    assignment : array_ele_ref assignment_operator expression
    """
    p[0] = p[1] + p[3] + 'STOREN\n'
    if global_debug:
        print('p_assignment')

def p_expression(p):
    """
    expression : sconst
               | prim_func_call
               | ref_value
               | arithmetic_exp_p0
    """
    p[0] = p[1]
    if global_debug:
        print('p_expression')

def p_arithmetic_exp_p0_1(p):
    """
    arithmetic_exp_p0 : arithmetic_exp_p1
    """
    p[0] = p[1]
    if global_debug:
      print('p_arithmetic_exp_p0_1')

def p_arithmetic_exp_p0_2(p):
    """
    arithmetic_exp_p0 : arithmetic_exp_p0 relational_operator arithmetic_exp_p1
    """
    match p[2]:
        case '=':
            command = 'EQUAL'
        case '/=':
            command = 'EQUAL\n NOT'
        case '<':
            command = 'INF'
        case '>':
            command = 'SUP'
        case '>=':
            command = 'SUPEQ'
        case '<=':
            command = 'INFEQ'
    p[0] = p[1] + p[3] + command + '\n'
    if global_debug:
      print('p_arithmetic_exp_p0_2')

def p_arithmetic_exp_p1_1(p):
    """
    arithmetic_exp_p1 : arithmetic_exp_p2
    """
    p[0] = p[1]
    if global_debug:
      print('p_arithmetic_exp_p1_1')

def p_arithmetic_exp_p1_add(p):
    """
    arithmetic_exp_p1 : arithmetic_exp_p1 '+' arithmetic_exp_p2
    """
```

```
    p[0] = p[1] + p[3] + 'ADD\n'
    if global_debug:
      print('p_arithmetic_exp_p1_add')

def p_arithmetic_exp_p1_sub(p):
    """
    arithmetic_exp_p1 : arithmetic_exp_p1 '-' arithmetic_exp_p2
    """
    p[0] = p[1] + p[3] + 'SUB\n'
    if global_debug:
      print('p_arithmetic_exp_p1_sub')

def p_arithmetic_exp_p2_1(p):
    """
    arithmetic_exp_p2 : arithmetic_exp_p3
    """
    p[0] = p[1]
    if global_debug:
      print('p_arithmetic_exp_p2_1')

def p_arithmetic_exp_p2_mul(p):
    """
    arithmetic_exp_p2 : arithmetic_exp_p2 '*' arithmetic_exp_p3
    """
    p[0] = p[1] + p[3] + 'MUL\n'
    if global_debug:
      print('p_arithmetic_exp_p2_mul')

def p_arithmetic_exp_p2_div(p):
    """
    arithmetic_exp_p2 : arithmetic_exp_p2 '/' arithmetic_exp_p3
    """
    p[0] = p[1] + p[3] + 'DIV\n'
    if global_debug:
      print('p_arithmetic_exp_p2_div')

def p_arithmetic_exp_p2_mod(p):
    """
    arithmetic_exp_p2 : arithmetic_exp_p2 '%' arithmetic_exp_p3
    """
    p[0] = p[1] + p[3] + 'MOD\n'
    if global_debug:
      print('p_arithmetic_exp_p2_mod')

def p_arithmetic_exp_p3_1(p):
  """
  arithmetic_exp_p3 : arithmetic_exp_p4
  """
  p[0]= p[1]
  if global_debug:
    print('p_arithmetic_exp_p3_1')

def p_arithmetic_exp_p3_not(p):
  """
  arithmetic_exp_p3 : '!' arithmetic_exp_p4
  """
  p[0]= p[2] + 'NOT\n'
  if global_debug:
    print('p_arithmetic_exp_p3_not')

def p_arithmetic_exp_p4(p):
```

```
    """
    arithmetic_exp_p4 : ref_value
                      | sconst
                      | prim_func_call
                      | '(' expression ')'
    """
    p[0] = p[1] if len(p) == 2 else p[2]
    if global_debug:
      print('p_arithmetic_exp_p4')

def p_relational_operator(p):
    """
    relational_operator : '='
                        | diferent_operator
                        | '<'
                        | '>'
                        | GEQ_operator
                        | LEQ_operator
    """
    p[0] = p[1]
    if global_debug:
      print('p_relational_operator')

def p_ref_value_array(p):
    """
    ref_value : array_ele_ref
    """
    p[0] = p[1] + 'LOADN\n'
    if global_debug:
      print('p_ref_value_array')

def p_ref_value_id(p):
    """
    ref_value : id_ref
    """
    p[0] = f'PUSHG {p[1]}\n'
    if global_debug:
      print('p_ref_value_id')

def p_flux_control(p):
    """
    flux_control : if_command
                 | while_command
    """
    p[0] = p[1]
    if global_debug:
        print('p_flux_control')

def p_if_command(p):
    """
    if_command : if_then
               | if_then_else
    """
    p[0] = p[1]
    if global_debug:
        print('p_if_command')

def p_if_then(p):
    """
    if_then : if expression then commands fi
    """
```

```
    endif = global_env.label
    p[0] = p[2] + f'JZ {endif}\n' + p[4] + f'{endif}:\n'
    if global_debug:
        print('p_if_then')

def p_if_then_else(p):
    """
    if_then_else : if expression then commands else commands fi
    """
    elsecond = global_env.label
    endif = global_env.label
    p[0] = p[2] + f'JZ {elsecond}\n' + p[4] + f'JUMP {endif}\n' + p[6] +f'{endif}:\n'
    if global_debug:
        print('p_if_then_else')

def p_while_do(p):
    """
    while_command : while expression do commands od
    """
    startwhile = global_env.label
    endwhile = global_env.label
    p[0] = f'{startwhile}:\n' + p[2] + f'JZ {endwhile}\n' + p[4] + f'JUMP {startwhile}\n' + f'{endwhile}:\
    if global_debug:
        print('p_while_do')

def p_prim_func_call(p):
    """
    prim_func_call : primitive_id_const '(' func_args ')'
    """
    p[0] = p[3] + p[1][1:].upper() + '\n'
    if global_debug:
        print('p_prim_func_call')

def p_func_args_1(p):
    """
    func_args : func_arg
    """
    p[0] = p[1]
    if global_debug:
        print('p_func_args_1')

def p_func_args_2(p):
    """
    func_args : func_args ',' func_arg
    """
    p[0] = p[1] + p[3]
    if global_debug:
        print('p_func_args_2')

def p_func_args_empty(p):
    """
    func_args :
    """
    p[0] = ''
    if global_debug:
        print('p_func_args_empty')

def p_func_arg(p):
    """
    func_arg : expression
    """
```

```
    p[0] = p[1]
    if global_debug:
        print('p_func_arg')

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

parser = yacc.yacc()
if __name__ == "__main__":
    for line in sys.stdin:
        parser.success = True
        result = parser.parse(line)
```