Processamento de Linguagens e Compiladores (3º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

Breno Fernando Guerra Marrão
A97768

Tales André Rovaris Machado
A96314

Tiago Passos Rodrigues
A96414

30 de dezembro de 2022

Conteúdo

Capítulo 1

Introdução

Este relatório no âmbito de processamento de linguagens e compiladores do segundo projeto teve como o objetivo a criação de uma simples linguagem de programação que seja capaz de criar variáveis, atribuir valores a essas variáveis, ler e escrever no terminal, escrever ciclos e efetuar instruções de seleção. Na execução deste projeto utilizamos os modulos ply.lex para a análise léxica e yacc para a gramática na linguagem python.

Capítulo 2

Escolhas Feitas

2.1 Desenho da Linguagem

Na nossa linguagem optamos por usar notações case sensitive , tais como os token if, while, etc. Onde é posicionado o habitual ponto e vírgula das outras linguagens de programação nos escolhemos por um ponto final para delimitar as linhas de código.

2.1.1 Declarar variáveis

Decidimos que a declaração de variaveis seria feita no inicio do programa, e é feita com a seguinte notação , segue exemplos :

```
int x . # declaração da variável sem argumentos
int x = 2 . #declacração da variável com atribuição de valor
int y = x + 2 . # declaração da variável com outra variável e com uma operação de soma
```

2.1.2 Operações

Decidimos manter as nossas operações como é costume na maior partes das linguagens , com os seus simbolos, segue exemplos :

```
x = x + 1 .# operação de soma entre variável e inteiro
x = x + y .# operação de soma entre variáveis
x = x - y .# operação de subtração entre variáveis
x = y * z .# operação de multiplicação entre variáveis
x = x / y .# operação de divisão entre variáveis
x = (x - 1 ) * 2 .# operação com prioridade
x = x + 2 + 3 + y * z - 10 / 5 .# Varias operações juntas.
```

2.1.3 Ler do standard input e escrever no standard output

Para escrever no standard output decidimos que deveria começar por print e deveria ser seguido por frases entre aspas e se quisesse colocar números fechava as aspas e escervia os números , variaveis ou expressõe , segue exemplos :

```
int x = 20 .
int y = 10 .
print x . # exemplo de escrever com variável .
print "Trabalho de PLC" # exemplo de escrever com uma frase
```

```
print "Este é nosso segundo projeto de PLC e merecemos " x " como nota " . # exemplo de escrever com frases
↪  e uma variável
print "A soma de variável x e y é  " x + y . # exemplo de escrever com frases e uma operação
```

Para ler no standard output dizemos para qual variável queremos atribuir o valor (que só pode ser inteiro)lido para e alguma frase .

```
int x .
x = input " Qual valor para x ?" .
```

2.1.4 Instruções Condicionais

Nas instruções condicionais esta estrutura tal que devemos primeiro escrever "if"a condição logica "then"as instruções que é para ser realizada e o mesmo para "else", segue exemplos:

```
int x  = 2 .
if x == 2 then x = x + 1 . else x = x - 1 .
```

2.1.5 Instruções Cíclicas

A instrução cíclica que fizemos foi 'do while' em que a estrutura é 'while' seguido da condição do ciclo e depois um 'do' após isso vem as instruções que queremos executar, e terminamos com um 'while .'

```
int x  = 2 .
while x > 0 do x = x - 1 . end while .
```

2.2 Desenho da gramática

Literais : ['+', '-', '*', '/', '=', '(', ')','.','!']
Tokens : 'IF','TRUE','FALSE','EQUAL','INF','INFEQ','SUP','SUPEQ','INPUT','PRINT','INT','OU','E','THEN'
,'ELSE','WHILE','begin','return','end','DO','NUM','ID','FRASE'
A gramática implementada utiliza o método recursivo à esquerda e é constituída pelas seguintes regras de derivação.

```
Programa : Vars Funcs Cod
Vars : Var Vars | €
Funcs : Funcs Func | €
Var : INT ID '.' | INT ID '=' expr '.'
Func : ID begin Cod end return expr '.' | ID begin Cod end '.'
Cod : Linha Cod | €
Linha : Escrever | atr | ID '(' ')' '.' | Ler | cond | SE | Ciclo
SE : IF cond THEN Cod ELSE Cod
Ler : ID '=' INPUT FRASE '.'
Escrever : PRINT corpoescreve '.'
corpoescreve : alter corpoescreve | €
alter : FRASE | expr
Ciclo : WHILE cond DO Cod end WHILE '.'
atr : ID '=' expr '.'
cond : bool | expr | expr oprelacao expr | cond E cond | cond OU cond
oprelacao : INF | EQUAL | DIFF | INFEQ | SUP | SUPEQ
```

```
bool : TRUE | FALSE
expr : termo | expr '+' termo | expr '-' termo
termo : fator | termo '*' fator | termo '/' fator
fator : NUM | ID | '(' expr ')'
```

Capítulo 3

Regras de tradução para Assembly

As regras de tradução para Assembly estão apresentadas no ...

Capítulo 4

Exemplos de Utilização

4.1 testar se o dolali

As regras de tradução para Assembly estão apresentadas no ...

4.2 culcular fatorial

Simples exemplo de programa que calcula fatorial

```
int aux = x - 1 .
while aux > 0 do
x = x * aux .
aux = aux - 1 .
end while .
print x .
```

o codigo assembly gerado foi o seguinte:

```
PUSHI 10
PUSHG 0
PUSHI 1
SUB
comeco0:
PUSHG 1
PUSHI 0
sup
JZ terminar0
PUSHG 0
PUSHG 1
MUL
storeg 0
PUSHG 1
PUSHI 1
SUB
storeg 1
JUMP comeco0
terminar0:
```

```
PUSHG 0
WRITEI
STOP
```

Capítulo 5

Conclusão

Este projeto foi do mais importante para a introdução a técnicas na área do processamento de linguagens e compiladores. Permitiu-nos aplicar em casos concretos os nossos conhecimentos de análise sintática e léxica pelo módulo yacc e ply.lex do python com o uso de tokens e da definição de gramáticas através da criação de uma linguagem e a sua sintaxe, o assembly gerado pela mesma e tratamento de erros.

Consideramos este projeto de grande importância para nos como alunos de ciência da Computação pois agora temos uma capacidade maior de entender como funcionam as linguagens de programação algo que usamos no diariamente e como elas geram código .

Apêndice A

Código do Programa

A.1 Analisador Léxico

```
import re
import ply.lex as lex

literals = ['+', '-', '*', '/', '=', '(', ')','.','!']

tokens = (
    'IF',
    'TRUE',
    'FALSE',
    'EQUAL',
    'INF',
    'INFEQ',
    'SUP',
    'SUPEQ',
    'INPUT',
    'PRINT',
    'INT',
    'OU',
    'E',
    'THEN',
    'ELSE',
    'WHILE',
    'begin',
    'return',
    'end',
    'DO',
    'NUM',
    'ID',
    'FRASE',

)

t_TRUE = r'true'

t_FALSE = r'false'

t_EQUAL = r'\=\='

t_INF = r'\<'
```

```
t_INFEQ = r'\<\='

t_SUP = r'\>'

t_SUPEQ = r'\>\='

t_INPUT = r'input'

t_PRINT = r'print'

t_INT = r'int'

t_OU = r'\|'

t_E = r'\^'

t_THEN = r'then'

t_ELSE = r'else'

t_WHILE = r'while'

t_begin = r'begin'

t_return = r'return'

t_end = r'end'

t_ID = r'\w+'

t_NUM = r'[0-9]+'

t_FRASE = r'\"[^\"]*\"'

t_ANY_ignore = ' \n\t'

def t_IF(t):
    r'if'
    return t

def t_DO(t):
    r'do'
    return t

def t_ANY_error(t):
    print('Illegal character: %s', t.value[0])

lexer = lex.lex()
```

A.2 Analisador Sintático

```
import ply.yacc as yacc
import sys

from linguagem_lex import tokens

def p_Programa(p):
    "Programa : Vars Funcs Cod"
    parser.assembly = f'START\n{p[1]}{p[3]}STOP\n{p[2]}'

def p_Escrever(p):
    "Escrever : PRINT corpoescreve '.'"
    p[0] = f"{p[2]}"

def p_corpoescreve_null(p):
    "corpoescreve : "
    p[0] = f''

def p_corpoescreve_alter(p):
    "corpoescreve : alter corpoescreve"
    p[0] = f'{p[1]}\n{p[2]}'

def p_alter_frase(p):
    "alter : FRASE"
    p[0] = f'pushs {p[1]}\nWRITES'

def p_alter_expr(p):
    "alter : expr"
    p[0] = f'{p[1]}WRITEI'

def p_Vars_Empty(p):
    "Vars : "
    p[0] = f''

def p_Vars_Var(p):
    "Vars : var Vars"
    p[0] = f'{p[1]}{p[2]}'

def p_Funcs_Empty(p):
    "Funcs : "
    p[0] = f''

def p_Funcs_Func(p):
    "Funcs : Funcs Func"
    p[0] = f'{p[1]}\n{p[2]}'

def p_Func_comRETURN(p):
    "Func : ID begin Cod end return expr '.'"
    p[0] = f'{p[1]}:\n{p[3]}\n{p[6]}return\n'

def p_Func(p):
    "Func : ID begin Cod end '.'"
    p[0] = f'{p[1]}:\n{p[3]}\nreturn\n'

def p_Cod_Empty(p):
    "Cod : "
    p[0] = f''
```

```
def p_var_tipoID(p):
    "var : INT ID '.'"
    var = p[2]
    p.parser.table[var] = parser.pc
    parser.pc += 1
    p[0] = "PUSHI 0\n"

def p_var_atribuicao(p):
    "var : INT ID '=' expr '.'"
    var = p[2]
    p.parser.table[var] = parser.pc
    parser.pc += 1
    p[0] = f"{p[4]}"

def p_Cod_linhas(p):
    "Cod : Linha Cod"
    p[0] = f'{p[1]} {p[2]}'

def p_Linha_Escrever(p):
    "Linha : Escrever"
    p[0] = p[1]

def p_Linha_atr(p):
    "Linha : atr"
    p[0] = p[1]

def p_linha_func(p):
    "Linha : ID '(' ')' '.'"
    p[0] = f'pusha {p[1]}\nCALL\n'

def p_Linha_Ler(p):
    "Linha : Ler"
    p[0] = p[1]

def p_Linha_Cond(p):
    "Linha : cond"
    p[0] = f'{p[1]}\n'

def p_Linha_Se(p):
    'Linha : SE'
    p[0] = p[1]

def p_se_else(p):
    'SE : IF cond THEN Cod ELSE Cod '
    p[0] = f'{p[2]}JZ l{p.parser.labels}\n{p[4]}JUMP l{p.parser.labels}f\nl{p.parser.labels}:
    ↪  NOP\n{p[6]}l{p.parser.labels}f: NOP\n'
    p.parser.labels += 1

def p_ler(p):
    "Ler : ID '=' INPUT FRASE '.'"
    p[0] = f"pushs {p[4]}\nWRITES\nread\natoi\nstoreg {parser.table[p[1]]}\n"

def p_Linha_Ciclo(p):
    "Linha : Ciclo"
    p[0] = p[1]

def p_ciclo(p):
    "Ciclo : WHILE cond DO Cod end WHILE '.'"
```

```
    p[0] = f'comeco{p.parser.labels}: \n{p[2]}\nJZ terminar{p.parser.labels}\n{p[4]}JUMP
    ↪  comeco{p.parser.labels}\nterminar{p.parser.labels}: \n'
    p.parser.labels += 1

def p_atr(p):
    "atr : ID '=' expr '.'"
    p[0] = f'{p[3]}storeg {parser.table[p[1]]}\n'

def p_bool_true(p):
    "bool : TRUE"
    p[0] = f'PUSHI 1'

def p_bool_false(p):
    "bool : FALSE"
    p[0] = f'PUSHI 0'

def p_cond_bool(p):
    "cond : bool"
    p[0] = f'{p[1]}'

def p_cond_expr(p):
    "cond : expr"
    p[0] = f'{p[1]}pushi 0\nsup'

def p_oprelacao_inf(p):
    "oprelacao : INF"
    p[0] = 'inf'

def p_oprelacao_EQUALS(p):
    "oprelacao : EQUAL"
    p[0] = 'EQUAL\n'

def p_oprelacao_DIFF(p):
    "oprelacao : DIFF"
    p[0] = 'EQUAL\nNOT\n'

def p_oprelacao_infeq(p):
    "oprelacao : INFEQ"
    p[0] = 'infeq'

def p_oprelacao_sup(p):
    "oprelacao : SUP"
    p[0] = 'sup'

def p_oprelacao_supeq(p):
    "oprelacao : SUPEQ"
    p[0] = 'supeq'

def p_cond_oprelacao(p):
    "cond : expr oprelacao expr"
    p[0] = f'{p[1]}{p[3]}{p[2]}'

def p_cond_e(p):
    "cond : cond E cond"
    p[0] = f'{p[1]}\n{p[3]}\nadd\npushi 2\nequal'

def p_cond_ou(p):
    "cond : cond OU cond"
    p[0] = f'{p[1]}\n{p[3]}\nadd\npushi 0\nsup'
```

```
def p_expr_add(p):
    "expr : expr '+' termo"
    p[0] = p[1] + p[3] + "ADD\n"

def p_expr_sub(p):
    "expr : expr '-' termo"
    p[0] = p[1] + p[3] + "SUB\n"

def p_expr_termo(p):
    "expr : termo"
    p[0] = p[1]

def p_termo_mul(p):
    "termo : termo '*' fator"
    p[0] = p[1] + p[3] + "MUL\n"

def p_termo_div(p):
    "termo : termo '/' fator"
    p[0] = p[1] + p[3] + "DIV\n"

def p_termo_fator(p):
    "termo : fator"
    p[0] = p[1]

def p_fator_NUM(p):
    "fator : NUM"
    p[0] = f"PUSHI {p[1]}\n"

def p_fator_ID(p):
    "fator : ID"
    p[0] = f'PUSHG {parser.table[p[1]]}\n'

def p_fator_expr(p):
    "fator : '(' expr ')'"
    p[0] = p[2]

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

#inicio do Parser e do Processamento
parser = yacc.yacc()
parser.table = {}
parser.pc = 0

parser.sucesso = True
parser.assembly = ""
parser.labels = 0

fonte = ""
for linha in sys.stdin:
    fonte += linha

parser.parse(fonte)

print(parser.table)
print(parser.assembly)
```