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

Trabalho Prático Nº2

Relatório de Desenvolvimento

Grupo 12

José Miguel Barbosa (Nºa95088)

Leonardo Lordello Fontes (Nºa96308)

Tiago Adriano Moreira (Nºa92046)

12 de janeiro de 2023

Resumo

Neste projeto de Processadores de Linguagens e Compiladores realizamos o desenvolvimento de processadores de linguagens e de um compilador. Este ajudou-nos a pôr em pratica muito do conhecimento que adquirimos durante o decorrer da unidade curricular, como a escrita e de gramáticas GIC e GT, treinamos também a geração de código para uma máquina de stack virtual.

Capítulo 1

Introdução

Neste projeto foi nos dado o desafio de enquanto grupo de definir uma linguagem de programação imperativa e desenvolver um compilador especificamente criado para reconhecer esta linguagem personalizada e programas que originem dela e gerar código assembly na maquina virtual colocada acima.

1.1 O Que é um Compilador?

Um compilador é um programa, que pode ser chamado de tradutor de linguagens, uma vez que a partir de um código fonte escrito a partir de uma linguagem compilada traduz este para código objeto. Estes também indicam se houve algum tipo de erro na sintaxe do código. Normalmente usamos compiladores para programas que traduzem linguagens de programação de alto nível (como Java ou C) para uma de baixo nível (como o Assembly). Num compilador existem 3 módulos distintos e importantes sendo que cada um deles faz uma função diferente. O **Analisador Léxico** transforma uma sequência de carateres que são vocábulos validos em códigos inteiros que identificam símbolos. O **Analisador Sintático** converte um sequência de símbolos numa árvore de derivação que corresponde ás regras da gramática reconhecidas. E o **Analisador Semântico** converte uma sequência de símbolos numa árvore de derivação que corresponde ás regras de gramática reconhecidas. Estes 3 ocorrem na ordem em que foram explicadas.

1.2 Problemas e Objetivos

Baseado na descrição na descrição do compilador anterior percebemos logo algumas coisas essenciais a este projeto. As que saltam imediatamente à vista são os 3 analisadores, o léxico, o sintático e o semântico. Precisamos de criar uma gramática e a partir desta sermos capazes de fazer a transição para o assembly.

Os objetivos deste relatório é tornar mais fácil a compreensão do nosso trabalho, decisões que tomamos durante o desenvolvimento deste e potencialmente retirar algumas duvidas que poderiam surgir se apenas o código fosse apresentado.

1.2.1 Estrutura

Tentamos tornar este relatório de projeto o mais organizado possível, principalmente na parte do código. Estamos agora no fim da nossa introdução e de seguida começaremos a mostrar pedaços do nosso código, que se encontram divididos e explicados com detalhe. Após mostramos o código referentes ao parser e analisador semântico irmos mostrar a nossa gramática, mas de um modo que facilite a sua leitura. Após todos estes passos do código mostraremos exemplos, testes que realizamos e os seus respetivos resultados e depois apresentaremos as nossas reflexões e conclusões.

Capítulo 2

Módulos e seu código

2.1 Analisador Léxico

Definimos desta secção de código o Analisador Léxico do nosso trabalho. Como aprendemos durante as aula de Processamento de Linguagens e Compiladores, o Lexer é encarregado de realizar a análise léxica.

Na primeira linha deste (após o import da biblioteca) temos os **Literals** (ou literais). Estes são caracteres que representam uma constante. Os literais que definimos são ***virgula, ponto virgula***, ***diferentes tipos de parênteses e chavetas*** e ***símbolos matemáticos***.

Seguem-se os **Tokens**, os elementos mais básicos sobre os quais se desenvolve toda a tradução de um programa. Estes são segmentos de texto ou símbolos que podem ser manipulados por um analisador sintático, dando assim um significado ao texto. Nós definimos vários tokens no nosso lexer (21 no total) observando cada um algo diferente no código. Temos dois tipos diferentes, símbolos que estão definidos em primeiro lugar, como por exemplo o LE (Less or Equal) ou palavras reservadas, que aparecem de seguida e são definidas sobre forma de função. Algumas destas funções são definidas de modo bastante simples, como por exemplo a t_TIPO_STRING, que apenas lê 'string' e devolve exatamente isto. E outras ligeiramente mais complexas, Estas são a t_TRUE (em que o t.value = 1, representando assim um valor de Verdadeiro) a t_FALSE (em que o t.value = 0, representando assim o valor de Falso) e a t_INT (em que se lê qualquer digito (+) e em que o t.value = int(t.value) ou seja t.value fica inteiro).

De resto temos o t_ignore que ocorre em situações que não estão definidas nos exemplos acima, nem devolvem nada em particular, mas são casos prováveis (como uma mudança de linha ou um tab) sendo que existe para prevenir erros sem sentido. E o t_error que engloba tudo o resto que não está no lexer (caracteres não suportados por exemplo) e trata-se da nossa condição de erro.

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

literals = [',','(',')','[',']','{','}',';','=','+','-','*','/']

tokens = ('ID','TIPO_INT','TIPO_STRING','NOT','AND','OR','TRUE','FALSE',
          'INT','IF','ELSE','WHILE','INPUT','PRINT','STRING',
          'GE', 'GT', 'LT', 'LE', 'EQUALS', 'DIFF')

t_EQUALS = r'=='

t_DIFF = r'!='
```

```
t_LE = r'<='

t_GE = r'>='

t_GT = r'>'

t_LT = r'<'

t_ID = r'[a-zA-Z]+'

def t_TIPO_STRING(t):
    r'string'
    return t

def t_TIPO_INT(t):
    r'int'
    return t

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

def t_PRINT(t):
    r'print'
    return t

def t_INPUT(t):
    r'input'
    return t

def t_WHILE(t):
    r'while'
    return t

def t_ELSE(t):
    r'else'
    return t

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

def t_OR(t):
    r'or'
    return t

def t_AND(t):
    r'and'
    return t
```

```
    def t_NOT(t):
        r'not'
        return t

    def t_TRUE(t):
        r'True'
        t.value = 1
        return t

    def t_FALSE(t):
        r'False'
        t.value = 0
        return t

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

    t_ignore = ' \r\n\t'

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

    lexer = lex.lex() # cria um AnaLex especifico a partir da especificação␣
↪acima usando o gerador 'lex' do objeto 'lex'

    # Reading input

    """
    f = input(">> ")
    lexer.input(f)
    simb = lexer.token()
    while simb:
        print(simb)
        simb = lexer.token()
    """
```

2.2 Parser e Analisador Semântico

Apresentamos a baixo a definição do nosso Parser e do nosso Analisador Semântico. Parsing é equivalente à Analise Sintática, ou seja, é o processo de analisar strings de símbolos seguindo uma determinada gramática. A analise semântica vai tratar da entrada sintática e transforma-la numa representação melhor para a se gerar o código. Dai acharmos por bem juntar estes dois módulos, facilita a sua compreensão.

Abaixo do código teremos a gramática de forma bastante mais legível, uma vez que, apesar de ser possível ler através do código, é muito mais complicado.

[ ]:
```
from analisador_lexico import tokens
import ply.yacc as yacc

endereço = 0
labels = 0
tabela = {}
resultado = ""

file = open("code2.txt", "r")

def p_prog(p):
    "prog : cod"
    global resultado
    resultado += p[1]

def p_cod_inst(p):
    "cod : cod inst ';' "
    p[0] = p[1] + p[2]

def p_cod_vazio(p):
    "cod : "
    p[0] = ""

def p_inst_print_string(p):
    "inst : PRINT '(' STRING ')' "
    p[0] = "PUSHS " + p[3] + "\n" + "WRITES\n"

def p_inst_print_id(p):
    "inst : PRINT '(' ID ')' "
    if p[3] not in tabela:
        print("Variable not defined")
        exit()

    elif tabela[p[3]][0] == "int":
        p[0] = "PUSHG " + str(tabela[p[3]][1]) + "\n" + "WRITEI\n"

    elif tabela[p[3]][0] == "string":
        p[0] = "PUSHG " + str(tabela[p[3]][1]) + "\n" + "WRITES\n"

def p_inst_print_id_array_index(p):
    "inst : PRINT '(' ID '[' INT ']' ')' "
    if p[3] not in tabela:
        print("Variable not defined")
        exit()

    else:
        p[0] = "PUSHG " + str(tabela[p[3]][1] + p[5]) + "\n" + "WRITEI\n"

def p_inst_print_ops(p):
    "inst : PRINT '(' ops ')' "
```

```
    p[0] = p[3] + "\n" + "WRITEI\n"


def p_inst_while(p):
    "inst : WHILE '(' ops ')' '{' cod '}' "
    global labels
    p[0] = "label" + str(labels) + ":\n" + p[3] + "JZ label" + str(labels +␣
 ↪1) + "\n" + p[6] + "JUMP " + "label" + str(labels) + "\n" + "label" +␣
 ↪str(labels + 1) + ":\n"
    labels += 2

def p_inst_if_else(p):
    "inst : IF '(' ops ')' '{' cod '}' ELSE '{' cod '}'"
    global labels
    p[0] = p[3] + "JZ label" + str(labels) + "\n" + p[6] + "JUMP label" +␣
 ↪str(labels+1) + "\n" + "label" + str(labels) + ":\n" + p[10] + "label" +␣
 ↪str(labels+1) + ":\n"
    labels += 2

def p_inst_if(p):
    "inst : IF '(' ops ')' '{' cod '}' "
    global labels
    p[0] = p[3] + "JZ label" + str(labels) + "\n" + p[6] + "label" +␣
 ↪str(labels) + ":\n"
    labels += 1

def p_inst_atr(p):
    "inst : expatr"
    p[0] = p[1]

def p_expatr(p):
    "expatr : atr"
    p[0] = p[1]

def p_atr_var_1d(p):
    "atr : ID '=' ops"
    if p[1] not in tabela:
        print("Variable not defined")
        exit()

    elif tabela[p[1]][0] != "int":
        print("Type do not match")

    else:
        p[0] = p[3] + "STOREG " + str(tabela[p[1]][1]) + "\n"

def p_atr_var_2d(p):
    "atr : ID '[' INT ']' '=' ops"
    if p[1] not in tabela:
        print("Variable not defined")
```

```
        exit()

    elif p[3] > tabela[p[1]][2]:
        print("out of index")

    elif tabela[p[1]][0] != "array_int":
        print("Type do not match")
        exit()

    else:
        p[0] = p[6] + "STOREG " + str(tabela[p[1]][1] + p[3]) + "\n"


def p_atr_decl_input(p):
    "atr : TIPO_STRING ID '=' INPUT "
    global endereço
    tabela[p[2]] = ("string", endereço)

    p[0] = "READ\n"
    endereço += 1

def p_atr_input(p):
    "atr : ID '=' INPUT"
    if p[1] not in tabela:
        print("Variable not defined")
        exit()

    elif tabela[p[1]][0] != "string":
        print("Type do not match")
        exit()

    else:
        p[0] = "READ\n" + "STOREG " + str(tabela[p[1]][1]) + "\n"

def p_atr_ops_INT(p):
    "atr : TIPO_INT ID '=' ops"

    tabela[p[2]] = ("int", endereço - 1)

    p[0] = p[4]

def p_atr_ops_INT_vazio(p):
    "atr : TIPO_INT ID"
    global endereço
    tabela[p[2]] = ("int", endereço )
    p[0] = "PUSHI " + str('0') + "\n"
    endereço += 1

def p_atr_STRING(p):
    "atr : TIPO_STRING ID '=' STRING"
    global endereço
```

```
    tabela[p[2]] = ("string", endereço)
    endereço += 1
    p[0] = "PUSHS " + p[4] + "\n"

def p_atr_STRING_Vazia(p):
    "atr : TIPO_STRING ID"
    global endereço
    tabela[p[2]] = ("string", endereço)
    endereço += 1
    p[0] = "PUSHS " + '""' + "\n"

def p_atr_array_vazio(p):
    "atr : TIPO_INT ID '[' INT ']' '=' '{' '}'"

    global endereço
    p[0] = "PUSHN " + str(p[4]) + "\n"

    tabela[p[2]] = ("array_int", endereço, p[4])

    endereço += p[4]

def p_atr_array_notvazio(p):
    "atr : TIPO_INT ID '[' INT ']' '=' '{' val '}'"

    p[0] = "PUSHN " + str(p[4]) + "\n" + p[8]

    tabela[p[2]] = ("array_int", endereço - p[4], p[4])

def p_val_int(p):
    "val : INT"
    global endereço
    p[0] = "PUSHI " + str(p[1]) + "\n" + "STOREG " + str(endereço) + "\n"
    endereço += 1

def p_val_neg_int(p):
    "val : '-' INT"
    global endereço
    p[0] = "PUSHI " + str(p[2]) + "\n" + "PUSHI " + "-1" + "\n" + "MUL\n" +␣
 ↪"\n" + "STOREG " + str(endereço) + "\n"
    endereço += 1

def p_val_vals(p):
    "val : val ',' INT"
    global endereço
    p[0] = "PUSHI " + str(p[3]) + "\n" + "STOREG " + str(endereço) + "\n" +␣
 ↪p[1]
    endereço += 1

def p_val_vals_neg(p):
    "val : val ',' '-' INT"
```

```
    global endereço
    p[0] = "PUSHI " + str(p[4]) + "\n" + "PUSHI " + "-1" + "\n" + "MUL\n" +␣
 ↪"STOREG " + str(endereço) + "\n" + p[1]
    endereço += 1

def p_atr_array_vazio_vazio(p):
    "atr : TIPO_INT ID '[' INT ']'"

    global endereço
    p[0] = "PUSHN " + str(p[4]) + "\n"

    tabela[p[2]] = ("array_int", endereço, p[4])

    endereço += p[4]

def p_ops(p):
    "ops : exl"
    p[0] = p[1]

def p_exl_tl(p):
    "exl : tl"
    p[0] = p[1]

def p_exl_or(p):
    "exl : exl OR tl"
    p[0] = p[1] + p[3] + "OR\n"
    global endereço
    endereço -= 1

def p_tl_fl(p):
    "tl : fl"
    p[0] = p[1]

def p_tl_and(p):
    "tl : tl AND fl"
    p[0] = p[1] + p[3] + "AND\n"
    global endereço
    endereço -= 1

def p_fl_rl(p):
    "fl : rl"
    p[0] = p[1]

def p_fl_not(p):
    "fl : NOT rl"
    p[0] = p[2] + "NOT\n"

def p_rl_exr(p):
    "rl : exr"
    p[0] = p[1]
```

```
def p_rl_EQUALS(p):
    "rl : rl EQUALS exr"
    p[0] = p[1] + p[3] + "EQUAL\n"
    global endereço
    endereço -= 1

def p_rl_DIFF(p):
    "rl : rl DIFF exr"
    p[0] = p[1] + p[3] + "EQUAL\nNOT\n"
    global endereço
    endereço -= 1

def p_rl_LE (p):
    "rl : rl LE exr"
    p[0] = p[1] + p[3] + "INFEQ\n"
    global endereço
    endereço -= 1

def p_rl_GE(p):
    "rl : rl GE exr"
    p[0] = p[1] + p[3] + "SUPEQ\n"
    global endereço
    endereço -= 1

def p_rl_GT(p):
    "rl : rl GT exr"
    p[0] = p[1] + p[3] + "SUP\n"
    global endereço
    endereço -= 1

def p_rl_LT(p):
    "rl : rl LT exr"
    p[0] = p[1] + p[3] + "INF\n"
    global endereço
    endereço -= 1

def p_exr_t(p):
    "exr : t"
    p[0] = p[1]

def p_exr_opadd(p):
    "exr : exr '+' t"
    p[0] = p[1] + p[3] + "ADD\n"
    global endereço
    endereço -= 1

def p_exr_opasub(p):
    "exr : exr '-' t"
    p[0] = p[1] + p[3] + "SUB\n"
    global endereço
    endereço -= 1
```

```
def p_t_f(p):
    "t : f"
    p[0] = p[1]

def p_t_opmmul(p):
    "t : t '*' f"
    p[0] = p[1] + p[3] + "MUL\n"
    global endereço
    endereço -= 1

def p_t_opmdiv(p):
    "t : t '/' f"
    p[0] = p[1] + p[3] + "DIV\n"
    global endereço
    endereço -= 1

def p_f_int(p):
    "f : INT"
    p[0] = "PUSHI " + str(p[1]) + "\n"
    global endereço
    endereço += 1

def p_f_id_array(p):
    "f : ID '[' INT ']' "
    global endereço
    endereço += 1
    if p[3] > tabela[p[1]][2]:
        print("Out of index")
        exit()

    else:
        p[0] = "PUSHG " + str(tabela[p[1]][1] + p[3]) + "\n"

def p_f_id(p):
    "f : ID"
    global endereço
    endereço += 1
    if tabela[p[1]][0] == "int":
        p[0] = "PUSHG " + str(tabela[p[1]][1]) + "\n"
    else:
        print("Invalid operation")
        exit()

def p_f_BOOL(p):
    "f : bool"
    p[0] = p[1]

def p_bool_FALSE(p):
    "bool : FALSE"
    p[0] = "PUSHI " + str(p[1]) + "\n"
```

```
    global endereço
    endereço += 1

def p_bool_TRUE(p):
    "bool : TRUE"
    p[0] = "PUSHI " + str(p[1]) + "\n"
    global endereço
    endereço += 1

def p_f_simetrico(p):
    "f : '-' f"
    p[0] = "PUSHI " + "-1" + "\n" + str(p[2]) + "MUL\n"
    global endereço
    endereço += 1

def p_f(p):
    "f : '(' exl ')'"
    p[0] = p[2]

def p_error(p):
    parser.success = False
    print('Syntax error!!')
    exit()


parser = yacc.yacc()
parser.success = True

parser.parse(file.read())
file.close()

res = open("resultado.txt", "w")
res.write(resultado)
res.close()

print(resultado)
print(tabela)
```

2.2.1 Gramática

```
prog :  cod


cod :  cod inst ';'
    |


inst: PRINT '(' STRING ')'
    | PRINT '(' ID ')'
    | PRINT '(' ID '[' INT '] ')'
    | PRINT '(' ops ')'
    | WHILE '(' ops ')' '{' cod '}'
    | IF '(' ops ')' '{' cod '}' ELSE '{' cod '}'
    | IF '(' ops ')' '{' cod '}'
    | expatr


expatr : atr


atr : ID '=' ops
    | ID '[' INT ']' '=' ops
    | TIPO_STRING ID '=' INPUT
    | ID '=' INPUT
    | TIPO_INT ID '=' ops
    | TIPO_INT ID
    | TIPO_STRING ID '=' STRING
    | TIPO_STRING ID
    | TIPO_INT ID '[' INT ']' '=' '{' '}'
    | TIPO_INT ID '[' INT ']' '=' '{' val '}'
    | TIPO_INT ID '[' INT ']'


val : INT
    | '-' INT
    | val ',' INT
    | val ',' '-' INT


ops : exl


exl : tl
    | exl OR tl


tl : fl
   | tl AND fl
```

```
    | rl
    | NOT rl


rl : exr
    | rl EQUALS exr
    | rl DIFF exr
    | rl LE exr
    | rl GE exr
    | rl GT exr
    | rl LT exr


exr : t
     | exr '+' t
     | exr '-' t


t : f
  | t '*' f
  | t '/' f


f : INT
  | ID '[' INT ']'
  | ID
  | bool
  | '-' f
  | '(' exl ')'


bool : FALSE
     | TRUE
```

Capítulo 3

Exemplos

Apresentamos agora alguns exemplos de testes que realizamos para testar os limites do que criamos.

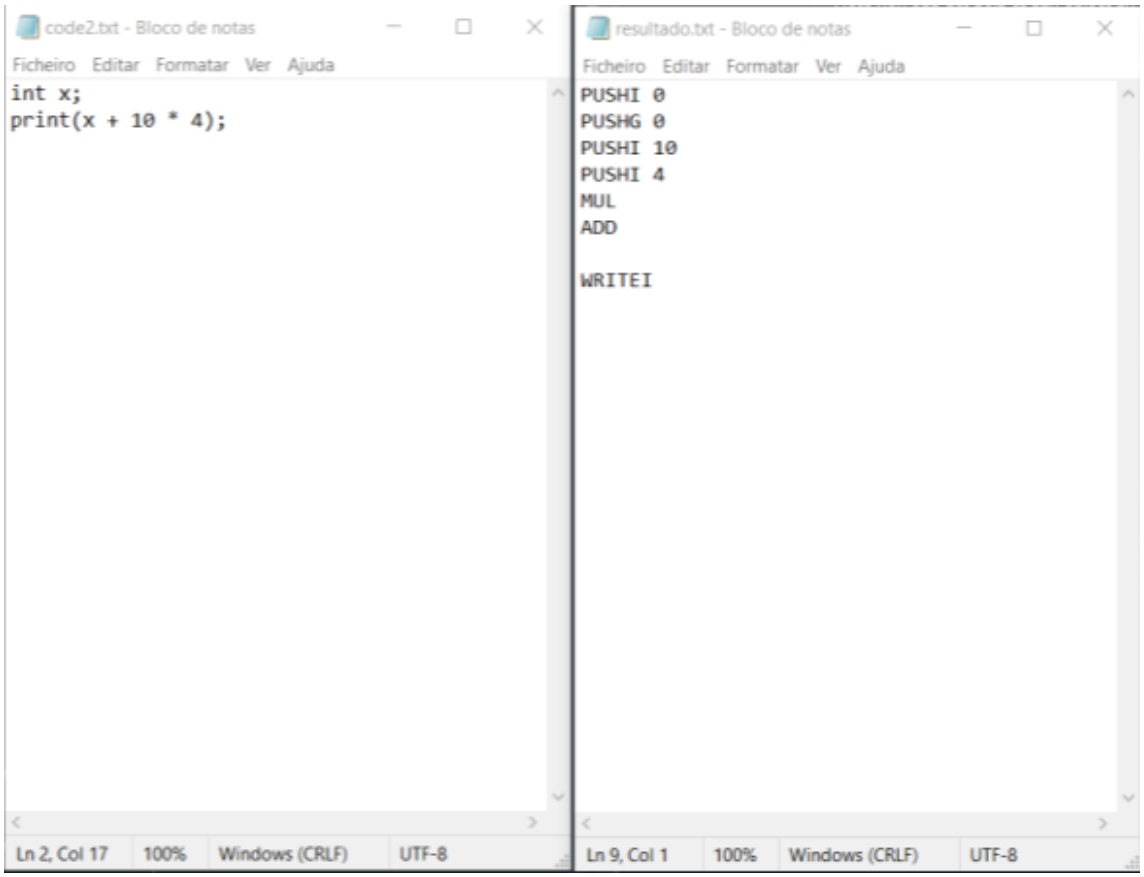

Figura 3.1: *Exemplo 1, print de uma equação simples*

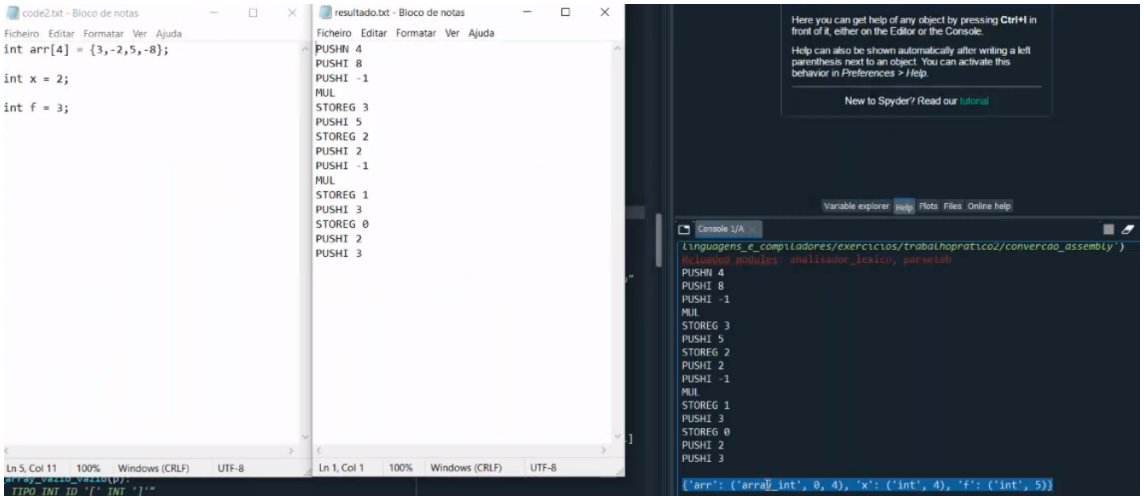

Figura 3.2: *Exemplo 2, declaração de array com 4 elementos e 2 dois inteiros*

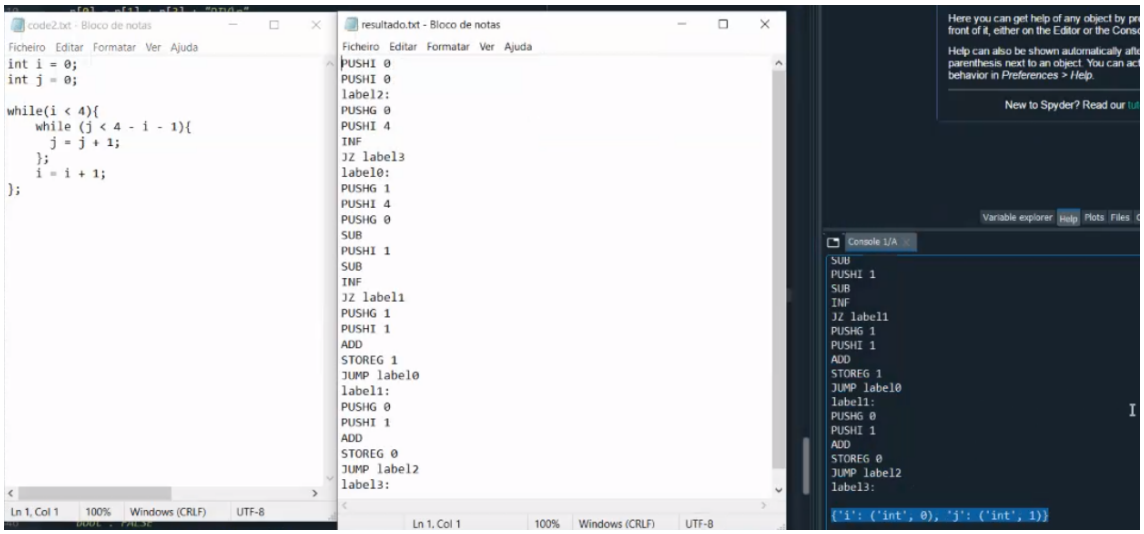

Figura 3.3: *Exemplo 3, Incrementador com 2 while*

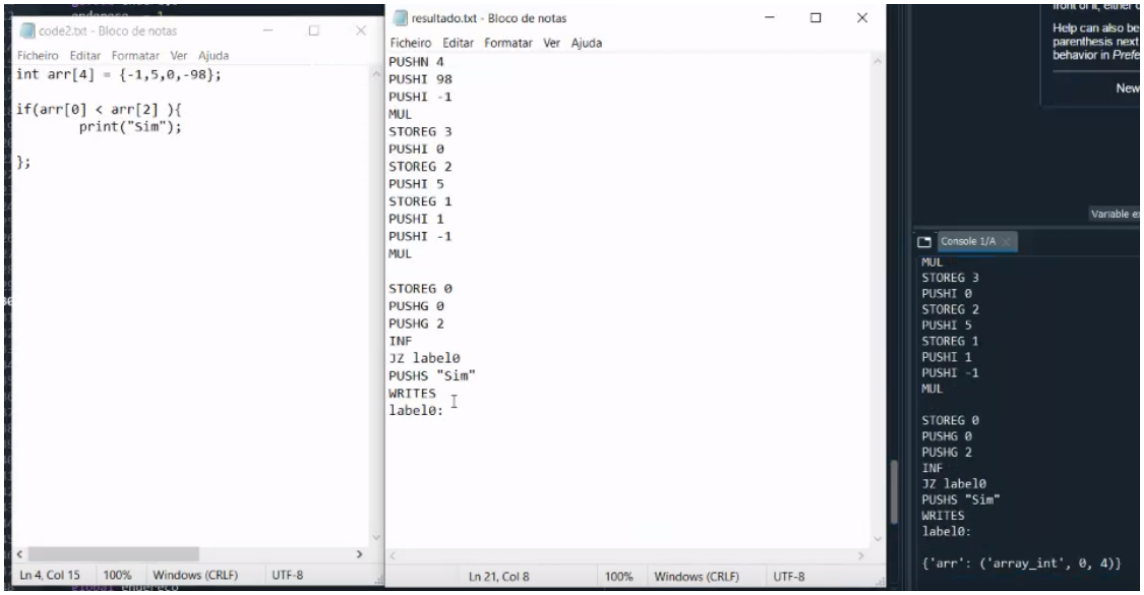

Figura 3.4: *Exemplo 4, If (sem Else) e Print*

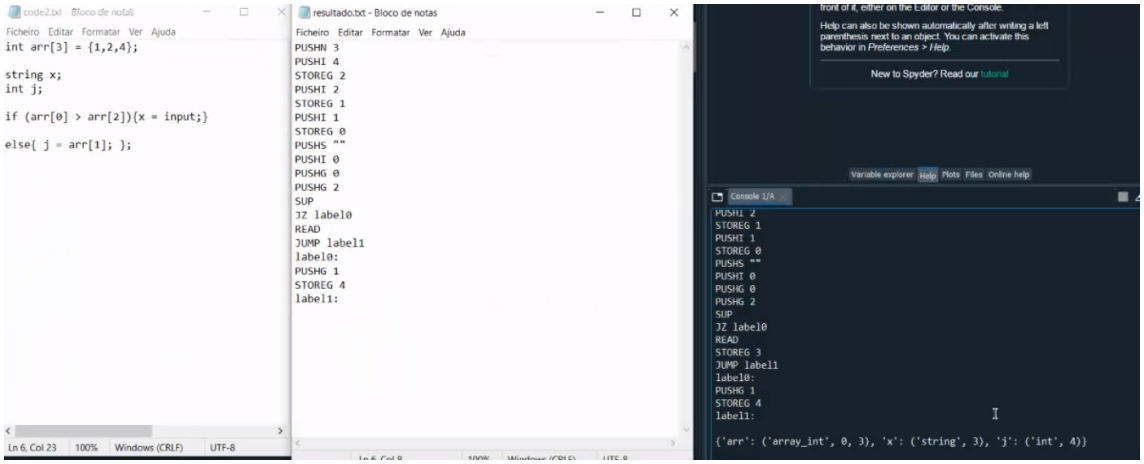

Figura 3.5: *Exemplo 5, If, Else*

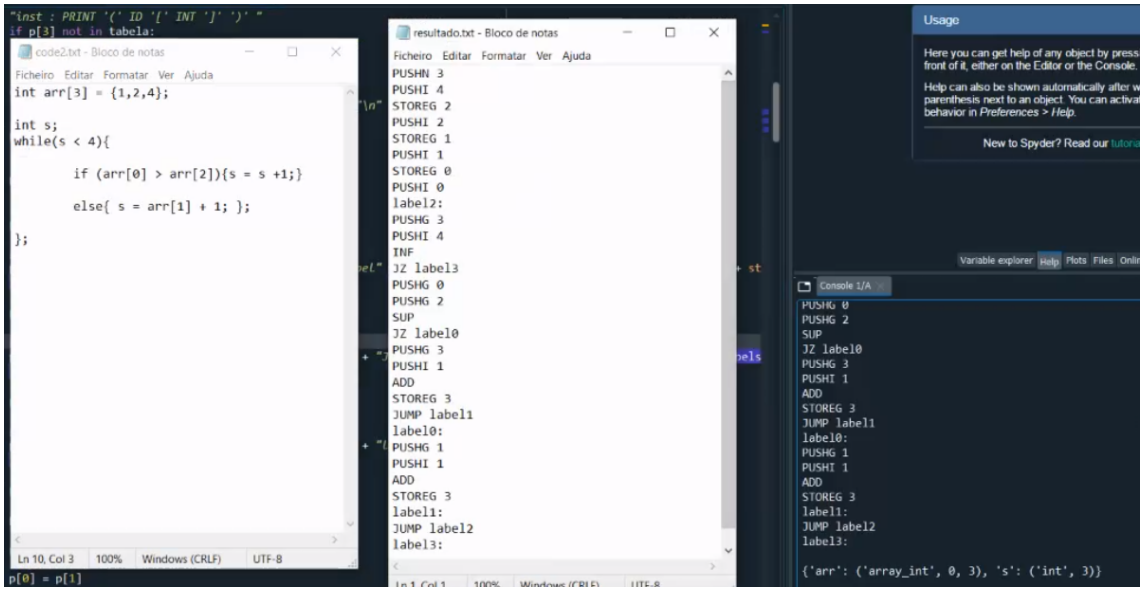

Figura 3.6: *Exemplo 6, Ciclo While com If e Else*

Capítulo 4

Conclusão e Reflexões

Com base daquilo que nos propusemos a fazer aliado aos objetivos que nos foram dados achamos que cumprimos o que nós queríamos. Conseguimos construir os 3 módulos de um compilador com sucesso e criar uma gramática funcional que depois transitamos para código assembly como nos foi pedido e como pretendíamos Consideramos a nossa linguagem bem feita e bastante completa para o que tencionávamos fazer. Temos algumas coisas que se tivermos em conta algumas linguagens de alto nível Como por exemplo o caso especifico abaixo, em que tentamos aceder a um array usando um variável em vez de um numero que seria possível em C. Pois tomamos a decisão de não guardar esses valores na tabela.

Ex: int arr[1,2,4]; int r = 2; print (arr[r]);

Este trabalho ajudou-nos não só a pôr em pratica o que aprendemos nas aulas, como também nos deu mais alguma experiência a trabalhar com a matéria. Concluímos assim o nosso relatório do segundo projeto da UC de processamento de Linguagens e Compiladores.

Capítulo 5

Bibliografia

Material Fornecido pelo Professor

Conhecimentos obtidos nas aulas

máquina de stack virtual.