

**PODER EXECUTIVO**

**MINISTÉRIO DA EDUCAÇÃO**

**UNIVERSIDADE FEDERAL DE RORAIMA**

**DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

**CONSTRUÇÃO DE COMPILADORES**

**Construção do CompilerExpressions**

**ALUNOS:**

**Pedro Aleph Gomes de Souza Vasconcelos – 2016.007150**

**Dezembro de 2020**

**Boa Vista/Roraima**

**Introdução**

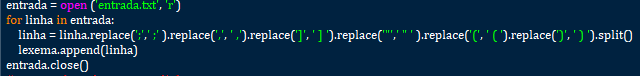
Este tem como objetivo relatar a construção feita detalhadamente do compilerExpress; para a disciplina construção de compiladores, semestre do (2020.2). É relatado a separação e rotulação; análise léxica, sintática e semântica; código intermediário; código em linguagem simbólica; e mensagens de erro na interface; junto ao funcionamento e comandos da linguagem fonte. O código a ser compilado foi baseado na linguagem C.

**Desenvolvimento**

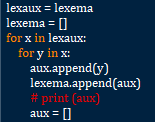
Para desenvolver foi utilizado a linguagem python por ter ferramentas conhecidas que são convenientes para a construção do programa. Foram usadas, basicamente, estruturas de dados de lista e dicionário, e comandos básicos de condição e laços de repetição, dentro de funções. Foram desenvolvidos as etapas, separação de lexemas, análise léxica, análise sintática e análise semântica até o momento.

**Separação de lexemas**

O programa lê o arquivo de entrada, coloca cada sentença adicionando em um vetor(lexema) usando append(), separado os quando há um espaço, através do split(), também considera sinais que podem estar juntos das variáveis e os separa com espaço, usando replace().



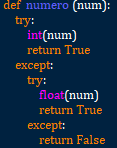
Depois o vetor lexema é reconstruído transformando-o em um matriz de forma a facilitar posteriormente a inserção de colunas para cada lexema do vetor.



Depois cada linha da matriz lexema é lida, levando em consideração que a primeira coluna é a qual será rotulada. As rotulações são separadas em $, num, ‘sinais’, PR e ID; adicionando o determinado a coluna seguinte do lexema sendo rotulado.

No caso de ser $, é quando o lexema é ‘;’.

No caso de num, isso quer dizer que o lexema é um número, para isso usa se função que retorna quando é verdadeiro para int ou float:



No caso de ser algum tipo de sinal, e verificado se está contido na lista de expressões:

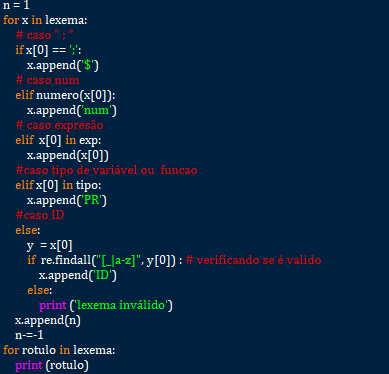


(deve conter todas as expressões possíveis a serem rotuladas, considerando que suas correções serão feitas em outra etapa)

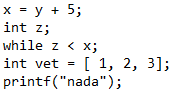
NO caso de PR, isto é para toda palavra reservada, é verificado se está contida na lista tipo:



E no último caso a ser verificado,o ID, estando no final por conveniência de rotulação, pelos anteriores serem prioridade, neste somente é um possível nome de variável ou função a ser usado, Adicionalmente é colocado um número na coluna seguinte ao rótulo, o COD de cada lexema. E no final é dado um print na matriz para verificar se os lexemas foram separados e rotulados corretamente.



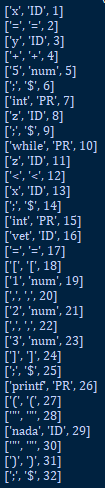
Para os casos de teste, foram usadas simples operações:



temos os casos de uso de vetor e comandos para o sistema com uso de strings, colocados para representar lexemas que requerem mais cuidado na correção em etapas posteriores.

a separação dos lexemas ocorreu da forma prevista dos casos testados.

lexema, rótulo e cod:



Nos casos de uso de sinais como parênteses, colchetes, e indicação de texto, observa se que é rotulado separadamente dos dados de onde são usados, e a correção será feita posteriormente com uma estrutura de dados de pilha, no qual empilha um inicio desse tipo de sinal, e desempilha o quando caso sinal de fim; o sinal para desempilhar dever ser sempre do mesmo tipo do sinal do no topo da pilha, e retorna corretamente caso no fim a pilha esteja vazia, retorna falso caso contrário.

**Análise léxica**

A gramática regular usada identifica se um nome de variável é válida, assim usa somente os rótulos com ID da etapa anterior, o primeiro estado(Si) analisa se o primeiro caractere é uma letra ou o sinal especial “\_”, após isso o próximo estado já é o estado final(Sf) onde há um self-loop que aceita tanto o mesmo tipo de caractere do primeiro estado, quanto se este for um número.

alfabeto{a, b}, tal que a = [\_] | [a-z] e b = [0-9]

L = a (a | b) \*

matriz:

|  |  |  |
| --- | --- | --- |
|  | Si | Sf |
| a | Sf | Sf |
| b |  | Sf |

código:

alfabeto = {'a' : 0, 'b': 1}

def verifica(i):

if re.findall("[\_|a-z]", i):

return 'a'

elif re.findall("[0-9]", i):

return 'b'

return False

def automatov (L):

'''matriz do automato de variáveis'''

M = [[ 1, 1], [ -1, 1]]

e = 0

for i in L:

l = verifica(i)

if l in alfabeto.keys():

e = M[alfabeto[l]][e]

else:

return False

'''Estado Final '''

if e == 1:

print ('reconhecido')

return True

else:

return False

**Análise sintática**

È usado ambos os analisadores , preditivo e de precedência fraca; para o de precedência fraca, é usado a gramática G adaptada para reconhecer expressões como ‘-’ e ‘/’, neste caso eles são aceitos como equivalentes de ‘+’ e ‘\*’; já o analisador preditivo é usada a gramática (A'').

A gramática (A'') faz um id (está como i) receber algo, que pode ser outro id (i = i), ou uma expressão complementando a gramática G (i = v). A gramática também aceita atribuições em sequência, neste caso, a expressão, se houver, deve estar somente na última atribuição (ex: i=i=v).

Gramática A’’

Vt = {i, =, v}, Vn = {A, B, I, J}, símbolo sentencial A

P = { P1: A -> iB,

P2: B -> =I,

P3: I->A,

P4: I -> vJ,

P5: B-> &,

P6: J-> &}

Tabela Sintática:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | i | = | v | $ |
| A | P1 |  |  |  |
| B |  | P2 |  | P5 |
| I | P3 |  | P4 |  |
| J |  |  |  | P6 |

Gramática no código:

gramatica\_A = {

'terminal' : {'i': 0, '=': 1, 'v': 2, '$': 3},

'simbolos' : {'A': 0, 'B': 1, 'I': 2, 'J': 3},

'producao' : {1: "Bi", 2: "I=", 3: "A", 4: "Jv", 5: "&", 6: "&"},

'sentencial' : 'A',

'M' : [[1, 0, 0, 0], [0, 2, 0, 5], [3, 0, 4, 0], [0, 0 , 0, 6]]

}

O analisador de procedência fraca foi implementado adaptando o código do analisador sintático disponibilizado. O Analisador de precedência fraca começa dado um while que percorre uma sentença até esta terminar, sendo desempilhando quando um caractere é analisado, e a verificação termina com sucesso caso a símbolo no topo da pilha seja o sentencial e o símbolo do topo da sentença seja $

if pilha[-1] == g['sentencial'] and s[-1] == '$':

return True

Após ter identificado os símbolos da sentença com sucesso, o analisador verifica qual produção será executada, caso seja D, o analisador irá para o próximo símbolo da sentença, mas caso R, ele primeiro verificar se os três primeiros símbolos do topo da pilha formam alguma produção a direita, e assim os colocará na mesma produção para que não haja ambiguidade a seguir, e depois é feita a análise do símbolo do topo transformando no símbolo da produção a direita.

while True:

l = g['simbolo'][pilha[-1]]

'''producao conforme a tabela'''

prod = M[l][c]

if prod == 'R':

if len(pilha) > 3:

if pilha[-1] in g['nterminal']:

r = pilha.copy()

q = ""

while len(q) < 3:

q = q + r.pop()

if q in g['producao'].keys():

r.append(q)

pilha = r

p = pilha.pop()

pilha.append(g['producao'][p])

else:

if prod == 'D':

break

else:

print("erro na producao da tabela sintatica")

return False

Para a árvore sintática do analisador de precedência fraca, é usando uma lista A que aramazena as producoes de direita e eesquerda pela qual analisador passou até chegar no sentencial. E assim percorre em loop até que a lista A esteja vazia, de modo que quando ele encontra um símbolo sentencial, este é movido para a lista de árvore, e segue para o próximo; quando um símbolo da lista da árvore e um símbolo da lista A se combinarem em alguma produção, o símbolo do topo da lista A é movido para árvore e seu símbolo anterior se torna nodo do novo top. Nesta fase é verificado primeiramente que os três primeiros símbolos no topo da árvore formam alguma produção, assim resolvendo a ambiguidade; e também trata caso seja uma produção com parentes.

def arvore\_sintatica(a):

a = a[::-1]

p = []

while a != []:

if a[-1] in terminais:

p.append(a.pop())

else:

q = ""

if len(p) > 2:

r = p.copy()

while len(q) < 3:

q = q + r[-1][0]

r.pop()

if (p[-1][0]==')')or(a[-1]==producao[p[-1][0]]):

nodo = []

nodo.append(a.pop())

if q in producao.keys():

t = []

t.append(p.pop())

t.append(p.pop())

t.append(p.pop())

nodo.append(t[::-1])

else:

nodo.append(p.pop())

p.append(nodo)

return p[0]

Depois de gerada, a árvore pode ser simplificada em uma função recursiva que retorna caso chegue num símbolo terminal; caso encontre parênteses a recursão é retornada para o símbolo do meio, assim eliminando o parênteses na árvore simplificada, e logo abaixo verifica o se o nodo tem mais de um símbolo, o tratando recebendo a recursão para cada nodo, caso seja o tenha; caso não, a recursão é recebida só pelo único nodo.

def simplifica(a):

if a[1] in terminais:

return a[1]

else:

if a[1][0] == '(':

return simplifica(a[1][1])

if len(a[1]) > 2:

t = a[1]

a = []

a.append(t[1])

a.append(simplifica(t[0]))

a.append(simplifica(t[2]))

else:

a = simplifica(a[1])

return a

Para a árvore sintática do analisador preditivo, da mesma forma, também utiliza uma lista A com cada produção gerada pela análise da sentença. Mas desta vez a árvore precisa ser criada de forma precedente, sendo assim utiliza se a estratégia de recursão, onde partindo do símbolo sentencial, cria se um nodo para cada símbolo, e o próximo símbolo da lista são tidos como filho de deste nodo, e é invocado a recursão destes, até que encontre um símbolo terminal, desta forma cada filho é retornado para o nodo anterior, assim formando a árvore. É usando uma pilha auxiliar para percorrer a lista, e desempilhar caso o símbolo no topo seja um terminal, assim os nodos folhas são conectados corretamente na árvore.

def arvore\_sintatica(a):

p = []

q = [0]

def add\_nodo(n):

nodo = []

nodo.append(n)

if n not in terminais:

q[0] = q[0] + 1

p.append(a[q[0]])

for i in p[-1]:

nodo.append(add\_nodo(i[0]))

else:

p.pop()

return nodo

A = []

p.append(a[q[0]])

A.append(add\_nodo(a[q[0]]))

print(A[0])

return A[0]

Já para a sua simplificação, é usado duas funções recursivas. Primeiramente, procura se remover toda subárvore que termine na folha com símbolo &, que no caso quando a folha é retornada, verifica se ela é contém este símbolo, e junto com o nodo pai, o símbolo é removido; e trata os casos com parênteses, no caso, retorna somente símbolo entre os parentes; e nodos com mais de um filho, eles são adicionado ao nodo pai como seus irmãos.

def simplifica(a):

print(a)

if len(a) == 2:

if a[1][0] == '&':

return '&'

else:

if a[1][0] == '(':

return simplifica(a[2])

elif a[1][0] in terminais:

a = a[::-1]

a.pop()

a = a[::-1]

a[1] = simplifica(a[1])

a[2] = simplifica(a[2])

if(a[2]) == '&':

a.pop()

else:

t = a.pop()

a.extend(t)

return a

Com as subárvores com os nodos folhas &, removidos, agora a árvore passa para uma função de simplificação semelhante a simplificação da árvore sintática do analisador de precedência fraca. Adaptando não precisar tratar parênteses, por já ter sido tratado, e também as propriedades da árvore sintática preditiva.

def ar\_sim(a):

if a[1][0] in terminais:

return a[1][0]

else:

if len(a) > 2:

t = a[::-1]

t.pop()

a = []

a.append(t[1][0])

a.append(ar\_sim(t[2]))

a.append(ar\_sim(t[0]))

else:

a = ar\_sim(a[1])

return a

**Análise semântica**

Nesta etapa , foi feita a verificação de tipos, levando em consideração que o código da entrada já esteja adaptado para ser percorrido por esta verificação. Para verificar, utiliza se estrutura de dados lista para identificar sinais básicos, e dicionário para verificação do tipo na variável.

# tipos de variaveis

tipo = {'int': int, 'float': float, 'double': float, 'char': str, 'void': None}

sinais = ['+', '-', '\*', '/', '%', '=']

(Também é usando um dicionário que armazenará as variáveis declaradas, além da lista que conterá cada elemento de cada linha do código)

A seguir a lista que contém cada trecho do código da entrada é percorrida, para identificar as declarações, para então serem adicionadas ao dicionário de declarações; estas são identificadas a partir do reconhecimento de um elemento em tipo, obtendo assim a variável e seu tipo, é armazenando também a linha do código onde ela foi declarada; também é adicionado ao dicionário de declarações, funções e suas variáveis invocadas; a forma de como elas são identificadas é tratado posteriormente, para as funções é adicionado a posição onde termina uma função , quando o símbolo ‘}’ é identificado; estando esta posição, igual a posição de início quando foi identificado inicialmente, isto para toda declaração; é usado uma pilha auxiliar para identificar a função.

Após uma declaração de variável ser reconhecida, ela é removida da lista dos trechos; esta lista posteriormente é usada para analisar expressões e usa o dicionário de declarações para reconhecer as variáveis. Nesta parte também é tratado os casos de declaração com atribuição na mesma linha, neste caso usa se uma pilha auxiliar que adiciona na lista a atribuição após a declaração ser removida, na mesma posição da lista.

for t in trechos:

if t[0] in tipo.keys():

# declaracao com atribuicao

t[0] = tipo[t[0]]

dt = []

declaracoes[t[1]] = [t[0], t[-1], t[-1]]

if t[2] == '=':

dt = t

dt.pop(0)

elif len(t) > 4:

isFunc.append(t[1])

n = 3

while 1:

if (t[n] in tipo) and (is\_var(t[n+1])):

t[n] = tipo[t[n]]

declaracoes[t[n+1]] = [t[n], t[-1], t[-1]]

n+=2

elif (t[n] == ','):

n+=1

else:

break

if dt != []:

t = dt

else:

t.clear()

elif len(t) == 2:

if t[0] == '}':

declaracoes[isFunc[-1]][2] = t[1]

isFunc.pop()

t.clear()

A seguir é tratado as declarações obtidas, de forma a determinar até onde uma declaração é válida no código; para isso, variáveis com a posição inicial declarada antes, e que já possui a posição final diferente da inicial, no caso as funções, são usadas para reconhecer as próximas se as declarações, se estas estão dentro ou fora; são consideradas dentro caso a posição de declaração for maior que a posição do início da função mas for menor que a posição de termino da função. O dicionário é percorrido usando uma lista auxiliar que recebe as chaves do dicionário de forma inversa; isso é necessário para que não e haja ambiguidade das posições no caso de função dentro de outra função, sendo o dicionário percorrido de forma inversa, a prioridade da declaração será da função interna, a função maior não identifica por já ter sua posição final alterada.

# tratando declaracoes nas funcoes

r = list(declaracoes.keys())

b = r[::-1]

for f in b:

i = declaracoes[f]

if (i[1] != i[2]):

for g in b:

j = declaracoes[g]

if (j[1] == j[2]):

if (i[1] <= j[1]):

if i[2] > j[2] :

declaracoes[g][2] = i[2]

Na próxima etapa, é analisado se nas expressões dos trechos, cada variável foi declarada previamente; para tanto, cada linha dos trechos é percorrida, e reconhece se um elemento pode ser uma variável com função auxiliar; caso seja, depois é verificado se está em alguma chave do dicionário de declarações, após isso suas posições serão analisadas; no caso se a posição inicial da declaração for menor que a posição da linha do elemento , então ela foi declarada previamente; e também analisa o caso dela estar sendo usada dentro da função, se a posição da função for maior que a posição onde a variável é usada, caso contrário, erro da declaração ser usada fora do escopo de onde foi declarada.

# analisando variaveis nas expressoes

for t in trechos:

for d in t:

if is\_var(d):

if d in declaracoes.keys():

i = declaracoes[d][1]

f = declaracoes[d][2]

if (t[-1] >= i):

if (i == f) or (t[-1] < f):

print("variavel ", d, " foi declarada")

else:

print("variavel ", d, " nao foi declarada previamente no escopo")

else:

print("variavel ", d, " nao foi declarada previamente")

break

else:

print("variavel ", d, " nao foi declarada")

break

Analisa se aqui se os sinais nas expressões são usados corretamente, para isso, quando identificado na expressão percorrida, o sinal é analisado a partir dos elementos esquerda e direita do sinal; para isso é usando uma variável auxiliar que conta os elementos, logo após isso os elementos são reconhecidos, retornando o tipo da variável ou tipo de elemento através de uma função auxiliar(convertendo\_) e recebem seu valor de tipo, e assim, a partir deste momento, só será usada os tipos dos elementos para cada sinal:

for e in trechos:

n = 0

for v in e:

if v in sinais:

x = e[n-1]

y = e[n+1]

if x in declaracoes.keys():

x = declaracoes[x][0]

else:

x = convertendo\_(x)

if y in declaracoes.keys():

y = declaracoes[y][0]

else:

y = convertendo\_(y)

Para o sinal ‘%’ só são aceitos elementos do tipo int:

if v == '%':

if (x is int) and (y is int):

print("operacao de tipos valida para '%'")

else:

print("operacao invalida para '%' ", x, "e", y)

Para o sinal ‘=’, o elemento a esquerda precisa ser igual direta, se este último for int ou char; no caso do elemento da direita ser tipo char, este é convertido, levando em conta que será recebido o seu valor numérico; e o mesmo é feito de o elemento a direita for int e o da esquerda for float:

elif v == '=':

if (x is int) and (y is not int):

print("atribuicao invalida, ", x, "e", y)

if (x is str) and (y is not str):

print("atribuicao invalida, ", x, "e", y)

if x is float:

y = declaracoes[x][0]

Para os demais sinais, levando em conta a lista de sinais, estes não requerem um tratamento especial, somente trata os elementos do tipo char, se necessário:

else:

if (number(x) and (y is str)):

y = declaracoes[x][0]

elif (number(y) and (x is str)):

x = declaracoes[y][0]

Ainda no mesmo percorrer dos sinais, caso o elemento não seja um sinal, mas seja um início de parênteses ‘(‘, então quer dizer que o elemento anterior é uma função, e verifica se foi usada corretamente com seus parâmetros. Para reconhecer se os parâmetros estão corretos, são usadas duas lista, a primeira pega os valores dentro dos parênteses; adicionando o tipo do elemento ou da variável :

if v == '(':

x = e[n-1]

v1 = []

n += 1

while e[n] != ')':

if (e[n] in declaracoes.keys()):

v1.append(declaracoes[e[n]][0])

elif(e[n] != ','):

v1.append(convertendo\_(e[n]))

n += 1

Para a segunda lista, é adicionado o tipo da variável onde a posição inicial é igual a posição da função, não está sendo a própria função, assim identificado como seu parâmetro:

v2 = []

for d in declaracoes:

z = declaracoes[d][1]

if d != x:

if z == declaracoes[x][1]:

v2.append(declaracoes[x][0])

Agora a duas lista são comparadas e a função foi usada corretamente quando são iguais, já que elas têm o mesmo número de parâmetros e tipo de cada:

if v1 == v2:

print("variaveis em ", x, " estao corretas",v1,v2)

else:

print("variaveis em ", x, " incorretas",v1, v2)

n += 1

**Código Intermediário**

O código intermediário começa separando cada sentença encontrada, separando as partes de atribuição e de operação, visando aplicar a gramática de atribuições e de expressões e cada uma cada. Quando há uma sentença com atribuição e expressão, na atribuição o último símbolo antes de um operador é substituído por um ‘<’(considerando que deve ser um símbolo não aceito em variáveis para não haver ambiguidade no uso), e a expressão é analisada, dessa forma, a expressão anterior a uma atribuição com ‘<’ na lista, indica que esta expressão é atribuída a próxima a atribuição. Depois das sentenças serem separadas, elas são analisadas e cria se outra lista onde as variáveis são substituídas, para serem aceitas quando forem analisadas por cada gramática, e assim cada sentença vai para sua respectiva gramática.

for a in V:

if a[0] == 'i':

if (asp.automatoM(a, asp.gramatica\_A)):

a = []

a = asp.arv

a = arsp.arvore\_sintatica(a)

a = arsp.ar\_simp(a)

A.append(a)

else:

ms.showerror('ERRO!', 'sentenca invalida na gramatica de atribuicoes')

return False

elif (a[0] == 'v') or (a[0] == '('):

if(apf.automatoM(a, apf.gramatica\_G)):

a = []

a = apf.arv

a = arpf.arvore\_sintatica(a)

a = arpf.simplifica(a)

A.append(a)

else:

ms.showerror('ERRO!', 'sentenca invalida na gramatica de expressoes')

return False

Após ocorrer tudo certo na análise das sentenças, seus valores originais são colocados novamente, levando em consideração as propriedades da árvore, e retirando parênteses, caso haja, já que na árvore simplificada,eles não são mais utilizados.

A optimização usada, é que diminui os trechos do código retirado e substitui partes iguais. Para isso, leva se em consideração que cada optimização não deve comprometer o código no geral, então as retiradas de trechos repetidos só é feita dentro de uma mesma árvore. Nesta etapa cada árvore é percorrida criando uma variáveis temporárias que recebem cada nodo desta; e assim verifica se cada uma destas temporárias; as que atribuem a mesma expressão são retiradas, e substituídas por uma só da mesma.

def is\_equal(cod\_in):

for c in cod\_in:

#print (c)

igual = c

for cod in cod\_in:

if cod != igual:

if cod[1] == igual[1]:

for con in cod\_in:

if con[1][0] == cod[0]:

con[1][0] = igual[0]

elif con[1][2] == cod[0]:

con[1][2] = igual[0]

cod\_in.pop(cod\_in.index(cod))

Após isso, a lista é tratada para então ser inserida no arquivo do código intermediário.

**Código em Linguagem Simbólica**

Para o código simbólica, usa se a linguagem assembly x86, uma máquina de dois endereços; será explicado o seu uso para operações aritméticas; têm se os registradores especiais para essas operações:

AX , BX, CX e DX; cada um pode ter uma específica funcionalidade para se auxiliarem, e será apresentada no seu uso; cada um possui 16 bits de dados, e podem se dividir em duas partes (AX por exemplo por se dividir em AL e AH, 8 bits cada), mas nesta utilização, nos restritaremos ao uso geral.

AX, pode ser usado como porta de entrada é saída, para para operações aritméticas, e este será o seu uso.

BX, pode ser usado como registrador auxiliar para operações aritméticas em AX, como multiplicação e divisão, e assim será utilizado. (EX.: a operação’ mul BX ‘, irá dividir o valor em AX pelo dividendo em BX.

CX, a sua principal função é como contador em laços de repetição e operações de deslocamento, não terá uso em particular para estas operações básicas.

DX, também pode ser usado em operações aritméticas, guardando por exemplo (e será sua utilização neste), o resto de uma operação de divisão, enquanto o valor dividido estará em AX.

Além destes registradores gerais, utilizamos as declarações de variáveis da linguagem x86, que funciona da seguinte forma:

{nome da variável} {tamanho da word} {valor}

para seu uso geral utilizaremos o tamanho da word com DW (double word, our seja com 16 bits), e assim fica, no exemplo com nome da variável ‘V’ com valor ‘0’: V DW 0

Além das operações básicas: ADD,SUB, MUL e DIV, para utilizamos a instrução MOV para auxiliar o uso das variáveis com os registradores gerais, de forma que os registradores primeiro são carregados com os valores das variáveis; os registradores executam a operação, e carregam o resultado nas respectivas variáveis. As instruções funcionam da seguinte forma:

ADD e SUB, {instrução} {registrador}, {registrador/ variável/valor}

MUL e DIV, {instrução} {registrador}

MOV, {instrução} {registrador/variável}, {registrador/valor}

Vale ressaltar que para atribuições de outra variável, é preciso, primeiro, que a segunda variável seja carregada para um registrador, e depois carregar esta a primeira variável.

A estrutura do arquivo com a linguagem simbólica, é feita de forma para, se esta for compilada, então, funcionar minimamente. Começando em ‘org 100h’, um padrão para o funcionamento em geral, e terminar em ‘ret’, que delimitada o fim do código.

Primeiro é identificado as variáveis usadas no código intermediário para serem declaradas na código simbólico.

variaveis = []

for line in codigo:

n = 0

while n < len(line):

if line[n] not in variaveis:

variaveis.append(line[n])

n += 1

Depois cria-se a parte no código simbólico onde as operações são realizadas, para cada linha do código simbólico; levando em consideração, o uso das instruções já explicadas.

def operation(line):

tamanho = len(line)

m = ""

if tamanho == 3:

if line[2] in variaveis:

m = '\nmov ax,' + line[2] + '\n'

m += 'mov ' + line[0] + ', ax\n'

else:

m = '\nmov ' + line[0] + ', '+ line[2] + '\n'

else:

if line[3] in mul.keys():

m = '\nmov ax, ' + line[2] + '\n'

m += 'mov bx, ' + line[4] + '\n'

m += operadores[line[3]] + ' bx\n'

m += 'mov ' + line[0] + ', '+ mul[line[3]] + '\n'

else:

m = '\nmov ax, ' + line[2] + '\n'

m += operadores[line[3]] + ' ax, ' + line[4] + '\n'

m += 'mov ' + line[0] + ', ax\n'

return m

E assim as declarações e operações são inseridas no código símbolo de forma a funcionar minimamente.

saida = open('codigo\_simbolico.txt', 'w')

saida.write('org 100h\n\n')

saida.write('; variaveis\n\n')

for var in variaveis:

saida.write(var + ' DW ' + '0\n')

saida.write('\n; expressoes\n')

for line in codigo:

saida.write(operation(line))

saida.write('\nret')

saida.close()

**Mensagens de erro na interface gráfica**

Para emitir as mensagens de erro foi utilizado o import tkinter.messagebox as ms,que mensagens no sistema; e usando especificamente, o método ms.showerror(title, message), para emitir mensagens de erro.

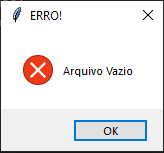
Como demonstração de uso, será usado a mensagem de erro, de quando tenta se salvar um arquivo com nada escrito dentro:

data = texto.get('1.0', 'end-1c')

if len(data) == 0:

ms.showerror('ERRO!', 'Arquivo Vazio')

A seguinte aba será aberta caso o este erro seja detectado:



E assim é feito para cada funcionalidade do código fonte (retornando False em seguida).

Na análise léxica:

Erro na transição de estado:

ms.showerror('ERRO!', 'variavel nao declarada corretamente\n' + L)

Erro de caractere não aceito pela linguagem:

ms.showerror('ERRO!', 'simbolo nao aceito\n'+ i + ' em ' + L)

Erro quando de não chega ao estado final apos toda a sentenca ser percorrida:

ms.showerror('ERRO!', 'variavel nao reconhecida: ' + L)

Na análise sintática:

Erro em cada gramática:

ms.showerror('ERRO!', 'sentenca invalida na gramatica de atribuicoes')

ms.showerror('ERRO!', 'sentenca invalida na gramatica de expressoes')

Na análise semântica:

Para cada tipo de erro em declaração de variável:

ms.showerror('ERRO!',"variavel "+ str(d) + " nao foi declarada previamente no escopo")

ms.showerror('ERRO!',"variavel " + str(d) + " nao foi declarada previamente")

ms.showerror('ERRO!',"variavel " + str(d) + " nao foi declarada")

Erro de tipo das variáveis no uso em operações:

ms.showerror('ERRO!',"operacao invalida para '%' " + str(x) + "e" + str(y))

ms.showerror('ERRO!',"atribuicao invalida, " + str(x) + "e" + str(y))

Erro de quando os parâmetros na função estão incorretos, em tipo e/ou em número.

ms.showerror('ERRO!', "variaveis em "+ x + " incorretas" + str(v1) + str(v2))

**Considerações finais**

Este Trabalho está sendo foi boa experiência e vou continuar desenvolvê-lo mesmo após apresentá-lo na disciplina. Mesmo trabalhando em desenvolvê-lo por muitos dias, percebi que não foi tempo suficiente, pois mesmo tendo tempo, em cada parte feita é necessário um intervalo, para então prosseguir para a próxima etapa, este é um trabalho que merece um foco maior em detrimento com outras disciplinas. A maior parte dos problemas em fazer, é em tratar cada caso de entrada do código; é a parte que mais levou tempo para mim, e depois decidi me focar em fazer as demais partes para fazê-lo funcionar minimamente; mas por isso mesmo eu estou motivado a depois continuar desenvolvendo para que funcione para todo caso de entrada aceito possível.

**Referências**

[http://excript.com/python/iterando-listas-em-python.htm](http://excript.com/python/iterando-listas-em-python.html)l

<https://docs.python.org/pt-br/3.8/howto/regex.html>

<https://www.w3schools.com/python/python_regex.asp>

<https://www.geeksforgeeks.org/python-remove-empty-list-from-list/>

<https://www.w3schools.com/python/python_ref_list.asp>

<https://www.geeksforgeeks.org/python-tkinter-scrolledtext-widget/>

<https://dev.to/eshleron/how-to-convert-py-to-exe-step-by-step-guide-3cfi>

<https://pythonbasics.org/tkinter-messagebox/>

<https://www.w3schools.com/python/python_ref_string.asp>

<https://www.w3schools.com/python/python_ref_list.asp>

<https://www.w3schools.com/python/python_ref_dictionary.asp>

<https://knowpapa.com/text-editor/>