

Roteiro 10 - Lógica da Computação

Aluna: Livia Sayuri Makuta.

Lógica principal da memória

Antes de começar explicando qual a lógica utilizada, primeiro vamos resgatar tudo o que foi desenvolvido na disciplina até então. Na versão do roteiro v3.0, foi quando implementamos funções e chamada de funções no compilador de Julia. Nesse cenário, havíamos separado as tabelas de símbolos em dois tipos: a FuncTable e as Symbol Tables gerais. A primeira seria a tabela que guardaria todos os nós das funções criadas no código e a segunda está relacionada a todas as Symbol Tables criadas logo após a chamada de uma função, além da Symbol Table criada e passada como argumento para o *evaluate* da raiz no método *run* do Parser. Além de que, como também já implementado anteriormente, temos as funcionalidades de loop, condição, declaração de variável, input, print, operações lógicas (or e and) e operações de soma, subtração, multiplicação, divisão, comparação, igualdade. Com isso, parti da versão v3.0 do roteiro e a complementei com a geração de código feita anteriormente para a versão v2.4, a fim de implementar a geração de código agora também para as funções.

O grande desafio então é gerar código considerando as várias Symbol Tables de são criadas durante cada chamada de função. Para isso teremos que mexer com a pilha (stack) durante a execução do programa. Dessa forma, temos que no assembly o EBP (Extended Base Pointer) e ESP (Extended Stack Pointer) que são registradores especiais utilizados para gerenciar a pilha como um “ponteiro de base” e um “ponteiro para o topo da pilha”, respectivamente. Primeiro será explicado como foi feito o gerenciamento da memória para as variáveis internas da função, e depois como isso foi feito também para as variáveis passadas como entradas da função.

Quando uma função for chamada, o valor atual do EBP é guardado, e isso é feito para que em qualquer ponto do programa seja possível restaurar o valor inicial do EBP

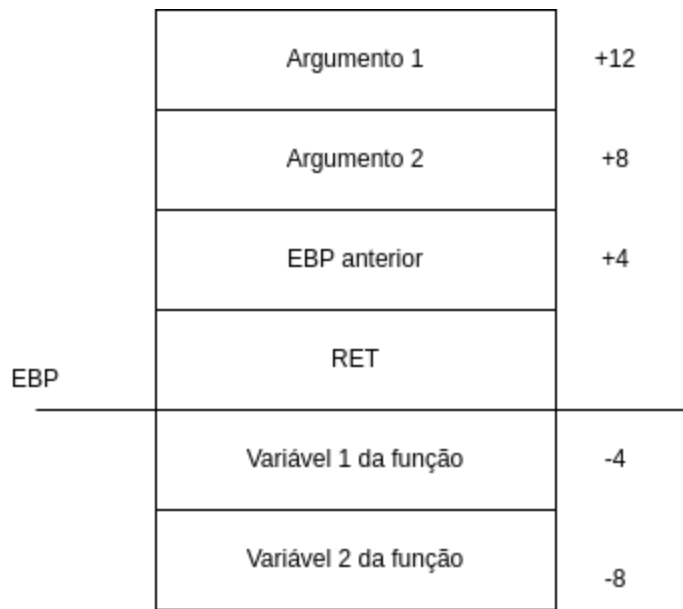
e ter acesso ao estado original da pilha, o que garante que os dados e variáveis sejam acessados corretamente em diferentes níveis de escopo. Por sua vez, caso a função seja chamada recursivamente, é necessário lidar com o contexto da função atual e criar um novo contexto para a chamada recursiva, e isso implica em criar uma nova tabela de símbolos e um novo espaço na pilha para armazenar as variáveis e parâmetros da função recursiva. Nesse contexto, o valor inicial guardado não é mais o valor original do EBP, mas sim o valor do EBP da chamada recursiva anterior. Isso permite que a função recursiva tenha acesso aos valores da tabela de símbolos e às variáveis da chamada anterior. Ao retornar da chamada recursiva, o valor do EBP é restaurado para o valor anterior, permitindo a continuação da execução no contexto anterior da função.

Agora, para passar os argumentos da função, posso acessá-los diretamente a partir dos offsets positivos em relação ao registrador EBP. Como os argumentos estão organizados em ordem decrescente na pilha, o último argumento estará no offset +8 em relação ao EBP, o penúltimo argumento no offset +12 e assim por diante.

Ao chamar a função, posso utilizar a instrução PUSH para empilhar os valores dos argumentos na pilha, de acordo com a ordem em que eles são passados. Dessa forma, os valores dos argumentos serão salvos na memória e poderão ser acessados posteriormente dentro do bloco de instruções da função. Além disso, durante a passagem dos argumentos, posso também atualizar a tabela de símbolos do compilador, armazenando os valores dos argumentos nos offsets correspondentes. Isso permite que a tabela de símbolos tenha acesso aos valores dos argumentos e que possam ser utilizados dentro da função. Essa estratégia de salvar os argumentos na pilha e atualizar a tabela de símbolos com os valores correspondentes facilita o acesso e manipulação dos argumentos dentro da função, tornando-os disponíveis para serem utilizados em expressões, atribuições e outras operações.

Por fim, sempre que for retornado da função com a instrução RET a pilha deve ser limpa, isto é, o EBP precisa estar com o valor antigo para que seja dado POP das variáveis que foram adicionadas como argumentos. Além disso, é importante salientar que o número de PUSH de argumentos e o número de POP deve ser o mesmo depois da chamada da função.

Basicamente a memória ficou organizada da seguinte maneira:



Construção do Assembly

Com a lógica explicada, agora vamos entender como o Assembly em si foi construído. Desde a entrega da versão v2.4 já tínhamos o cabeçalho (header), o código gerado pelo compilador e o rodapé (footer). O que muda para essa versão é que agora teremos um espaço especial antes do *start do header* para salvar as funções. Assim a estrutura do assembly fica sendo:

- Cabeçalho;
- Funções;
- Start;
- Código geral do programa;
- Rodapé.

Para concatenar tudo isso sem precisar modificar outros nós além dos nós de declaração de função (FuncDec), chamada de função (FuncCall), identifier e assign utilizei a ideia que o aluno Francisco Pinheiro Janela me sugeriu: criar um arquivo de stash temporário ("main_temp-stash.asm") que guarda todo o código gerado no documento principal ("main_tem.asm") até então, para que depois quando for dado o evaluate do bloco da função, o texto dela possa ser preenchido no arquivo "functions.asm" e depois possa ser passado para o arquivo principal junto com o código

salvo no stash. Isso basicamente faz a organização para que todas as funções possam ser colocadas logo após o cabeçalho e antes do start e para que o código geral do programa não seja perdido até aquele ponto.

Declaração das Funções

Para formalizar, temos que as funções possuem o seguinte formato no assembly do compilador:

```
nome_funcao:
    PUSH EBP
    MOV EBP, ESP
    ; Bloco de execução da função
    ; Mover o resultado do retorno para EBX
    MOV ESP, EBP
    POP EBP
    RET
```

Todas as funções possuem um label, além de como citado anteriormente, terem uma tabela de símbolos temporária. E para garantir que os argumentos das funções sejam alocados corretamente nela, foi criada uma função para ajustar os offsets desses argumentos, garantindo que seus valores fossem positivos e adequados para o acesso às variáveis dentro do corpo da função (a ideia dessa função também foi indicada pelo aluno Francisco Pinheiro Janela).

Por fim, para chamar a função, basta realizar o evaluate dos filhos passados para o nó de chamada de função e empilhá-los utilizando o comando PUSH. Como o resultado sempre estará em EBX, é necessário dar PUSH nesse registrador logo após o evaluate de cada filho.

Testes para a versão 3.1

Por fim, a fim de testar o compilador com geração de código, montei 2 programas simples em Julia que utilizam funções não recursivas e funções recursivas. Lembrando que para testar cada código gerado em “.asm” foram utilizados os comandos passados no roteiro 8 que fazem um link e transformam o código em executável no Linux/x86:

```
nasm -f elf32 -F dwarf -g program.asm
ld -m elf_i386 -o program program.o
```

Exemplo 1- Funções e chamada de funções simples

A função a ser testada no primeiro exemplo foi:

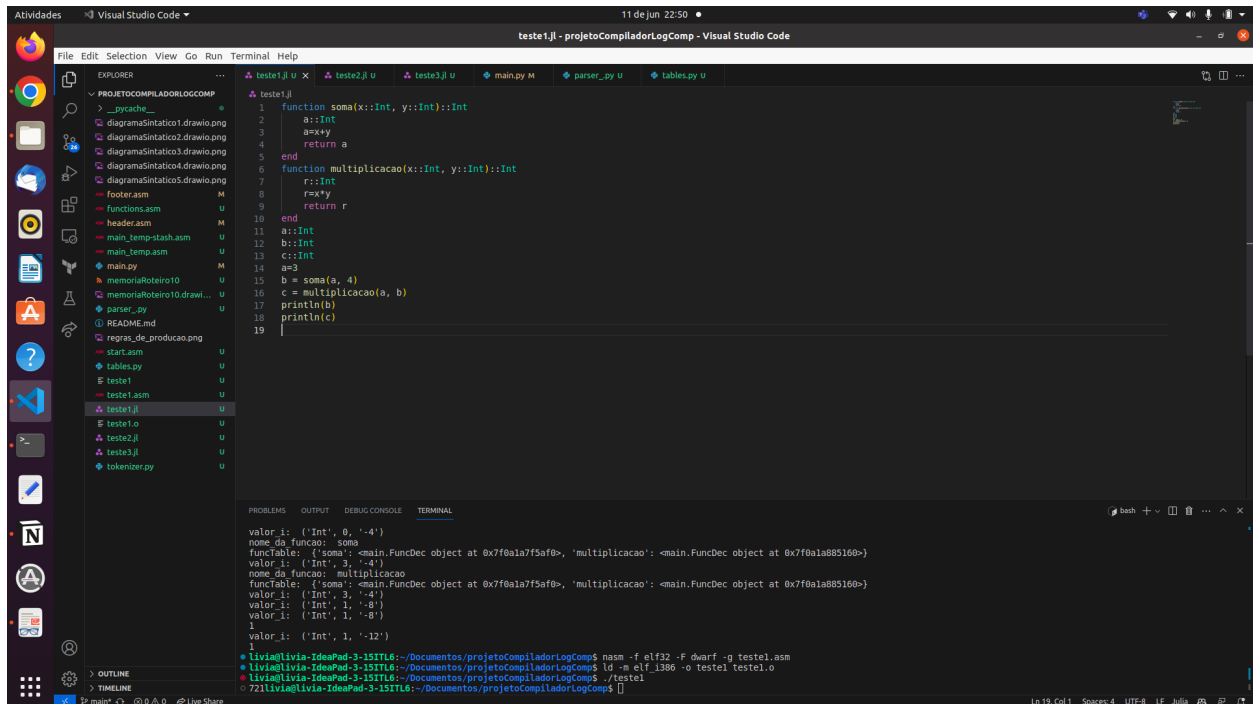
```
function soma(x::Int, y::Int)::Int
    a::Int
    a=x+y
    return a
end
function multiplicacao(x::Int, y::Int)::Int
    r::Int
    r=x*y
    return r
end
a::Int
b::Int
c::Int
a=3
b = soma(a, 4)
c = multiplicacao(a, b)
println(b)
println(c)
```

Resultado esperado:

```
7
21
```

Resultado obtido (721):

Observação: saiu tudo concatenado, pois os prints estão sem '\n'.



The screenshot shows the Visual Studio Code interface. The Explorer panel on the left lists files in a project named 'projetoCompiladorLogComp'. The main editor displays a Python file 'teste1.jl' with the following code:

```
1 function soma(x::Int, y::Int)::Int
2     a::Int
3     a=x+y
4     return a
5 end
6 function multiplicacao(x::Int, y::Int)::Int
7     r::Int
8     r=x*y
9     return r
10 end
11 a::Int
12 b::Int
13 c::Int
14 a=3
15 b = soma(a, 4)
16 c = multiplicacao(a, b)
17 println(b)
18 println(c)
19
```

The terminal at the bottom shows the output of the program:

```
valor i: ('Int', 0, '4')
nome da funcao: soma
funcTable: {'soma': <main.FuncDec object at 0x7f0a1a7f5af0>, 'multiplicacao': <main.FuncDec object at 0x7f0a1a885160>}
valor i: ('Int', 3, '-4')
nome da funcao: multiplicacao
funcTable: {'soma': <main.FuncDec object at 0x7f0a1a7f5af0>, 'multiplicacao': <main.FuncDec object at 0x7f0a1a885160>}
valor i: ('Int', 3, '-4')
valor i: ('Int', 1, '-8')
valor i: ('Int', 1, '-8')
1
valor i: ('Int', 1, '-12')
```

O assembly gerado pode ser visto abaixo:

```
; constantes
SYS_EXIT equ 1
SYS_READ equ 3
SYS_WRITE equ 4
STDIN equ 0
STDOUT equ 1
True equ 1
False equ 0

segment .data

segment .bss ; variaveis
    res RESB 1

section .text
    global _start

print: ; subrotina print

    PUSH EBP ; guarda o base pointer
    MOV EBP, ESP ; estabelece um novo base pointer
```

```

MOV EAX, [EBP+8] ; 1 argumento antes do RET e EBP
XOR ESI, ESI

print_dec: ; empilha todos os digitos
MOV EDX, 0
MOV EBX, 0x000A
DIV EBX
ADD EDX, '0'
PUSH EDX
INC ESI ; contador de digitos
CMP EAX, 0
JZ print_next ; quando acabar pula
JMP print_dec

print_next:
CMP ESI, 0
JZ print_exit ; quando acabar de imprimir
DEC ESI

MOV EAX, SYS_WRITE
MOV EBX, STDOUT

POP ECX
MOV [res], ECX
MOV ECX, res

MOV EDX, 1
INT 0x80
JMP print_next

print_exit:
POP EBP
RET

; subrotinas if/while
binop_je:
JE binop_true
JMP binop_false

binop_jg:
JG binop_true
JMP binop_false

binop_jl:
JL binop_true
JMP binop_false

binop_false:
MOV EBX, False
JMP binop_exit

binop_true:
MOV EBX, True

binop_exit:

```

```

    RET

soma:
PUSH EBP
MOV EBP, ESP
PUSH DWORD 0
MOV EBX, [EBP+12]
PUSH EBX
MOV EBX, [EBP+8]
POP EAX
ADD EAX, EBX
MOV EBX, EAX
MOV [EBP -4], EBX
MOV EBX, [EBP-4]
MOV ESP, EBP
POP EBP
RET
multiplicacao:
PUSH EBP
MOV EBP, ESP
PUSH DWORD 0
MOV EBX, [EBP+12]
PUSH EBX
MOV EBX, [EBP+8]
POP EAX
IMUL EAX, EBX
MOV EBX, EAX
MOV [EBP -4], EBX
MOV EBX, [EBP-4]
MOV ESP, EBP
POP EBP
RET

_start:

    PUSH EBP ; guarda o base pointer
    MOV EBP, ESP ; estabelece um novo base pointer

    PUSH DWORD 0
    PUSH DWORD 0
    PUSH DWORD 0
    MOV EBX, 3
    MOV [EBP -4], EBX
    MOV EBX, [EBP-4]
    PUSH EBX
    MOV EBX, 4
    PUSH EBX
    CALL soma
    POP EDX
    POP EDX
    MOV [EBP -8], EBX
    MOV EBX, [EBP-4]
    PUSH EBX
    MOV EBX, [EBP-8]

```



```

PUSH EBX
CALL multiplicacao
POP EDX
POP EDX
MOV [EBP -12], EBX
MOV EBX, [EBP-8]
PUSH EBX
CALL print
POP EBX
MOV EBX, [EBP-12]
PUSH EBX
CALL print
POP EBX

; interrupcao de saida
POP EBP
MOV EAX, 1
INT 0x80

```

Exemplo 2- Código recursivo (cálculo do fatorial):

A função a ser testada no segundo exemplo foi:

```

function fatorial(n::Int)::Int
    if (n == 0) || (n == 1)
        return 1
    else
        return n * fatorial(n - 1)
    end
end

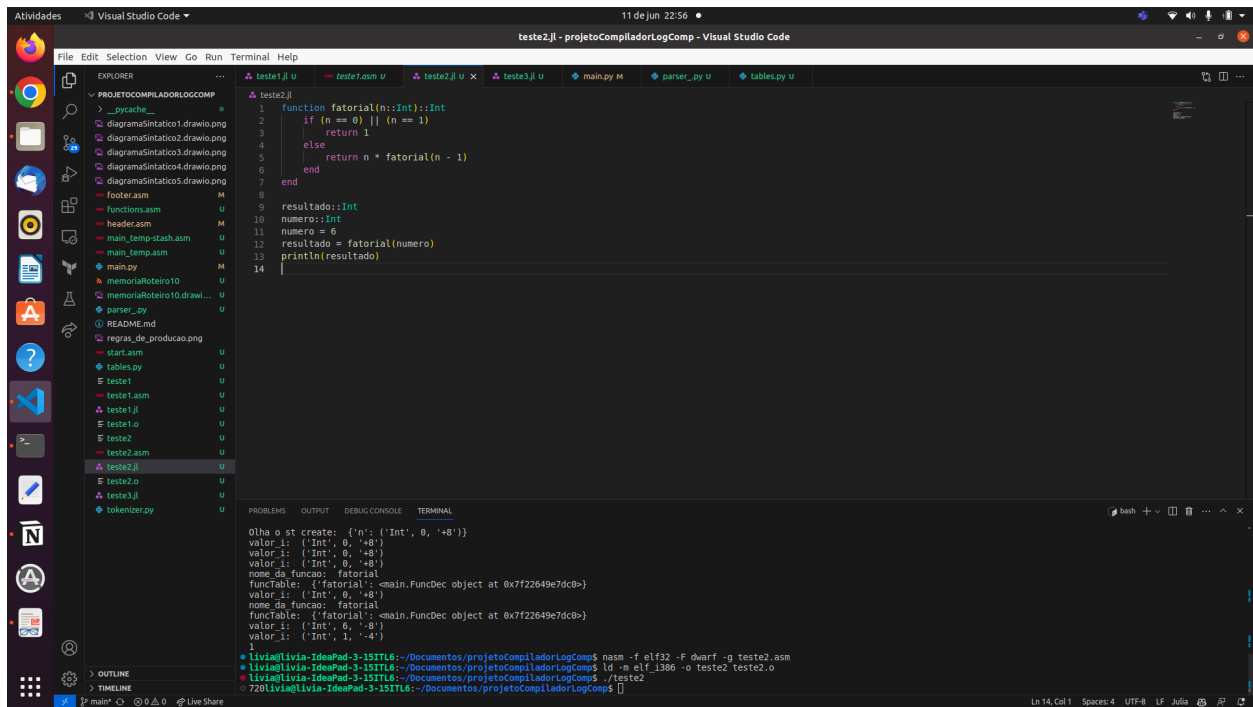
resultado::Int
numero::Int
numero = 6
resultado = fatorial(numero)
println(resultado)

```

Resultado esperado:

```
Resultado do fatorial: 720
```

Resultado obtido (720) :



O assembly gerado pode ser visto abaixo:

```
; constantes
SYS_EXIT equ 1
SYS_READ equ 3
SYS_WRITE equ 4
STDIN equ 0
STDOUT equ 1
True equ 1
False equ 0

segment .data

segment .bss ; variaveis
    res RESB 1

section .text
    global _start

print: ; subrotina print

    PUSH EBP ; guarda o base pointer
    MOV EBP, ESP ; estabelece um novo base pointer
```

```

MOV EAX, [EBP+8] ; 1 argumento antes do RET e EBP
XOR ESI, ESI

print_dec: ; empilha todos os digitos
MOV EDX, 0
MOV EBX, 0x000A
DIV EBX
ADD EDX, '0'
PUSH EDX
INC ESI ; contador de digitos
CMP EAX, 0
JZ print_next ; quando acabar pula
JMP print_dec

print_next:
CMP ESI, 0
JZ print_exit ; quando acabar de imprimir
DEC ESI

MOV EAX, SYS_WRITE
MOV EBX, STDOUT

POP ECX
MOV [res], ECX
MOV ECX, res

MOV EDX, 1
INT 0x80
JMP print_next

print_exit:
POP EBP
RET

; subrotinas if/while
binop_je:
JE binop_true
JMP binop_false

binop_jg:
JG binop_true
JMP binop_false

binop_jl:
JL binop_true
JMP binop_false

binop_false:
MOV EBX, False
JMP binop_exit

binop_true:
MOV EBX, True

binop_exit:
RET

```

```

fatorial:
PUSH EBP
MOV EBP, ESP
IF_25:
MOV EBX, [EBP+8]
PUSH EBX
MOV EBX, 0
POP EAX
CMP EAX, EBX
call binop_je
PUSH EBX
MOV EBX, [EBP+8]
PUSH EBX
MOV EBX, 1
POP EAX
CMP EAX, EBX
call binop_je
POP EAX
OR EAX, EBX
MOV EBX, EAX
CMP EBX, False
JE ELSE_25
MOV EBX, 1
MOV ESP, EBP
POP EBP
RET
JMP END_IF_25
ELSE_25:
MOV EBX, [EBP+8]
PUSH EBX
MOV EBX, [EBP+8]
PUSH EBX
MOV EBX, 1
POP EAX
SUB EAX, EBX
MOV EBX, EAX
PUSH EBX
CALL fatorial
POP EDX
POP EAX
IMUL EAX, EBX
MOV EBX, EAX
MOV ESP, EBP
POP EBP
RET
END_IF_25:

_start:

    PUSH EBP ; guarda o base pointer
    MOV EBP, ESP ; estabelece um novo base pointer

PUSH DWORD 0

```

```

PUSH DWORD 0
MOV EBX, 6
MOV [EBP -8], EBX
MOV EBX, [EBP-8]
PUSH EBX
CALL fatorial
POP EDX
MOV [EBP -4], EBX
MOV EBX, [EBP-4]
PUSH EBX
CALL print
POP EBX

; interrupcao de saida
POP EBP
MOV EAX, 1
INT 0x80

```

Exemplo 3- Função countdown:

A função a ser testada no terceiro exemplo foi:

```

function countdown(n::Int)::Int
    if n == 0
        println(42)
    else
        println(n)
        countdown(n - 1)
    end
    return 0
end

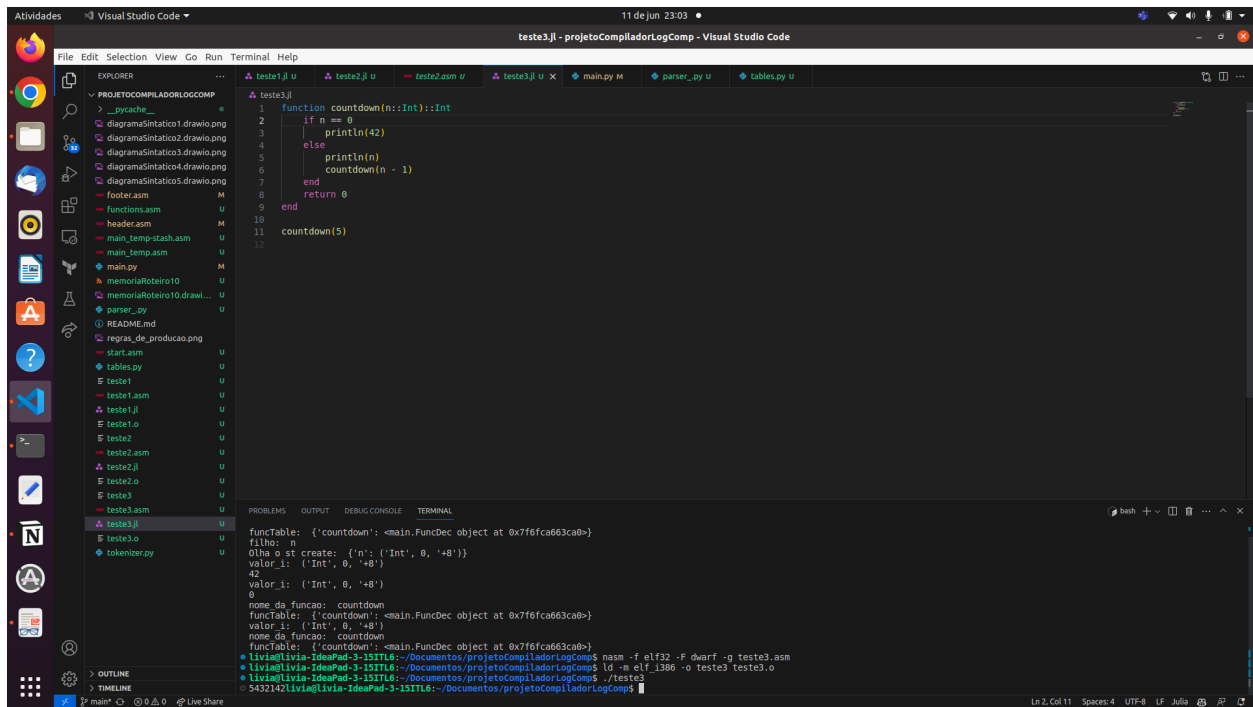
countdown(5)

```

Resultado esperado:

```
5 4 3 2 1 42
```

Resultado obtido (5432142):



O assembly gerado pode ser visto abaixo:

```
; constantes
SYS_EXIT equ 1
SYS_READ equ 3
SYS_WRITE equ 4
STDIN equ 0
STDOUT equ 1
True equ 1
False equ 0

segment .data

segment .bss ; variaveis
res RESB 1

section .text
global _start

print: ; subrotina print

    PUSH EBP ; guarda o base pointer
    MOV EBP, ESP ; estabelece um novo base pointer
```

```

MOV EAX, [EBP+8] ; 1 argumento antes do RET e EBP
XOR ESI, ESI

print_dec: ; empilha todos os digitos
MOV EDX, 0
MOV EBX, 0x000A
DIV EBX
ADD EDX, '0'
PUSH EDX
INC ESI ; contador de digitos
CMP EAX, 0
JZ print_next ; quando acabar pula
JMP print_dec

print_next:
CMP ESI, 0
JZ print_exit ; quando acabar de imprimir
DEC ESI

MOV EAX, SYS_WRITE
MOV EBX, STDOUT

POP ECX
MOV [res], ECX
MOV ECX, res

MOV EDX, 1
INT 0x80
JMP print_next

print_exit:
POP EBP
RET

; subrotinas if/while
binop_je:
JE binop_true
JMP binop_false

binop_jg:
JG binop_true
JMP binop_false

binop_jl:
JL binop_true
JMP binop_false

binop_false:
MOV EBX, False
JMP binop_exit

binop_true:
MOV EBX, True

binop_exit:
RET

```

```

countdown:
PUSH EBP
MOV EBP, ESP
IF_20:
MOV EBX, [EBP+8]
PUSH EBX
MOV EBX, 0
POP EAX
CMP EAX, EBX
call binop_je
CMP EBX, False
JE ELSE_20
MOV EBX, 42
PUSH EBX
CALL print
POP EBX
JMP END_IF_20
ELSE_20:
MOV EBX, [EBP+8]
PUSH EBX
CALL print
POP EBX
MOV EBX, [EBP+8]
PUSH EBX
MOV EBX, 1
POP EAX
SUB EAX, EBX
MOV EBX, EAX
PUSH EBX
CALL countdown
POP EDX
END_IF_20:
MOV EBX, 0
MOV ESP, EBP
POP EBP
RET

_start:

    PUSH EBP ; guarda o base pointer
    MOV EBP, ESP ; estabelece um novo base pointer

MOV EBX, 5
PUSH EBX
CALL countdown
POP EDX

; interrupcao de saida
POP EBP
MOV EAX, 1
INT 0x80

```


Logo, conclui-se que todos os programas testados alcançaram os resultados desejados.

EOF