

Compilador de Linguagem de Montagem

Marcello Mello

ESBOÇO

- INTRODUÇÃO
- REQUISITOS DE SISTEMA
- MÓDULOS
- IMPLEMENTAÇÕES
- UML DIAGRAMA (CLASSE)
- RESULTADOS
- REFERÊNCIAS

INTRODUÇÃO

Um compilador é um programa que converte instruções em um código de máquina ou em um formulário de nível inferior para que possam ser lidos e executados por um computador. O conjunto de instruções da linguagem é predefinido e a folha de dados correspondente às instruções é a seguinte:

Existem 8 registros, a saber:

AX, BX, CX, DX, EX, FX, GX, HX

Qualquer operação aritmética pode ser feita apenas usando registradores.

Existem duas instruções de entrada / saída.

Os operadores aritméticos suportados são ADD, SUB, MUL, DIV.

Operações lógicas IF THEN ELSE são suportadas.

A instrução JUMP é usada para pular para o rótulo correspondente no programa

A execução do programa começa com a palavra-chave START e termina com a palavra-chave END

REQUISITOS DE SISTEMA

REQUISITOS DE SOFTWARE:

Sistema operacional: WINDOWS

Idioma: Pascal

MÓDULOS

- Módulo de Compilação
- Módulo de Execução

Módulo de Compilação:

- Primeiro, verificamos se o arquivo fornecido pelo usuário está com a extensão .asm ou não e depois analisamos o código de montagem linha por linha.
- Intermediate Language, Symbol Table, Block Address Table e Memory Table são gerados e armazenados em um arquivo .obj.
- As instruções presentes no código de montagem são convertidas em seus opcodes correspondentes.

Módulo de Execução:

- A Linguagem Intermediária gerada e armazenada na forma de uma tabela no módulo de Compilação é usada para executar os Códigos de Operação (opcodes) e finalmente a saída é gerada neste módulo.

IMPLEMENTAÇÃO

FOLHA DE DADOS

- As especificações para o montador / simulador são folha de código / folha de dados para o idioma de montagem.
- O conjunto de instruções da linguagem é predefinido e a folha de dados correspondente às instruções é a seguinte:
- Existem 8 registros, a saber:
- AX, BX, CX, DX, EX, FX, GX, HX
- Qualquer operação aritmética pode ser feita apenas usando registradores. Exemplo:
 - DATA A : Isso irá alocar 4 bytes para A
- CONST C =5 : Isso fará com que a constante 5 seja atribuída a C
- MOV instrução é usada para mover valores entre registradores ou entre registradores e variáveis. Exemplo :
 - MOV AX, C : Agora AX tem valor de C
 - MOV C, AX : Valor do AX se move para C
 - MOV AX, DX : Valor do DX passa para o AX

Existem duas instruções de entrada / saída além dessas

READ AX : Valor lido e atribuído ao registro

PRINT AX : Para imprimir os valores de AX

Operadores aritméticos suportados são ADD, SUB, MUL ,DIV

ADD DX, AX, BX : DX= AX + BX

SUB EX, DX, CX : EX = DX - CX

MUL EX, DX, CX : EX = DX * CX

DIV EX, DX, CX : EX = DX / CX

Operações lógicas IF THEN ELSE ENDIF são suportadas. Exemplo:

IF condição THEN

Bloco de instruções

ELSE

Bloco de instruções

ENDIF

As verificações de condição suportadas são :

GT : Maior que (>).

LT : Menor que (<).

EQ : Igual a (=).

GTEQ : Maior que ou igual a (>=)

LTEQ : Menor que ou igual a (<=).

Onde a condição pode estar entre operadores e registros somente.

A instrução JMP é usada para pular para o rótulo correspondente no programa.

X:

MOV AX, C

JMP X : Vai pular a execução do programa para X

A execução do programa começa com a palavra-chave START e termina com a palavra-chave END.

START : Execução do programa começa aqui

END : Final da execução do programa.

CONJUNTO DE INSTRUÇÕES

REGISTRADORES	AX,BX,CX,DX,EF,FX,GX,HX
DECLARAÇÃO / INICIALIZAÇÃO	DATA,CONSTANT
ARITMÉTICA	ADD,SUB,MUL,DIV
CONDICIONAL	IF THEN ELSE
SALTO INCONDICIONAL	JMP
ENTRADA / SAÍDA	READ,PRINT
PROCESSAMENTO DE DADOS	MOV
VERIFICAÇÕES DE CONDIÇÃO	GT, LT ,EQ ,GTEQ , LTEQ
OUTRAS PALAVRAS-CHAVE	START,END, <label>:

OP CODES PARA INSTRUÇÕES

Instrução	Op code
MOV(Registrador p/ Memória)	1
MOV(Memória p/ Registrador)	2
ADD	3
SUB	4
MUL	5
JUMP/ ELSE	6
IF	7
EQ	8
LT	9
GT	10
LTEQ	11
GTEQ	12
PRINT	13
READ	14

Exemplo de código de montagem

- DATA B
- DATA A
- DATA C[4]
- DATA D
- CONST E = 8
- START:
- READ AX
- READ BX
- MOV A, AX
- MOV B, BX
- ADD CX, AX, BX
- MOV DX, E
- X:
- IF CX EQ DX THEN
- MOV C[0], CX
- MOV D, CX
- ELSE
- MOV C[1], CX
- ENDIF
- JUMP X
- END

DATA B

DATA A

DATA C[4]

DATA D

CONST E = 0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

DATA B

LINGUAGEM INTERMEDIÁRIA

REGISTRADOR

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

ENDEREÇO ATUAL DA MEMÓRIA = 8

ENDEREÇOS DE BLOCO

Nome do bloco	Endereço

MEMÓRIA

TABELA DE SÍMBOLOS

Nome	Endereço	Tamanho
B	8	1

DATA B

DATA A

DATA C[4]

DATA D

CONST E = 0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

DATA A

REGISTRADOR CÓDIGOS

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

LINGUAGEM INTERMEDIÁRIA

ENDEREÇO ATUAL DA MEMÓRIA = 9

ENDEREÇOS DE BLOCO

Nome do bloco	Endereço

MEMORIA

TABELA DE SÍMBOLOS

Nome	Endereço	Tamanho
B	8	1
A	9	1

DATA B

DATA A

DATA C[4]

DATA D

CONST E = 0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

DATA C[4]

REGISTER CODES

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

LANGUAGE

MEMORY CURRENT ADDRESS = 10

BLOCK ADDRESSES

Block name	Address

MEMORY

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4

DATA B

DATA A

DATA C[4]

DATA D

CONST E = 0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

DATA B

DATA A

DATA C[4]

DATA D

CONST E=0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

CONST E = 0

REGISTER CODES							
AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

MEMORY CURRENT ADDRESS = 15

BLOCK ADDRESSES	
Block name	Address

INTERMEDIATE LANGUAGE

MEMORY									
						0			

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4
D	14	1
E	15	0

Aqui tamanho constante é especificado como 0 para indicar como uma constante (uma determinada especificação especifica que a constante é sempre de 1 byte e nós armazenamos no respectivo local de memória)

START

Até este ponto todas as declarações estão feitas.

A partir deste ponto, analise o código e gere o código intermediário

DATA B
DATA A
DATA C[4]
DATA D
CONST E = 0
START:

READ AX

READ BX
MOV A, AX
MOV B, BX
ADD CX, AX, BX
MOV DX, E
X:
IF CX EQ DX THEN
 MOV C[0], CX
 MOV D, CX
ELSE
 MOV C[1], CX
ENDIF
JUMP X
END

1. READ AX

REGISTER CODES

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

MEMORY CURRENT ADDRESS = 15

BLOCK ADDRESSES

Block name	Address

INTERMEDIATE LANGUAGE

In No	Op code	PARAMETERS			
1	14	0			

MEMORY

						0			

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4
D	14	1
E	15	0

```
DATA B
DATA A
DATA C[4]
DATA D
CONST E = 0
START:
READ AX
READ BX
MOV A, AX
MOV B, BX
ADD CX, AX, BX
MOV DX, E
X:
```

```
IF CX EQ DX THEN
```

```
    MOV C[0], CX
```

```
    MOV D, CX
```

```
ELSE
```

```
    MOV C[1], CX
```

```
ENDIF
```

```
JUMP X
```

```
END
```

7. IF CX EQ DX THEN

REGISTER CODES

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

LABEL TABLE	
Block name	Address
X	7

INTERMEDIATE LANGUAGE

In No	Op code	PARAMETERS			
1	14	0			
2	14	1			
3	2	1	0		
4	2	0	1		
5	3	2	0	1	
6	1	3	7		
7	7	2	3	8	*

STACK
7

MEMORY									
						0			

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4
D	14	1
E	15	0

DATA B
DATA A
DATA C[4]
DATA D
CONST E = 0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

10. ELSE

REGISTER CODES

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

LABEL TABLE	
Block name	Address
X	7

INTERMEDIATE LANGUAGE

In No	Op code	PARAMETERS			
1	14	0			
2	14	1			
3	2	1	0		
4	2	0	1		
5	3	2	0	1	
6	1	3	7		
7	7	2	3	8	*
8	1	10	2		
9	1	14	2		
10	6	*			

STACK
10
7

MEMORY									
						0			

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4
D	14	1
E	15	0

DATA B
DATA A
DATA C[4]
DATA D
CONST E = 0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

11. MOV C[1], CX

REGISTER CODES

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

LABEL TABLE

Block name	Address
X	7

INTERMEDIATE LANGUAGE

In No	Op code	PARAMETERS			
1	14	0			
2	14	1			
3	2	1	0		
4	2	0	1		
5	3	2	0	1	
6	1	3	7		
7	7	2	3	8	*
8	2	10	2		
9	2	14	2		
10	6	*			
11	2	11	2		

STACK
10
7

MEMORY

						0			

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4
D	14	1
E	15	0

11. MOV C[1], CX

REGISTER CODES

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

LABEL TABLE

Block name	Address
X	7

INTERMEDIATE LANGUAGE

In No	Op code	PARAMETERS			
1	14	0			
2	14	1			
3	2	1	0		
4	2	0	1		
5	3	2	0	1	
6	1	3	7		
7	7	2	3	8	*
8	2	10	2		
9	2	14	2		
10	6	12			
11	2	11	2		

STACK

7

Quando nos deparamos com “ENDIF”, colocamos a pilha e armazenamos esse valor em uma variável temporária. Passamos para essa instrução no idioma intermediário e substituímos o * pelo número de instrução atual

MEMORY

						0			

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4
D	14	1
E	15	0

11. MOV C[1], CX

REGISTER CODES

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

LABEL TABLE

Block name	Address
X	7

INTERMEDIATE LANGUAGE

In No	Op code	PARAMETERS			
1	14	0			
2	14	1			
3	2	1	0		
4	2	0	1		
5	3	2	0	1	
6	1	3	7		
7	7	2	3	8	11
8	2	10	2		
9	2	14	2		
10	6	12			
11	2	11	2		

STACK

Nós estouramos a pilha novamente e nos movemos para essa Instrução em Linguagem Intermediária e substituímos o * por um valor previamente estourado (isto é, que é armazenado na variável temporária) + 1

MEMORY

						0			

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4
D	14	1
E	15	0

DATA B
DATA A
DATA C[4]
DATA D
CONST E = 0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

12. JUMP X

REGISTER CODES

AX	BX	CX	DX	EX	FX	GX	HX
0	1	2	3	4	5	6	7

LABEL TABLE	
Block name	Address
X	7

INTERMEDIATE LANGUAGE

In No	Op code	PARAMETERS			
1	14	0			
2	14	1			
3	2	1	0		
4	2	0	1		
5	3	2	0	1	
6	1	3	7		
7	7	2	3	8	11
8	2	10	2		
9	2	14	2		
10	6	12			
11	2	11	2		
12	6	7			

STACK

MEMORY									
						0			

SYMBOL TABLE

Name	Address	Size
B	8	1
A	9	1
C	10	4
D	14	1
E	15	0

DATA B
DATA A
DATA C[4]
DATA D
CONST E = 0

START:

READ AX

READ BX

MOV A, AX

MOV B, BX

ADD CX, AX, BX

MOV DX, E

X:

IF CX EQ DX THEN

MOV C[0], CX

MOV D, CX

ELSE

MOV C[1], CX

ENDIF

JUMP X

END

DIAGRAMA UML

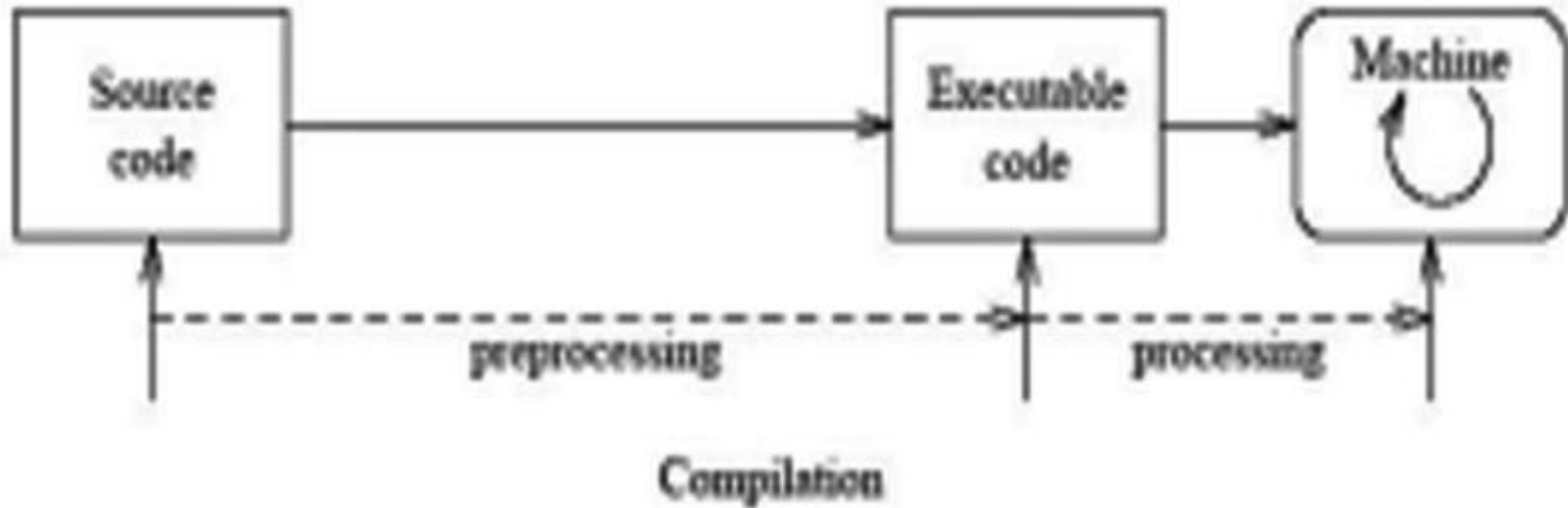
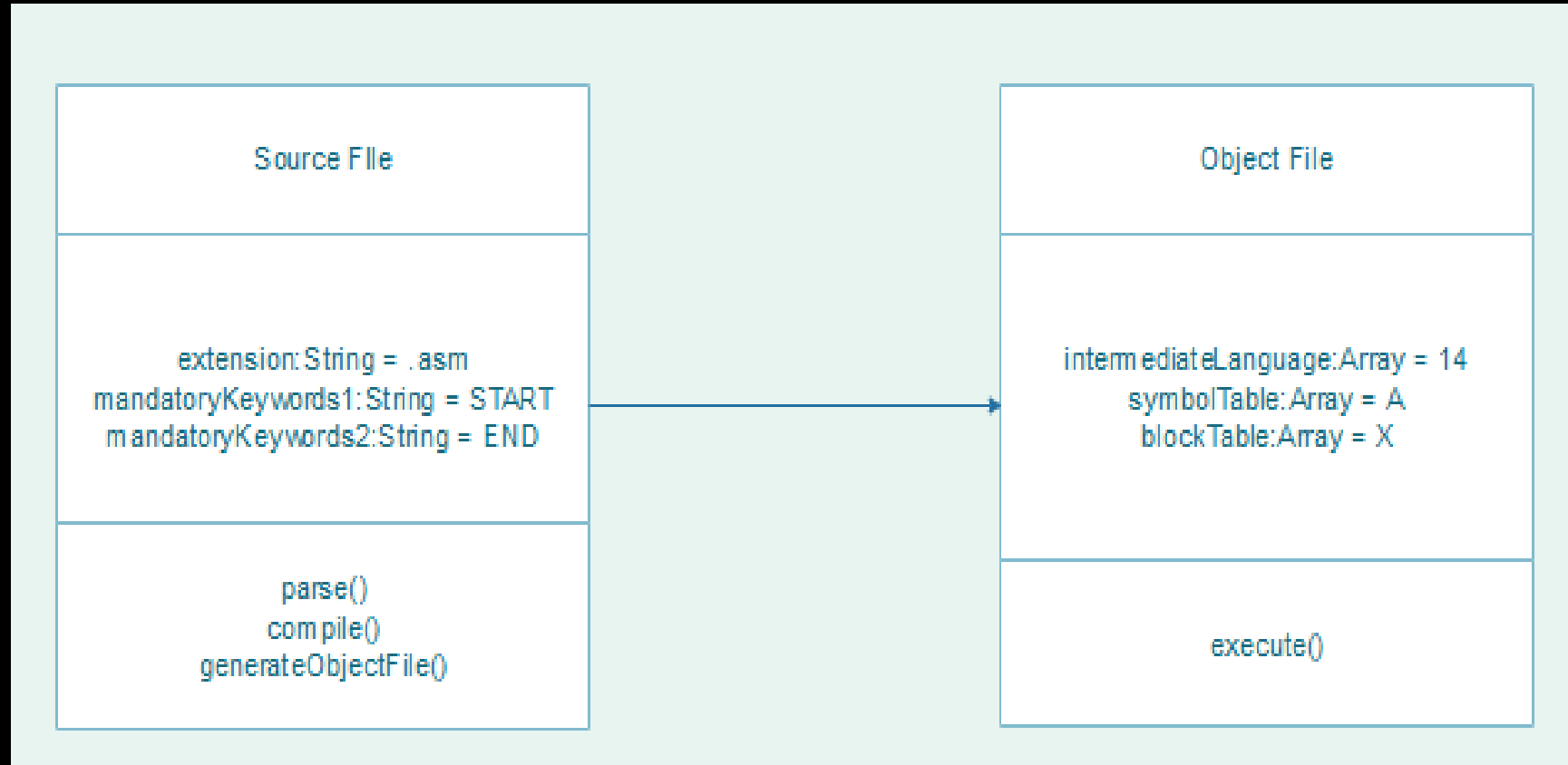
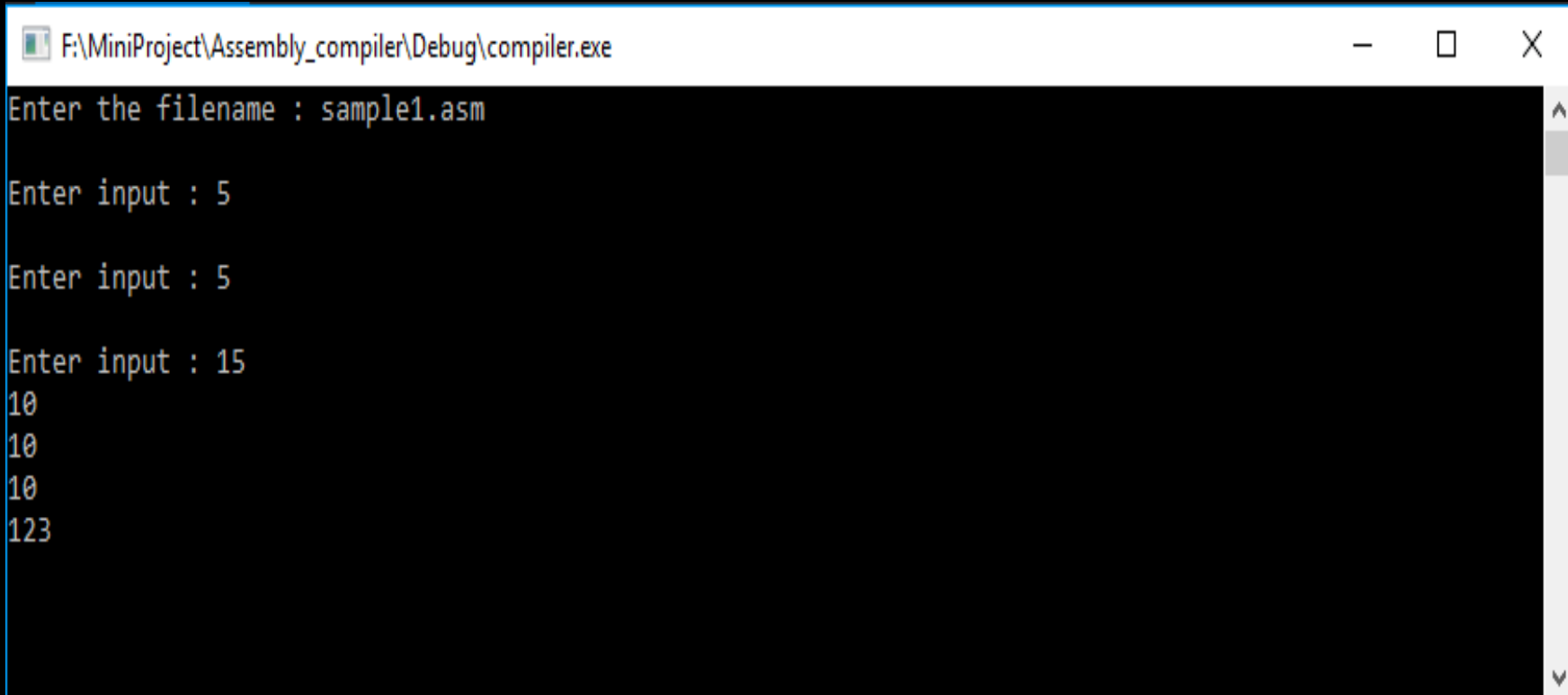


DIAGRAMA DE CLASSE UML



SAÍDA

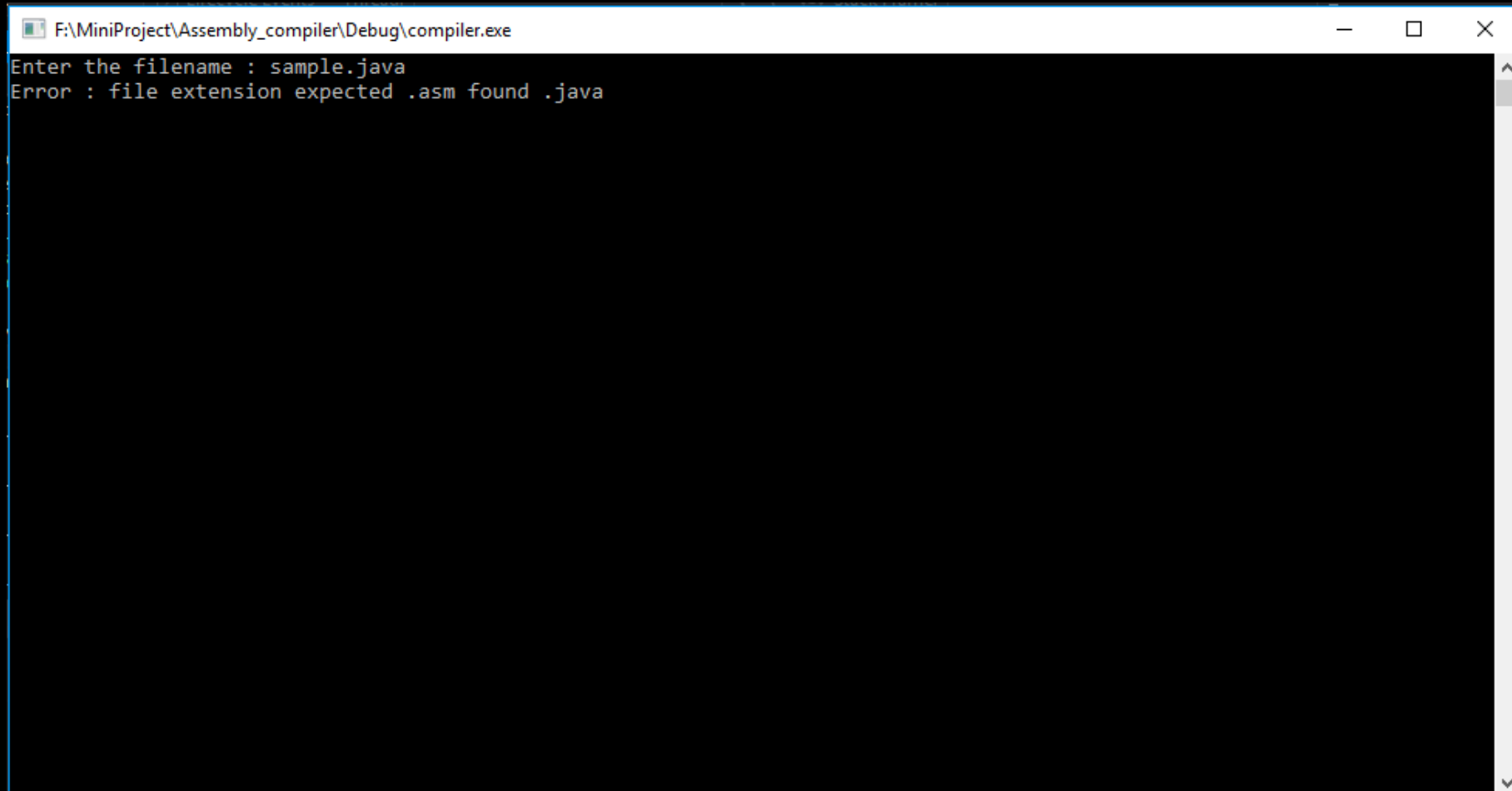


A screenshot of a Windows command prompt window. The title bar at the top reads "F:\MiniProject\Assembly_compiler\Debug\compiler.exe" and includes standard minimize, maximize, and close buttons. The command prompt has a black background with white text. It shows the following sequence of input and output:

```
Enter the filename : sample1.asm  
  
Enter input : 5  
  
Enter input : 5  
  
Enter input : 15  
10  
10  
10  
123
```

The output consists of four lines of numbers: 10, 10, 10, and 123, which appear to be the results of the program's execution for the given inputs.

SAÍDA DE ERRO



A screenshot of a Windows command prompt window. The title bar at the top reads "F:\MiniProject\Assembly_compiler\Debug\compiler.exe" and includes standard minimize, maximize, and close buttons. The command prompt shows the user has entered "sample.java" in response to the prompt "Enter the filename :". Below this, an error message is displayed: "Error : file extension expected .asm found .java". The window has a vertical scrollbar on the right side.

```
F:\MiniProject\Assembly_compiler\Debug\compiler.exe
Enter the filename : sample.java
Error : file extension expected .asm found .java
```


Arquivo de Objetos de exemplo(.obj)

- -----Tabela de símbolos-----
- B 8 1
- A 9 1
- C 10 4
- D 14 1
- E 15 0
- -----Tabela de blocos-----
- X 6
- -----tabela de instruções-----
- 1 14 0
- 2 14 1
- 3 1 12 0
- 4 1 8 1
- 5 3 2 0 1
- 6 14 0
- 7 4 3 0 1
- 8 13 3
- 9 13 2
- 10 7 2 3 8 14
- 11 1 10 2
- 12 13 10
- 13 6 17
- 14 1 11 3
- 15 13 11
- 16 6 6
- 17 13 15

REFERÊNCIAS

- Compiler Design Concepts : https://www.tutorialspoint.com/compiler_design/
- Alfred V Aho, Ravi Sethi, Jeffrey D.Ullman, *“Compilers-Principles Techniques and Tools”, 2nd Edition, Pearson Education 2008..*
- Kenneth C.Louden, *“Compiler Construction-Principles and Practice”, 2nd Edition, Cengage, 2010*

PERGUNTAS??

OBRIGADO