Pontifícia Universidade Católica de Minas Gerais
Departamento de Ciência da Computação
Disciplina: Compiladores - TP: Grafo de Fluxo de Controle
Professor: Pedro Ramos

VALOR: 20 PONTOS
ENTREGA: 26/06/2025

INTERPRETADOR BYTECODE

Neste trabalho você deverá projetar um interpretador bytecode em uma *tiny language* (linguagem pequena).

1. Especificação da linguagem

A linguagem desse interpretador é uma máquina de pilha simples com operações aritméticas, suporte para declaração e inicialiação de variáveis e fluxo de controle. Também há suporte para chamadas de função.

Nesta linguagem, o programa:

```
let x = 3 + 5;
print(x);
```

Produziria a saída:

```
PUSH 3
PUSH 5
ADD
STORE x
LOAD x
PRINT
```

1.2 Instruções

As tabelas abaixo representam as instruções da linguagem *bytecode*.

Operações aritméticas e de Pilha:

Opcode	**Exemplo**	**Conteúdo da Pilha**
PUSH <val>	PUSH 10	[10]
POP	POP	[]
ADD	PUSH 2, PUSH 3, ADD	[5]
SUB	PUSH 5, PUSH 3, SUB	[2]
MUL	PUSH 4, PUSH 2, MUL	[8]
DIV	PUSH 8, PUSH 4, DIV	[2]
MOD	PUSH 10, PUSH 3, MOD	[1]
NEG	PUSH 5, NEG	[-5]

Variáveis:

Opcode	**Exemplo**	**Conteúdo da Pilha**
STORE <var>	PUSH 10, STORE x	[]
LOAD <var>	LOAD x	[10]

Fluxo de controle:

Opcode	**Exemplo**	**Conteúdo da Pilha**
JMP <addr>	JMP 8	[]
JZ <addr>	PUSH 0, JZ 20	[]
JNZ <addr>	PUSH 1, JNZ 32	[]
HALT	PUSH 3, HALT	[3]

Comparação:

Opcode	**Exemplo**	**Conteúdo da Pilha**
EQ	PUSH 3, PUSH 3, EQ	[1]
NEQ	PUSH 3, PUSH 4, NEQ	[1]
LT	PUSH 2, PUSH 3, LT	[1]
GT	PUSH 5, PUSH 2, GT	[1]
LE	PUSH 2, PUSH 2, LE	[1]
GE	PUSH 4, PUSH 3, GE	[1]

Funções e E/S:

Opcode	**Exemplo**	**Conteúdo da Pilha**
CALL <addr>	CALL 16	[<addr_ret>]
RET	RET	[]
PRINT	PUSH 42, PRINT	[42]
READ	READ, STORE x	[]

Obs: As instruções READ e PRINT trabalham com a entrada e saída padrão, respectivamente. Há também a instrução LABEL (que representa um rótulo) e instruções vazias (que devem ser ignoradas).

3. Máquina virtual

Projete uma máquina de pilha virtual para interpretar programas na linguagem especificada acima.

- **Pilha e armazenamento**

  O programa acontece em uma pilha de endereços inteiros limitada pela memória virtual dedicada ao espaço do seu interpretador, e considera que não há registradores livres na CPU para reduzir acessos à memória em instruções aritméticas.

  Para simular a memória da VM, considere o uso de um *map* entre variáveis (endereços) e valores.

- **Apontador para a próxima instrução a ser executada**

  Este apontador rastreia o próximo endereço a ser executado. É atualizado durante fluxo de controle.

- **Comportamento das instruções bytecode**.

  Atenção ao comportamento das instruções bytecode. Algumas instruções como LOAD, STORE, CALL e RET fazem operações implícitas na pilha. CALL empilha o endereço de retorno; LOAD empilha o valor da variável carregada; STORE desempilha o valor salvo na memória; RET desempilha o endereço de retorno. Fique atento a essas e outras funções.

4. Implementação

- **Linguagem:** Escreva o interpretador em Python, C, C++ ou Java.

- **O que será avaliado:**

  (1) (10pts) Código compilável e executável que lê da entrada padrão um arquivo de texto utf-8 representando um **código em bytecode** na linguagem especificada neste trabalho, e **produz na saída padrão o resultado da execução deste programa.** Submetido no Canvas.

  (2) (10pts) Entrevista com o professor no dia 30/06 em sala de aula durante o horário da disciplina. Será realizada em rodem de chegada dos trios. Durante a entrevista serão executados testes não divulgados. Portanto, atente-se para os casos extremos.

- O trabalho pode ser realizado de forma individual, dupla ou trio.
- Uma lista não exaustiva de exemplos está disponível em anexo na tarefa correspondente no Canvas.
- Uma implementação para o interpretador pode ser 80% gerada por uma IA atual. Seja responsável no seu aprendizado.
- **Atenção,** não é preciso escrever um tradutor de linguagem imperativa para Bytecode. Os exemplos acima são didáticos.

3. Exemplos

Programa:

```
let a = 10;
let b = a * 2;
print(b);
```

Bytecode:

```
PUSH 10
STORE a
LOAD a
PUSH 2
MUL
STORE b
LOAD b
PRINT
```

Saída:

```
20
```

Programa:

```
let x = 4;
if (x > 2) {
    print(1);
} else {
    print(0);
}
```

Bytecode:

```
PUSH 4
STORE x
LOAD x
PUSH 2
GT
JZ ELSE_BLOCK
PUSH 1
PRINT
JMP END_IF

ELSE_BLOCK:
PUSH 0
PRINT

END_IF:
```

Saída:

```
1
```

Programa:

```
function add(int a, int b) {
    print(a + b);
    return;
}

function main() {
    add(3, 4);
    return 0;
}
```

Bytecode:

```
CALL 12 # MAIN_START
HALT

MAIN_START:
PUSH 3
PUSH 4
STORE b
STORE a
CALL 30 # ADD_INICIO
PUSH 0
STORE r
RET

ADD_INICIO:
LOAD a
LOAD b
ADD
PRINT
POP
RET
```

Saída:

```
7
```

Programa:

```
let x = 5;
while (x > 0) {
    print(x);
    x = x - 1;
}
```

Bytecode:

```
PUSH 5
STORE x
LOOP_START:
LOAD x
PUSH 0
GT
JZ LOOP_END
LOAD x
PRINT
LOAD x
PUSH 1
SUB
STORE x
JMP LOOP_START

LOOP_END:
```

Saída:

```
5
4
3
2
1
```

Programa:

```
function add(a, b) {
    return a + b;
}
let result = add(3, 7);
print(result);
```

Bytecode:

```
PUSH 3
PUSH 7
STORE b
STORE a
CALL FUNC_START
LOAD r
PRINT
JMP END

FUNC_START:
LOAD a
LOAD b
ADD
STORE r
RET

END:
```

Saída:

```
10
```

5. Extra

- Implemente pelo menos uma otimização que reduza o tamanho do código bytecode sem modificar sua semântica. (4pts)

Bom trabalho!