Compiladores, 2022/2023

Trabalho prático, parte 1
– Expressões aritméticas para a linguagem Mar –

1 Introdução

Pretende-se que faça um compilador de uma linguagem muito simples, que apenas permite o cálculo de expressões aritméticas. Um programa nesta linguagem consiste numa ou mais instruções. A linguagem só permite um tipo de instrução: a palavra reservada `print`, seguido de uma expressão aritmética, seguido de um ponto e vírgula. Eis um exemplo de um programa válido:

```
print (1 + 2) * (3 + 4);
print -2 + 3.14;
```

A expressão aritmética pode ter os 4 operadores usuais: `+`, `-`, `*` e `/`, bem como os habituais parentesis curvos.

Os operandos podem ser literais inteiros ou reais, sem sinal. Note que o símbolo menos (`-`) tanto pode ser usado quer como operador binário, quer unário.

O seu compilador deve geral código para uma máquina virtual de stack, onde todas as operações aritméticas são realizadas no stack de execução. Internamente, quer os inteiros quer os reais são manipulados pela máquina virtual como se fossem um double.

O compilador deverá ler o input a partir de um ficheiro que contém o código fonte a compilar, e deverá ter a extensão '.mar'. O compilador deverá gerar os byte-codes correspondentes num ficheiro com o mesmo nome mas com extensão 'marbc'. Por exemplo, dado o input 'teste.mar', deverá ser gerado (caso não haja erros de compilação) um ficheiro de output com o nome 'teste.marbc'.

Para além do compilador, deverá também programar uma máquina virtual que receba como input um ficheiro '.marbc' e execute o código nele contido.

A título de exemplo, se o ficheiro de input '.mar' tivesse apenas as duas instruções apresentadas anteriormente, a execução dos bytecodes contidos no '.marbc' correspondente deveria produzir o seguinte output:

```
21.0
1.14
```

2 Instruções da máquina virtual

A máquina virtual tem apenas 7 instruções:

```
DCONST
ADD
SUB
MUL
DIV
UMINUS
PRINT
HALT
```

Destas 7 instruções, apenas DCONST tem um argumento que consiste num double (representado por 8 bytes, tal como em Java).

Numa máquina de stack, as expressões artiméticas são realizadas no stack: os operandos são empilhados no stack. Para se realiar uma operação, faz-se pop do(s) operando(o), efectua-se a operação, e empilha-se o resultado no stack.

A isntrução DCONST empilha o seu argumento no stack de execução.

A instrução PRINT deve desempilhar o valor que está no topo do stack e escrevê-lo para o ecrã.

A instrução HALT termina o programa.

2.1 Exemplo

Ao ser compilado, o seguinte input:

```
print (1 + 2) * (3 + 4);
print -2 + 3.14;
```

deverá geral o seguinte output (em assembly):

```
 0: DCONST 1.0
 1: DCONST 2.0
 2: ADD
 3: DCONST 3.0
 4: DCONST 4.0
 5: ADD
 6: MULT
 7: PRINT
 8: DCONST 2.0
 9: UMINUS
10: DCONST 3.14
11: ADD
12: PRINT
13: HALT
```

O que é escrito no ficheiro 'marbc' correspondente é uma sequência de bytes (os bytecodes) correspondentes ao código assembly acima apresentado.

Por sua vez, quando a vossa máquina virtual corre o código '.marbc', deverá aparecer no output o resultado da execução. No exemplo acima seria:

```
21.0
1.14
```

2.2 Output obtido pela minha implementação

Apresento os outputs obtidos pela minha implementação do trabalho, quando executado para o input apresentado anteriormente. Estes exemplos foram mostrados na aula téorica de 21-mar-2023.

Quer o meu compilador, quer a minha máquina virtual, permitem passar o argumento `-debug`, que faz com que se envie para o ecrã o código gerado (no caso do compilador), e um trace da execução dos bytecodes (no caso da máquina virtual). Recomenda-se que façam algo análogo no vosso trabalho.

java marCompiler inputs/in2.mar -debug

```
Generated assembly code:
0: DCONST 1.0
1: DCONST 2.0
2: ADD
3: DCONST 3.0
4: DCONST 4.0
5: ADD
6: MULT
7: PRINT
8: DCONST 2.0
9: UMINUS
10: DCONST 3.14
11: ADD
12: PRINT
13: HALT
Corresponding bytecodes:
7 63 -16 0 0 0 0 0 0 7 64 0 0 0 0 0 0 0 0 7 64 8 0 0 0 0 0 0 7 64 16 0 0 0 0 0 0 0 2 5 7 64 0 0 0 0 0 0 0 4 7 64 9 30 -72 81 -21 -123 31 0 5 6
Saving bytecodes to inputs/in2.marbc
```

java marVM inputs/in2.marbc -debug

```
Reading bytecodes from inputs/in2.marbc
7 63 -16 0 0 0 0 0 0 7 64 0 0 0 0 0 0 0 0 7 64 8 0 0 0 0 0 0 7 64 16 0 0 0 0 0 0 0 2 5 7 64 0 0 0 0 0 0 0 4 7 64 9 30 -72 81 -21 -123 31 0 5 6
                        Stack: []
 DCONST 1.0
                        Stack: [1.0]
 DCONST 2.0
                        Stack: [1.0, 2.0]
 ADD
                        Stack: [3.0]
 DCONST 3.0
                        Stack: [3.0, 3.0]
 DCONST 4.0
                        Stack: [3.0, 3.0, 4.0]
 ADD
                        Stack: [3.0, 7.0]
 MULT
                        Stack: [21.0]
 PRINT
21.0
                        Stack: []
 DCONST 2.0
                        Stack: [2.0]
 UMINUS
                        Stack: [-2.0]
 DCONST 3.14
                        Stack: [-2.0, 3.14]
 ADD
                        Stack: [1.1400000000000001]
 PRINT
1.1400000000000001
                        Stack: []
 HALT
```

3 Sobre a leitura e escrita de ficheiros em Java

Recomenda-se que usem um DataOutputStream e um DataInputStream para a leitura e escrita dos ficheiros binários. Podem ver exemplos de utilização neste link:

https://jenkov.com/tutorials/java-io/dataoutputstream.html

4 Condições de realização

O projeto deve ser realizado em grupo, de acordo com as inscrições em grupo do laboratório. Deverão submeter o vosso código num ficheiro ZIP por via eletrónica, através da tutoria, até às 23:59 do dia 29/03/2023.

O código ZIP deverá conter a gramática '.g4' em ANTLR, bem como todo o código Java desenvolvido.

- O vosso compilador deverá chamar-se `marCompiler`
- A vossa máquina virtual deverá chamar-se `marVM`
- Recomenda-se que a vossa gramática se chame `mar.g4`

Apenas é necessário que 1 dos elementos do grupo submeta o trabalho.