Compiladores, 2022/2023

Trabalho prático, parte 2
– Outros tipos de dados primitivos para a linguagem Mar –

1 Introdução

No 1ª parte do trabalho prático, as expressões da linguagem mar apenas permitiam ter números como operandos. Na 2ª parte do trabalho, pretende-se que extenda a linguagem de modo a que as expressões possam ter outros tipos de dados. No total, deverá ser permitido 4 tipos de dados primitivos: *Number*, *Bool*, *String*, e *Nil*.

O tipo *Number* foi usado no 1ª parte do trabalho. É usado para valores inteiros e reais, mas ambos são manipulados intermente como se fossem doubles com 4 bytes.

O tipo *Bool* é usado para valores booleanos e tem 2 valores possíveis: `true` e `false`.

O tipo *String* é usado para strings de caracteres, que devem ser delimitadas por aspas.

O tipo *Nil* só tem 1 valor possível que é `nil`. É um tipo de dados especial que denota a ausência de valor (semelhante a `null` em C ou Java).

Com estes novos tipos de dados, podemos ter não só expressões aritméticas (como no 1º trabalho), mas também expressões booleanas, e expressões que envolvem strings. E, inclusivamente, podemos ter uma expressão com o valor `nil`.

2 Operadores válidos consoante o tipo de dados

De seguida apresenta-se os operadores que são válidos na linguagem mar, consoante o tipo do(s) operando(s).

Number

- O operador `-` unário só pode ser aplicado a um operando do tipo *Number*. O resultado da operação é um valor do tipo *Number*.
- Os operadores binários `+`, `-`, `*`, `/`, podem ser aplicados a operandos do tipo *Number*. O resultado da operação é um valor do tipo *Number*. Estes operadores têm o significado usual da aritmética.
- Os operadores relacionais `>`, `>=`, `<`, `<=`, `==`, `!=`, podem ser aplicados a operandos do tipo *Number*. O resultado da operação é um valor do tipo *Bool*. Estes operadores têm o significado usual tal como nas linguagens C ou Java.

Bool

- O operador `not` é unário e só pode ser aplicado a um operando do tipo *Bool*. O resultado da operação é um valor do tipo *Bool*. Este operador tem o significa idêntico ao do operador `!` do C e Java.
- Os operadores binários `and`, `or`, podem ser aplicados a operandos do tipo *Bool*. O resultado da operação é um valor do tipo o *Bool*. Estes operadores têm o significado idêntico aos operadores `&&` e `||` do C e Java.
- Note que `not`, `and`, e `or`, são palavras reservadas na linguagem mar.
- Os operadores relacionais `==` e `!=` podem ser aplicados quando os operandos são do tipo *Bool*. O resultado da operação é um valor do tipo *Bool*.

String

- É permitido o operador binário `+` quando os operandos são do tipo *String*. O resultado da operação é um valor do tipo *String* e consiste na concatenação do operando esquerdo com o operando direito, tal como em Java.
- Os operadores relacionais `==` e `!=` podem ser aplicados quando os operandos são do tipo *String*. O resultado da operação é um valor do tipo *Bool*.

Nil

- Os operadores relacionais `==` e `!=` podem ser aplicados quando os operandos são do tipo *Nil*. O resultado da operação é um valor do tipo o *Bool*.

A aplicação de operadores fora do contexto acima descrito deverá originar um erro semântico de inconsistência de tipos (não de parsing), que deve ser reportado pelo compilador de forma apropriada.

2.1 Precedência dos operadores

A tabela seguinte apresenta a precedência dos operadores da linguagem mar. Quanto mais acima na tabela, maior é a precedência do operador (e menor é o número que aparece na 1ª coluna da tabela). Dizer que um operador tem maior precedência que outro, significa que esse operador deve ser avaliado primeiro do que o operador com precedência mais baixa.

Os operadores que aparecem na mesma linha, têm o mesmo nível de precedência. Nesse caso, a ordem pela qual as operações são feitas é da esquerda para a direita.

Precedência	**Descrição**	**Operadores**
1	Parênteses	`()`
2	Unário	`- not`
3	Multiplicação e Divisão	`* /`
4	Soma e Subtração	`+ -`
5	Relacional	`< > <= >=`
6	Igualdade e desigualdade	`== !=`
7	E lógico	`and`
8	Ou lógico	`or`

Exemplo 1

`1 + 2 * 3` é equivalente a `1 + (2 * 3)`

Exemplo 2

`1 + 2 - 3 + 4` é equivalente a `((1 + 2) - 3)) + 4`

3 Exemplo de programa válido

```
print not (5 > 3);
print nil;
print nil == nil;
print nil != nil;
print "uni" + "ver" + "sidade";
print true and (false or true);
print 1 + 2 * 3;
print "bolo" == "pastel";
print "ma" + "ria" == "maria";
```

Ao ser executado pela máquina virtual deveria dar o seguinte output:

```
false
nil
true
false
universidade
true
7.0
false
true
```

4 Exemplo de programa com erros de inconsistência de tipos

O seguinte programa não tem erros de parsing, mas tem erros semânticos (de inconsistência de tipos) e por isso não é compilado para bytecodes.

```
print 1 + "ola";
print 2+3 == 5;
print "ola" - "maria";
print true;
print true + false;
print nil * 2;
print -true;
```

O output do compilador neste caso deve ser algo deste género:

```
line 1:10 error: operator + is invalid between number and string
line 3:14 error: operator - is invalid between string and string
line 5:13 error: operator + is invalid between bool and bool
line 6:12 error: operator * is invalid between nil and number
line 7:7 error: unary operator - is invalid for bool
```

Note que as mensagens de erro indicam a linha e coluna no ficheiro de input onde é detectado o erro.

5 Instruções da máquina virtual

A máquina virtual passa a ter instruções especializadas consoante o tipo de dados. DCONST deixa de existir, e passa a haver a instrução CONST que tem um argumento inteiro (4 bytes) que representa o índice para uma entrada da *constant pool*, onde está a constante desejada (número ou string).

Eis o conjunto completo de instruções da máquina virtual para o 2º trabalho.

Instrução com 1 argumento

OpCode	Argumento	Descrição
CONST	inteiro x	empilha a constante que está na posição x da *constant pool*, no stack.

Instruções sem argumentos

OpCode	Descrição
ADD	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha $a + b$ no stack.
SUB	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha $a - b$ no stack.
MULT	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha $a * b$ no stack.
DIV	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha a/b no stack.
UMINUS	faz pop do operando a, e empilha $-a$ no stack.
CONCAT	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha a concatenado com b no stack.
HALT	termina a execução do programa.
NIL	empilha o valor *nil* no stack.
TRUE	empilha o valor *true* no stack.
FALSE	empilha o valor *false* no stack.
AND	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico a and b no stack.
OR	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico a or b no stack.
NOT	faz pop do operando a, e empilha o valor lógico not a no stack.
GT	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a > b$ no stack.
LT	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a < b$ no stack.

GEQ	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a \geq b$ no stack.
LEQ	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a \leq b$ no stack.
EQ_N	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a == b$ no stack. (Assume que a e b são do valores do tipo *Number*)
EQ_B	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a == b$ no stack. (Assume que a e b são do valores do tipo *Bool*)
EQ_S	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a == b$ no stack. (Assume que a e b são do valores do tipo *String*)
EQ_NIL	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a == b$ no stack. (Assume que a e b são do valores do tipo *Nil*)
NEQ_N	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a \neq b$ no stack. (Assume que a e b são do valores do tipo *Number*)
NEQ_B	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a \neq b$ no stack. (Assume que a e b são do valores do tipo *Bool*)
NEQ_S	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a \neq b$ no stack. (Assume que a e b são do valores do tipo *String*)
NEQ_NIL	faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a \neq b$ no stack. (Assume que a e b são do valores do tipo *Nil*)
PRINT_N	faz pop do operando a, e escreve a no output seguido de uma mudança de linha. (Assume que a é do tipo *Number*)
PRINT_B	faz pop do operando a, e escreve a no output seguido de uma mudança de linha. (Assume que a é do tipo *Bool*)
PRINT_S	faz pop do operando a, e escreve a no output seguido de uma mudança de linha. (Assume que a é do tipo *String*)
PRINT_NIL	faz pop do operando a, e escreve a no output seguido de uma mudança de linha. (Assume que a é do tipo *Nil*)

6 Output obtido pela minha implementação com a opção -debug activa

Apresento os outputs obtido pela minha implementação do compilador e máquina virtual para o input apresentado anteriormente, com a opção `-debug` activa. Recomenda-se que façam algo análogo no vosso trabalho.

6.1 Compilador

```
java marCompiler inputs/in1.mar -debug

... no parsing errors
... no type errors
... code generation
Constant pool:
0: <NUMBER:5.0>
1: <NUMBER:3.0>
2: <STRING:"uni">
3: <STRING:"ver">
4: <STRING:"sidade">
5: <NUMBER:1.0>
6: <NUMBER:2.0>
7: <NUMBER:3.0>
8: <STRING:"bolo">
9: <STRING:"pastel">
10: <STRING:"ma">
11: <STRING:"ria">
12: <STRING:"maria">
Generated assembly code:
0: CONST 0
1: CONST 1
2: GT
3: NOT
4: PRINT_B
5: NIL
6: PRINT_NIL
7: NIL
8: NIL
9: EQ_NIL
10: PRINT_B
11: NIL
12: NIL
13: NEQ_NIL
```

```
14: PRINT_B
15: CONST 2
16: CONST 3
17: CONCAT
18: CONST 4
19: CONCAT
20: PRINT_S
21: TRUE
22: FALSE
23: TRUE
24: OR
25: AND
26: PRINT_B
27: CONST 5
28: CONST 6
29: CONST 7
30: MULT
31: ADD
32: PRINT_N
33: CONST 8
34: CONST 9
35: EQ_S
36: PRINT_B
37: CONST 10
38: CONST 11
39: CONCAT
40: CONST 12
41: EQ_S
42: PRINT_B
43: HALT
Saving the constant pool and the bytecodes to inputs/in1.marbc
```

6.2 Máquina virtual

```
java marVM inputs/in1.marbc -debug

Constant pool:
0: <NUMBER:5.0>
1: <NUMBER:3.0>
2: <STRING:"uni">
3: <STRING:"ver">
4: <STRING:"sidade">
5: <NUMBER:1.0>
6: <NUMBER:2.0>
```

```
7: <NUMBER:3.0>
8: <STRING:"bolo">
9: <STRING:"pastel">
10: <STRING:"ma">
11: <STRING:"ria">
12: <STRING:"maria">
Instructions:
0: CONST 0
1: CONST 1
2: GT
3: NOT
4: PRINT_B
5: NIL
6: PRINT_NIL
7: NIL
8: NIL
9: EQ_NIL
10: PRINT_B
11: NIL
12: NIL
13: NEQ_NIL
14: PRINT_B
15: CONST 2
16: CONST 3
17: CONCAT
18: CONST 4
19: CONCAT
20: PRINT_S
21: TRUE
22: FALSE
23: TRUE
24: OR
25: AND
26: PRINT_B
27: CONST 5
28: CONST 6
29: CONST 7
30: MULT
31: ADD
32: PRINT_N
33: CONST 8
34: CONST 9
35: EQ_S
36: PRINT_B
37: CONST 10
```

```
38: CONST 11
39: CONCAT
40: CONST 12
41: EQ_S
42: PRINT_B
43: HALT
Trace while running the code:
                      Stack: []
 CONST 0 '<NUMBER:5.0>
                      Stack: [<NUMBER:5.0>]
 CONST 1 '<NUMBER:3.0>
                      Stack: [<NUMBER:5.0>, <NUMBER:3.0>]
 GT
                      Stack: [<BOOL:true>]
 NOT
                      Stack: [<BOOL:false>]
 PRINT_B
false
                      Stack: []
 NIL
                      Stack: [<NIL>]
 PRINT_NIL
nil
                      Stack: []
 NIL
                      Stack: [<NIL>]
 NIL
                      Stack: [<NIL>, <NIL>]
 EQ_NIL
                      Stack: [<BOOL:true>]
 PRINT_B
true
                      Stack: []
 NIL
                      Stack: [<NIL>]
 NIL
                      Stack: [<NIL>, <NIL>]
 NEQ_NIL
                      Stack: [<BOOL:false>]
 PRINT_B
false
                      Stack: []
 CONST 2 '<STRING:"uni">
                      Stack: [<STRING:"uni">]
 CONST 3 '<STRING:"ver">
```

```
                    Stack: [<STRING:"uni">, <STRING:"ver">]
 CONCAT
                    Stack: [<STRING:"univer">]
 CONST 4 '<STRING:"sidade">
                    Stack: [<STRING:"univer">, <STRING:"sidade">]
 CONCAT
                    Stack: [<STRING:"universidade">]
 PRINT_S
universidade
                    Stack: []
 TRUE
                    Stack: [<BOOL:true>]
 FALSE
                    Stack: [<BOOL:true>, <BOOL:false>]
 TRUE
                    Stack: [<BOOL:true>, <BOOL:false>, <BOOL:true>]
 OR
                    Stack: [<BOOL:true>, <BOOL:true>]
 AND
                    Stack: [<BOOL:true>]
 PRINT_B
true
                    Stack: []
 CONST 5 '<NUMBER:1.0>
                    Stack: [<NUMBER:1.0>]
 CONST 6 '<NUMBER:2.0>
                    Stack: [<NUMBER:1.0>, <NUMBER:2.0>]
 CONST 7 '<NUMBER:3.0>
                    Stack: [<NUMBER:1.0>, <NUMBER:2.0>, <NUMBER:3.0>]
 MULT
                    Stack: [<NUMBER:1.0>, <NUMBER:6.0>]
 ADD
                    Stack: [<NUMBER:7.0>]
 PRINT_N
7.0
                    Stack: []
 CONST 8 '<STRING:"bolo">
                    Stack: [<STRING:"bolo">]
 CONST 9 '<STRING:"pastel">
                    Stack: [<STRING:"bolo">, <STRING:"pastel">]
 EQ_S
                    Stack: [<BOOL:false>]
 PRINT_B
false
                    Stack: []
```

```
 CONST 10 '<STRING:"ma">
                      Stack: [<STRING:"ma">]
 CONST 11 '<STRING:"ria">
                      Stack: [<STRING:"ma">, <STRING:"ria">]
 CONCAT
                      Stack: [<STRING:"maria">]
 CONST 12 '<STRING:"maria">
                      Stack: [<STRING:"maria">, <STRING:"maria">]
 EQ_S
                      Stack: [<BOOL:true>]
 PRINT_B
true
                      Stack: []
 HALT
```

7 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 26/04/2023.

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.