Compiladores, 2022/2023

Trabalho prático, parte 3
– Variáveis globais. Instruções: atribuição, bloco, while, if –

1 Introdução

Nesta parte do trabalho prático, vamos extender a linguagem mar para permitir declarações de variáveis globais, e mais 4 tipos de instruções, nomeadamente:

- afectação
- bloco
- while
- if-then e if-then-else

Uma vez que é permitido variáveis, também é obtivamente permitido ter variáveis no contexto de expressões. Por exemplo, `x + 1` passa a ser uma expressão sintáticamente válida na linguagem mar.

A linguagem mar também deverá permitir a utilização de comentários no código.

2 Alterações à gramática

Devarão alterar a vossa gramática de modo a permitir estas extensões.

2.1 Programa

Um programa em mar passa agora a ser consituído por uma sequência de zero ou mais declarações de variáveis, seguido de uma sequência de uma ou mais instruções.

2.2 Declaração de variável

Uma declaração de variável é feita indicando o tipo de dados (number, bool, string, ou nil), que passam a ser palavras reservadas da linguagem, seguido do nome da variável, seguido de um ponto-e-vírgula.

O nome da variável obedece à mesma regra que é usada na linguagem C. Isto é, deve começar por uma letra ou por um underscore, e depois pode vir uma sequência de letras, digítos, ou underscores.

2.3 Novas instruções

Afectação

A instrução de afectação permite atribuir um valor a uma variável. A sintaxe é análoga à da linguagem C ou Java: nome da variável, seguido de um sinal de `=`, seguido de uma expressão, seguido de um ponto-e-vírgula. Exemplo:

```
x = 4;
```

Bloco

Um bloco, usualmente também designado por instrução composta, é uma instrução que serve para agrupar instruções. Tal como em C e Java, podemos ter um bloco especificando o seu início com uma chaveta a abrir, seguido de zero ou mais instruções, seguido de uma chaveta a fechar. Exemplo:

```
{
    print x;
    x = x+1;
}
```

While

A instrução while serve para fazer ciclos, tal como em C e Java. A instrução deve começar com a palavra reservada `while`, seguido de uma expressão booleana, seguido da palavra reservado `do`, seguido de uma instrução. Exemplo:

```
while x < 10 do {
```

```
    print x;
    x = x+1;
}
```

if-then e if-then-else

A instrução if deve começar com a palavra reservada `if`, seguido de uma expressão booleana, seguido da palavra reservado `then`, seguido de uma instrução, e opcionalmente poderá ter a parte else, em que aparece a palavra reservada `else` seguido de uma instrução. Exemplo:

```
if x < 10 then print "ola";
else {
   print x;
   x = x+1;
}
```

2.4 Comentários

Tal como em C e Java, podemos ter comentários de uma só linha (começam com `//` e vão até ao final da linha) ou multi-linha (começam com `/*` e terminam com `*/`).

Para facilitar-vos a vida, dou-vos as regras lexicais em ANTLR que permitem ter este tipo de comentários.

```
SL_COMMENT :   '//' .*? (EOF|'\n') -> skip;  // single-line comment
ML_COMMENT :   '/*' .*? '*/' -> skip ;       // multi-line comment
```

3 Análise semântica

A fase da análise semântica deverá criar uma tabela de símbolos para guardar informação relevante para as variáveis globais. Deverá também fazer a verificação de tipos (type checking).

Para além disso, deverá ainda garantir que uma variável só pode ser referenciada no código se tiver sido inicializada previamente com um valor. Caso contrário, deverá ser reportado um erro.

Deverá igualmente ser reportado erros se fizermos referência a uma variável não declarada, ou se tentarmos declarar uma varável que já foi declarada anteriormente.

3.1 Exemplo

O seguinte exemplo mostra um programa que não têm erros sintáticos, mas tem vários erros semânticos que devem ser reportados com mensagens apropriadas.

```
number i;
number j;
bool b;
i = 1;
b = false;
i = i + b;
while i < 10 do {
   i = i + j;
   j = k;
}
if i*i then print "Fim";
```

Ao ser executado, o compilador deveria dar um output deste género:

```
line 6:8 error: operator + is invalid between number and bool
line 8:11 error: j has not been initialized
line 9:7 error: k is not defined
line 11:3 error: if expression must be of type bool
```

4 Novas instruções da máquina virtual

Para além de todas as instruções já descritas no trabalho anterior, a máquina virtual passa a ter mais estas 5 instruções, todas elas com um argumento inteiro.

OpCode	Argumento	Descrição
GLOBAL	inteiro n	aloca n posições num array que permite armazenar variáveis globais. Designemos esse array por *Globals*
LOAD	inteiro *addr*	empilha $Globals[addr]$ no stack.
STORE	inteiro *addr*	faz pop do stack e guarda o valor em $Globals[addr]$
JUMP	inteiro *addr*	actualiza o *instruction pointer* de modo a que a próxima instrução a ser executada seja aquela que se encontra na posição *addr* do array de instruções.
JUMPF	inteiro *addr*	faz pop do stack. Se o valor for `false`, actualiza o *instruction pointer* de modo a que a próxima instrução a ser executada seja aquela que se encontra na posição *addr* do array de instruções.

Durante a fase da análise semântica, ao insirir uma entrada na tabela de símbolos é conveniente atribuir um endereço a cada variável global, endereço esse que será usado pela máquina virtual para aceder à respectiva variável. Sugere-se que atribua os endereços sequencialmente começando em zero. Por exemplo, no programa ilustrado na secção 3.1, a variável `i` ficaria com o endereço 0, a variável `j` ficaria com o endereço 1, e a variável `b` ficaria com o endereço 2.

5 Exemplos obtidos pela minha implementação

Apresento os outputs obtido pela minha implementação do compilador e máquina virtual para alguns inputs.

5.1 Exemplo 1

inputs/gvars1.mar

```
number x;
bool b;
b = true;
x = 5;
x = x + 2;
print x;
print b;
```

Compilador com opção -debug

```
java marCompiler inputs/gvars1.mar -debug

... no parsing errors
... no type errors
... code generation
Constant pool:
0: <NUMBER:5.0>
1: <NUMBER:2.0>
Generated assembly code:
0: GLOBAL 2
1: TRUE
2: STORE 1
3: CONST 0
4: STORE 0
```

```
5: LOAD 0
6: CONST 1
7: ADD
8: STORE 0
9: LOAD 0
10: PRINT_N
11: LOAD 1
12: PRINT_B
13: HALT
Saving the constant pool and the bytecodes to inputs/gvars1.marbc
```

Máquina virtual, sem opção -debug

```
java marVM inputs/gvars1.marbc

7.0
true
```

Máquina virtual, com opção -debug

```
java marVM inputs/gvars1.marbc -debug

Constant pool:
0: <NUMBER:5.0>
1: <NUMBER:2.0>
Instructions:
0: GLOBAL 2
1: TRUE
2: STORE 1
3: CONST 0
4: STORE 0
5: LOAD 0
6: CONST 1
7: ADD
8: STORE 0
9: LOAD 0
10: PRINT_N
11: LOAD 1
12: PRINT_B
13: HALT
Trace while running the code:
                    Globals: []
                      Stack: []
```

```
 GLOBAL 2
                    Globals: [<NIL>, <NIL>]
                      Stack: []
 TRUE
                    Globals: [<NIL>, <NIL>]
                      Stack: [<BOOL:true>]
 STORE 1
                    Globals: [<NIL>, <BOOL:true>]
                      Stack: []
 CONST 0 '<NUMBER:5.0>
                    Globals: [<NIL>, <BOOL:true>]
                      Stack: [<NUMBER:5.0>]
 STORE 0
                    Globals: [<NUMBER:5.0>, <BOOL:true>]
                      Stack: []
 LOAD 0
                    Globals: [<NUMBER:5.0>, <BOOL:true>]
                      Stack: [<NUMBER:5.0>]
 CONST 1 '<NUMBER:2.0>
                    Globals: [<NUMBER:5.0>, <BOOL:true>]
                      Stack: [<NUMBER:5.0>, <NUMBER:2.0>]
 ADD
                    Globals: [<NUMBER:5.0>, <BOOL:true>]
                      Stack: [<NUMBER:7.0>]
 STORE 0
                    Globals: [<NUMBER:7.0>, <BOOL:true>]
                      Stack: []
 LOAD 0
                    Globals: [<NUMBER:7.0>, <BOOL:true>]
                      Stack: [<NUMBER:7.0>]
 PRINT_N
7.0
                    Globals: [<NUMBER:7.0>, <BOOL:true>]
                      Stack: []
 LOAD 1
                    Globals: [<NUMBER:7.0>, <BOOL:true>]
                      Stack: [<BOOL:true>]
 PRINT_B
true
                    Globals: [<NUMBER:7.0>, <BOOL:true>]
                      Stack: []
 HALT
```

5.2 Exemplo 2

inputs/while1.mar

```
number i;
i = 1;
while i <= 3 do {
   print i;
   i = i + 1;
}
print "Fim";
```

Compilador com opção -debug

```
java marCompiler inputs/while1.mar -debug

... no parsing errors
... no type errors
... code generation
Constant pool:
0: <NUMBER:1.0>
1: <NUMBER:3.0>
2: <NUMBER:1.0>
3: <STRING:"Fim">
Generated assembly code:
0: GLOBAL 1
1: CONST 0
2: STORE 0
3: LOAD 0
4: CONST 1
5: LEQ
6: JUMPF 14
7: LOAD 0
8: PRINT_N
9: LOAD 0
10: CONST 2
11: ADD
12: STORE 0
13: JUMP 3
14: CONST 3
15: PRINT_S
16: HALT
Saving the constant pool and the bytecodes to inputs/while1.marbc
```

Máquina virtual, sem opção -debug

```
java marVM inputs/while1.marbc

1.0
2.0
3.0
Fim
```

Máquina virtual, com opção -debug

```
java marVM inputs/while1.marbc -debug

Constant pool:
0: <NUMBER:1.0>
1: <NUMBER:3.0>
2: <NUMBER:1.0>
3: <STRING:"Fim">
Instructions:
0: GLOBAL 1
1: CONST 0
2: STORE 0
3: LOAD 0
4: CONST 1
5: LEQ
6: JUMPF 14
7: LOAD 0
8: PRINT_N
9: LOAD 0
10: CONST 2
11: ADD
12: STORE 0
13: JUMP 3
14: CONST 3
15: PRINT_S
16: HALT
Trace while running the code:
                    Globals: []
                      Stack: []
 GLOBAL 1
                    Globals: [<NIL>]
                      Stack: []
 CONST 0 '<NUMBER:1.0>
                    Globals: [<NIL>]
```

```
                    Stack: [<NUMBER:1.0>]
 STORE 0
                  Globals: [<NUMBER:1.0>]
                    Stack: []
 LOAD 0
                  Globals: [<NUMBER:1.0>]
                    Stack: [<NUMBER:1.0>]
 CONST 1 '<NUMBER:3.0>
                  Globals: [<NUMBER:1.0>]
                    Stack: [<NUMBER:1.0>, <NUMBER:3.0>]
 LEQ
                  Globals: [<NUMBER:1.0>]
                    Stack: [<BOOL:true>]
 JUMPF 14
                  Globals: [<NUMBER:1.0>]
                    Stack: []
 LOAD 0
                  Globals: [<NUMBER:1.0>]
                    Stack: [<NUMBER:1.0>]
 PRINT_N
1.0
                  Globals: [<NUMBER:1.0>]
                    Stack: []
 LOAD 0
                  Globals: [<NUMBER:1.0>]
                    Stack: [<NUMBER:1.0>]
 CONST 2 '<NUMBER:1.0>
                  Globals: [<NUMBER:1.0>]
                    Stack: [<NUMBER:1.0>, <NUMBER:1.0>]
 ADD
                  Globals: [<NUMBER:1.0>]
                    Stack: [<NUMBER:2.0>]
 STORE 0
                  Globals: [<NUMBER:2.0>]
                    Stack: []
 JUMP 3
                  Globals: [<NUMBER:2.0>]
                    Stack: []
 LOAD 0
                  Globals: [<NUMBER:2.0>]
                    Stack: [<NUMBER:2.0>]
 CONST 1 '<NUMBER:3.0>
                  Globals: [<NUMBER:2.0>]
                    Stack: [<NUMBER:2.0>, <NUMBER:3.0>]
 LEQ
```

```
                    Globals: [<NUMBER:2.0>]
                      Stack: [<BOOL:true>]
 JUMPF 14
                    Globals: [<NUMBER:2.0>]
                      Stack: []
 LOAD 0
                    Globals: [<NUMBER:2.0>]
                      Stack: [<NUMBER:2.0>]
 PRINT_N
2.0
                    Globals: [<NUMBER:2.0>]
                      Stack: []
 LOAD 0
                    Globals: [<NUMBER:2.0>]
                      Stack: [<NUMBER:2.0>]
 CONST 2 '<NUMBER:1.0>
                    Globals: [<NUMBER:2.0>]
                      Stack: [<NUMBER:2.0>, <NUMBER:1.0>]
 ADD
                    Globals: [<NUMBER:2.0>]
                      Stack: [<NUMBER:3.0>]
 STORE 0
                    Globals: [<NUMBER:3.0>]
                      Stack: []
 JUMP 3
                    Globals: [<NUMBER:3.0>]
                      Stack: []
 LOAD 0
                    Globals: [<NUMBER:3.0>]
                      Stack: [<NUMBER:3.0>]
 CONST 1 '<NUMBER:3.0>
                    Globals: [<NUMBER:3.0>]
                      Stack: [<NUMBER:3.0>, <NUMBER:3.0>]
 LEQ
                    Globals: [<NUMBER:3.0>]
                      Stack: [<BOOL:true>]
 JUMPF 14
                    Globals: [<NUMBER:3.0>]
                      Stack: []
 LOAD 0
                    Globals: [<NUMBER:3.0>]
                      Stack: [<NUMBER:3.0>]
 PRINT_N
3.0
                    Globals: [<NUMBER:3.0>]
```

```
                    Stack: []
 LOAD 0
                  Globals: [<NUMBER:3.0>]
                    Stack: [<NUMBER:3.0>]
 CONST 2 '<NUMBER:1.0>
                  Globals: [<NUMBER:3.0>]
                    Stack: [<NUMBER:3.0>, <NUMBER:1.0>]
 ADD
                  Globals: [<NUMBER:3.0>]
                    Stack: [<NUMBER:4.0>]
 STORE 0
                  Globals: [<NUMBER:4.0>]
                    Stack: []
 JUMP 3
                  Globals: [<NUMBER:4.0>]
                    Stack: []
 LOAD 0
                  Globals: [<NUMBER:4.0>]
                    Stack: [<NUMBER:4.0>]
 CONST 1 '<NUMBER:3.0>
                  Globals: [<NUMBER:4.0>]
                    Stack: [<NUMBER:4.0>, <NUMBER:3.0>]
 LEQ
                  Globals: [<NUMBER:4.0>]
                    Stack: [<BOOL:false>]
 JUMPF 14
                  Globals: [<NUMBER:4.0>]
                    Stack: []
 CONST 3 '<STRING:"Fim">
                  Globals: [<NUMBER:4.0>]
                    Stack: [<STRING:"Fim">]
 PRINT_S
Fim
                  Globals: [<NUMBER:4.0>]
                    Stack: []
 HALT
```

5.3 Exemplo 3

inputs/ifThen.mar

```
/*
   Programa que ilustra um if-then
*/
if 3 > 2 then {
   print "3 é maior que 2";
   print "Bingo!";
}
print "Fim";
```

Compilador com opção -debug

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

... no parsing errors
... no type errors
... code generation
Constant pool:
0: <NUMBER:3.0>
1: <NUMBER:2.0>
2: <STRING:"3 é maior que 2">
3: <STRING:"Bingo!">
4: <STRING:"Fim">
Generated assembly code:
0: CONST 0
1: CONST 1
2: GT
3: JUMPF 8
4: CONST 2
5: PRINT_S
6: CONST 3
7: PRINT_S
8: CONST 4
9: PRINT_S
10: HALT
Saving the constant pool and the bytecodes to inputs/ifThen.marbc
```

Máquina virtual, sem opção -debug

```
java marVM inputs/ifThen.marbc

3 é maior que 2
Bingo!
Fim
```

Máquina virtual, com opção -debug

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

Constant pool:
0: <NUMBER:3.0>
1: <NUMBER:2.0>
2: <STRING:"3 é maior que 2">
3: <STRING:"Bingo!">
4: <STRING:"Fim">
Instructions:
0: CONST 0
1: CONST 1
2: GT
3: JUMPF 8
4: CONST 2
5: PRINT_S
6: CONST 3
7: PRINT_S
8: CONST 4
9: PRINT_S
10: HALT
Trace while running the code:
                    Globals: []
                      Stack: []
 CONST 0 '<NUMBER:3.0>
                    Globals: []
                      Stack: [<NUMBER:3.0>]
 CONST 1 '<NUMBER:2.0>
                    Globals: []
                      Stack: [<NUMBER:3.0>, <NUMBER:2.0>]
 GT
                    Globals: []
                      Stack: [<BOOL:true>]
 JUMPF 8
                    Globals: []
```

```
                      Stack: []
 CONST 2 '<STRING:"3 é maior que 2">
                    Globals: []
                      Stack: [<STRING:"3 é maior que 2">]
 PRINT_S
3 é maior que 2
                    Globals: []
                      Stack: []
 CONST 3 '<STRING:"Bingo!">
                    Globals: []
                      Stack: [<STRING:"Bingo!">]
 PRINT_S
Bingo!
                    Globals: []
                      Stack: []
 CONST 4 '<STRING:"Fim">
                    Globals: []
                      Stack: [<STRING:"Fim">]
 PRINT_S
Fim
                    Globals: []
                      Stack: []
 HALT
```

5.4 Exemplo 4

inputs/ifThenElse.mar

```
/*
   Programa que ilustra um if-then-else
*/
if 3 > 2 then print "Then";
else print "Else";
print "Fim";
```

Compilador com opção -debug

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

... no parsing errors
... no type errors
... code generation
```

```
Constant pool:
0: <NUMBER:3.0>
1: <NUMBER:2.0>
2: <STRING:"Then">
3: <STRING:"Else">
4: <STRING:"Fim">
Generated assembly code:
0: CONST 0
1: CONST 1
2: GT
3: JUMPF 7
4: CONST 2
5: PRINT_S
6: JUMP 9
7: CONST 3
8: PRINT_S
9: CONST 4
10: PRINT_S
11: HALT
Saving the constant pool and the bytecodes to inputs/ifThenElse.marbc
```

Máquina virtual, sem opção -debug

```
java marVM inputs/ifThenElse.marbc

Then
Fim
```

Máquina virtual, com opção -debug

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

Constant pool:
0: <NUMBER:3.0>
1: <NUMBER:2.0>
2: <STRING:"Then">
3: <STRING:"Else">
4: <STRING:"Fim">
Instructions:
0: CONST 0
1: CONST 1
2: GT
3: JUMPF 7
```

```
4: CONST 2
5: PRINT_S
6: JUMP 9
7: CONST 3
8: PRINT_S
9: CONST 4
10: PRINT_S
11: HALT
Trace while running the code:
                    Globals: []
                      Stack: []
 CONST 0 '<NUMBER:3.0>
                    Globals: []
                      Stack: [<NUMBER:3.0>]
 CONST 1 '<NUMBER:2.0>
                    Globals: []
                      Stack: [<NUMBER:3.0>, <NUMBER:2.0>]
 GT
                    Globals: []
                      Stack: [<BOOL:true>]
 JUMPF 7
                    Globals: []
                      Stack: []
 CONST 2 '<STRING:"Then">
                    Globals: []
                      Stack: [<STRING:"Then">]
 PRINT_S
Then
                    Globals: []
                      Stack: []
 JUMP 9
                    Globals: []
                      Stack: []
 CONST 4 '<STRING:"Fim">
                    Globals: []
                      Stack: [<STRING:"Fim">]
 PRINT_S
Fim
                    Globals: []
                      Stack: []
 HALT
```

6 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 10/05/2023.

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.