# Compiladores, 2022/2023

Trabalho prático, parte 4
– Funções –

## 1 Introdução

Nesta parte do trabalho prático, vamos extender a linguagem mar para permitir a declaração e uso de funções. Devido à instrodução de funções, a linguagem mar terá de permitir mais dois tipos de instrução, nomeadamente:

- return
- expressão

Para além disso, um bloco vai passar a permitir declaração de variáveis, fazendo com que possamos ter variáveis locais.

Vamos ainda assumir que uma função só pode ser chamada após ter sido definida previamente no programa fonte. (Esta restrição simplifica a implementação da fase de identificação de símbolos.)

O tipo primitivo `nil` usado nos trabalhos anteriores deixa de ser válido.[1] Isto é, passamos apenas a ter 3 tipos primitivos: `number`, `bool` e `string`.

## 2 Alterações à gramática

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

[1] Porém, a instrução `NIL` da máquina virtual que empinha o valor `nil` no stack de execução continua a existir. Por outras palavras, existe um valor `nil` mas não existe o tipo de dados `nil`.

## 2.1 Programa

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

## 2.2 Declaração de função

Uma declaração de função é feita indicando o tipo de retorno (number, bool, ou string), seguido do nome da função, seguido de um parêntesis a abrir, seguido de uma lista de definição de argumentos separados por vírgula, seguido de um parêntesis a fechar, seguido de um bloco.

A indicação do tipo de retorno é opcional. Isto é, se não for especificado significa que a função não retorna um valor (equivalente a uma função void em C ou Java).

A definição de argumento é especificado indicando o tipo e o seu nome. O nome da função e argumento(s) obedecem às mesmas regras lexicais usadas para as declarações de variáveis. Eis um exemplo com a declaração de 2 funções:

```
hello( string s ) {
    print "Hello " + s;
}

number max( number a, number b ) {
    if a > b then return a;
    else return b;
}
```

## 2.3 Novo tipo de expressão: chamada de função

Uma chamada de função passa a ser considerado uma expressão. Uma chamada de função é especificada tal como em C e Java: nome da função, seguido de um parêntesis a abrir, seguido de uma lista (porventura vazia) de argumentos separados por vírgula, seguido de um parêntesis a fechar. Cada argumento, por sua vez, é também uma expressão.

```
hello("Maria")
max(a,b)
max(a, max(b,c))
```

A passagem de parâmetros é feita por valor. Isto é, uma cópia do valor do argumento é copiado para o parâmetro formal da função. Exemplos:

## 2.4 Novas instruções

### Return

Esta instrução permite terminar a execução de uma função, retornando um valor que deverá ser do mesmo tipo que foi especificado na definição da função. A sintaxe da instrução é a palavra `return` seguida de uma expressão, seguida de um ponto-e-vírgula. A expressão é opcional, porque podemos querer terminar a execução de uma função sem retornar qualquer valor (função void). Este comportamento é análoogo ao que acontece em C e Java. Exemplos:

```
return a+b;
return ;
```

### Expressão

Uma expressão seguida de um ponto-e-vírgula passa a ser considerado uma instrução válida. Exemplos:

```
hello("Maria");
5;
true and false;
```

### Bloco (alteração)

A sintaxe de um bloco passa a ser: chaveta a abrir, seguido de zero ou mais declarações de variáveis, seguido de zero ou mais instruções, seguido de uma chaveta a fechar. Exemplo:

```
{
    bool b1;
    bool b2;
    b1 = true; b2 = false;
    print b1 and b2;
}
```

# 3 Análise semântica

A fase da análise semântica deverá criar uma tabela de símbolos para guardar informação relevante para todo o tipo de símbolos (variáveis globais, variáveis locais, nomes de funções, argumentos de funções). Deverá também fazer a verificação de tipos (type checking).

Quando uma função é definida como retornando um valor, devem garantir que de facto a função retorna algo.

No trabalho anterior, a análise semântica verificava se uma variável tinha sido inicializada antes do seu uso. Nesta fase do trabalho, essa verificação continua a ser exigida, exceptuando a seguinte situação: no caso de o corpo de uma função tentar aceder a uma variável global, a verificação não é feita (pois tal verificação não é trivial). Porém, em tempo de execução a marVM deverá fazer tal verificação, e no caso da variável não estar inicializada, deverá reportar um *runtime error*.

## 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.

```
1  number n;
2  bool b;
3
4  number xpto( number a, string s ) {
5     if a > 5 then return a;
6     if a > 3 then return s;
7     return a+1;
8  }
9
10 xpto2() {
11    print "ola";
12    return 53;
13    return;
14 }
15
16 xpto3() {
17    print "ola";
18 }
19
20 number xpto4( number a ) {
21    if a > 0 then return a;
```

```
22  }
23
24  // main program starts below
25  xpto = 4;
26  n = fact(5);
27  n = xpto(10);
28  n = xpto(10,"ola");
29  b = xpto(10,"ola");
30  n = xpto(10,true);
```

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

```
... parsing done
line 6:24 error: xpto should return a value of type number, not string
line 12:10 error: xpto2 should not return a value
line 22:0 error: missing return in function xpto4
line 25:0 error: xpto is not a variable
line 26:4 error: fact is not defined
line 27:4 error: xpto has 2 arguments, not 1
line 29:0 error: cannot assign an expression of type number to a variable of type bool
line 30:12 error: actual parameter 'true' is of type bool, but type string expected
... identification and type checking done
inputs/func-erros.mar has 8 semantic errors
```

Os exemplos acima estão na tutoria. Ver ficheiros:

- `inputs/func-erros.mar`
- `output/func-erros-marCompiler.txt`

# 4 Novas instruções da máquina virtual

As instruções LOAD e STORE usadas no trabalho anterior deixam de existir, sendo substituídas por 4 novas instruções análogas: LOADG e STOREG para variáveis globais, e LOADL e STOREL para variáveis locais.

Passa tambem a haver as instruções LOCAL, POP, CALL, e RETURN, todas elas com um argumento inteiro. A descrição de cada uma destas novas instruções aparece na tabela abaixo.

| **OpCode** | Argumento | Descrição |
|---|---|---|
| LOCAL | inteiro $n$ | aloca (empilha) $n$ posições no stack, que vão servir para armazenar variáveis locais. |
| POP | inteiro $n$ | desempilha $n$ elementos do stack |
| LOADG | inteiro $addr$ | empilha $Globals[addr]$ no stack. |
| STOREG | inteiro $addr$ | faz pop do stack e guarda o valor em $Globals[addr]$ |
| LOADL | inteiro $addr$ | empilha o conteúdo de $Stack[LB + addr]$ no stack. |
| STOREL | inteiro $addr$ | faz pop do stack e guarda o valor em $Stack[LB + addr]$ |
| CALL | inteiro $addr$ | guarda o estado da máquina virtual (isto é: empilha o valor do registo $LB$, e empilha o endereço retorno (O endereço da instrução imediatamente após o CALL)) |
| RETURN | inteiro $n$ | faz $x = pop()$, desempinha o espaço reservado para as variáveis locais usadas pela função, restaura o estado da máquina virtual, desempilha os $n$ argumentos do stack, e depois empilha $x$ |

Durante a fase da análise semântica ou durante a geração de código, devem atribuir um endereço a cada variável, seja ela global ou local.

# 5 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 24/05/2023.

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.

# 6 Exemplos obtidos pela minha implementação

Apresento de seguida alguns exemplos de inputs de programas válidos. O output gerado pelo compilador e pela execução da máquina virtual em modo debug estão disponíveis em ficheiros de texto na tutoria.

## Exemplo 1: inputs/local-vars.mar

```
number g1;
number g2;
g1 = 10;
g2 = 20;
{
    number a;
    number b;
    a = 1;
    {
        number c;
        c = 2;
        {
            number d;
            number e;
            d = 3;
            {
                number f;
                f = 4;
            }
            e = 5;
            d = g1 + a;
            print d;
        }
   }
   b = 6;
   {
        number g;
        g = 7;
   }
}
```

Após compilado e executado pela marVM deve dar o seguinte output:

```
11.0
```

Os outputs do marCompiler e marVM com a opção debug estão na tutorian em ficheiros de texto com estes nomes:

- outputs/local-vars-marCompiler-debug.txt
- outputs/local-vars-marVM-debug.txt

## Exemplo 2: inputs/func-args0.mar

```
hello() {
   print "Hello";
}

print 1;
hello();
print 2;
hello();
print 3;
```

Após compilado e executado pela marVM deve dar o seguinte output:

```
1.0
Hello
2.0
Hello
3.0
```

Os outputs do marCompiler e marVM com a opção debug estão na tutorian em ficheiros de texto com estes nomes:

- outputs/func-args0-marCompiler-debug.txt
- outputs/func-args0-marVM-debug.txt

## Exemplo 3: inputs/func-hello.mar

```
hello( string s ) {
   print "Hello " + s;
}

// main program starts below
hello("Maria");
```

Após compilado e executado pela marVM deve dar o seguinte output:

```
Hello Maria
```

Os outputs do marCompiler e marVM com a opção debug estão na tutorian em ficheiros de texto com estes nomes:

- outputs/func-hello-marCompiler-debug.txt
- outputs/func-hello-marVM-debug.txt

## Exemplo 4: inputs/func-hello2.mar

```
string mau( string s ) {
   string s2;
   s2 = "!";
   return "Mau " + s + s2;
}

hello( string s ) {
   string s1;
   string s2;
   s2 = "!";
   s1 = "Hello " + s + s2;
   print s1;
   s = mau(s);
   print s;
}

// main program starts below
hello("Maria");
```

Após compilado e executado pela marVM deve dar o seguinte output:

```
Hello Maria!
Mau Maria!
```

Os outputs do marCompiler e marVM com a opção debug estão na tutorian em ficheiros de texto com estes nomes:

- outputs/func-hello2-marCompiler-debug.txt
- outputs/func-hello2-marVM-debug.txt

## Exemplo 5: inputs/globalVar-in-func.mar

```
string g;
string g2;

xpto() {
   print "xpto " + g2;
}

hello() {
   string s;
   s = "Hello";
   print "Hello " + g;
}

g = "Maria";
hello();
xpto();      // xpto() references g2 and g2 is not initialized,
             // must give runtime error
```

Após compilado e executado pela marVM deve dar o seguinte output:

```
Hello Maria
runtime error: marVM accessed a nil value.
               global variable might not be initialized.
```

Os outputs do marCompiler e marVM com a opção debug estão na tutorian em ficheiros de texto com estes nomes:

- outputs/globalVar-in-func-marCompiler-debug.txt
- outputs/globalVar-in-func-marVM-debug.txt

## Exemplo 6: inputs/func-max.mar

```
number x;
number y;
number z;

number max( number a, number b ) {
   if a > b then return a;
   else return b;
}

x = 3;
y = 5;
z = max(x,y);
print z;
```

Após compilado e executado pela marVM deve dar o seguinte output:

```
5.0
```

Os outputs do marCompiler e marVM com a opção debug estão na tutorian em ficheiros de texto com estes nomes:

- outputs/func-max-marCompiler-debug.txt
- outputs/func-max-marVM-debug.txt

## Exemplo 7: inputs/func-max3.mar

```
number x;
number y;
number z;

number max( number a, number b ) {
   if a > b then return a;
   else return b;
}

number max3( number a, number b, number c) {
    return max(a, max(b,c));
}

// main program starts below
x = 3;
y = 5;
z = 4;
print max3(x,y,z);
```

Após compilado e executado pela marVM deve dar o seguinte output:

```
5.0
```

Os outputs do marCompiler e marVM com a opção debug estão na tutorian em ficheiros de texto com estes nomes:

- outputs/func-max3-marCompiler-debug.txt
- outputs/func-max3-marVM-debug.txt

## Exemplo 8: inputs/func-factorial.mar

```
number fact( number n ) {
   if n == 0 then return 1;
   return n * fact(n-1);
}

print fact(3);
```

Após compilado e executado pela marVM deve dar o seguinte output:

```
6.0
```

Os outputs do marCompiler e marVM com a opção debug estão na tutorian em ficheiros de texto com estes nomes:

- outputs/func-factorial-marCompiler-debug.txt
- outputs/func-factorial-marVM-debug.txt