Compiladores, 2024/2025

Trabalho prático, parte 3
– Tuga: Funções –

Fernando Lobo

1 Introdução

Nesta parte do trabalho prático, vamos extender a linguagem Tuga para permitir a declaração e uso de funções, bem como a declaração e uso de variáveis locais.

Como consequência, a tabela de símbolos terá de suportar enquadramentos encadeados (nested scopes) como foi explicado nas aulas teóricas e ser capaz de guardar informação relevante para todo o tipo de símbolos (variáveis globais, variáveis locais, nomes de funções, argumentos de funções).

Devido à introdução de funções, a linguagem Tuga terá de permitir mais dois tipos de instrução, nomeadamente:

- chamada de função
- return

Por outro lado, teremos de acrescentar novas instruções à máquina virtual de modo a poder suportar a execução de funções.

A chamada de uma função poderá aparecer no programa fonte antes da declaração da função, sem que isso cause qualquer erro.

2 Alterações à linguagem

2.1 Programa

Um programa em Tuga passa agora a ser constituído por uma sequência de zero ou mais declarações de variáveis globais, seguido de uma sequência de uma ou mais

declarações de funções, sendo que uma dessas funções terá de se chamar `principal`, o ponto de entrada para a execução do programa.

2.2 Declaração de função

Uma declaração de função é feita indicando a palavra reservada `funcao`, seguido do nome da função, seguido de um parêntesis curvo a abrir, seguido de uma lista de definição de argumentos separados por vírgula, seguido de um parêntesis curvo a fechar, seguido do caracter `:`, seguido do tipo de retorno (que pode ser um dos 4 tipos base da linguagem: `inteiro`, `real`, `booleano` ou `string`), seguido de um bloco.

Há porém uma excepção para o caso da função não retornar um valor (o equivalente a uma função void em C ou Java). Nesse caso não se coloca o caracter `:`, nem o tipo de retorno. Neste caso dizemos que a função é do tipo `vazio`.

A definição de argumento é especificada indicando o nome do argumento, seguido do caracter `:`, seguido do tipo do argumento. 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:

```
funcao hello( s: string )
inicio
    escreve "Hello " + s;
fim

funcao max( a: inteiro, b: inteiro ): inteiro
inicio
    se (a > b) retorna a;
    senao retorna b;
fim
```

O compilador deverá fazer a verificação de tipos. Quando uma função é definida como retornando um valor de determinado tipo, o compilador deve garantir que de facto a função retorna um valor, e que esse valor tem um tipo de dados consistente com o tipo de dados declarado.

Consistência de tipos significa que os tipos têm de ser iguais, havendo apenas a excepção de algo declarado como sendo do tipo `real` poder receber um valor do tipo `inteiro`, devendo neste caso o compilador emitir código para fazer a conversão usando a instrução `itod` da máquina virtual.

No caso da função não retornar um valor, a existência de uma instrução de return (`retorna` em Tuga) explícito por parte do programador é opcional.

2.3 Novas instruções

2.3.1 Chamada de função

Uma chamada de função seguido de um ponto e vírgula, passa a ser uma instrução válida da linguagem Tuga, se essa função estiver declarada como não retornando valor. Se a função for definida como retornando um valor, então deixa de ser uma instrução válida. Por exemplo, mediante as declarações de funções mencionadas anteriormente, apenas a 1ª linha do seguinte excerto de código corresponde a uma instrução válida:

```
hello("Maria");     // é válido porque hello não retorna valor
max(a,b);           // não é válido porque max retorna um valor do tipo inteiro
```

Note porém, que `max(a,b)` é considerado uma expressão válida (ver Secção 2.4 adiante).

Retorna

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 `retorna` 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. Este comportamento é análogo ao que acontece em C e Java. Exemplos:

```
retorna a+b;
retorna ;
```

Bloco (alteração)

A sintaxe de um bloco passa a ser: palavra reservada `inicio`, seguido de zero ou mais declarações de variáveis, seguido de zero ou mais instruções, seguido de palavra reservada `fim`. Exemplo:

```
inicio
    b1, b2: booleano;
    x: inteiro;
    b1 <- verdadeiro;
    b2 <- falso;
    x <- 5;
    escreve b1 e b2;
fim
```

2.4 Novo tipo de expressão

Chamada de função

Uma chamada de função passa a ser uma expressão válida. 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. Exemplo de expressão válida:

```
max(a,max(b,c))
```

Durante a verificação de tipos, deve ser feita também a verificação de consistência entre o tipo de dados dos argumentos/parâmetros formais e dos argumentos/parâmetros actuais, bem como do seu número. No exemplo dado, a função max foi declarada como tendo 2 argumentos do tipo inteiro. Logo, quando a função é chamada terá de ser garantido que lhe estamos a passar duas (e apenas duas) expressões, e cada uma delas terá de ser de um tipo consistente com a definição.

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

3 Máquina Virtual S

De modo a suportar a execução de funções, a máquina virtual vai passar a ter um registo especial chamado FP (Frame Pointer) que aponta para a base do *frame* em execução. Para além disso devem ser acrescentadas 7 novas instruções à máquina virtual, todas elas contendo um argumento inteiro. O nome e descrição destas novas instruções é apresentado na tabela abaixo.

Opc	**Nome**	**Arg.**	**Descrição**
46	lalloc	inteiro n	*local memory allocation*: aloca n posições no topo do stack para armazenar variáveis locais. Essas n posições de memória ficam inicializadas com o valor NIL.
47	lload	inteiro *addr*	*local load*: empilha o conteúdo de $Stack[FP + addr]$ no stack.
48	lstore	inteiro *addr*	*local store*: faz pop do stack e guarda o valor em $Stack[FP + addr]$
49	pop	inteiro n	desempilha n elementos do stack

50	call	inteiro *addr*	Cria um novo frame no stack, que passará a ser o frame currente. Para tal terá de guardar o estado da máquina virtual (isto é, empilha o valor do registo FP, actualiza o valor de FP para apontar para a base do novo frame, e empilha o endereço de retorno: o endereço da instrução imediatamente após o call). Depois actualiza o *instruction pointer* IP de modo a que a próxima instrução a ser executada seja aquela que se encontra na posição *addr* do array de instruções.
51	retval	inteiro n	*return from non-void function*: faz $x = pop()$, desempilha 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
52	ret	inteiro n	*return from void function*: desempilha o espaço reservado para as variáveis locais usadas pela função, restaura o estado da máquina virtual, e desempilha os n argumentos do stack

4 Reporte de erros

Tal como no 2º trabalho, vamos querer reportar os eventuais erros semânticos que possam haver. Aa mensagens erro devem indicar a linha onde o erro ocorre, seguido de mensagem apropriado. As mensagens de erro devem vir ordenadas pelo número da linha onde o erro ocorre. (Ver exemplo A no apêndice).

No caso de haver erros lexicais ou de parsing, o programa deve limitar-se a dizer `Input tem erros lexicais` ou `Input tem erros de parsing`, consoante o caso, e terminar a execução.

5 Requisitos

- O trabalho deve ser feito em Java usando o ANTLR 4, e submetido ao mooshak.
- Devem submeter um ficheiro zip que deverá conter uma pasta chamada `src` onde está todo o código que desenvolveram.
- A gramática deve chamar-se `Tuga.g4` e deverá estar na pasta `src`
- A pasta `src` deverá conter o código gerado pelo ANTLR, supostamente numa pasta/package chamada `Tuga`

- Nota importante: NÃO DEVEM incluir o ficheiro `antlr-4.13.2-complete.jar` na vosso zip. O servidor do mooshak já lá tem esse ficheiro.
- No código apenas pode haver um ficheiro java que tenha o método `main`. Não pode haver mais nenhum `main`, mesmo que esteja comentado. O nome do ficheiro que tem o método main deve chamar-se `TugaCompileAndRun.java`
- O vosso código é submetido ao mooshak, pelo que deverá poder ler o input a partir do *standard input*. Não obstante, o código deve estar também preparado para poder receber o input (nome do ficheiro tuga a compilar, e enventuais flags) a partir de argumentos passados à função `main`.
- O método `main` deverá definir as seguintes 2 flags que servem para controlar o modo como a emissão de erros é feita.

  ```
  boolean showLexerErrors = false;
  boolean showParserErrors = false;
  ```

  Quando o código é submetido ao mooshak, as flags devem ter o valor `false` de modo a não mostrar as mensagens de erro. Porém, se o valor das flags for alterado para `true`, o vosso programa é suposto emitir os respectivos erros com mensagens apropriadas.
- Para efeitos de submissão ao mooshak:
  - se o input tiver erros lexicais, o programa deverá enviar a seguinte mensagem para o output e terminar: `Input tem erros lexicais`
  - se não tiver erros lexicais, mas tiver erros de parsing, o programa deverá enviar a seguinte mensagem para o output e terminar: `Input tem erros de parsing`
  - se não tiver erros lexicais nem de parsing, mas tiver erros semânticos, o programa deverá enviar mensagens de erro apropriadas para o output e terminar (ver exemplos).
  - se não tiver erros lexicais, nem de parsing, nem semânticos, o programa deverá gerar bytecodes de acordo com a escpecificação da máquina virtual e guardá-los num ficheiro com o nome `bytecodes.bc`. Após ser gerado, esse ficheiro deverá ser lido de imediato e a máquina virtual deverá executar os bytecodes.

6 Sobre o Mooshak

- http://deei-mooshak.ualg.pt/~flobo/, concurso Comp25, problema C.
- Não use caracteres acentuados no código, nem mesmo nos comentários, uma vez que o mooshak poderá dar erros por causa disso.

- O compilador de Java instalado no Mooshak é o openjdk version 21.0.6. Se no vosso desenvolvimento usarem um JDK mais recente, recomendo que configurem o vosso IDE para não usar funcionalidades do Java posteriores à versão que está no Mooshak. (Se usarem o IntelliJ vão a 'Project Structure' → 'Language Level', e escolham 21.

7 Validação e avaliação

Os trabalhos serão validados e avaliados após a discussão individual a ter lugar nos dias 20, 22, 23, e 27 de Maio, em horário que será anunciado através da tutoria eletrónica.

Os trabalhos serão avaliados de acordo com a clareza e qualidade do código implementado, pela correcta implementação do compilador e máquina virtual, pelo cumprimento dos requisitos acima enunciados, e pelo desempenho individual durante a validação/discussão.

8 Prazo de entrega

O trabalho deve ser realizado em grupo, de acordo com as inscrições em grupo escolhida na tutoria. Deverão submeter o vosso código ao mooshak até às 23:59 do dia 18/Mai/2025. (**NOTA:** Não serão aceites entregas fora de prazo, nem que seja por um só minuto. Não devem deixar a submissão para os últimos instantes. Podem submeter o vosso trabalho várias vezes. Apenas será considerada a última submissão que tiver 'Accepted', ou no caso de não terem Accepted contará a última submissão.

Apêndice A Exemplos de inputs/outputs para o mooshak

Exemplo A

Input

```
1  funcao sqr( x: inteiro ): inteiro
2  inicio
3      retorna x * x;
4  fim
5
6  funcao sqrsum( a: inteiro, b: inteiro ): inteiro
7  inicio
8      s: inteiro;
9      s <- sqr(a + b);
10     retorna s;
11 fim
12
13 funcao principal()
14 inicio
15     escreve sqrsum(3,2);
16 fim
```

Output

```
*** Constant pool ***
*** Instructions ***
0: call 14
1: halt
2: lload -1
3: lload -1
4: imult
5: retval 1
6: lalloc 1
7: lload -2
8: lload -1
9: iadd
10: call 2
11: lstore 2
12: lload 2
13: retval 2
```

```
14: iconst 3
15: iconst 2
16: call 6
17: iprint
18: ret 0
*** VM output ***
25
```

Exemplo B

Input

```
1  funcao principal()
2  inicio
3     escreve fact(3);
4  fim
5
6  funcao fact( n: inteiro ): inteiro
7  inicio
8     se (n igual 0) retorna 1;
9     retorna n * fact(n-1);
10 fim
```

Output

```
*** Constant pool ***
*** Instructions ***
0: call 2
1: halt
2: iconst 3
3: call 6
4: iprint
5: ret 0
6: lload -1
7: iconst 0
8: ieq
9: jumpf 12
10: iconst 1
11: retval 1
12: lload -1
13: lload -1
14: iconst 1
15: isub
16: call 6
17: imult
18: retval 1
*** VM output ***
6
```

Exemplo C

Input

```
funcao principal()
inicio
   aa, bb: inteiro;
   aa <- 1;
   inicio
      cc: inteiro;
      cc <- 2;
      inicio
         dd, ee: inteiro;
         dd <- 3;
         inicio
            ff: inteiro;
            ff <- 4;
         fim
         ee <- 5;
      fim
   fim
   bb <- 6;
   inicio
      gg: inteiro;
      gg <- 7;
      escreve gg;
   fim
fim
```

Output

```
*** Constant pool ***
*** Instructions ***
0: call 2
1: halt
2: lalloc 2
3: iconst 1
4: lstore 2
5: lalloc 1
6: iconst 2
7: lstore 4
8: lalloc 2
9: iconst 3
10: lstore 5
```

```
11: lalloc 1
12: iconst 4
13: lstore 7
14: pop 1
15: iconst 5
16: lstore 6
17: pop 2
18: pop 1
19: iconst 6
20: lstore 3
21: lalloc 1
22: iconst 7
23: lstore 4
24: lload 4
25: iprint
26: pop 1
27: pop 2
28: ret 0
*** VM output ***
7
```

Exemplo D

Input

```
funcao func(): inteiro
inicio
   x, y: inteiro ;
   inicio
      a, b, c: inteiro;
      escreve "Oi";
      retorna 1;
   fim
   escreve "Ai";
fim

funcao principal()
inicio
   x: inteiro;
   x <- func();
fim
```

Output

```
*** Constant pool ***
0: "Oi"
1: "Ai"
*** Instructions ***
0: call 10
1: halt
2: lalloc 2
3: lalloc 3
4: sconst 0
5: sprint
6: iconst 1
7: retval 0
8: sconst 1
9: sprint
10: lalloc 1
11: call 2
12: lstore 2
13: pop 1
14: ret 0
*** VM output ***
```

```
Oi
```

Exemplo E

Input

```
1  funcao sqr( x: real ): real
2  inicio
3      retorna x * x;
4  fim
5
6  funcao sqrsum( a: real, b: real ): real
7  inicio
8      s: real;
9      s <- a + b;
10     retorna sqr(s);
11 fim
12
13 funcao principal()
14 inicio
15     escreve sqrsum(3,2);
16 fim
```

Output

```
*** Constant pool ***
*** Instructions ***
0: call 14
1: halt
2: lload -1
3: lload -1
4: dmult
5: retval 1
6: lalloc 1
7: lload -2
8: lload -1
9: dadd
10: lstore 2
11: lload 2
12: call 2
13: retval 2
14: iconst 3
15: itod
16: iconst 2
17: itod
18: call 6
```

```
19: dprint
20: ret 0
*** VM output ***
25.0
```

Exemplo F

Input

```
1  funcao hello( s: string )
2  inicio
3     s <- s + " SILVA";
4     escreve "hello(): " + s;
5  fim
6
7  funcao f( a: inteiro, b: inteiro ): real
8  inicio
9      retorna a + b;
10 fim
11
12 funcao principal()
13 inicio
14     s: string;
15     s <- "Maria";
16     hello(s);
17     escreve "principal(): " + s;
18     escreve f(4,5);
19 fim
```

Output

```
*** Constant pool ***
0: " SILVA"
1: "hello(): "
2: "Maria"
3: "principal(): "
*** Instructions ***
0: call 16
1: halt
2: lload -1
3: sconst 0
4: sconcat
5: lstore -1
6: sconst 1
7: lload -1
8: sconcat
9: sprint
10: ret 1
11: lload -2
```

```
12: lload -1
13: iadd
14: itod
15: retval 2
16: lalloc 1
17: sconst 2
18: lstore 2
19: lload 2
20: call 2
21: sconst 3
22: lload 2
23: sconcat
24: sprint
25: iconst 4
26: iconst 5
27: call 11
28: dprint
29: pop 1
30: ret 0
*** VM output ***
hello(): Maria SILVA
principal(): Maria
9.0
```

Exemplo G

Input

```
zzz: inteiro;

funcao principallll()
inicio
    n : inteiro;
    n : booleano;
    n <- fun(1,2,3);
    fun(1,2);
    hello("Maria");
    hello(5);
    n <- hello("Maria");
    n <- hello("Maria") + 44;
    fun <- 8;
    n <- misterio(n);
fim

funcao hello( s: string )
inicio
   escreve "Hello " + s;
fim

funcao zzz( x: inteiro ): real
inicio
   retorna x + 1;
fim

funcao hello()
inicio
   escreve "Hello";
fim

funcao fun( x: inteiro, y: inteiro ): inteiro
inicio
   b: booleano;
   b <- hello;
   retorna x + y;
fim
```

Output

```
erro na linha 6: 'n' ja foi declarado
erro na linha 7: 'fun' requer 2 argumentos
erro na linha 8:  valor de 'fun' tem de ser atribuido a uma variavel
erro na linha 10: '5' devia ser do tipo string
erro na linha 11: operador '<-' eh invalido entre inteiro e vazio
erro na linha 12: operador '+' eh invalido entre vazio e inteiro
erro na linha 13: 'fun' nao eh variavel
erro na linha 14: 'misterio' nao foi declarado
erro na linha 22: 'zzz' ja foi declarado
erro na linha 27: 'hello' ja foi declarado
erro na linha 35: 'hello' nao eh uma variavel
erro na linha 38: falta funcao principal()
```