Compiladores, 2024/2025

Trabalho prático, parte 2
– Tuga: variáveis globais, instruções de controlo de fluxo –

Fernando Lobo

1 Introdução

Neste trabalho vamos extender a linguagem Tuga para permitir declarações de variáveis globais, instrução de afectação, e instruções de controlo de fluxo. Especificamente, para além da instrução *escreve* existente no trabalho 1, passaremos a ter as seguintes instruções:

- afectação
- bloco
- enquanto
- se / se-senao
- vazia

2 Alterações à linguagem

2.1 Programa

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

2.2 Declaração de variáveis

A declaração de variáveis é feita indicando uma sequência de nomes de variáveis separados por uma vírgula, seguido do caracter `:`, seguido do tipo de dados que pode

ser um dos 4 tipos de dados suportados pela linguagem Tuga (`booleano`, `inteiro`, `real`, `string`), que passam a ser palavras reservadas [1], 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. Exemplo:

```
x, y, soma: inteiro;
b: booleano;
```

Na fase de análise semântica deve ser feito a verificação de tipos. Para além disso, deve ser reportado um erro se fizermos referência a uma variável não declarada, ou se tentarmos declarar uma variável que já foi declarada anteriormente.

Uma variável que não é inicializada fica armazenada na memória da máquina virtual com o valor `NULO`, e o acesso a uma variável nessas condições deve gerar um erro de runtime pela máquina virtual.

2.3 Novo tipo de expressão

Uma vez que é permitido variáveis, também é obviamente permitido ter variáveis no contexto de expressões. Por exemplo, `n + 1` passa a ser uma expressão sintáticamente válida na linguagem Tuga. Na fase de verificação de tipos (*type checking*) deve usar as regras especificadas no trabalho anterior, tendo presente que o tipo de uma variável é obtido através da sua declaração.

2.4 Novos tipos de instrução

Afectação

A instrução de afectação permite atribuir um valor a uma variável. A sintaxe é parecida à da linguagem C ou Java, mas usa-se o sinal `<-` como sinal da afectação, em vez do sinal de `=` usado em C e Java. A sintaxe é: nome de variável, seguido do sinal de afectação `<-`, 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. Ao invés das chavetas usadas em C e Java, o início

[1] Palavras reservadas não podem ser usadas como nomes de variáveis, funções, etc.

do bloco é especificado com a palavra reservada `inicio` e o fim do bloco é especificado com a palavra reservada `fim`. O contéudo do bloco é uma sequência de zero ou mais instruções. Exemplo:

```
inicio
    escreve x;
    x <- x+1;
fim
```

Enquanto

A instrução *enquanto* serve para fazer ciclos, e é análoga ao ciclo while em C e Java. A instrução deve começar com a palavra reservada `enquanto`, seguido de um parentesis curvo a abrir, seguido de uma expressão, seguido de uma parentesis curvo a fechar, seguido de uma instrução. Exemplo:

```
enquanto (x < 10)
inicio
    escreve x;
    x <- x+1;
fim
```

Na fase de análise semântica deve ser garantido que a expressão de controlo do ciclo *escreve* tem de ser do tipo booleano.

Se / Se-Senão

Esta instrução é análoga â instrução *If / If-Else* do C e Java. A instrução começa com a palavra reservada `se`, seguido de um parentesis curvo a abrir, seguido de uma expressão, seguido de uma parentesis curvo a fechar seguido de uma instrução, e opcionalmente poderá ter a parte `senao`, em que aparece a palavra reservada `senao` seguido de uma instrução. Exemplo:

```
se (x < 10)
     escreve "ola";
senao
     inicio
         escreve x;
         x <- x+1;
     fim
```

Na fase de análise semântica deve ser garantido que a expressão do *se* tem de ser do tipo booleano.

Vazia

Um ponto-e-virgula é uma instrução. (É uma instrução vazia que não faz nada.)

3 Máquina Virtual S

O vosso compilador de Tuga deverá gerar bytecodes para a máquina virtual chamada S (versão mini 2). Esta nova versão da máquina virtual é uma extensão daquela que fizeram no 1º trabalho. Para além da *constant pool* e do *runtime stack*, vai também incluir uma *memória* para armazenar as variáveis globais, bem como mais algumas instruções necessárias para alocar posições de memória, fazer leitura e escrita da memória, e ainda instruções que efectuam saltos condicionais e incondicionais.

A especificação desta nova versão da máquina virtual S está disponível num ficheiro à parte disponível na tutoria: `SVM-mini-2.pdf`

4 Reporte de erros

Ao invés do que aconteceu no 1º trabalho, desta vez vamos querer reportar os eventuais erros semânticos que possam haver. Caso haja um erro de consistência de tipos numa expressão, a mensagem de erro deve indicar a linha onde o erro ocorre, seguido de mensagem apropriada (ver exemplos 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 B.
- 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 apenas serão validados e avaliados após a discussão individual a ter lugar nas 2 últimas semanas de aulas do semestre. Faz-se notar que até ao final do semestre haverá mais um trabalho para além deste.

Não basta terem accepted no mooshak para automaticamente terem boa nota. Aliás, é possível terem boa nota mesmo que não tenham accepted no mooshak.

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 30/Abr/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

```
x,y,soma: inteiro;
b: booleano;
r: real;
s: string;
b <- 99;
r <- 3;
s <- 88;
x <- 33;
x <- 3.14;
escreve x;
```

Output

```
erro na linha 5: operador '<-' eh invalido entre booleano e inteiro
erro na linha 7: operador '<-' eh invalido entre string e inteiro
erro na linha 9: operador '<-' eh invalido entre inteiro e real
```

Exemplo B

Input

```
x,y,soma,y: inteiro;
b,x: booleano;
escreve 5;
```

Output

```
erro na linha 1: variavel 'y' ja foi declarada
erro na linha 2: variavel 'x' ja foi declarada
```

Exemplo C

Input

```
se ("esta a chover")
    escreve "abre o guarda-chuva";
senao
    escreve "tuga eh fixe";
```

Output

```
erro na linha 1: expressao de 'se' nao eh do tipo booleano
```

Exemplo D

Input

```
i: inteiro;
i <- 1;
enquanto (i)
inicio
     escreve i;
     i <- i + 1;
fim
escreve "tchau";
```

Output

```
erro na linha 3: expressao de 'enquanto' nao eh do tipo booleano
```

Exemplo E

Input

```
escreve "ola";
;;;
```

Output

```
*** Constant pool ***
0: "ola"
*** Instructions ***
0: sconst 0
1: sprint
2: halt
*** VM output ***
ola
```

Exemplo F

Input

```
x: inteiro;
x <- 6;
se (x % 2 igual 0)
    escreve x + " eh par";
senao
    escreve x + " eh impar";
```

Output

```
*** Constant pool ***
0: " eh par"
1: " eh impar"
*** Instructions ***
0: galloc 1
1: iconst 6
2: gstore 0
3: gload 0
4: iconst 2
5: imod
6: iconst 0
7: ieq
8: jumpf 15
9: gload 0
10: itos
11: sconst 0
12: sconcat
13: sprint
14: jump 20
15: gload 0
16: itos
17: sconst 1
18: sconcat
19: sprint
20: halt
*** VM output ***
6 eh par
```

Exemplo G

Input

```
i: inteiro;
i <- 1;
enquanto (i <= 5)
inicio
     escreve i;
     i <- i + 1;
fim
escreve "tchau";
```

Output

```
*** Constant pool ***
0: "tchau"
*** Instructions ***
0: galloc 1
1: iconst 1
2: gstore 0
3: gload 0
4: iconst 5
5: ileq
6: jumpf 14
7: gload 0
8: iprint
9: gload 0
10: iconst 1
11: iadd
12: gstore 0
13: jump 3
14: sconst 0
15: sprint
16: halt
*** VM output ***
1
2
3
4
5
tchau
```

Exemplo H

Input

```
i,j,k: inteiro;
b: booleano;

b <- verdadeiro;
escreve b;
i <- 5;
escreve i + j;
```

Output

```
*** Constant pool ***
*** Instructions ***
0: galloc 3
1: galloc 1
2: tconst
3: gstore 3
4: gload 3
5: bprint
6: iconst 5
7: gstore 0
8: gload 0
9: gload 1
10: iadd
11: iprint
12: halt
*** VM output ***
verdadeiro
erro de runtime: tentativa de acesso a valor NULO
```