Compiladores, 2024/2025

Trabalho prático, parte 1
– Linguagem Tuga: expressões, consistência de tipos –

Fernando Lobo

1 Introdução

Neste e nos restantes trabalhos da disciplina iremos desenvolver um compilador para uma linguagem imperativa chamada Tuga. Para começar, iremos nos restringir a fazer a compilação de programas que apenas permitem um tipo de instrução: escreve.

Nos restantes trabalhos iremos acrescentar outro tipo de instruções e outro tipo de funcionalidades à linguagem.

Caso o input não tenha erros, o vosso compilador deve gerar bytecodes para uma máquina virtual baseada num stack, que também deverá ser implementada por vós.

2 Sintaxe da linguagem Tuga

Um programa em Tuga é uma sequência de uma ou mais instruções, cada qual consistindo na palavra reservada `escreve`, seguido de uma expressão, seguido de um ponto-e-vírgula.

A expressão pode envolver operadores aritméticos, lógicos, e também de concatenação de strings. A sintaxe de uma expressão pode ser descrita de forma indutiva através das seguintes regras.

1. um literal (um número inteiro, um número real, uma string, verdadeiro, falso) é por definição uma expressão.
2. um operador unário seguido de uma expressão, é uma expressão.
3. uma expressão, seguida de um operador binário, seguido de outra expressão, é também uma expressão.

4. Um parêntesis a abrir, seguido de uma expressão, seguido de um parêntesis a fechar, é também uma expressão.

A linguagem Tuga suporta 4 tipos de dados: booleano, inteiro, real, string. O tipo booleano suporta os valores `verdadeiro` e `falso`. O tipo inteiro suporta inteiros. O tipo real suporta números reais (equivalente a um double em Java). O tipo string suporta sequência de caracteres delimitados por aspas.

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
```

2.1 Exemplo de um programa em Tuga

```
/*
   Exemplo de programa em Tuga
*/
escreve 1 + 2 * 3;
escreve 1 + 2.0 * 3;
escreve 7 % (1 + 4);
escreve verdadeiro e (5 < 3.14159);
escreve "pi = " + 3.14159;
escreve "ma" + "ria" igual "maria";
```

2.2 Precedência dos operadores

A tabela seguinte apresenta a precedência dos operadores da linguagem Tuga. 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, i.e. todos têm associatividade à esquerda.

Precedência	Descrição	Operadores
1	Parênteses	`()`
2	Unário	`- nao`
3	Multiplicação e Divisão	`* / %`
4	Soma e Subtração	`+ -`
5	Relacional	`< > <= >=`
6	Igualdade e Desigualdade	`igual diferente`
7	E lógico	`e`
8	Ou lógico	`ou`

Os operadores listados na tabela acima têm o significado usual e são em tudo semelhantes a operadores que existem no C e Java. Note porém que o símbolo usado para alguns deles é diferente, nomeadamente os operadores `nao`, `igual`, `diferente`, `e`, `ou` (que em C e Java são escritos como `!`, `==`, `!=`, `&&`, `||`, respectivamente).

3 Regras sobre tipos de dados

Tal como acontece na maioria das linguagens de programação, a utilização dos operadores acima descritos obedecem a algumas regras, dependendo do tipo de dados do(s) operando(s). Por exemplo, a seguinte instrução,

```
escreve verdadeiro - 6;
```

apesar de ser sintaticamente correcta, não é válida porque o operador binário '`-`' não pode ser aplicado quando um dos operandos é do tipo booleano. Como tal, o compilador deverá emitir uma mensagem de erro apropriada neste tipo de situação. De seguida apresenta-se as regras sobre os tipos de dados da linguagem Tuga.

3.1 Regras para os literais

- um literal inteiro é uma expressão do tipo inteiro
- um literal real é uma expressão do tipo real
- uma string delimitada por aspas é uma expressão do tipo string
- `verdadeiro` é uma expressão do tipo booleano
- `falso` é uma expressão do tipo booleano

3.2 Regras para operadores unários

op	*expr*	*op expr*
`-`	inteiro	inteiro
`-`	real	real
`nao`	booleano	booleano

Cada uma das linhas da tabela especifica uma regra. Por exemplo, a 1ª linha da tabela acima diz que o operador unário `-` aplicado a uma expressão do tipo inteiro, dá origem a uma expressão do tipo inteiro. Como se pode constatar, o operador unário `-` só pode ser aplicado para expressões do tipo inteiro ou real. Por sua vez o operador unário `nao` só pode ser aplicado para expressões do tipo booleano.

3.3 Regras para operadores binários

op	$expr_1$	$expr_2$	$expr_1$ *op* $expr_2$
`+ - * / %`	inteiro	inteiro	inteiro
`+ - * /`	inteiro	real	real
`+ - * /`	real	inteiro	real
`+ - * /`	real	real	real
`+`	booleano	string	string
`+`	string	booleano	string
`+`	inteiro	string	string
`+`	string	inteiro	string
`+`	real	string	string
`+`	string	real	string
`+`	string	string	string
`and or`	booleano	booleano	booleano
`< > <= >=`	inteiro	inteiro	booleano
`< > <= >=`	inteiro	real	booleano
`< > <= >=`	real	inteiro	booleano
`< > <= >=`	real	real	booleano
`igual diferente`	booleano	booleano	booleano
`igual diferente`	inteiro	inteiro	booleano
`igual diferente`	inteiro	real	booleano
`igual diferente`	real	inteiro	booleano
`igual diferente`	real	real	booleano
`igual diferente`	string	string	booleano

Novamente, cada linha da tabela acima especifica uma regra. 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.

Como referido anteriormente, o significado dos operadores é em tudo idêntico ao da linguagem Java. Note por exemplo que o operador `+` tanto pode ser usado com o significado de soma, quando os operandos são valores numéricos (inteiro ou real), como também pode ser usado para fazer concatenação de strings, quando pelo menos um dos operandos é uma string, sendo que o compilador deve gerar código para fazer conversões de tipos se necessário. Por exemplo, na expressão:

```
"pi = " + 3.14159
```

a expressão do lado esquerdo é uma string e a expressão do lado direito é um real. Como tal, é suposto o compilador gerar código para converter 3.14159 para a string "3.14159", e só depois gerar código para fazer a concatenação das strings.

4 Máquina Virtual S

O vosso compilador de Tuga deverá gerar bytecodes para uma máquina virtual chamada S (versão mini). Trata-se de uma máquina virtual *stack-based*, e é uma extensão daquela que já vimos na aula teórica 12 e aula PL 6.

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

5 Exemplos de inputs/outputs para o mooshak

Exemplo A

Input

```
escreve 1 + 2 * 3;
```

Output

```
*** Constant pool ***
*** Instructions ***
0: iconst 1
1: iconst 2
2: iconst 3
```

```
3: imult
4: iadd
5: iprint
6: halt
*** VM output ***
7
```

Exemplo B

Input

```
escreve 1 + 2 * 3.0;
escreve "ola " + "maria";
```

Output

```
*** Constant pool ***
0: 3.0
1: "ola "
2: "maria"
*** Instructions ***
0: iconst 1
1: itod
2: iconst 2
3: itod
4: dconst 0
5: dmult
6: dadd
7: dprint
8: sconst 1
9: sconst 2
10: sconcat
11: sprint
12: halt
*** VM output ***
7.0
ola maria
```

Exemplo C

Input

```
/*
   Exemplo de programa em Tuga
*/
escreve 1 + 2 * 3;
escreve 1 + 2.0 * 3;
escreve 7 % (1 + 4);
escreve verdadeiro e (5 < 3.14159);
escreve "pi = " + 3.14159;
escreve "ma" + "ria" igual "maria";
```

Output

```
*** Constant pool ***
0: 2.0
1: 3.14159
2: "pi = "
3: "ma"
4: "ria"
5: "maria"
*** Instructions ***
0: iconst 1
1: iconst 2
2: iconst 3
3: imult
4: iadd
5: iprint
6: iconst 1
7: itod
8: dconst 0
9: iconst 3
10: itod
11: dmult
12: dadd
13: dprint
14: iconst 7
15: iconst 1
16: iconst 4
```

```
17: iadd
18: imod
19: iprint
20: tconst
21: iconst 5
22: itod
23: dconst 1
24: dlt
25: and
26: bprint
27: sconst 2
28: dconst 1
29: dtos
30: sconcat
31: sprint
32: sconst 3
33: sconst 4
34: sconcat
35: sconst 5
36: seq
37: bprint
38: halt
*** VM output ***
7
7.0
2
falso
pi = 3.14159
verdadeiro
```

Exemplo D

Input

```
escreve 2 & + 5;
```

Output

```
Input has lexical errors
```

Exemplo E

Input

```
escreve azul 4;
escreve verde 33;
```

Output

```
Input has parsing errors
```

Exemplo F

Input

```
escreve 2 + falso;
```

Output

```
Input has type checking errors
```

6 Recomendações

1. Começem por especificar a gramática em ANTLR e certifiquem-se que está correcta. Para tal devem testar vários inputs (com e sem erros gramaticais).

2. Após certificarem-se que a gramática está correcta, implementem a deteção de erros semânticos, que no contexto deste trabalho se resumem a erros de inconsistência de tipos. Para tal devem percorrer a árvore sintática e anotar os nós das expressões com informação sobre o seu tipo.

3. Após certificarem-se que não há erros de inconsistência de tipos, podem implementar o gerador de código. Para tal, devem percorrer novamente a árvore sintática e, usando a informação obtida no ponto anterior, gerar instruções adequadas para a máquina virtual.

4. Não basta testarem os exemplos ilustrados no enunciado. O vosso trabalho deve ser testado de forma exaustiva para garantir com o máximo de certeza possível que o vosso compilador funciona do modo esperado.

7 Requisitos

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 22/Abr/2024. (**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, e só a última submissão é que conta.)

O código ZIP deverá conter a gramática em ANTLR, bem como todo o código Java desenvolvido.

- O trabalho deve ser feito em Java usando o ANTLR 4, e submetido ao mooshak.
- O ficheiro zip 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 3 flags que servem para controlar o modo como a emissão de erros é feita.

```
boolean showLexerErrors = false;
boolean showParserErrors = false;
boolean showTypeCheckingErrors = 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 has lexical errors`
  - se não tiver erros lexicais, mas tiver erros de parsing, o programa deverá enviar a seguinte mensagem para o output e terminar: `Input has parsing errors`
  - se não tiver erros lexicais nem de parsing, mas tiver erros de consistência de tipos, o programa deverá enviar a seguinte mensagem para o output e terminar: `Input has type checking errors`
  - se não tiver erros lexicais, nem de parsing, nem de consistência de tipos, 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.
- Link para o mooshak: http://deei-mooshak.ualg.pt/~flobo/, concurso Comp25, problema A.
- Apenas um dos elementos do grupo se deve registar no mooshak, devendo partilhar a password com o(a) colega de grupo. Ao fazer o registo o nome do User deve ser forçosamente o nome do vosso grupo como está na tutoria, algo do género `PL3-g12`. Registos que não cumpram esta regra serão ignorados e/ou eliminados.

8 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 osrestantes trabalhos serão uma continuação 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.