Linguagem para análise dimensional

Universidade de Aveiro

Bruno Caseiro, Gabriel Saudade, Isac Cruz,
Joaquim Ramos, João Génio, Renato Valente

Linguagem para análise dimensional

DETI

Universidade de Aveiro

(88804) caseiro@ua.pt, (89304) gabrielfsaudade@ua.pt,
(90513) isac.cruz@ua.pt, (88012) joaquimramos@ua.pt,
(88771) joaogenio@ua.pt, (89077) renatovalente5@ua.pt

Junho 2019

Resumo

A analise dimensional é indispensável na área da física sendo especialmente usada na previsão, verificação e resolução de equações que relacionam diferentes grandezas. A análise dimensional toma cada dimensão como grandeza algébrica, ou seja, apenas subtrai ou adiciona grandezas com a mesma dimensão.

A linguagem de programação criada durante este projeto ajuda neste tipo de cálculos de grandezas. Ao invés de apenas ser possível realizar cálculos entre tipos de dados primitivos tal como entre "Integers", "Doubles"e "Floats", esta permite-nos manipular vários tipos de grandezas sem correr o risco de as baralhar de forma absurda.

Agradecimentos

Queremos agradecer aos nosso professores da unidade curricular de Linguagens Formais e Autómatos, Miguel Oliveira e Silva, Artur Pereira e André Zúquete, por nos ter guiado durante o projeto e incentivado a realizá-lo da melhor forma possível. Aprendemos mais sobre como os compiladores e interpretadores realmente funcionam, demonstrando assim que este último trabalho prático é indispensável na cadeira de LFA.

Conteúdo

Capítulo 1

Introdução

Este relatório aborda a maneira de como foi criado e programado o interpretador e o compilador que origina a linguagem de programação final.

Este documento está dividido em três capítulos principais, o Capítulo 2 é dedicado à descrição do interpretador, isto é, a linguagem final que será usada pelo programador que deseja realizar os cálculos entre diversas grandezas. No Capítulo 3 é demonstrada a forma como a linguagem principal é compilada para Java.

O Capítulo 4 é usado para descrever como é possível utilizar a linguagem criada. Finalmente, no Capítulo 6 são apresentadas as conclusões retiradas do trabalho.

Capítulo 2

Interpretador

2.1 Declaração

A declaração de variáveis pode ser do tipo *Integer*, *Double* ou qualquer tipo (*TYPENAME*) definido pelo utilizador. É obrigatório o *TYPENAME* começar por um caráter maiúsculo e os seguintes serem carateres minusculos, enquanto *VARNAME* tem de começar apenas por um caráter. No final tem de ter um símbolo terminal ';'. Ex:

```
Integer     var1   ;
Integer     var2   ;
Double      var3   ;
Distancia   var4   ;
```

```
13  declaration:
14
15          'Integer' VARNAME           #IntegerType
16        | 'Double' VARNAME            #DoubleType
17        | TYPENAME VARNAME            #NewType
18        ;
```

```
37  TYPENAME: [A-Z][a-z]+;
38  VARNAME: [a-zA-Z_][a-zA-Z_0-9]*;
```

Figura 2.1: Declarações

2.2 Atribuição

O **Assignment** permite inicializar uma variável ou associar a uma já declarada o valor de uma **Expression**, que pode ser um literal, uma junção de valores ou uma variável já definida e atribuída (mas com o mesmo tipo). Entre a variável e a **Expression** tem de estar presente o símbolo '=' e no final tem de ter um símbolo terminal ';'. Ex:

```
var1   '='   1
var2   '='   var1
var3   '='   3.0
var4   '='   4
```

```
20  assignment:
21          declaration '=' expression    #AssignDeclaration
22          | VARNAME '=' expression      #AssignVariable
23          ;
```

Figura 2.2: Assignment

2.3 Print

Quando escrevemos 'print(**Expression**)', vamos imprimir qualquer expressão, pode ser um valor literal, uma variável, ou até carateres que não estão declarados nem têm valor atribuído. Não é possível somar, subtrair, multiplicar ou dividir variáveis de tipos *TYPENAME* diferentes. No final tem de ter um símbolo terminal ';'. Ex:

```
print(      -var1       )
print(   var1 + var2    )
print(   var1 * var2    )
print(        5         )
```

```
21  print:    'print' '(' expression ')'              #PrintExpression
22          ;
```

Figura 2.3: Print

2.4 Expressão

A **Expression** pode ser um inteiro, *Double*, *String*, ou uma operação entre duas **Expression**, isto é a operação entre várias variáveis da mesma ordem de grandeza, salvo quando são somados dois números não declarados (por ex.: 2 + 2.0 -> *Integer* + *Double*). No final tem de ter um símbolo terminal ';'.

```
expression:'('expression')'                                    #ExpressionAssociation
        |op=('+'|'-') expression                               #ExpressionSignal
        | e1=expression op=('*'|'/') e2=expression             #ExpressionMulDiv
        | e1=expression op=('+'|'-') e2=expression             #ExpressionAddSub
        | DOUBLE                                               #ExpressionDouble
        | INTEGER                                              #ExpressionInteger
        | VARNAME                                              #ExpressionVariable
        ;
```

Figura 2.4: Expression

Capítulo 3

Compilador

3.1 Verificação

Durante a fase da verificação, o interpretador impede o utilizador de baralhar grandezas de forma incompatível. É possível somar duas variáveis do tipo "metro"por exemplo, resultando numa nova variável também do tipo "metro". Se tentássemos subtrair litros a metros, o programador seria travado pelo interpretador.

Foi implementado um *hashmap* com as combinações entre grandezas e operações possíveis. Quando é pedida uma operação, consulta-se o *hashmap* de forma a verificar se é realmente possível operar sobre as duas variáveis. A operação "divisão"entre uma variável do tipo "metros"e uma segunda variável do tipo "segundo"resultaria noutra do tipo "metros/segundo", descrevendo uma velocidade a partir duma operação sobre duas grandezas que representam, respetivamente, distância e tempo.

```
32      while(lineText != null) {
33          // create a CharStream that reads from standard input:
34          CharStream input = CharStreams.fromString(lineText + "\n");
35          // create a lexer that feeds off of input CharStream:
36          SourceLanguageLexer lexer = new SourceLanguageLexer(input);
37          lexer.setLine(numLine);
38          lexer.setCharPositionInLine(0);
39          // create a buffer of tokens pulled from the lexer:
40          CommonTokenStream tokens = new CommonTokenStream(lexer);
41          // create a parser that feeds off the tokens buffer:
42          parser.setInputStream(tokens);
43          // begin parsing at main rule:
44          ParseTree tree = parser.main();
45          if (parser.getNumberOfSyntaxErrors() == 0) {
46              // print LISP-style tree:
47              // System.out.println(tree.toStringTree(parser));
48
49              visitor1.visit(tree);
50
51              ST code = (ST)visitor0.visit(tree);
52              pw.print(code.render());
53          }
```

Figura 3.1: O ciclo while corre o programa a partir de um ficheiro e "visitor1.visit(tree)"(o interpretador, sendo o visitor0 o compilador) verifica se é necessário fechar o processo devido a alguma incompatibilidade

Capítulo 4

Utilização

A definição de uma variável é feita através de <tipo de variável> <id da variável> <;>. Por exemplo » Integer x;

A atribuição de um valor a essa variável é dada por <id da variável> <=> <valor> <;>. Por exemplo » x = 5;

Também podemos definir uma variável e atribuir-lhe um valor só com uma instrução <tipo de variável> <id da variável> <=> <valor> <;>. Por exemplo » Integer x = 5;

A escrita no terminal é feita através de <print> <(> <valor> <)> <;> Por exemplo » print(var1);

Capítulo 5

Manual de Execução

Antes de executar o programa, é necessário criar dois ficheiros: um deles com as grandezas que vão ser utilizadas e um segundo ficheiro com as instruções a correr pelo programa.

Para combinar estes dois ficheiros e realmente correr o programa, é necessário executar o comando "java SourceLanguageMain [TYPELIST] [INSTR] [fileOutput]", sendo [TYPELIST] um ficheiro com as grandezas que vão ser utilizadas, [INSTR] um ficheiro com todas as instruções e [fileOutput] o nome desejado para o ficheiro .java resultante.

Partindo deste último ficheiro, depois de executar o comando "java fileOutput.java", obtemos o resultado final.

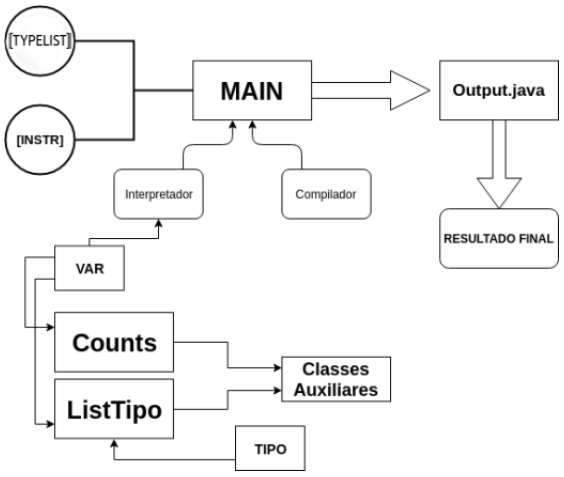

Figura 5.1: Esquema geral simplificado

```
1  Integer [int];
2  Double [double];
3  Distance [m];
4  Time [s];
5  Velocity = Distance / Time;
6  Area = Distance * Distance;
```

Figura 5.2: Ficheiro [TYPELIST] válido

```
1  Integer var1;
2  Double var2;
3  Integer var3 = 3;
4  Distance var4 = 4;
5  print(var4);
6  Double var5;
7  Integer var6;
8
9  var1 = 1;
10 var2 = 2;
11 var5 = var2;
12 var6 = var1 + 2;
13 var3 = var1 * var2 * var3;
14 var4 = var4 * 2;
15 print(var4);
16 print(var1);
17 print(2);
18 print(var1+var3);
19 print(var5 / var2);
20 print(var6);
21
22 print(var4);
```

Figura 5.3: Ficheiro [INSTR] válido

```java
import java.util.*;
public class Output{
Run | Debug
public static void main(String[] args) throws Exception {
Map<String, Var> symb = new HashMap<>();
ListTipo.readCSV("listTipoCSV.csv");symb.put("var1", new Var("Integer", 0));
symb.put("var2", new Var("Double", 0));
symb.put("var3", new Var("Integer", 0));
symb.get("var3").setVal(((new Var("Integer",3))).getVal());
symb.put("var4", new Var("Distance", 0, ListTipo.getTipo("Distance").units()));
symb.get("var4").setVal(((new Var("Integer",4))).getVal());
System.out.println(symb.get("var4"));
symb.put("var5", new Var("Double", 0));
symb.put("var6", new Var("Integer", 0));
symb.put("var6", new Var("Integer", 0));
symb.get("var1").setVal(((new Var("Integer",1))).getVal());
symb.get("var1").setNumDen(ListTipo.getTipo(symb.get("var1").getType()).units());
symb.get("var2").setVal(((new Var("Integer",2))).getVal());
symb.get("var2").setNumDen(ListTipo.getTipo(symb.get("var2").getType()).units());
symb.get("var5").setVal((symb.get("var2")).getVal());
symb.get("var5").setNumDen(ListTipo.getTipo(symb.get("var5").getType()).units());
symb.get("var6").setVal(((Counts.add(symb.get("var1"),(new Var("Integer",2))))).getVal());
symb.get("var6").setNumDen(ListTipo.getTipo(symb.get("var6").getType()).units());
symb.get("var3").setVal(((Counts.mult((Counts.mult(symb.get("var1"),symb.get("var2"))),symb.get("var3")))).getVal());
symb.get("var3").setNumDen(ListTipo.getTipo(symb.get("var3").getType()).units());
symb.get("var4").setVal(((Counts.mult(symb.get("var4"),(new Var("Integer",2))))).getVal());
symb.get("var4").setNumDen(ListTipo.getTipo(symb.get("var4").getType()).units());
System.out.println(symb.get("var4"));
System.out.println(symb.get("var1"));
System.out.println((new Var("Integer",2)));
System.out.println((Counts.add(symb.get("var1"),symb.get("var3"))));
System.out.println((Counts.div(symb.get("var5"),symb.get("var2"))));
System.out.println(symb.get("var6"));
System.out.println(symb.get("var6"));
System.out.println(symb.get("var4"));
}
}
```

Figura 5.4: Exemplo de um ficheiro Output.java

```
saki@saki-lap:~/Documentos/UA/ua/ano2/LFA/trabalho4/lfa-1819-g15/Final$ java SourceLanguageMain DimensionsTable.txt test.txt Output.java
1 : Integer var1;
2 : Double var2;
3 : Integer var3 = 3;
4 : print(var3);
5 : Distance var4 = 4;
6 : print(var4);
7 : Double var5;
8 : Integer var6;
9 :
10 : var1 = 1;
11 : var2 = 2;
12 : var5 = var2;
13 : var6 = var1 + 2;
14 : var3 = var1 * var2 * var3;
15 : var4 = var4 * 2;
16 : print(var4);
17 : print(var1);
18 : print(2);
19 : print(var1+var3);
20 : print(var5 / var2);
21 : print(var6);
22 :
23 : print(var4);
```

Figura 5.5: Output do comando 'java SourceLanguageMain [TYPELIST] [INSTR] [fileOutput]'

```
saki@saki-lap:~/Documentos/UA/ua/ano2/LFA/trabalho4/lfa-1819-g15/Final/Files/Exec$ java Output
3.0Integer
4.0Distance
8.0Distance
1.0Integer
2.0Integer
7.0Integer
1.0Double
3.0Integer
3.0Integer
```

Figura 5.6: Output do comando 'java Output.java'

Capítulo 6

Conclusões

Olhando, agora, em retrospetiva para a informação exposta neste relatório, a nossa abordagem ao domínio da análise dimensional traduziu-se numa ferramenta que ajuda o programador a criar novas grandezas, de forma a manter uma "estabilidade matemática"no momento do cálculo ou alteração de uma variável.

Reconhecemos que este projeto foi fundamental na extensão do nosso interesse pela Unidade Curricular, pois aumentou a nossa compreensão do processo de compilação de uma linguagem.

Contribuições dos autores

Consideramos que o trabalho foi dividido de forma igual por todos os elementos do grupo. O Joaquim Ramos, João Génio e o Renato Valente foram responsáveis maioritariamente pelo interpretador e pelas gramáticas enquanto que os restantes três elementos do grupo, Bruno Caseiro, Isac Cruz e Gabriel Saudade foram responsáveis em grande parte pelo compilador.

Apesar de termos dividido as tarefas de igual forma, fomo-nos ajudando mutuamente com a ocorrência de dúvidas, bugs ou outro tipo de problemas.

Portanto, cada membro foi responsável por aproximadamente 16,(6)% do trabalho.