

LINGUAGEM PARA ANÁLISE DIMENSIONAL

Compiladores 2018/19



universidade de aveiro

Alexandre Lopes - 88969
André Amarante - 89198
Carolina Marques - 85084
Edgar Morais - 89323
Rui Coelho - 86182

Índice

Introdução	3
Gramática Dim	4
Sintaxe	4
DimExecute.java	4
DimTable.java	5
Análise Semântica	5
Gramática Geral	6
Instruções	6
Declaração e atribuição de valores a variáveis	6
Operações algébricas	8
Operações condicionais	8
Operações iterativas/ciclos	9
Expressões de comparação	10
Operações booleanas	10
Impressão de itens na consola	11
Pedido de input ao utilizador	11
Importe de dimensões	11
Estrutura	12
Análise Semântica	13
O Compilador	14
Conversão de variáveis no compilador	14
Criação do ficheiro a compilar	14
Importe de ficheiros das dimensões	14
Tipo de input a pedir ao utilizador	15
Exemplos de programas	16
Programas errados	16
err1.txt – Atribuição de um real a um inteiro	16
err2.txt - atribuição de um real a uma dimensão do que só aceita valores inteiros	16
err3.txt - variável com uma dimensão que não foi importada/não existe	16
err4.txt - atribuição de uma velocidade a um tempo: dimensões diferentes são incompatíveis	16
err5.txt - criação de variável com o mesmo nome de uma já existente, mesmo com dimensão diferente	17
err6.txt - soma de operandos de dimensões diferentes	17
err7.txt - uso de uma variável não definida numa operação	17




err8.txt - atribuição de valor a uma variável não existente	17
Programas funcionais.....	18
p2.txt – definição e impressão de variáveis	18
p3.txt – ciclos e condições.....	18
p4.txt – interação com o utilizador	20
Autoavaliação.....	21

Introdução

A linguagem a que nos propusemos criar tem por objetivo a manipulação de unidades dimensionais sendo possível definir desde dimensões distintas a expressões numéricas. O sistema proposto deve permitir operações algébricas sobre as dimensões existentes como validar a respetiva correção, isto é, temos de garantir que a soma entre variáveis de diferentes tipos não é permitida. Por exemplo, apesar de ambas serem numéricos, não é possível somar distância com tempo.

Para alcançar o objetivo proposto foram criadas duas linguagens: numa primeira instância, a linguagem de especificação, a linguagem interpretada, na qual podemos definir dimensões e unidades que definiriam o novo tipo de dados numéricos de forma a poderem ser utilizados na linguagem genérica, a linguagem compilada, uma vez que os erros dimensionais estariam prevenidos.

Para o bom funcionamento destas linguagens devemos ter em atenção alguns problemas da álgebra dimensional:

-  Somas e subtrações apenas podem ser efetuadas entre operandos de dimensões iguais;
-  Multiplicações e divisões geram sempre uma nova dimensão (caso ambos os operandos possuam uma dimensão);
-  Quando multiplicada uma dimensão por um valor adimensional é mantida a mesma dimensão.

É de salientar que não deve ser permitido somar uma grandeza adimensional com uma grandeza dimensional (pelo menos, sem proceder ao `cast` desse adimensional, como explicaremos mais adiante).

Para tornar a linguagem um pouco mais desenvolvida e para que pudesse ser usada em contexto real, munimos a mesma de algumas funcionalidades básicas das linguagens de programação comuns, isto é, é permitido criar expressões condicionais, utilização de ciclos, imprimir valores na consola, fazer importe de pacotes e permitir inputs por parte do utilizador.

Gramática Dim

Sintaxe

A gramática Dim permite a definição de dimensões, constituídas pelo seu nome, sigla (ex.: m, s, m/s) e o seu tipo (**'real'** ou **'integer'**). Para definirmos, então, uma dimensão a sintaxe a usar será:

`<NOME DA DIMENSÃO> [<SIGLA>] : <TIPO>`

Exemplos:

distancia [m] : REAL;

massa [g] : REAL;

velocidade [m/s] : REAL;

NOTA:

Inicialmente as siglas seriam usadas da seguinte forma no código fonte:

`<DIMENSÃO> = <VALOR> <SIGLA>`

Após deliberação as siglas tornaram-se obsoletas, na medida em que passámos a adotar uma notação semelhante ao “cast”:

`(<DIMENSÃO>) <VALOR>`

Contudo, estas são usadas no código da análise semântica da gramática Geral para efetuar verificações.

DimExecute.java

De forma a armazenar e aceder mais facilmente às informações das dimensões criadas, optámos por criar a classe Unidade.

DimExecute é uma classe que estende *DimBaseVisitor*. Nesta classe, serão visitadas as ações apresentadas na gramática: **#realType** e **#integerType**. No fundo, ao visitar cada uma destas ações iremos criar as unidades usando a classe Unidade.

No caso de **visitIntegerType** geramos a nova unidade da seguinte forma:

Unidade elemento = new Unidade(grandeza, sigla, "integer");

e faremos o mesmo para o tipo “real”, apenas mudando “integer” para “real”.

DimTable.java

Esta classe Java será usada para colocar uma unidade numa tabela (**TreeSet**) caso ela não exista. Posto isto, podemos pensar então na **DimTable** como uma classe que armazena todas as unidades criadas pelo utilizador. Esta tabela é instanciada no cabeçalho da gramática para que seja passada ao **DimParser**.

Análise Semântica

Visto que não existem erros semânticos, a análise semântica desta gramática não foi feita uma vez que não havia erros para lidar.

Gramática Geral

Apesar de serem independentes, a gramática Dim e Geral podem ser usadas em conjunto, passando assim a comunicar entre si, deste modo, as dimensões que forem definidas na Dim passam a poder ser usadas na Geral como novos tipos que terão variáveis e valores associados.

Instruções

Tendo em conta o que foi anteriormente mencionado podemos dizer que é possível realizar definir variáveis, atribuir valores, somar, subtrair, multiplicar e dividir de uma forma segura sem o utilizador se ter que preocupar visto que cabe à linguagem tratar de todas as verificações no que diz respeito à realização destas operações.


Salientamos ainda que é possível o uso de ciclos, no caso, o ciclo *while*, realizar operações booleanas (*and*, *or*, *true*, *false*), podemos realizar operações de comparação (igualdade, diferença, maior, menor, maior ou igual e menor ou igual), suportamos ainda o importe de ficheiros, a utilização de comentários, o *input* de informação que permite ao programador receber informação por parte do utilizador e, ainda, o *print* de modo a que possamos saber o que se passa no sistema.

Declaração e atribuição de valores a variáveis

Para a definição de variáveis decidimos usar uma sintaxe à *la C*, isto é, optámos pela utilização dos *casts* para utilizar as unidades que são definidas na **Dim**.

Deste modo, é de salientar que suportamos os seguintes tipos:

 Real

 Integer

 Todas as dimensões definidas pelo utilizador na Dim

Para definir as variáveis temos sempre de associar um valor às mesmas, algo que é definido pela nossa gramática. É de notar que a declaração e atribuição de valores pode ser feita de duas formas, tendo em conta se são grandezas adimensionais ou dimensionais.

Adimensionais

No caso das grandezas adimensionais a definição é mais simples, a sintaxe assemelha-se muito à de Java tendo, assim, de se indicar o tipo de dados que vamos usar (*real*, *integer* ou dimensão definida na gramática Dim), nome que queremos atribuir e o seu valor sempre delimitado por um ponto e vírgula.

<dimensão> <nome da variável> = <valor>;

Exemplo:

real a = 1.6;

integer b = 2;

Se a variável já está definida e queremos atribuir um novo valor à mesma temos apenas que fazer:

<nome da variável> = <valor>;

Exemplo:

b = 3;

Dimensionais

No caso de as grandezas serem dimensionais temos que indicar o seu tipo, ou seja, usar os *casts* anteriormente mencionados. Usamos isto para garantir que é impossível que os tipos sejam incompatíveis.

<dimensão> <nome da variável> = (<dimensão>) valor;

Caso já existam variáveis de um mesmo tipo da variável que queremos definir e, a queremos usar para a nova definição podemos omitir os *casts*, isto é, imaginemos que existem, por exemplo, uma velocidade *x* já definida e uma grandeza adimensional *d*, se queremos definir *w* às custas de *x* e *d*, podemos fazer sem usar *casts*. Podemos ainda não usar os *casts* num caso em que apenas se defina às custas de apenas uma variável do mesmo tipo, por exemplo, definir *w* apenas baseado em *x*.

<dimensão> <nome da variável> = <variável da mesma dimensão já definida> <operação> <variável adimensional>;

<dimensão> <nome de variável> = <variável da mesma dimensão já definida>;

É de notar que a operação aqui seria ou a multiplicação ou divisão visto que não é possível somar e subtrair grandezas adimensionais a grandezas dimensionais (sem *cast*).

Caso queiramos definir uma nova variável às custas de duas já existentes é também possível omitir este *cast*.

<dimensão> <nome de variável> = <variável da mesma dimensão já definida> <operação> <variável da mesma dimensão já definida>;

Neste caso devemos ter em atenção que apenas podemos considerar esta operação somas e subtrações visto que multiplicar e dividir duas grandezas dimensionais geram uma nova dimensão.

Exemplo:

real d = 2;

velocidade x = (velocidade)10.1;

velocidade z = (velocidade)20.8;







*velocidade w = d*x;*

tempo t = d(tempo)2;*

Operações algébricas

Como seria de esperar como qualquer linguagem de programação, a nossa linguagem permite a realização de operações como a subtração, a adição, multiplicação e divisão.

É de notar que não é possível fazer todo o tipo de somas, subtrações, divisões e multiplicações. Devemos ter em atenção os seguintes tópicos:

-  Apenas podem ser somadas e subtraídas as mesmas grandezas;
-  Não podemos ter divisão por zero;
-  Não é possível somar uma grandeza adimensional com uma grandeza dimensional;
-  Multiplicações de uma grandeza adimensional por uma grandeza dimensional geram sempre a grandeza dimensional;
-  Multiplicações de duas grandezas dimensionais geram uma nova grandeza;
-  Divisões de duas grandezas dimensionais geram uma nova grandeza.

Na linguagem que criamos para executar este tipo de operações deve ser feito:

<dimensão> <nome da variável> = <variável a> <operação> <variável b>;

É de notar que a sintaxe do tipo *cast*, tal como foi mencionado anteriormente, é apenas dispensável caso as unidades sejam do mesmo tipo.

Exemplos:

real r = 1.2;

velocidade v = (velocidade)10.1;

velocidade va = (velocidade)20.8;

*velocidade vb = r*v; // como são do mesmo tipo (real) não existe necessidade de
// indicar explicitamente a unidade.*

tempo t = r(tempo)2;*

tempo ta = (tempo)10.9;

tempo tb = ta + t;

Operações condicionais

No que concerne às operações com condições, isto é, operações que nos permitem inferir a validade de uma afirmação, suportamos a típica estrutura, **if**, **else if** e **else**.

Para a realização deste tipo de operações são necessários definir os sinais de comparação no caso, a nossa linguagem suporta '==', '!=', '>', '<', '>=' e o '<=' os quais serão mencionados nos pontos seguintes. Podem também ser usados os valores lógicos **true** e **false**.

Realçamos que foi adicionado suporte ainda para os operadores booleanos **AND**, **OR** que nos permitem realizar expressões mais complexas e fazer código mais compacto.

A sintaxe que escolhemos é muito similar à do Java por isso temos que a estrutura das operações condicionais pode ser definida por:

```

if ( <condição> ){
    <código>
} else if ( <condição> ) {
    <código>
} else{
    <código>
}

```

Exemplos:

```

tempo t = (tempo)2;
tempo ta = (tempo)10.9;
tempo tb = ta + t;
if ( t == ta ){
    print "entrou num if";
    print -20;
} else if ( ta == tb)
    print "entrou num else if";
} else{
    print "entrou no else";
    print(10);
}

```

Operações iterativas/ciclos

Na linguagem criada está disponível o ciclo ‘while’. Este tem a função do ‘while’ comum nas várias linguagens, ou seja, iterar sobre linhas de código verificando, em cada iteração, se a condição é verdadeira e continuando a iterar se esse for o caso.

Este comando tem a seguinte estrutura de uso:

```
while ( < condição > ) { <código> }
```

Exemplo:

```

while ( v >= (velocidade) 0 ) {
    t = t + (tempo) 1;
}

```

Expressões de comparação

Na nossa linguagem é possível realizar comparações no contexto de operações condicionais (como o **if**) e do ciclo implementado (**while**). Estas expressões vão originar um valor booleano que é utilizado nos comandos referidos. Os sinais de comparação disponíveis são '==', '!=', '>', '<', '>=' e '<='. No lugar destas expressões, a nossa linguagem também permite a utilização de um valor booleano fixo através da utilização das palavras '**true**' e '**false**' que representam os dois valores booleanos possíveis. De notar que não é possível comparar dimensões diferentes.

Estas expressões têm a seguinte estrutura:

<valor> <sinal comparação> <valor>

Exemplo:

`v1 == v2`

Exemplo de uso:

```
if ( v1 == v2 ){
    v1 = v2 + (velocidade) 1;
}
```

Operações booleanas

Também implementámos a linguagem de forma a ser possível realizar operações entre várias condições. Os operadores disponíveis para realizar este tipo de operações são o 'AND' e o 'OR'. O 'AND' tem a mesma função que o operador '&&' tem na linguagem Java, sendo que realiza a operação designada como 'e'. O 'OR' tem a mesma função que o operador '||' tem na linguagem Java, sendo que realiza a operação designada como 'ou'.

Estas operações têm a seguinte estrutura:

< expressão de comparação > < operador booleano > < expressão de comparação >

Exemplo:

`v2 >= v1 AND t1==(tempo) 1`

Exemplo de uso:

```
while ( v2 >= v1 AND t1==(tempo) 1 ) {
    v2 = v2 + (velocidade) 1;
}
```

Impressão de itens na consola

Para implementação de prints foi escolhida a seguinte estrutura:

```
print <algo a imprimir>;
```

Podem ser impressas no terminal strings ou contas (que por sua vez podem ser valores de variáveis, inputs ou outras contas).

Exemplos de uso:

```
print "olá";
```

```
distancia d = (distancia)2.4;
```

```
print d;
```

Pedido de input ao utilizador

Para permitir interação com o utilizador, foi implementado um sistema de inputs com a seguinte estrutura:

```
input "<Informação do pedido ao utilizador>" ;
```

Exemplos de uso:

```
distancia di = input "Insira distancia: "; // associa a di o valor inserido pelo utilizador
```

```
print input "algo-> "; // imprime o valor inserido pelo utilizador)
```

Importe de dimensões

Para que o utilizador possa utilizar as dimensões definidas noutra ficheiro no código fonte, foi criado o comando **'import'** para simular o importe dessas dimensões, à semelhança dos importes de pacotes em Java.

Para implementação deste mecanismo foi escolhida a seguinte estrutura:

```
import <nome/caminho do ficheiro>;
```

Exemplos de uso:

```
import dim1.txt; // importa as dimensões criadas em dim1.txt
```

```
import dim2.txt; // importa as dimensões criadas em dim2.txt
```

Estrutura

Como referido anteriormente, a nossa linguagem permite ao programador definir dimensões num ficheiro separado do código fonte. Esse ficheiro pode depois ser importado (comando **'import'**) para o código fonte, sendo que, a partir desse momento, se torna possível utilizar as dimensões definidas em cada ficheiro de configuração dimensional. Assim, tornou-se necessário criar um mecanismo que permitisse transferir os dados das dimensões (armazenados na **DimTable** do **DimParser**) para os Visitors da gramática Geral, quer para, por exemplo, verificar a existência ou a compatibilidade entre dimensões na análise semântica, quer para verificar o tipo de cada dimensão (**'real'** ou **'integer'**) de modo a fazer a respetiva conversão para tipos Java válidos na parte do Compilador.

Para isso, na análise semântica da gramática Geral, quando este *visitor* visita a regra **'pacote'**, irá ter acesso ao nome do ficheiro (eventualmente um caminho para o ficheiro), invocando de seguida a função **main(String[] args)** da classe **DimMain**, passando como argumento o nome/caminho desse ficheiro. Assim, a **DimMain** irá ter como fonte o ficheiro importado com as dimensões definidas para que o *visitor* **DimExecute** preencha a tabela de dimensões (**dimTable**) com as dimensões do ficheiro. Depois deste *visitor* executar, a **DimMain** guarda essa tabela, que é depois obtida pelo *visitor* **GeralSemanticAnalysis** para que possa guardar as dimensões na sua própria tabela de dimensões (como este processo ocorre a cada uso do **'import'**, a tabela de dimensões da **GeralSemanticAnalysis** irá conter todas as dimensões de todos os ficheiros de dimensões importados).

No final da análise semântica, a classe **GeralMain** obtém desta as tabelas (de símbolos e de dimensões) e transmite-as ao **GeralCompiler** para que este possa então renomear as variáveis do código fonte em nomes válidos em Java, bem como converter os tipos do código fonte em tipos de dados Java (nomeadamente, **int** e **double**, consoante as variáveis sejam do tipo **integer** ou **real**, respetivamente).

De notar que, para este processo, dividimos cada gramática numa pasta e tirámos partido dos pacotes Java, tendo importado os itens necessários do pacote **'dim'** para o pacote **'geral'** (principalmente as classes **Unidade**, **DimTable**, **DimMain** e **DimExecute**).

Análise Semântica

Para realizarmos a análise semântica, identificámos os possíveis erros semânticos presentes na linguagem. Para alguns desses erros verificámos que precisávamos de obter informação das folhas das árvores, sendo que tivemos de adicionar à gramática alguns valores de retorno para algumas regras.

Além disso foi necessário criar alguns mapas e sets de forma a armazenar algumas informações necessárias para realizar verificações semânticas (como dimensões e tabela de símbolos). Estas incluem verificar se uma variável já foi criada quando esta é chamada numa operação e a verificação da existência de algum tipo de dados na criação de uma variável ou na chamada de um valor numérico desse tipo.

Também foi preciso verificar os tipos dos operandos na operação de soma/subtração, pois, ambos têm de ser do mesmo tipo. Esta verificação também tem de ser feita em comparações de valores.

Por último também foi necessário criar novos tipos de dados nas operações de multiplicação e divisão entre dados dimensionais. Consideramos que o nome da nova dimensão é originado a partir dos nomes das dimensões dos operandos. Se a operação for uma multiplicação o nome da dimensão originado é a concatenação dos nomes das dimensões (**ex:** (tempo) 5 * (distancia) 2 = (tempodistancia) 10). No caso de a operação ser uma divisão o nome da dimensão originada é a concatenação dos dois nomes separados por 'por' (**ex:** (tempo) 10 / (distancia) 2 = (tempopordistancia) 5). É de notar que este processo só acontece caso a dimensão resultante da operação tenha uma sigla diferente das dimensões importadas. Neste caso, a dimensão do resultado da operação será a dimensão com a sigla originada na operação (ou seja, se for importada "distancia" com a sigla "m", tempo com a sigla "s" e velocidade com a sigla "m/s", a dimensão resultante da divisão de uma "distancia" por "tempo" será, portanto, "velocidade" e não "distanciaportempo". No caso de, na multiplicação e divisão, um dos operandos ser adimensional, a dimensão do resultado da operação é o mesmo do operador dimensional (**ex:** (tempo) 10 * 2 = (tempo) 20.0).

O Compilador

Conversão de variáveis no compilador

Uma vez que, como linguagem destino do compilador, optámos por Java, tornou-se necessário fazer com que o compilador renomeasse as variáveis do código fonte para variáveis válidas em Java. Isto é, no nosso código fonte é possível ter variáveis com nomes como 'for' ou 'instanceof', enquanto que em Java essas são palavras reservadas, não podendo ser utilizadas para nomear variáveis.

Assim, o nosso compilador irá, a cada nova variável que surja no código fonte, renomeá-la para 'varX', em que 'X' é o valor de um contador que incrementa a cada nova variável (ex.: 'var1', 'var2', 'var3', etc).

É de salientar que o compilador contém o mapeamento dos nomes das variáveis do código fonte para código destino para que possa usar a nova designação cada vez que uma variável anteriormente definida seja usada noutra instrução.

Criação do ficheiro a compilar

Para executar a **GeralMain** é necessário fornecer como argumento o ficheiro/caminho do código fonte. Durante a execução da **main()** é instanciado o compilador (**GeralCompiler**) ao qual é passado esse mesmo ficheiro/caminho com o código fonte para que o possa compilar na linguagem destino.

Para isso, o compilador conta com o método **prepareClassName()** que vai aceder ao ficheiro do código fonte e extrair o seu nome (removendo o resto do caminho e a extensão), para que o nome da classe Java gerada seja igual ao nome do ficheiro (embora com a inicial maiúscula, respeitando a convenção de nomeação de classes em Java).

Após a visita de toda a árvore do *Visitor* do compilador, é invocado o método **compile()** deste que vai receber o resultado do *String Template* como *String* e escrevê-lo no ficheiro **<NomeDaClasse>.java**. De seguida, executa o comando **javac** para compilar o código Java gerado num **<NomeDaClasse>.class**.

Para ilustrar este processo, consideremos que o código fonte se encontra em **geral/p3.txt**. O nome da classe será **P3**, o nome do ficheiro com código gerado em Java será **P3.java** e o respetivo ficheiro compilado será, portanto, **P3.class**.

Importe de ficheiros das dimensões

Foi decidido que o início do ficheiro .java iria conter comentários com referência aos importes dos ficheiros usados para definir dimensões, pois estes ficheiros não são pacotes válidos em Java, nem são precisos no código compilado. Para isso foi criado em java.stg, o seguinte *template*:

```
head(library_name) ::= <<

// import <library_name; separator="\n">;

>>
```

No *GeralCompiler*, no **visitPacote**, substituímos `<library_name>` pelos nomes dos ficheiros de dimensões.

Tipo de input a pedir ao utilizador

Para compilação para código Java, foram criados 2 *templates* no `java.stg`, o **inputInt** e o **inputDouble**, preparados para inserir no código compilado `"sc.nextInt()"` e `"sc.nextDouble()"`, respetivamente.

No **GeralCompiler**, foram criadas *strings* globais para guardar informação acerca do tipo de *input* a pedir ao utilizador (*int* ou *double*): **tipolInput** e **stringAimprimir**. A primeira guarda o nome de um de dois *templates* do `java.stg` (**inputInt** e **inputDouble**). Nas visitas a produções que podem produzir um *input*, esta *string* é alterada consoante o tipo de variável ('real' ou 'integer').

A **stringAimprimir** terá o valor `"System.out.println("Informação a dar ao utilizador")"` nas visitas a *input*, e vazia, caso contrário (visitas a valores). Esta *string* será colocada nos *templates* de produções que podem produzir *input* (para aparecer no código Java antes de pedir o *input*).

No **visitInput**, iremos apenas buscar o módulo `"inputInt"` ou `"inputDouble"`, consoante está atualmente definido na *string* `"tipolInput"`, garantido que lemos valor no formato pretendido.

Exemplos de programas

Programas errados

err1.txt – Atribuição de um real a um inteiro

```
import testes/FunctionalProg/dim/dim1.txt;
import testes/FunctionalProg/dim/dim2.txt;
```

```
integer d = 1.0; // erro
velocidade x = (velocidade)10;
velocidade z = (velocidade)20;
velocidade w = d*x;
tempo t = d*x;
```

err2.txt - atribuição de um real a uma dimensão do que só aceita valores inteiros

```
import testes/FunctionalProg/dim/dim1.txt;
import testes/FunctionalProg/dim/dim2.txt;
unidadeawesome x = (unidadeawesome)10.1; // unidadeawesome é do tipo integer
```

err3.txt - variável com uma dimensão que não foi importada/não existe

```
import testes/FunctionalProg/dim/dim1.txt;
import testes/FunctionalProg/dim/dim2.txt;
velocidade x = (velocidade)10.2;
potencia p = (potencia)17; // potencia não foi definida em nenhum dos ficheiros
importados
```

err4.txt - atribuição de uma velocidade a um tempo: dimensões diferentes são incompatíveis

```
import testes/FunctionalProg/dim/dim1.txt;
import testes/FunctionalProg/dim/dim2.txt;

real d = 1.2;
velocidade x = (velocidade)10.1;
velocidade z = (velocidade)20.4;
tempo t = (tempo)1.2;
t = d*x; // erro
```

err5.txt - criação de variável com o mesmo nome de uma já existente, mesmo com dimensão diferente

```
import testes/FunctionalProg/dim/dim1.txt;
```

```
import testes/FunctionalProg/dim/dim2.txt;
```

```
real d = 1.2;
```

```
velocidade x = (velocidade)10.1;
```

```
velocidade z = (velocidade)20.4;
```

```
tempo t = (tempo)4.3;
```

```
tempo x = (tempo)6.5; // erro
```

err6.txt - soma de operandos de dimensões diferentes

```
import testes/FunctionalProg/dim/dim1.txt;
```

```
import testes/FunctionalProg/dim/dim2.txt;
```

```
real d = 1.6;
```

```
velocidade x = (velocidade)10.1;
```

```
tempo y = (tempo)10.9;
```

```
tempo o = y + x; // erro
```

err7.txt - uso de uma variável não definida numa operação

```
integer d = 4;
```

```
integer i = 2;
```

```
integer f = x + 1; // erro: a variável x não existe
```

err8.txt - atribuição de valor a uma variável não existente

```
import testes/FunctionalProg/dim/dim1.txt;
```

```
import testes/FunctionalProg/dim/dim2.txt;
```

```
temperatura temp = (temperatura)4.8;
```

```
temperatura t = (temperatura)2.0;
```

```
f = temp*t; // erro: f tem de ser definida primeiro
```

Programas funcionais

p2.txt – definição e impressão de variáveis

```
import testes/FunctionalProg/dim/dim1.txt;

import testes/FunctionalProg/dim/dim2.txt;


print "Welcome!";

real r = 1.6; // definicao variavel adimensional e print da mesma

print "variavel real: ";

print r;


r = 1.2; // redefinicao de valor de variavel

real ra = 2.1;

integer i = 2;

input "Insira um valor inteiro: "; // este input será sc.nextInt()

print input "Insira outro valor inteiro (este eu mostro-lhe de volta!): ";
```

p3.txt – ciclos e condições

```
import testes/FunctionalProg/dim/dim1.txt;

import testes/FunctionalProg/dim/dim2.txt;


real r = 1.2; // definicao variavel adimensional e print da mesma

print "variavel real: ";

print r;


velocidade v = (velocidade)10.1;

velocidade va = (velocidade)20.8;

velocidade vb = r*v;


tempo t = r*(tempo)2;

while ( v < va ){

    v = v + (velocidade)1;
```

```

}

print "ciclo while: incremento de 1 à velocidade 10.1 até ser maior que 20.8";

print v;

tempo ta = (tempo)10.9;

tempo tb = ta + t;

print "resultado de tempo 10.9 + 2.4";

print tb;

tempo tc = (tempo)12.4;

velocidadevelocidade vv = v * v; // var nao existente

if ( t == ta ){
    print "entrou num if";
    print -20;
}else{
    print "entrou no else";
    print(10);
}

if( vv == v*va ){
    tc = (tempo)13.7;
}

if(true){
    tc = (tempo)3 + (tc - (tempo)2.3);
}

if(true OR v==v AND tc==(tempo)3){
    print "Done :)";
}

```

```
}
```

p4.txt – interação com o utilizador

```
import testes/FunctionalProg/dim/dim1.txt;
```

```
import testes/FunctionalProg/dim/dim2.txt;
```

```
distancia di = input "Insira distancia: ";
```

```
tempo te = input "Tempo: ";
```

```
integer j = input "Insira um inteiro: ";
```

```
print j;
```

```
velocidade dpt = di / te;
```

```
print "distancia inserida";
```

```
print di;
```

```
print "tempo inserido";
```

```
print te;
```

```
print "velocidade resultante (distancia/tempo): ";
```

```
print dpt;
```

```
unidadeawesome unia = input "Insira unidadeawesome: ";
```

```
print "valor inserido: ";
```






```
print unia;
```

NOTA: No repositório encontra-se ainda o p1.txt, que engloba as funcionalidades dos ficheiros acima expostos.

Autoavaliação

Se olharmos para o repositório veremos que existem elementos sem *commits*, contudo, não foi por falta de trabalho dos mesmos. Abordámos este projeto de uma forma mais clássica, optamos por juntar o grupo em diversos dias e realizar o trabalho em apenas um computador, sendo que os elementos que realizavam o código iam trocando para todos poderem trabalhar.

Deste modo, achamos que a distribuição justa da nota do trabalho (em percentagem) seria:

-  Alexandre Lopes – 21,25%
-  André Amarante – 21,25%
-  Carolina Marques – 15%
-  Edgar Morais – 21,25%
-  Rui Coelho – 21,25%