Compiladores

Miguel Oliveira e Silva
Artur Pereira

Linguagem para análise dimensional: Chubix

Bruno Bastos, 93302
Hugo Almeida, 93195
Mário Silva, 93430
Leandro Silva, 93446
Rui Fernandes, 92952

DETI
Universidade de Aveiro
18 de Junho de 2020

Conteúdo

1 Introdução

Este documento visa descrever e guiar o leitor por todo o processo de desenvolvimento do projeto final da unidade curricular de Compiladores. O trabalho foi desenvolvido pelos estudantes identificados na capa sendo que recorreram maioritariamente à assistência do docente Prof. Dr. Miguel Oliveira e Silva.

Este projeto tem como principal objetivo o desenvolvimento de um compilador assente em duas linguagens, uma para o compilador em si, e outra complementar que auxilia o funcionamento da primeira. Ao longo do desenvolvimento tentou-se ao máximo seguir todas as fases de construção de linguagens de programação.

Foram utilizadas as tarefas e métodos aprendidos nas aulas da unidade curricular, o ANTLR4 e programação em Java.

O grupo decidiu escolher como tema para o projeto, um dos temas sugeridos no guião do trabalho, sendo este o desenvolvimento de uma linguagem de programação para análise Dimensional. Este tema tem por base estender o sistema de tipos de uma linguagem de programação possibilitando a definição de dimensões distintas e interoperáveis, a expressões numéricas. Sendo uma linguagem destas de grande utilidade para problemas de Física ou Química em que existem várias dimensões nas quais é preciso operar, é importante que as operações possíveis façam sentido.

Tendo isto em conta, foi criada uma linguagem de programação principal com as funcionalidades básicas de uma linguagem como Java ou Python com o acrescento de incorporar análise dimensional, servindo de base à construção do compilador. Desenvolveu-se também uma linguagem complementar a esta que permite criar as dimensões e unidades físicas pretendidas pelo utilizador.

2 Concepção e Definição da Linguagem

Desde o inicio soube-se que o maior desafio seria como criar novas dimensões e unidades, e posteriormente utilizá-las e definir variáveis usando então o sistema de dimensões. Tirando isto, na linguagem principal tencionava-se incluir todas as funcionalidades básicas necessárias como instruções iterativas e condicionais, bem como funções, declaração de variáveis e todas as operações matemáticas base.

A sintaxe de criação de dimensões e unidades passou por várias fases no momento de idealização, o objetivo era principalmente que fosse fácil de utilizar e de entender para o utilizador. Segue-se então a evolução da mesma:

1ª Fase:

- create Metro as m;
- create Centímetro as m*10^-2;

Este primeiro conceito é obviamente defeituoso, visto que não está a ser criada uma dimensão, mas sim duas unidades e uma relação entre elas, acrescenta-se ainda que deste modo, o centímetro não tem símbolo da unidade, e apenas um símbolo que representa a sua relação ao metro.

2ª Fase:

- new dim distance;
- new unit distance(m : 1);
- new unit distance(cm : 1/100);

Nesta segunda fase, separou-se então as criações de dimensões e unidades, criando explicitamente uma dimensão com identificação primeiro, e apenas depois criar unidades para a mesma, sendo que a unidade centímetro tem agora um símbolo, mantendo a sua relação ao metro. Aqui também encontraram-se problemas, por exemplo as unidades apenas se podiam relacionar com uma unidade base passada como *"1"* e faltava também diferenciação entre dados *Integer* e *Double*.

Sintaxe Final:

- dim Distance(m : Double);
- unit Distance(cm : 0.01*m);
- unit Distance(in : 2.54*cm);

Já na sintaxe final, os problemas anteriores foram resolvidos, sendo de notar que achou-se irrelevante a presença do elemento *"new"* antes de cada dimensão e unidade novas, e ainda que optámos por não permitir a declaração de uma dimensão sem que tenha pelo menos a sua unidade base.

Quanto à sintaxe de definir variáveis de determinadas dimensões e unidades, a evolução foi análoga ao caso anterior, visto que era alterada consoante a sintaxe de criação de dimensões/unidades. Para demonstrar:

- **Primeira Fase:** Meter d = 10;
- **Segunda Fase:** d = 10(m);
- **Sintaxe Final:** Distance d = 10 [m];

3 Linguagem Complementar

As seguintes secções abordam a linguagem complementar destinada a definir as dimensões e unidades que vão ser utilizadas na linguagem principal. Explicam, respectivamente, as bases da implementação em ANTLR4, o funcionamento e sintaxe da linguagem, e ainda a análise semântica desta.

3.1 Implementação em ANTLR4

Para o funcionamento desta linguagem, foi implementada a gramática ***dimensions.g4***. No que toca ao *parser*, foi criado um mapa que tem como objetivo guardar toda a informação relativa a dimensões e unidades criadas, este auxilia também bastante o processo de análise semântica.

```
@parser::members {
    static protected Map<String,DimensionsType> dimTable = new HashMap<>();
}
```

Figura 1: Criação do mapa com a informação relativa às dimensões e unidades criadas

Quanto à gramática em si, trata-se de uma simples lista de instruções que suporta apenas duas, *"dim"* e *"unit"*, usadas para criar dimensões e unidades respetivamente. O funcionamento destas será explicado em detalhe na próxima secção.

```
main: statList EOF;

statList: (stat? ';')*;

stat: dim
    | unit
    ;

dim : 'dim' ID '(' ID ':' type ')'        #PrimitiveDim
    | 'dim' ID '(' (ID ':')? unitdim ')'  #RelativeDim
    ;

unit : 'unit' ID '(' ID ':' expr ')';
```

Figura 2: Parte da gramática correspondente à Linguagem Complementar

3.2 Funcionamento da Linguagem

3.2.1 Unidades de Dimensões

Para as unidades das dimensões recorreu-se a dois *HashMaps*, um que possui um *ID* como chave e como valor um *HashMap* representativo da unidade e outro que possui o *ID* dessa mesma unidade como chave e que guarda o seu valor de conversão, em relação à unidade default, como valor.

Relativamente ao *HashMap* da unidade, possui como chaves *Strings* correspondentes a cada unidade envolvida, e como valor, o expoente associado a essa unidade. Para comparar se duas unidades pertencem à mesma dimensão verifica-se através da função *equals* entre os *HashMaps* das unidades *default* de cada dimensão.

Um outro exemplo de como os *HashMaps* foram úteis, é nas operações de multiplicação, divisão e exponencial.

No caso da multiplicação, faz-se um *merge*, em que se as chaves (unidades) forem iguais, soma-se os valores (expoentes) associados, e se forem diferentes, adiciona-os. Depois são removidas todas as chaves que possuírem expoente a zero.

```
case "*":
   map1.forEach((k, v) -> map2.merge(k, v, (v1, v2) -> v1 + v2));
   map2.values().removeIf(f -> f == 0f);
```

Figura 3: Parte da multiplicação entre unidades de Dimensões

3.2.2 Criação de nova dimensão

A criação uma nova dimensão pode ser de raiz ou relativa a outras dimensões:

- **De raiz:** *dim [nome]([identificador da unidade base] : [tipo de dados])*
- **Relativa a outras dimensões:** *dim [nome]([operações entre dimensões])*

O tipo de dados apenas pode ser *Integer* ou *Double* e as operações entre dimensões apenas suportam a multiplicação ou a divisão, sendo que, no caso de alguma dimensão relativa possuir um tipo de dados *Double*, a nova dimensão a ser criada ficará com este tipo. A unidade base da nova dimensão relativa, será a resultante das operações entre as unidades bases das outras dimensões, suportando também as unidades adicionais destas.

Como exemplos de utilização:

- *dim Distance(m : Integer);*
- *dim Time (s : Double);*
- *dim Velocity(Distance/Time);*

3.2.3 Criação de nova unidade

A criação uma nova unidade associada a uma dimensão necessita das operações relativas à unidade base:

- *unit [nome da dimensão]([identificador da nova unidade] : [operações relativas a uma unidade da dimensão])*

Como exemplos de utilização:

- *unit Time(h : 3600*s);*
- *unit Velocity(mach: 200*m/s);*

3.3 Análise Semântica

A análise semântica é realizada com o *Visitor* denominado ***DimSemantic.java***. Este trata de todas as situações que se achou relevantes controlar para que se mantivesse a integridade da linguagem. Segue-se então a lista de regras semânticas que foram incluídas:

- Não é permitida a criação de dimensões ou unidades que já existam, no caso das unidades isto é verdade mesmo que em dimensões distintas.
- Não é permitido criar uma unidade para uma Dimensão não previamente existente.
- Na criação de unidades, a relação que define a mesma não pode incluir unidades pertencentes a outra Dimensão.
- Não é permitido criar unidades cuja definição corresponda ao valor 0.
- Na criação de unidades, a relação que define a mesma não pode incluir unidades não definidas previamente.
- Na criação de unidades, não é possível elevar uma expressão a uma unidade nem ao valor 0.

4 Linguagem Principal

As seguintes secções abordam a linguagem principal do nosso projeto e visam explicar, respectivamente, as bases da implementação em ANTLR4, o funcionamento e sintaxe da linguagem, e ainda a análise semântica desta.

4.1 Implementação em ANTLR4

Para o funcionamento desta linguagem, foi implementada a gramática ***chubix.g4***. Quanto ao parser foram criadas os contadores *"insideLoop"* e *"insideFunc"*, que são efectivamente predicados semânticos relativos a restrições por contexto das instruções iterativas e de funções respectivamente.

Adicionalmente foi criada uma *SymbolTable*, classe que visa guardar e interagir com as variáveis criadas, bem como o alcance (*scope*) delas, assimilando uma estrutura em árvore. Para a mesma árvore de *SymbolTables* existem duas referências, sendo que a *"global"* serve como referência para a raiz da árvore e trata do alcance global de um programa e a *"current"* para a *SymbolTable* a ser visitada num dado momento. Portanto, vão existir *SymbolTables* relativas a cada bloco que necessite de variáveis locais (condicionais, iterativos e funções) tendo em conta que a *"current"* tem também acesso às variáveis de todas as tabelas *"parent"* da mesma.

```
@parser::members{
  int insideLoop = 0;
  int insideFunc = 0;
  public static final SymbolTable global = new SymbolTable();
  public static SymbolTable current = global;
}
```

Figura 4: Criação de variáveis auxiliares à gramática e da tabela de símbolos utilizada no processamento da Linguagem Principal

Para exemplificar, na análise semântica da linguagem principal, sempre que se entra num bloco de código com um *scope* restrito internamente, é criada uma nova *SymbolTable* filha da *current*, e esta, a *current*, é alterada para se referir à filha criada. Na saída da função é feito um *up*, que coloca a *current* de volta para a *SymbolTable* anterior à entrada na função.

No compilador, para respeitar os scopes das variáveis também é feita a mesma estratégia. No entanto, o método que cria uma *SymbolTable* filha é agora substituído por um método que retorna a filha já criada e com os respetivos símbolos guardados. O retorno deste método obedece à ordem de criação das filhas na passagem do *visitor* para a semântica, obtendo assim o resultado pretendido.

```
Symbol sym = new Symbol(idFunc, new FunctionType(ctx.ret_type.res));
chubixParser.global.addSymbol(idFunc, sym);
chubixParser.current = chubixParser.current.addChild();
```

Figura 5: Criação de uma nova *SymbolTable* na mudança de *scope*

A gramática em si baseia-se em três partes do símbolo *main*, primeiramente um cabeçalho opcional de *imports* seguido de um bloco opcional de definição de funções e por fim a lista de instruções que constituem o corpo do programa. Para analisar a gramática completa dirija-se ao ficheiro ***chubix.g4***.

```
main: (importDim? ';')* (function? ';')* instList EOF ;

instList: (instruction? ';')*;

instruction:  print
            | assignment
            | conditional
            | forLoop
            | breakLoop
            | continueLoop
            | whileLoop
            | declare
            | callFunction
            | declAssig
            ;
```

Figura 6: Parte da gramática correspondente à Linguagem Principal

É ainda de mencionar, para futura referência, que uma expressão na nossa linguagem pode ser qualquer um dos seguintes:

- Uma variável.
- Um valor.
- Um input.
- A chamada a uma função.
- Uma variável seguida de incrementação/decrementação.
- A conversão de uma expressão.
- Uma expressão com sinal (+-).
- Uma operação entre expressões, utilizando os operadores aritméticos permitidos (+-*/^).
- Uma comparação entre expressões, utilizando os operadores relacionais permitidos (== != < > <= >=).

```
expr returns[Type exprType, String varName]:
     expr '['unitdim']'                                          #exprConvUnit
    | <assoc=right> e1=expr '^' e2=expr                          #powExpr
    | sign=('+'|'-') expr                                        #signExpr
    | e1=expr op=('*' | '/') e2=expr                             #multDivRestExpr
    | e1=expr op=('+' | '-') e2=expr                             #addSubExpr
    | e1=expr op=('==' | '!=' | '<' | '>' | '>=' | '<=') e2=expr #conditionalExpr
    | '(' expr ')'                                               #parenExpr
    | ID op=('++' | '--')                                        #doubleSumMin
    | 'input' '(' (STRING ',')? type')'                          #inputExpr
    | BOOLEAN                                                    #booleanExpr
    | ID                                                         #idExpr
    | STRING                                                     #stringExpr
    | callFunction                                               #functionExpr
    | DOUBLE                                                     #doubleExpr
    | INTEGER                                                    #integerExpr
    ;
```

Figura 7: Implementação em ANTLR4 da expressão

4.2 Funcionamento da Linguagem

4.2.1 Importar ficheiros da linguagem complementar

As instruções de importação que forem necessárias incluir no programa têm de ser efectuadas no inicio do mesmo. É possível importar qualquer ficheiro com terminação *.ubi* através do caminho para o mesmo, relativo ao directório do programa. **Como exemplos de utilização:**

- *import Exemplo.ubi;*
- *import ../dic1/dic2/Example.ubi;*

4.2.2 Definição de funções

Todas as funções têm de ser definidas no inicio do programa, mas após as instruções de import, sendo esta a sintaxe a usar:

```
function <tipo> <nome>(<argumentos>){
    <corpo da funcao>
;}
```

Tipo: O tipo de retorno da função, pode ser qualquer um dos existentes (Integer, Double, String, Boolean), qualquer dimensão criada previamente, ou ainda do tipo Void.

Nome: O nome atribuído à função, tem de começar por uma letra ou underscore.

Argumentos: lista de argumentos da função sob a forma de declarações de variáveis sem valor atribuído.

Corpo da função: Lista de instruções que definem a função, tem de incluir uma ou mais instruções de retorno:

return <expressão>;

No caso da função ser do tipo Void a instrução de retorno não inclui expressão.

Como exemplo de utilização:

```
function Integer Sum(Integer a, Integer b){
    return a + b;
};
```

4.2.3 Declaração de Variáveis

A declaração de variáveis é feita segundo a sintaxe a seguinte:

```
<tipo> <nome>;
```

Tipo: O tipo da variável, pode ser qualquer um dos existentes (Integer, Double, String, Boolean) ou qualquer dimensão criada previamente.

Nome: O nome atribuído à variável, tem de começar por uma letra ou underscore.

Como exemplos de utilização:

- *Integer a;*
- *String s;*
- *Time t;*

4.2.4 Atribuição de valor a Variáveis

É possível atribuir valor a uma variável previamente declarada, ou então na declaração de uma, segue-se a sintaxe:

```
<nome> = <expressao>;
```

ou então:

```
<tipo> <nome> = <expressao>;
```

É de notar que o tipo da expressão a atribuir tem de se conformar ao tipo da variável, e que no caso da mesma ser dimensional, a expressão deve incluir a unidade a utilizar.

Como exemplos de utilização:

- *Velocity v = 10.0 [m/s];*
- *Integer a = 10;*
- *t = 20 [s];*
- *i = a + 5;*

4.2.5 Instrução de Impressão

Para imprimir conteúdo utiliza-se a seguinte instrução:

```
print(<expressao>);
```

Como exemplos de utilização:

- *print("Olá!");*
- *print(2 [s] + t);*
- *print(5.55);*
- *print(5 > 2);*

4.2.6 Expressão Input

Embora não seja uma instrução em si, é possível obter dados pelo utilizador através do *input*, relembrando que trata-se apenas de um contexto de expressão. Sendo a sintaxe a seguinte:

```
input(<texto>, <tipo>);
```

ou então:

```
input(<tipo>);
```

Como exemplos de utilização:

- *Mass m = input("Insira a sua massa em kg",Mass) [kg] ;*
- *i = 100 - input(Integer);*
- *print(input("Diga o seu nome",String));*

4.2.7 Instrução Condicional

Uma das instruções mais importantes para uma típica linguagem de programação, foi utilizada uma estrutura básica de *if/else*:

```
if( <expressao> ){
    <lista de instrucoes>
} else if( <expressao> ){
    <lista de instrucoes>
} else{
    <lista de instrucoes>
};
```

É de notar que as expressões a passar têm de ser do tipo booleano.

Como exemplo de utilização:

```
if(num > 0){
    print("Positivo");
} else if (num == 0){
    print("Zero");
} else{
  print("Negativo");
};
```

4.2.8 Instruções Iterativas

Outro tipo de instrução essencial á maior parte das linguagens, no nosso caso implementámos dois blocos simples, a instrução *for* e *while*. Segue-se a sintaxe do ciclo for e while respectivamente:

```
for( <atribuicao> ; <expressao de termino> ; <incremento> ){
    <lista de instrucoes>
};
```

Atribuição: Instrução de atribuição de valor a variável que servirá para controlar o ciclo.

Expressão de término: Expressão do tipo booleano que visa definir quando o ciclo deve terminar.

Incremento: Instrução de atribuição de valor que dita o tipo de incremento que a variável de controlo vai ter.

```
while( <expressao> ){
    <lista de instrucoes>
};
```

Quanto ao ciclo while, a expressão passada tem de ser do tipo booleano. Há que mencionar que para ambos os ciclos é possível utilizar as instruções *break;* e *continue;* que terminam o ciclo e ignoram uma iteração respectivamente.

Como exemplos de utilização:

```
for(Integer i = 0; i<=100; i=i+10){
    print(i*2);
};

while(a<b){
    a++;
    b--;
    print("Dentro do ciclo");
};
```

4.3 Análise Semântica

A análise semântica é realizada com o *Visitor* denominado *SemanticChubix.java*. Este trata de todas as situações que se achou relevantes controlar para que se mantivesse a integridade da linguagem. Segue-se então a lista de regras semânticas que foram incluídas:

- Não é permitido criar funções com o mesmo nome.
- O valor de retorno de uma função tem de se conformar ao tipo da mesma.
- Não se pode chamar uma função não definida.
- Os argumentos passados na chamada a uma função tem de se conformar aos tipos, ou dimensões se dimensionais, requeridos pela mesma.
- O numero de argumentos na chamada de uma função tem de ser igual ao numero de argumentos definidos na mesma.
- Não é possível declarar variáveis já existentes no mesmo alcance (*scope*) ou alcances mais abrangentes.
- Para atribuir valor a uma variável esta tem de ser previamente declarada.
- Ao atribuir valor a uma variável, o valor tem de ser do tipo da variável.
- Ao atribuir valor a uma variável dimensional, a sua unidade passada tem de pertencer à dimensão requerida, e o seu valor tem de ser análogo à mesma, havendo excepção se a dimensão for do tipo *double*, e o valor do tipo *integer*.
- Numa instrução condicional é necessário passar uma expressão booleana, o mesmo acontece nas condições de instruções iterativas.
- Não é permitida a soma de valores pertencentes a dimensões distintas.
- É permitida a conversão de expressões numéricas a dimensões, exceptuando se a expressão for do tipo *double* e a dimensão *integer*.
- Não é permitido converter uma expressão dimensional noutra dimensão.
- Não é permitido chamar uma função do tipo void no contexto de uma expressão.
- Numa expressão não é possível elevar um valor dimensional a um valor não inteiro, ou a zero.
- Ao utilizar operadores relacionais, os elementos a comparar devem ser de tipos análogos.
- Para utilizar uma variável numa expressão, esta deve estar declarada e com valor atribuído.

5 Geração de Código

Para a compilação e geração de código foi utilizada a ferramenta *String Template* com o objetivo de gerar o código-fonte da linguagem, após a compilação, em *Java*.

Terminadas as verificações realizadas na análise semântica, a linguagem procede à compilação através de um visitor, *ChubixComp.java*, onde são introduzidas as instruções passadas pelo utilizador no ficheiro *chubix.stg*, um *STGroupFile* que procede à renderização do *String Template* com o código *Java* final, pronto para ser compilado e executado, de acordo com as regras implementadas pelo grupo.

O compilador da linguagem trabalha com o auxílio da tabela de símbolos, preenchida durante a análise semântica, para associar as variáveis criadas durante a compilação com o nome que o utilizador lhes deu, para obter o tipo de dados de variáveis, entre outros. Desta forma, é assegurada a compilação de acordo com as verificações feitas previamente. Ao trabalhar com a tabela de símbolos, todas as instruções passadas ao ficheiro *chubix.stg* permitem a compilação sem erros em Java.

6 Utilização

De modo a permitir a utilização da linguagem desenvolvida, seguem-se instruções para a sua devida utilização:

- **1º - Linguagem Complementar**

  Primeiramente é necessário a criação de um ficheiro da linguagem complementar com o formato *.ubi* que permita a definição de Dimensões e das respectivas Unidades de modo a serem utilizadas na Linguagem Principal.

- **2º - Linguagem Principal**

  Após a criação do ficheiro para a Linguagem Complementar, é, também, necessário a criação de um para a Linguagem Principal. O ficheiro deverá estar no formato *.ubix* e deve seguir as regras explicadas previamente, nomeadamente a importação do ficheiro relativo à Linguagem Complementar.

- **3º - Geração do ficheiro Java**

  Com os ficheiros de ambas as linguagens prontos, deve-se executar o *Main* da Linguagem Principal com o nome do ficheiro como argumento:

  ```
  java -ea chubixMain [ficheiro da Linguagem Principal]
  ```

  Após esta execução sem erros, é gerado um ficheiro *.java* com o nome do ficheiro da Linguagem Principal utilizado como argumento.

- **4º - Compilação e execução**

  Por fim, deve-se compilar o ficheiro *.java* com recurso ao comando *javac* e executá-lo usando o comando *java*.

Em caso de necessidade, existem programas exemplo preparados na secção seguinte.

7 Programas de Exemplo

Nesta secção serão demonstrados alguns exemplos de utilização da linguagem criada. Os programas de exemplo seguintes encontram-se disponíveis dentro da pasta */tests* na raíz do repositório, sendo que os referentes à Linguagem Complementar encontram-se na pasta */tests/dimensions* e os referentes à Linguagem Principal em */tests/chubix*.

7.1 Programas Corretos

7.1.1 Linguagem Complementar

Para o bom funcionamento da Linguagem Principal no seu programa de exemplo correto, todas as dimensões e unidades foram definidas no ficheiro */tests/dimensions/dimensionsExamples.ubi*:

```
dim Time(s: Integer);
unit Time(h: 3600*s);

dim Distance(m: Double);
unit Distance(km : 1000*m);

dim Velocity(Distance/Time);
unit Velocity(mach: 200*m/s);

dim Acceleration(Distance/Time^2);

dim Mass(kg: Double);

dim Force(N: Mass*Acceleration);

dim Potential(V: Double);
unit Potential(MV: 10.0^6.0*V);

dim Resistence(O: Double);

dim Current(A: Potential/Resistence);

dim Energy(J: Mass*(Distance^2.5)*Time^-2);

dim Power(W: Energy/Time);
unit Power(KW: W*10.0^3.0);

dim Price(euro : Double);

dim PricePerPowerTime(Price/(Power*Time));

dim Charge(C: Current*Time);

dim IMC(imc : Mass/Distance^2);
```

7.1.2 Linguagem Principal

Após a definição do ficheiro da Linguagem Complementar, foram criados os seguintes exemplos corretos para a Linguagem Principal no ficheiro */tests/chubix/compilerExamples.ubix*:

Definição de funções e importação das dimensões e unidades:

```
import ../tests/dimensions/dimensionsExamples.ubi;

function Void evalIMC(IMC imc){
    if (imc<16[imc]) {
        print("You are severely underweight.");
    } else if (imc<=19.9[imc]) {
        print("You are underweight.");
    } else if (imc<=24.9[imc]) {
        print("You have a healthy weight.");
    }else if (imc<=29.9[imc]) {
        print("You are overweight.");
    }else if (imc<=39.9[imc]) {
        print("You are obese.");
    }else if (imc>40[imc]) {
        print("You are morbidly obese.");
    };
    return;
};

function Energy CalcCinetica(Velocity v, Mass  m) {
    Energy ec = 0.5*m*v^2;
    return ec;
};

function IMC getIMC(Mass m, Distance height){
    IMC imc = m/height^2;
    return imc;
};
```

Exemplo 1:

Determinar distância percorrida por uma massa de 5kg em 5 segundos.

```
Time t = 5[s];
Mass m = 5 [kg];
Velocity v0 = 5[m/s];
Force f = 5 [kg*m*s^-2];
Acceleration a = f / m;

Distance res = (v0 * t +(a*t^2)/2)[km];

print("Distancia percorrida: ");
print(res);
```

Exemplo 2:

Um circuito elétrico tem uma resistência de o Ohms e uma fonte de tensão de v Volts. Diminuindo a intensidade de corrente para 1/2,1/4...1/32 e mantendo a mesma tensão, pretende-se descobrir o valor em Ohms da nova resistência a aplicar em cada caso.

```
Potential U =  input("Insira a tens o de corrente em volts", Potential) [V];
Resistence R = input("Insira a intensidade da for a , em Newton", Resistence) [O];

Double taxa = 1/2;
Current I = U/R;  # V / O  {'A' : 1, 'V/O' : 1}

while(taxa>=1.0/32.0){
    print("Taxa: ");
    Resistence result = (U/taxa*I^-1) [O];
    print(taxa);
    print("Nova Resist ncia: ");
    print(result);
    taxa = taxa / 2;
};
```

Exemplo 3:

Determinar a energia consumida mensalmente por um chuveiro elétrico de potência 4000W em uma residência onde vivem quatro pessoas que tomam, diariamente, 2 banhos de 15 min.

```
Power P = input("Insira a pot ncia: ", Power) [KW];
Time T = input("Insira o tempo da dura  o de 1 banho: ", Time) [h];

# Da equa  o da energia consumida temos que E = P x  t
# Sabendo que s o 8 banhos com dura  o total de 120 min (2h) e considerando os 30 dias do m s , temos:
#   E = 4000 . 2 . 30 = 192.000 = 192 Kwh

Energy E = P * 8 * T * 30;

print("Energia consumida: ");
print(E);
```

Exemplo 3.1:

Na residência pretende-se gastar menos de 30 euros mensalmente com a energia gasta pelo chuveiro. Sabendo que o preço per kWh é de 0.14810 euros, conseguem alcançar este objetivo?

```
PricePerPowerTime p = input("Pre o por kWh: ", PricePerPowerTime) [euro/(KW*h)];

Price p_mensal = p * E;

if (p_mensal < 30[euro]){
    print("N o conseguem gastar menos de 30");
}
else {
    print("Conseguem gastar menos de 30");
};
```

Exemplo 4:

Calcular o a Energia cinética em intervalos de 10s no primeiro minuto do movimento de queda livre. Considere que a velocidade inicial é 0m/s e que a aceleração gravítica é 10m/s^2 e que não existem forças de atrito.

```
Mass m1 = input("Insira a massa do objeto em kg",Mass) [kg] ;
Acceleration g = 10 [m*s^-2];
Velocity v = 0[m/s];
Energy ec;

for (Integer i = 0; i<=60; i=i+10){
    v = v+i[s]*g;
    ec = CalcCinetica(v, m1);
    print(ec);
};
```

Exemplo 5:

Uma corrente de 5,0 A atravessa um fio durante 4,0 min. Quanto vale (a) a carga que passa por uma secção desse fio e (b) a quantos eletrões corresponde?

```
#a)

Current cur = 5 [A];
Time time = 4*60 [s];

#Visto que a Intensidade = carga / intervalo de tempo
Charge ch = cur*time ;

print(ch);

#b)

#N mero de eletr es = carga / |carga por eletr o|
Double eletroes = ch / (1.602*10^-19)[C];

print(eletroes);
```

Exemplo 6:

Calcular e avaliar o valor de IMC do utilizador.

```
Mass weight = input("Insert your weight in kg: ", Mass) [kg];
Distance height = input("Insert your height in meters: ", Distance) [m];


IMC imc = getIMC(m,height);
evalIMC(weight/height^2);
```

7.2 Programas com Erros

Foram desenvolvidos alguns programas com erros com o objetivo de verificar se toda a análise semântica da Linguagem funcionava como o esperado. Em cada linha onde é esperado um erro, existe um comentário com o erro.

7.2.1 Linguagem Complementar

```
dim Time(s:Integer);

dim Time2(c:Time);              #[ERROR at line 3] Unit "s" already defined.

unit Time(h: 0 * s);            #[ERROR at line 5] Cannot add a unit with value 0.
unit Time(s: 12*s);             #[ERROR at line 6] Unit "s" already defined.
unit Time(h: 1000);             #[ERROR at line 7] Conversion expression must be a unit.
unit Time(h: s*10);             #[ERROR at line 8 Dimension "Time" and "s^0" are not compatible.

unit Tempo(h : 3600*s);         #[ERROR at line 10] Dimension "Tempo" not defined.

dim Distance(m:Double);

dim Distance(km : Double);      #[ERROR at line 14] Dimension "Distance" already defined.

unit Time(h:3600*m);            #[ERROR at line 16] Dimension "Time" and "3600*m" are not compatible.

unit Distance(cm : 0.01*m);
unit Distance(in: 2.54*cm);     #No error when refering to a unit which is not the default.

dim Velocity(Distance/Time^0);  #[ERROR at line 21] Power of 0 is not possible when defining a dimension.
```

7.2.2 Linguagem Principal

```
import ../tests/dimensions/dimensionsExamples.ubi;

function Random CalcCinetica(Velocity v, Mass  m) {#[ERROR at line 3] Dimension "Random" does not exist.
    Energy ec = 0.5*m*v^2;
    return ec;
};

function Energy CalcCinetica(Velocity v, Mass  m) {
    Energy ec = 0.5*m*v^2;
    return ec;
};

function Energy CalcCinetica(Velocity v, Mass  m) {#[ERROR at line 13] Function "CalcCinetica" already exis
    Energy ec = 0.5*m*v^2;
    return ec;
};


function Void catch(Double x) {
    print(x);
    return;
};

# main

catch(10.0);
catch();                    #[ERROR at line 27] Number of arguments do not match.
CalcCinetica(1 [m], 1 [kg]);#[ERROR at line 28] Incomparable types: m and m / s^1
```

```
Distance m = 10;            #[ERROR at line 30] Expression type does not conform to variable "m" type!

Distance m3 = 10[km];       #[ERROR at line 32] There is no dimension with that unit!

Distance m2 = 10[m];
print(m2);

print(m3[m]);
print(m3[s]);               #[ERROR at line 38] Cannot convert to another Dimension!

Time t = 1.2 [s];           #[ERROR at line 40] Cannot convert Double to a Dimension of Integers!

print(catch(10.0) + 1);     #[ERROR at line 42] Expression cannot be void
print("ola"*2);             #[ERROR at line 43] Cannot fetch a type between "ola" and 2

String name = input("Insert your name",String);
Distance r=input("Insert distance in km",Distance)[km];

Distance mm = 1[m];
if(mm>1){                   #[ERROR at line 49] Incomparable types: mm and 1
    print("Maior");
}else{
    print("Menor");
};

Double x = 0;
for (Integer i = 0;i<=2;i=i+1){
    x = 10;
    Integer y = i;
};
print(y);                   #[ERROR at line 60] Variable "y" does not exist!


Integer i = 0;
Integer y = 2;
for (i = 0;i<=2;i=i+1){
    i = 10;
    Integer y = i;          #[ERROR at line 67] Variable "y" already declared!
};

while(1){                   #[ERROR at line 70] Condition "1" isnt a Boolean expression!
    print("Not boolean");
};

while(true) {
    Double p;
    print(-"hello");        #[ERROR at line 76] Numeric operator applied to a non-numeric operand!
};
Double p;                   #No error

a = 2;

Boolean qwe = p>mm;         #[ERROR at line 82] Variable "p" not defined!
Boolean qwq = 1>mm;         #[ERROR at line 83] Incomparable types: 1 and mm
```

8 Conclusão

Devido à falta de tempo perante as múltiplas entregas que se acumularam no final do semestre, alguns aspetos do trabalho não foram tão bem explorados. A documentação é um exemplo prático deste problema, onde o grupo, perante outras datas de avaliação e cansaço, não conseguiu dedicar o tempo que pretendia inicialmente.

Outro fator que se considera ter influenciado negativamente o desenvolvimento do trabalho incide na situação atual que se vive, uma vez que o grupo teve de se adequar a um método de trabalho à distância, algo que anteriormente nunca tinha feito.

No geral, a análise semântica foi a parte do trabalho onde se sentiu mais dificuldades, tendo sido necessário a reescrita do código várias vezes após nova análise da documentação do docente.

Todos os elementos do grupo concluíram que o desenvolvimento de trabalho favoreceu a compreensão dos tópicos abordados e que será uma mais valia para para a avaliação prática individual.

Por fim, acredita-se que todos os objetivos do trabalho foram cumpridos com sucesso.

9 Contribuições dos autores

Segue-se a contribuição de cada elemento do grupo:

- Bruno Bastos - 20%
- Hugo Almeida - 20%
- Mário Silva - 20%
- Leandro Silva - 20%
- Rui Fernandes - 20%

10 Bibliografia

[1] Material fornecido pelo docente da disciplina