DIMX - Linguagem para Análise Dimensional

Trabalho Realizado por: Vasco Ramos - nº mec 88931
Diogo Silva - nº mec 89348
Alina Yanchuk - nº mec 89093
Tiago Mendes - nº mec 88886
João Vasconcelos - nº mec 88808

Data de Preparação: Aveiro, 07 de Julho de 2019

Cadeira: Compiladores

Corpo Docente: Professor Miguel Oliveira e Silva
Professor Artur Pereira
Professor André Zúquete

universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Índice

1. Introdução

O presente documento tem como objetivo descrever detalhadamente todo o trabalho prático desenvolvido no âmbito da unidade curricular de Compiladores da Universidade de Aveiro. Este trabalho foi desenvolvido pelos alunos apresentados na capa deste documento, maioritariamente sob a orientação do docente Prof. Dr. Miguel Oliveira e Silva.

O principal objetivo deste trabalho prático era o de desenvolver um compilador, tendo em conta todas as fases de construção de uma linguagem de programação.

Para atingir esse fim, foi escolhido um dos temas propostos no guião do trabalho prático, referente ao desenvolvimento de uma linguagem de programação para **análise dimensional**.

Assim, foi desenvolvida uma linguagem de programação principal para a análise dimensional, servindo de base à construção do compilador, bem como uma linguagem complementar, incorporada pela primeira, que permite especificar as unidades e grandezas físicas a serem utilizadas.

Para garantir a integridade da mesma, foram implementadas análises léxica e sintática, bem como todas as regras semânticas que permitam um correto funcionamento de programas.

No geral, a nossa linguagem apresenta as seguintes características:

- Definição de novos tipos de dados (isolados ou dependentes de outros).
- Operações de álgebra dimensional de caráter lógico, resolvendo assim o problema de operações que gerem erros dimensionais.
- Operações comuns a qualquer linguagem de programação (definição de variáveis, operações interativas com o utilizador, expressões algébricas, instruções iterativas, expressões booleanas e instruções condicionais).
- Gramática bastante compreensível e fácil de utilizar.
- Geração de código na linguagem destino Java.

2. Linguagem Secundária

2.1 Gramática

Foi desenvolvida a gramática **Quantities.g4**, que permite declarar grandezas físicas e associar-lhes unidades. Estas grandezas podem ser grandezas totalmente independentes ou podem ser definidas em função de grandezas previamente definidas.

2.1.1 Declaração de Grandezas Físicas

A declaração de uma grandeza física é feita de seguinte modo:

```
Grandeza : tipo [ unidade ];
```

Ou então, em função de outras grandezas:

```
Grandeza : Grandeza operador Grandeza [ unidade operador unidade ];
```

Tendo em conta que:

- O tipo apenas pode ser inteiro ou real.
- O operador pode ser '/' ou '*'.

Exemplos de declarações permitidas:

- `Distance : real [m];`
- `Time : int [s];`
- `Speed : Distance / Time [m/s];`

Exemplos de declarações não permitidas:

- `Distance real [m];`
- `Time: int [m];` (a unidade m já tinha sido previamente usada em `Distance: real [m]`)
- `Speed: Distance / Time [s/m];` (caso Distance e Time sejam definidos como anteriormente - `Distance: real [m] ; Time: int [s]`).

2.2 Análise Semântica

A análise semântica é efetuada através de um visitor no ficheiro ***SemanticCheckQuantities.java***. Para garantir a integridade da linguagem desenvolvida são verificados os seguintes aspetos:

- Não são permitidas declarações de grandezas já anteriormente declaradas.
- Não são permitidas declarações de grandezas em função de outras ainda não declaradas.
- Não são permitidas unidades repetidas em grandezas diferentes.
- No caso das grandezas definidas em função de outras, não é permitido discrepâncias entre as unidades da nova grandeza e as operações entre as grandezas constituintes da mesma.

3. Linguagem Principal

3.1 Gramática

Foi desenvolvida a gramática **General.g4**, que permite não só um grande conjunto de operações típicas de uma linguagem de programação, bem como operações próprias da nossa linguagem, tais como:

- **Declaração de variáveis;**
- **Atribuição de valores a variáveis;**
- **Impressão de conteúdo (print);**
- **Entrada de conteúdo pelo utilizador (input);**
- **Expressões booleanas;**
- **Instruções condicionais;**
- **Instruções iterativas;**
- **Verificação de grandezas.**

A gramática aceita também, no início, o **import de ficheiros de texto** (.txt) que contenham a linguagem secundária, ou seja, grandezas declaradas, realizando toda a verificação necessária. Este import é **opcional** e, caso seja o desejado, pode ser feito import de vários ficheiros. Para referências futuras, a expressão pode ser:

- Outra variável;
- Um valor;
- Um valor com sinal (+ -);
- Potências de valores;
- Uma operação entre expressões (com os operadores aritméticos permitidos: + - * /);
- Uma operação entre expressões (com os operadores lógicos permitidos: and or);
- Uma comparação entre expressões (com os operadores relacionais permitidos: == != === !== < > <= >=)
- Negação de uma expressão;

- Input.

3.1.1 Declaração de Variáveis

A declaração de variáveis é feita de seguinte modo:

```
<tipo>  nome_da_variável ;
```

Os tipos permitidos são:

- Tipos já existentes (Integer, String, Boolean, Real);
- Tipos novos definidos pelo utilizador na linguagem secundária **(ver secção 2)**.

O **nome_da_variável** pode ser um carater ou String.

Exemplos de declarações permitidas:

- `Time t;`
- `Distance d;`
- `Speed speed;`
- `Integer i;`

Exemplos de declarações não permitidas:

- `Integer i; Integer i;` (erro na segunda declaração)
- `Time t;` (se a grandeza Time não estiver definida no ficheiro de import, então a declaração seguinte não é permitida)

3.1.2 Atribuição de Valores a Variáveis

A atribuição de valores a variáveis é feita de seguinte modo:

```
nome_da_variável  =  expressão ;
```

Ou declarando a variável primeiro:

```
Grandeza   nome_da_variável   =  expressão ;
```

Exemplos de declarações permitidas:

- `Time t = 470;`
- `Distance d; d = 10;`

Exemplos de declarações não permitidas:

- `Time t = 4; Distance d = 1; t = d;` (apesar de a gramática aceitar a atribuição <variável_1> = <variável_2>, as duas variáveis têm de ser da mesma grandeza)

3.1.3 Impressão de Conteúdo

A impressão de conteúdo é feita de seguinte modo:

```
print ( expressão ) ;
```

Exemplos:

- `print ("Hello World!");`
- `print(2.11);`
- `print(d);` (seja a variável d da declaração: Distance d = 2 [m];)

3.1.4 Entrada de Conteúdo pelo Utilizador

A entrada de conteúdo é feita dos seguintes modos (alternativos):

```
var_name =  input ( prompt ) ;
String var_name = input ( prompt ) ;
```

Em que **var_name** e **prompt** têm de ser obrigatoriamente do tipo String, sendo este último uma mensagem representativa mostrada ao utilizador.

Exemplos corretos:

- `String s = input ( "Mensagem representativa" );`

Exemplos incorretos:

- `input ("Input") ;`

3.1.5 Expressões Booleanas

Relativamente à utilização de expressões booleanas na nossa linguagem, definimos na nossa gramática diferentes operadores de comparação, em particular de igualdade, relação, negação e reunião/interseção. É de realçar que estas expressões booleanas não podem por si só constituir um *statement* comum (i.e.: 2 == 1;).

3.1.5.1 Operadores de igualdades

Quanto aos operadores de igualdade '==' e '!=', estes permitem comparar dois valores do mesmo tipo (ou numérico ou booleano), enquanto que os operadores '===' e '!==' apenas permitem comparar Strings.

Exemplos corretos:

- *1 == 2*
- *2 != 1.0*
- *true == false*
- *“String” === “Another String”*

Exemplos incorretos:

- *1 == true*
- *“String” != “Another String”*
- *1.0 === 2.0*
- *“String” === 1.0*

3.1.5.2 Operadores de relação (relacionais)

Quanto aos operadores de relação ‘<’, ‘>’, ‘<=’ ou ‘>=’, estes permitem apenas comparar dois valores numéricos.

Exemplos corretos:

- *1 > 2*
- *1 < 2*
- *1 >= 2*
- *1 <= 2.0*

Exemplos incorretos:

- *1 < true*
- *1 > “string”*
- *2.0 >= “another string”*
- *1.0 <= false*

3.1.5.3 Operador de negação

Relativamente ao operador de negação ‘not’, ele apenas inverte o valor de uma **expressão booleana**.

Exemplos corretos:

- `not true` (false)
- `not 1 == 1` (false)
- `not “string” === “string”` (false)
- `not 1 > 2` (true)

Exemplos incorretos:

- `not 1`
- `not a`
- `not “string”`

3.1.5.4 Operadores de interseção/reunião (and/or)

Relativamente aos operadores de interseção/reunião, respetivamente, ‘and’ e ‘or’, e à semelhança do operador de negação, apenas podem ser utilizados para comparar duas expressões booleanas.

Exemplos corretos:

- `1 < 2 and 2 <= 3` (true and true == true)
- `1 > 2 and true` (false and true == false)
- `not “string” === “another string” or false` (not false or false == true)

Exemplos incorretos:

- `1 and 2`
- `true or “string”`
- `not false or 1.0`

3.1.6 Instruções Condicionais

Qualquer linguagem de programação que se preze deve possuir instruções condicionais, de modo a construir programas minimamente interessantes. Portanto, decidimos implementar a instrução condicional 'if' de acordo com a seguinte sintaxe:

```
if <expr>  {
   <statements_list>
} else if <expr> {
   <statements_list>
} else {
   <statements_list>
};
```

Em que <expr> deve ser uma expressão booleana.

Exemplo correto:

```
if  1 < 2 {
        print("1 é menor que 2");
} else if 1 > 2 {
        print("1 é maior que 2");
} else {
        print("1 é igual a 2");
};
```

3.1.7 Instruções Iterativas

À semelhança das instruções condicionais, a presença de instruções iterativas numa linguagem de programação é essencial, de modo a construir ciclos lógicos, em particular utilizando as instruções 'for' e 'while'. De seguida, apresenta-se a sintaxe destas duas instruções, bem como exemplos de utilização.

```
for <assign>, <break_condition>, <increment_condition> {
      <statements_list>
};

while <expr> {
      <statements_list>
};
```

Exemplos de utilização:

```
Integer i;
for i = 0, i < 10, i + 1 {
       print(i);
};
```

```
Integer i2 = 0;
while i2 < 10 {
       print(i2);
       i2 = i2 + 1;
};
```

3.1.8 Verificação de grandezas

A verificação de grandezas é feita de seguinte modo:

```
( expr ).dimension ;
```

Ou

```
( expr ).unit ;
```

O resultado é um print de uma frase que indica qual é o tipo ou unidade de uma dada expressão.

Exemplos:

- `(t).dimension;`
- `(d).unit;` (sejam t e d duas variáveis previamente declaradas e definidas)

3.2 Análise Semântica

Relativamente à análise semântica existem vários aspetos que temos de garantir, tais como a consistência nas dimensões utilizadas, a proibição de certas operações para certos tipos de dados, entre outros.

Para facilitar esta análise criámos duas classes abstratas **Type** e **Value** e um conjunto de classes que herdam das mesmas (como, p. ex., **StringType** e **StringValue**) que representam os vários tipos de dados permitidos na nossa linguagem.

Sempre que uma expressão é utilizada, ela retorna o **Tipo** da mesma, a **Dimensão**, a **Unidade** e o **nome da variável** (utilizado na geração de código) facilitando, portanto, a análise semântica, permitindo facilmente verificar se duas variáveis pertencem à mesma dimensão, se têm a mesma unidade e outros aspetos igualmente importantes.

Para garantir a integridade da linguagem principal verificamos vários aspetos como, por exemplo, se uma variável está previamente declarada para poder ser utilizada e ter um valor associado. Ao declarar uma variável e ao associar-lhe um valor, se o tipo for **não primitivo**, verificar se o tipo foi previamente declarado na linguagem de Quantidades e se a unidade que está presente na declaração está associada a esse tipo. Além disso se for declarada uma variável de um tipo não primitivo, é obrigatório especificar a unidade associada.

Nas operações de soma e subtração garantimos que ambos os operandos pertencem à mesma dimensão e que, se especificarmos unidade para um dos operandos, somos obrigados a especificar para ambos. Além disso garantimos que ambos os operandos têm que ter a mesma unidade para as operações puderem ser realizadas.

Nas operações de multiplicação e divisão existe uma maior "liberdade" no que é permitido fazer. Ao contrário da adição e subtração se um dos operandos tiver como unidade "Void" o resultado final irá ficar com a unidade do outro operando e se os operandos tiverem unidades diferentes simplesmente multiplicamos as unidades ou dividimos as mesmas, por exemplo:

```
Distance t = 2.0 [m];
Time h = 1.0[s];
```

```
Speed = t/h; // a unidade final irá ser [m/s];
```

Ao dividir as unidades, se as mesmas forem iguais, o resultado final irá ser **adimensional**. Por fim, verificamos se ambos os operandos são numéricos e se as operações podem ser realizadas para aquele tipo de operandos.

Na potência garantimos que o expoente é adimensional e que a sua unidade é Void. Se estas condições não se verificarem a operação não poderá ser realizada.

Na utilização dos operadores de interseção/reunião *AND* e *OR*, garantimos que ambos os operadores são booleanos e, na utilização das operações “>”, ”>=”, ”<” e ”<=” , garantimos que os operadores são numéricos.

Na utilização de operações de igualdade garantimos que ‘===’ e ‘!==’ apenas podem ser utilizados para strings e que ‘==’ e ‘!=’ podem ser utilizados para operadores numéricos, mas não para strings.

Na utilização de **if**, **while** e **for** garantimos que as condições são booleanas e, na utilização das operações de módulo (resto) garantimos que apenas pode ser utilizada em operadores numéricos.

Também asseguramos que o “-” ou “+” e a operação de valor absoluto apenas pode ser aplicado a operadores numéricos.

Por fim, na presença de uma ou mais instruções **import**, verificamos se os ficheiros a ser importados estão semanticamente corretos através do respectivo visitor da árvore sintática ***SemanticCheckQuantities.java***. Se estiverem corretos, as quantidades declaradas nos mesmos poderão ser utilizadas na linguagem principal.

4. Geração de Código

Relativamente à compilação e consequente geração de código executável, optámos por escolher a linguagem de programação **Java** como linguagem destino. A escolha não foi arbitrária, uma vez que esta linguagem é aquela na qual nos encontramos mais confiantes e confortáveis em termos de conhecimento da sintaxe e funcionamento.

O primeiro passo da geração de código foi a criação de um **StringTemplate GroupFile - java.stg -** que, utilizando a ferramenta String Template, nos permitiu gerar o código Java equivalente ao código da nossa **linguagem DimX**.

A compilação (e consequente geração de código na linguagem destino) é efetuada pelo visitor ***DimXCompiler.java*** que, após ter sido realizada uma verificação semântica através dos visitors mencionados nas seções anteriores, irá percorrer novamente todas as regras presentes na árvore sintática e preencher todos os espaços presentes no java.stg. No final, é efetuada uma **renderização** do String Template final.

Para efetuar a compilação da nossa linguagem, foi criado um ficheiro em bash, denominado de **dimx-compiler.sh**, que apenas aceita ficheiros com o formato **<file_name>.dmx**, passados como argumento. Após este ficheiro com a extensão .dmx ter sido analisado pelo ***DimXCompiler.java***, irá ser gerado um **<file_name>.java** com o código executável Java, já renderizado como explicado no parágrafo anterior.

Para executar o código gerado, foi também criado um ficheiro em bash, denominado de **dimx.sh**, que aceita como argumento nomes de ficheiro com o formato <file_name> (à semelhança do Java). Estes dois scripts criados servem apenas para criar uma abstração à compilação e execução dos nosso ficheiros .dmx, escondendo ao programador a linguagem destino (neste caso o Java).

Exemplo de compilação e execução de ficheiros .dmx:

```
$ ./dimx-compiler.sh p1.dmx p2.dmx
$ ./dimx.sh p1
$ ./dimx.sh p2
```

É ainda de realçar que todo o código gerado na linguagem destino está escrito utilizando a notação **TAC - Triple Address Code**, visto ser a notação mais adequada para a robusta compilação de código.

Por fim, note-se ainda que aquando da especificação do ficheiro .dmx a ser compilado, este deve-se encontrar no mesmo diretório dos restantes ficheiros necessários a uma correta compilação.

5. Exemplos de Programas

5.1 Programas Corretos

5.1.1 Declaração de grandezas (programa_definicao_dimensoes_1.txt)

```
Distance : real [m];
Time : int [s];
Speed : Distance / Time [m/s];
Capacidade : real [l];
```

5.1.2 Programa 1 (programa_de_fisica_1.dmx)

Um condutor sabe que percorreu 150 000 metros em 1 hora, num autoestrada, e desconfia que a sua velocidade pode ter ultrapassado o valor máximo (120 km/h ou, aproximadamente, 33,5 m/s). O condutor poderá ser multado? O seguinte programa dá-nos a resposta.

```
import programa_definicao_dimensoes_1; //ver DECLARAÇÃO DE GRANDEZAS

Speed velocidade;
Time tempo = 3600 [s] ;
Distance distancia_percorrida = 150000.0 [m] ;
Velocidade = distancia_percorrida/tempo;

if velocidade> 33.5 {
	print("Excesso de velocidade! Multado.");
}
else {
	print("Não houve excesso de velocidade. Não multado.");
};
```

5.1.3 Programa 2 (programa_de_fisica_2.dmx)

Um aluno pretende saber o volume de solução que tem, sabendo que a densidade da mesma é de 660 g/L.

Sabendo que a densidade calcula-se em função da massa e do volume, o seguinte programa apresenta uma "tabela" que mostra a densidade da solução preparada pelo aluno, em função de vários volumes possíveis, para que o aluno a consulte e obtenha a resposta à sua dúvida.

```
// ver DECLARAÇÃO DE GRANDEZAS
import programa_definicao_dimensoes_1;
import programa_definicao_dimensoes_2;

Massa soluto = 30 [g];
Massa solvente = 300 [g];
Massa massa_total = soluto + solvente;

(soluto).dimension;
(soluto).unit;

Capacidade volume = 0.1[l];
Densidade densidade;

while volume < 1 {

	densidade = massa_total / volume;
	print ( "Densidade:  " + densidade + "  Volume necessário: " + volume
	);
	Volume = volume + 0.1[l];

};
```

5.2 Programas Errados

5.2.1 Programa 1 (programa_erro_1.dmx)

Programa simples que dá erro devido à impressão de uma variável não declarada.

```
print(d);
```

5.2.2 Programa 2 (programa_erro_2.dmx)

Programa simples que dá erro por a grandeza **Height** não estar declarada na linguagem secundária (importada).

```
import programa_definicao_dimensoes_1; // ver DECLARAÇÃO DE GRANDEZAS

Height m =2 [m];
```

5.2.3 Programa 3 (programa_erro_3.dmx)

Programa simples que dá erro devido à soma de operandos de diferentes grandezas e unidades.

```
import programa_definicao_dimensoes_; // ver DECLARAÇÃO DE GRANDEZAS

Distance m = 2.0 [m];

Time h = 2[s];

print(h + m);
```

5.2.4 Programa 4 (programa_erro_4.dmx)

Programa simples que dá erro devido à multiplicação de um operando numérico com outro não numérico.

```
import programa_definicao_dimensoes_1; // ver DECLARAÇÃO DE GRANDEZAS

Distance m = 2.0 [m];

Time h = 2[s];

print(h*"ola");
```

6. Classificação

Nesta tabela encontram-se os temas abordados neste projeto e a respetiva divisão entre os vários elementos do grupo.

Temas	Autor
Gramática de Dimensões	Vasco Ramos, João Vasconcelos, Alina Yanchuk
Semantic Check Dimensões	João Vasconcelos, Vasco Ramos
Gramática de DimX	Diogo Silva, Vasco Ramos, João Vasconcelos, Tiago Mendes, Alina Yanchuk
Semantic Check DimX	João Vasconcelos, Vasco Ramos, Tiago Mendes
Compilador DimX	Diogo Silva, Tiago Mendes

Apresentamos agora a contribuição de cada elemento do grupo, em percentagem, no qual acreditamos ter sido a mais realista:

- Vasco Ramos - 20 %
- Diogo Silva - 20 %
- Alina Yanchuk - 20 %
- Tiago Mendes - 20 %
- João Vasconcelos - 20 %

7. Notas Finais

Antes de tecer as últimas conclusões, gostaríamos de fazer alguns comentários e, sobretudo analisar algumas das limitações da nossa solução e discutir algumas opções de melhoria para o futuro.

Devido a limitações, e com o objetivo de garantir que conseguiríamos desenvolver uma linguagem funcional e que pudesse ser utilizada, tomámos algumas decisões que limitaram um pouco a nossa linguagem.

As principais limitações da nossa linguagem relacionam-se com a incapacidade de definir prefixos na definição de novas dimensões/ quantidades, como m (mili), k (quilo), entre outros, com a definição de unidades e simplificação das unidades resultantes de operações entre dimensões como por exemplo simplificar [m.t/m] para só [t].

No futuro gostaríamos de nos focar nestas funcionalidades com o objetivo de enriquecer a linguagem DimX e acrescentar outras funcionalidades como a definição e utilização de funções definidas pelo utilizador e outras operações e funções aritméticas.

8. Conclusão

A realização deste trabalho foi fundamental para desenvolver e consolidar a nossa compreensão sobre as diferentes etapas do processo de criação de uma linguagem. Foi também uma excelente oportunidade para melhorar os nossos conhecimentos sobre as ferramentas String Template, ANTLR4 e Java, bem como capacidades transversais de pesquisa e autonomia que se provaram fulcrais para superar os problemas encontrados e obter as melhores soluções possíveis para os mesmos.

Em retrospectiva, cremos que os objetivos principais relativos a este trabalho foram atingidos, sendo que, como já foi falado no ponto anterior, outros ficaram por testar e/ou implementar. Para além de tudo o que foi descrito neste documento, é possível observar toda a extensão das funcionalidades implementadas através dos vários exemplos de código fonte que se encontram em anexo.

9. Bibliografia

- Enunciado e guião fornecidos pela equipa docente para apoio do trabalho.
- Slides das aulas teóricas fornecidos na página do elearning da cadeira.