Compiladores

Miguel Oliveira e Silva
Artur Pereira

Linguagem para análise dimensional

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

DETI
Universidade de Aveiro
16-06-2020

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 á 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 á 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 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);*

3.2.2 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 achámos relevantes controlar para que se mantivesse a integridade da nossa linguagem. Segue-se então a lista de regras semânticas que incluímos:

- 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.
- Dim semantic linha 190 e 202
- 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 parser foram criadas os contadores *"insideLoop"* e *"insideFunc"*, 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. Assimilam uma estrutura em árvore sendo que a *"global"* serve como raiz da árvore e trata do alcance global de um programa. Vão existir *SymbolTables* relativas a cada bloco que necessite de variáveis locais (condicionais, iterativos e funções) tendo em conta que a *"current"* será a *SymbolTable* que num dado momento contém as variáveis locais, tendo também acesso ás 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 3: Criação de variáveis auxiliares à gramática e da tabela de símbolos utilizada no processamento da Linguagem Principal

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 4: 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 5: 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 função>
};

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 á função, tem de começar por uma letra ou underscore.

Argumentos:

4.3 Análise Semântica

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

- 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 á dimensão requerida, e o seu valor tem de ser análogo á 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:

```
1          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 .

7.1 Programas Corretos

7.2 Programas com Erros

8 Conclusão

9 Contribuições dos autores

Segue-se a contribuição de cada elemento do grupo:

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

10 Bibliografia

[1] Material fornecido pelo docente da disciplina