

Manual do Utilizador

Linguagens Formais e Autómatos(LFA)

Daniel Marques, Francisco Morgado, Jorge
Catarino,
Óscar Pimentel, Paulo Vasconcelos



universidade de aveiro
theoria poiesis praxis

DETI - Universidade de Aveiro

Daniel Marques, Francisco Morgado, Jorge Catarino,
Óscar Pimentel, Paulo Vasconcelos
(85070) danielmarques@ua.pt, (85009) fmpfmorgado@ua.pt,
(85028) jorge.catarino@ua.pt, (80247) oscarpimentel@ua.pt
(84987) paulobvasconcelos@ua.pt

28 de junho de 2018

Conteúdo

1	Linguagem	2
1.1	BaseGrammar	2
1.1.1	Declaração de Variáveis	2
1.1.2	Atribuição	2
1.1.3	Imprimir no Terminal	3
1.1.4	Operação	3
1.1.5	Incrementos e Decrementos	4
1.1.6	Conversões	4
1.1.7	Condições <i>if and else</i>	4
1.1.8	Ciclo <i>for</i>	5
1.1.9	Ciclo <i>while</i>	5
1.1.10	Ciclo <i>do-while</i>	6
1.1.11	Condições	6
1.1.12	Potência $*10^n$	7
1.2	Unidades	8
1.2.1	<i>create unit</i>	8
1.2.2	<i>raise</i>	8
1.2.3	<i>compose</i>	8
1.2.4	<i>setConvValue</i>	9
2	Compilação	10
3	Execução	10
4	Programas de Teste e Gestão de Erros	11

1 Linguagem

1.1 BaseGrammar

1.1.1 Declaração de Variáveis

```
1 instruction returns[String varName]:      1 varType:
2     varType NAME          #varDec          2     'simpVar'    #simple
3     (...)                  3     | 'unitVar'  #unit
4     ;                               4     ;
```

```
1 NAME:[a-zA-Z][a-zA-Z_0-9]*;
```

A definição das variáveis é feita de duas maneiras, isto é, podem ser de dois tipos, tanto pode ser definida como uma variável simples sem unidade (*simpVar*) ou como uma variável com uma unidade associada (*unitVar*).

```
1 unitVar a
2 simpVar b
3 simpVar res89
4 simpVar kos
```

Esta variável é composta por letras maiúsculas e minúsculas de *A* a *Z* e eventualmente números compreendidos entre *0* e *9*, tendo que começar primeiro por uma letra.

1.1.2 Atribuição

```
1 instruction returns[String varName]:
2     (...)
3     | (varType)? NAME '=' operation      #assignment
4     (...)
5     ;
```

O **assignment** permite associar a uma variável já declarada um valor de uma operação. Para isso escreve-se o nome da variável, à qual pretendemos atribuir o valor, seguido pelo sinal '='. Qualquer operação escrita à frente desse sinal será aceite para a atribuição.

Caso a variável não tenha sido declarada é também possível declará-la de uma forma dinâmica, bastando escrever o tipo da variável que se pretende antes da atribuição.

```
1 unitVar a = 1 kg
2 b = 2.5e^2 lfa
```

1.1.3 Imprimir no Terminal

```
1 instruction returns [String varName]:
2     (...)
3     | print                #instPrint
4     (...)
5     ;
```

```
1 print: 'Print'
   '(', NAME ')';
```

Uma das instruções possíveis é a de imprimir no terminal uma variável que anteriormente teve que ser definida e ter sido atribuído a si um valor. Para imprimir basta escrever o comando **'Print'** seguido de uma variável dentro de parênteses.

```
1 unitVar lfa = 23kg * 3cd
2 Print(lfa)
```

1.1.4 Operação

```
1 operation returns [vartype ty] :
2     '(', n=operation ')',                                     #par
3     | left=operation NUMERIC_OPERATOR right=operation #op
4     | NAME                                                    #assignVar
5     | value                                                    #val
6     ;
```

Uma operação é realizada entre dois operandos, em que cada um pode ser também uma operação.

Dá prioridade à realização de operações que estejam dentro de parênteses.

Os operandos são separados por um **operador numérico**, que é definido na *BaseLexerRules* e pode ser observado no excerto de código ao lado. Este operador pode ser, por ordem de prioridade, uma potência (ex: $2^3 = 2^3$), uma divisão ou multiplicação e por fim uma adição ou uma subtração.

Só é possível operar sobre variáveis ou valores de tipos iguais.

```
1 NUMERIC_OPERATOR:
2     OP01
3     | OP02
4     | OP03
5     ;
6 OP01: '^';
7 OP02: '*'| '/';
8 OP03: '+'| '-';
```

```
1 unitVar centripetalAcceleration2 = 5 m/(1s^2)+45kg
2 centripetalAcceleration = ((velocity)*(velocity))/radius
```

1.1.5 Incrementos e Decrementos

```
1 deincrement returns [vartype ty]:
2     NAME '++'                                     #increment
3     | NAME '--'                                    #decrement
4     ;
```

Implementou-se o uso de incrementos e decrementos, estes são reconhecidos adicionando um '++', para incremento, ou um '--', para decremento, à frente da variável que se pretende.

```
1 lfa++
2 kg--
```

1.1.6 Conversões

```
1 convValue: src=value '€' dest=NAME;
```

Para o utilizador poder converter unidades entre o mesmo valor, tem à sua disposição a instrução "num unidadeX € unidadeY" como demonstrado de seguida:

```
1 2 arvores '€' cadeiras
```

Esta conversão mantém o valor original visto que a proporcionalidade entre as unidades terá de ter sido previamente definida no ficheiro das unidades.

O valor é impresso após a conversão para a nova unidade.

1.1.7 Condições *if* and *else*

```
1 if_else:
2 'if' '(' (cc=condition | bc=booleanCondition) ')' (ifA=ifArg)
3 ('else' (elseA=ifArg))?;
4 ifArg: '{' statList '}'          #ifStatList
5     | stat?                      #ifStat
6     ;
```

```
1 statList: (stat)*;
2 stat:
3     loop
4     | if_else
5     | instruction
6     ;
```

Adicionou-se à nossa linguagem a possibilidade de realizar condições de *if* and *else* e de ciclos *for*, *while* e *do-while*.

Nas condições *if* and *else* começa-se por colocar, tal como nas linguagens de programação, a palavra reservada **if** seguida de uma **condição** (ver *Condições*) entre parênteses. De seguida deverá ser colocado dentro de **duas chavetas** um *statList*, que pode ser um loop, outra condição *if* and *else* ou uma instrução.

Para colocar, eventualmente, um *else* basta, como para o *if*, colocar a palavra reservada **else** seguida de um *statList* entre chavetas.

```

1 if(i<t){
2     frequency = 1/1*t
3     Print(frequency)
4
5 } else{
6     k = 2
7     Print(k)
8 }

```

1.1.8 Ciclo *for*

```

1 //LOOPS SECTION
2 loop:
3     // FOR LOOP
4     'for' '(' var=NAME ';' min=INT ';' max=INT ')',
5     '{' statList '}' #loopFor

```

Para a realização de um ciclo *for*, deverá ser colocada a palavra reservada *for* seguido de três argumentos dentro de parênteses:

- Nome de uma variável
- Valor inicial de contagem
- Valor final de contagem(exclusive)

Este ciclo *for* funciona como um *for* em python, onde uma variável recebe um número inicial e a cada ciclo esse número é incrementado até atingir o valor final. De seguida, aceita um *statList* dentro de chavetas, tal como no *if and else*.

```

1 for(i;0;10){
2     freq = 1/1*t
3     Print(freq)
4 }

```

1.1.9 Ciclo *while*

```

1 // WHILE LOOP
2 | 'while' '(' (cc=condition|bc=booleanCondition) ')',
3 '{' statList '}' #loopWhile

```

O ciclo *while* segue a mesma lógica das instruções acima, primeiro escreve-se a palavra reservada *while* seguida de uma condição (ver *Condições*), e depois dentro de chavetas um *statList*.

Irão ser realizar todas as instruções que estão dentro do ciclo enquanto a condição for verdadeira.

```

1 while(a<b){
2     Print(a)
3 }

```

1.1.10 Ciclo *do-while*

```
1 // DO-WHILE LOOP
2 | 'do' '{' statList '}' 'while' '(' (cc=condition|bc=
   booleanCondition) ')', #
   loopDoWhile
3 ;
```

```
1 do{
2   inc++
3   if(inc<=15){
4     kos = b+c
5     Print(kos)
6   }
7 }while(inc<flag)
```

Num ciclo do-while começa-se por escrever a palavra reservada **do** seguida de um *statList* dentro de chavetas e logo a seguir a palavra reservada **while** com uma condição (ver *Condições*).

O ciclo corre o *statList* enquanto a condição é verdadeira, a diferença, deste ciclo com o ciclo *while*, é que este corre pelo menos uma vez o *statList* antes de verificar a condição.

1.1.11 Condições

```
1 condition:
2   left=conditionE
3   CONDITIONAL_OPERATOR
4   right=conditionE #compare
5   |conditionE      #soloCond
6   ;
```

Definiu-se como uma das condições, a **comparação condicional** entre dois operandos.

A comparação só pode ser feita entre variáveis e valores do mesmo tipo.

Esta comparação é feita com um **operador condicional**, definido na *BaseLexerRules*, que pode ser uma comparação de igualdade, uma de maior (ou igual) ou de menor (ou igual).

```
1 CONDITIONAL_OPERATOR:
2   '=='
3   | ('>' | '<') ('=')?
4   ;
```

```
1 not cd==m
2 centripetalAcceleration < centripetalAcceleration2
3 2 > 1
```

```
1 booleanCondition:
2   left=booleanCondition BOOLEAN_OPERATOR
3   right=booleanCondition #boolCondOp
4   |NOT condition        #boolNotCond
5   |condition             #boolCond
6   ;
```

A outra condição possível é a **comparação do valor booleano** de cada operando.

Neste caso, os operandos não se limitam a ser simplesmente variáveis ou valores, como descrito na primeira condição, mas sim comparações condicionais, e podem ser verificadas as condições necessárias.

Os operandos são comparados usando um **operador booleano** que, de acordo com a sua ordem de prioridade, pode ser um *and* ou um *or*.

```
1 BOOLEAN_OPERATOR:
2   'and'
3   | 'or'
4   ;
5 NOT: 'not';
```

Foi implementada também a possibilidade de verificar a negação de uma condição, para isso basta colocar a palavra *not* antes da condição que se pretende negar.

```
1 not a<b
2 a<b and b>c or kg==lfa
```

```
1 conditionE returns [vartype type]:
2   value          #condiEValue
3   | NAME         #condiEVar
4   ;
```

Cada operando pode ser um *value* ou uma variável.

Um *value* pode ser um valor positivo/negativo com uma unidade associada ou simplesmente um valor positivo/negativo.

Para representar o um número negativo basta colocar antes do número um '!', podendo cada um ter uma **potência** (ver *Potência* * 10^n).

```
1 value returns[vartype typ,String unit,String nr]:
2   num=(INT|REAL) pow? NAME          #valueUnit
3   | '!' num=(INT|REAL) pow? NAME     #valueUnitNeg
4   | num=(INT|REAL) pow?              #valueS
5   | '!' num=(INT|REAL) pow?         #valueSNeg
6   ;
```

```
1 6.7e^!3 m
2 !6.7e^!3 m
3 6.7^!3
4 !6.7e^!3
```

1.1.12 Potência * 10^n

```
1 // Equivalent to "*10^"
2 pow: 'e^' (min='!')? exp=(INT|REAL);
```

Para possibilitar operações com números muito grandes ou muito pequenos, implementou-se a potência.

Com "potência" assume-se que é a multiplicação de 10 elevado a um número (inteiro ou real), que pode ser positivo para números grandes ou negativo para números pequenos, a um valor.

Para escrever um valor deste tipo, basta colocar a seguir a um valor o 'e^'(elevado) seguido de um número positivo ou negativo, caso seja positivo não se coloca qualquer sinal, mas por causa de uma melhor gestão, para um número negativo, coloca-se um ponto de exclamação (!).

```
1 unitVar r = 6.7e^!3 m
```


1.2 Unidades

```
1 stat: 1 create:
2 create 2 'create' 'unit' uname=unit 'named' NAME;
3 |pow 3 pow:
4 |compose 4 'raise' NAME 'to power of' INT;
5 |setConvValue 5 compose:
6 ; 6 'compose' composedUnit 'named' NAME;
7 setConvValue:
8 src=NAME '$' dtn=value;
```

```
1 unit returns[String varName]: NAME #unitUNIT
2 ;
```

A gramática Unidades serve para criar unidades, para isso são usados quatro comandos distintos:

1.2.1 *create unit*

Para criar e atribuir um nome a uma unidade, o utilizador terá de usar a intrução *create unit* seguido da unidade ao que se acrescenta ainda *named* finalizando com o nome que se quer atribuir à unidade.

```
1 create unit Kilogram named Kg;
2 create unit liter named l;
3 create unit candela named cd;
4 create unit Newton named N;
```

1.2.2 *raise*

Este comando serve para elevar uma qualquer unidade, previamente definida, a um número inteiro.

Para isto, executa-se o comando com o nome da unidade à frente, *to power of* e o número inteiro pretendido.

```
1 raise m to power of 2;
```

1.2.3 *compose*

É usando o comando *compose* que o utilizador pode compor uma unidade, isto é, atribuir um nome a uma ou a um conjunto de unidades que estão associadas entre si por uma multiplicação/divisão.

Esta operação é realizada colocando o comando seguido do conjunto de unidades que se quer associar, escrevendo depois *named* para atribuir um nome e, por fim, o pretendido nome.

Na parte da associação, tal como nas operações, dá-se prioridade aos parênteses seguido de multiplicação/divisão. Um bom exemplo é o caso da velocidade, que corresponde à divisão da distância pelo tempo ($v = \frac{d}{t}$).

```

1 composedUnit returns [String varName]:
2     NAME                                #cUnitName
3     | '(' p=composedUnit ')'           #cUnitParents
4     | left=composedUnit op=(':'|'*')
5     right=composedUnit                 #cUnitDivMult
6     ;

1 compose m:s named vel;

```

1.2.4 *setConvValue*

É dada ao utilizador a possibilidade de definir unidades como proporcionais.

Para explicitar a relação entre duas unidades, o utilizador deve digitar o nome de uma unidade, o símbolo '\$' e a conversão correspondente noutra unidade.

```

1 setConvValue BarraDeChocolate $ 15 QuadradosDeChocolate

```

2 Compilação

Para facilitar a compilação dos dois ficheiros (ficheiro de unidades e ficheiro de código), foram desenvolvidos *scripts* que automatizam o processo.

São esses os ficheiros *runUnidades.py* e *runBaseGrammar.py*, respetivamente.

Estes *scripts* poderão ser executados da seguinte forma (considerando *FILEPATH_UNITS* e *FILEPATH_PROGRAM* os paths desde o diretório base até aos ficheiros que pretendemos compilar):

```
1 python3 runUnidades.py FILEPATH_UNITS
2 python3 runBaseGrammar.py FILEPATH_PROGRAM
```

Em alternativa, podem ser usados os seguintes comandos:

```
1 antlr4 -visitor Unidades.g4
2 javac Unidades*.java
3 java UnidadesMain FILEPATH_UNITS
4 antlr4 -visitor BaseGrammar.g4
5 javac BaseGrammar*.java
6 java BaseGrammarMain FILEPATH_PROGRAM
7 antlr4-clean
```

A utilização destes comandos resultará em dois ficheiros: *unitClass.py* e *Output.py*. São estes os ficheiros "finais" que contém o código a ser executado e as unidades que podem (ou não) ser utilizadas no programa escrito.

Para consultar o código gerado na linguagem destino (*Python*), pode ser executado o comando

```
1 nano Output.py
```

ou um semelhante.

Relembra-se que, para a correta compilação do programa, a máquina deve ter instalada a versão 3.5 do *Python* ou uma mais recente.

Foram desenvolvidos alguns programas de teste que podem ser encontrados na pasta *testFiles*.

3 Execução

Como seria de esperar, não pode haver execução do programa se não houver a compilação do mesmo.

O ficheiro *unitClass.py* não deverá ser executado pelo utilizador pois existe apenas para ser importado pelo ficheiro *Output.py*.

A execução deste último dá-se como se daria a execução de qualquer programa escrito na linguagem *Python*, com o comando:

```
1 python3 Output.py
```

4 Programas de Teste e Gestão de Erros

```
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$ python3 runUnidades.py testFiles/progXunits.txt
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$ python3 runBaseGrammar.py testFiles/prog7basegrammar.txt
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$ python3 Output.py
3.0 g^1/l^1
2
3
5
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$
```

Figura 1: Compilação e execução do *testFiles/prog7basegrammar.txt*

```
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$ python3 runUnidades.py
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$ python3 runBaseGrammar.py testFiles/erro1basegrammar.txt
[ERROR at line 1] Type mismatch!
[ERROR at line 2] Type mismatch!
[ERROR at line 6] Type mismatch!
[ERROR at line 8] You cannot make that operation between a simple variable and an unit variable!
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$
```

Figura 2: Menssagens de erro ao compilar o *testFiles/erro1basegrammar.txt*

```
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$ python3 runUnidades.py testFiles/erro1unidades.txt
[ERROR at line 2] A variable named "lfa" was already declared!
[ERROR at line 3] Variable "m" does not exist!
[ERROR at line 3] Variable "s" does not exist!
DansMarquis@LEGION_Y520:/mnt/c/Users/danie/Desktop/lfa-1718-g9$
```

Figura 3: Menssagens de erro ao compilar o *testFiles/erro1unidades.txt*