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) danielfmarques@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

1.1 BaseGrammar

1.1.1 Declaração de Variáveis

```
instruction returns[String varName]:
    varType NAME          #varDec
    (....)
    ;
```

```
varType:
 'simpVar'    #simple
 | 'unitVar' #unit
 ;
```

```
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***).

```
unitVar a
simpVar b
simpVar res89
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

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

O **assigment** 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.

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

1.1.3 Imprimir no Terminal

```
instruction returns[String varName]:
    (....)
    | print                #instPrint
    (....)
    ;
```

```
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.

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

1.1.4 Operação

```
operation returns [vartype ty] :
    '(' n=operation ')'                                #par
    | left=operation NUMERIC_OPERATOR right=operation  #op
    | NAME                                             #assignVar
    | value                                            #val
    ;
```

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: $2\hat{}3 = 2^3$), uma divisão ou multiplicação e por fim uma adição ou uma subtração.
Só é possível operar sobre variáveis ou valores de tipos iguais.

```
NUMERIC_OPERATOR:
          OP01
        | OP02
        | OP03
        ;
OP01: '^';
OP02: '*'|'/';
OP03: '+'|'-';
```

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

1.1.5 Incrementos e Decrementos

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

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.

```
lfa++
a--
```

1.1.6 Conversões

```
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:

```
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*

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

```
statList: (stat)*;
stat:
    loop
    |if_else
    |instruction
    ;
```

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 desntro 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.

```
if(i<t){
    frequency = 1/1*t
    Print(frequency)

} else{
    k = 2
    Print(k)
}
```

1.1.8 Ciclo *for*

```
//LOOPS SECTION
loop:
    // FOR LOOP
    'for' '(' var=NAME ';' min=INT ';' max=INT ')'
    '{' 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*.

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

1.1.9 Ciclo while

```
    // WHILE LOOP
    | 'while' '(' (cc=condition|bc=booleanCondition) ')'
    '{' 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.

```
while(a<b){
    Print(a)
}
```

1.1.10 Ciclo *do-while*

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

```
do{
    inc++
    if(inc<=15){
        kos = b+c
        Print(kos)
    }
}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 cilo 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

```
condition:
    left=conditionE
    CONDITIONAL_OPERATOR
    right=conditionE  #compare
    |conditionE       #soloCond
    ;
```

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).

```
CONDITIONAL_OPERATOR:
    '=='
    | ('>'|'<') ('=')?
    ;
```

```
not a>=b
centripetalAcceleration < centripetalAcceleration2
2 > 1
```

```
booleanCondition:
    left=booleanCondition BOOLEAN_OPERATOR
    right=booleanCondition                    #boolCondOp
    |NOT condition                            #boolNotCond
    |condition                                #boolCond
    ;
```

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.

```
BOOLEAN_OPERATOR:
    'and'
    | 'or'
    ;
NOT: 'not';
```

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

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.

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

```
conditionE returns [vartype type]:
    value           #condiEValue
    |NAME           #condiEVar
    ;
```

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* $*10^n$).

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

```
6.7e^!3 m
!6.7e^!3 m
6.7^!3
!6.7e^!3
```

1.1.12 Potência $*10^n$

```
// Equivalent to "*10^"
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 ('**!**').

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

1.2 Unidades

```
stat:
  create
  |pow
  |compose
  |setConvValue
  ;
```

```
create:
  'create' 'unit' uname=unit 'named' NAME;
pow:
  'raise' NAME 'to power of' INT;
compose:
  'compose' composedUnit 'named' NAME;
setConvValue:
  src=NAME '$' dtn=value;
```

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

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.

```
create unit Kilogram named Kg;
create unit liter named l;
create unit candela named cd;
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.

```
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
($v = \frac{d}{t}$).

```
composedUnit returns[String varName]:
                NAME                            #cUnitName
                |'(' p=composedUnit ')'         #cUnitParents
                |left=composedUnit op=(':'|'*')
                right=composedUnit              #cUnitDivMult
                ;
```

```
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.

```
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):

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

Em alternativa, podem ser usados os seguintes comandos:

```
antlr4 -visitor Unidades.g4
javac Unidades*.java
java UnidadesMain FILEPATH_UNITS
antlr4 -visitor BaseGrammar.g4
javac BaseGrammar*.java
java BaseGrammarMain FILEPATH_PROGRAM
antlr4-clean
```

A utilização destes comandos resultará em dois ficheiros: *units.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

```
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 *units.py* não deverá ser executado pelo utilizador pois exite 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:

```
python3 Output.py
```

4 Programas de Teste e Gestão de Erros

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

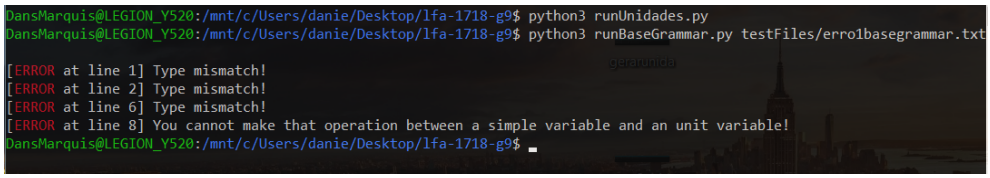

Figura 2: Menssagens de erro ao compilar o *testFiles/erro1basegrammar.txt*

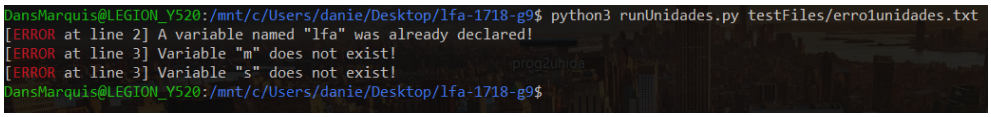

Figura 3: Menssagens de erro ao compilar o *testFiles/erro1unidades.txt*