Decrição da Linguagem

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 Gramáticas

1.1 BaseGrammar

1.1.1 Instrução

```
instruction returns[String varName]:
    // Variable declaration
    varType NAME                                    #varDec
    // Print/Read variable
    | print                                         #instPrint
    // Value atribution to variable
    // (This also accepts values that are not the result of
        an operation)
    |(varType)? NAME '=' operation                  #assignment
    // Operation without storing result or (most common)
        variable increment/decrement
    | operation                                     #soloOp
    | deincrement                                   #instDeincr
    ;
```

Nesta regra *instruction* pode-se observar que a gramática aceita a definição das variáveis, podendo esta ser apenas uma variável simples sem unidade (***simpVar***) ou uma variável com uma unidade associada (***unitVar***).
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.
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, que é a instrução seguinte (*assigment*). Para imprimir basta escrever o comando **'Print'** seguido de uma variável dentro de parênteses.
O **assigment** permite associar a uma qualquer variável um valor de uma operação. Este assigment pode ser dinâmico, podendo atribuir uma operação ao declarar uma variável.
Depois pode-se ter uma operação independente (*soloOp*) ——————-ACABAR

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

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

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

```
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.

1.1.2 Operação

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

A regra *operation* dá prioridade à realização de operações que estejam dentro de parênteses.
De seguida temos a operação propriamente dita que aceita dois operandos, em que cada um pode ser também uma operação. Estes operandos são separados por um operador numérico, que é definido na *BaseLexerRules* e pode ser observado no excerto de código abaixo. Este operador pode ser, por ordem de prioridade, uma potência, uma divisão ou multiplicação e por fim uma adição ou uma subtração.
Só é possível operar sobre variáveis ou valores de tipos iguais.

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

1.1.3 Condições *if and else*

```
//CONDITIONALS SECTION
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** (que irá ser explicada mais à frente) 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.

1.1.4 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*.

1.1.5 Ciclo while

```
    // WHILE LOOP
    | 'while' '(' condition ')' '{' statList '}'  #loopWhile
```

O ciclo *while* segue a mesma lógica das instruções acima, primeiro escreve-se a palavra reservada ***while*** seguida de uma condição, e depois, dentro de chavetas um *statList*.
Irão ser realizar todas as instruções que estão dentro do ciclo enquanto a condição for verdadeira.

1.1.6 Ciclo *do-while*

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

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.
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.

1.1.7 Condições

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

```
// Conditional Operators
CONDITIONAL_OPERATOR:
    '=='                  // Equal
    | ('>'|'<') ('=')?  // Great.(or equal)/Small.(or equal)
    ;
```

Definiu-se como uma das condições, a comparação condicional entre dois operandos.
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).
A comparação só pode ser feita entre variáveis e valores do mesmo tipo.

```
booleanCondition:
    left=booleanCondition BOOLEAN_OPERATOR
    right=booleanCondition                       #boolCondOp
```

```
    |NOT condition                                    #boolNotCond
    |condition                                       #boolCond
    ;
```

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

A outra condição possível a de comparação booleana entre dois operandos————————ACABAR————————-

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

Cada operando pode ser um valor ou uma variável. Se for um ————————————————ACABAR————————

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

1.1.8 Potência

```
// 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 (!).

1.2 Unidades

```
create: 'create' 'unit' uname=unit 'named' NAME;

pow: 'raise' NAME 'to power of' INT;

compose:'compose' composedUnit 'named' NAME;
```

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

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

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

- `create unit`, este comando serve para criar e atribuir um nome a uma unidade, para isso escreve-se o comando seguido da unidade, depois acrescenta-se ***named*** seguido do nome que se quer atribuir à unidade.
- `raise`, este comando serve para elevar uma qualquer unidade, anteriormente definida, a um número inteiro. Isto é feito da seguinte maneira, escreve-se novamente o comando com o nome da unidade à frente, depois acrescenta-se ***to power of*** seguido do número inteiro pretendido.
- `compose`, este comando serve para compor uma unidade, isto é, atribuir a uma unidade————————————ACABAR————————————

2 Instruções

Para facilitar a compilação de ambas as gramáticas foi desenvolvido um script.....
BASEGRAMMAR:
antlr4 -visitor BaseGrammar.g4
javac BaseGrammar*.java
java BaseGrammarMain [filepath]

3 Gestão de Erros