Linguagem i–

Hugo Frade*, Miguel Costa†, and Milton Nunes‡

Análise e Transformação de Software,
UCE30 Análise e Concepção de Software,
Mestrado em Engenharia Informatica,
Universidade do Minho

18 de Fevereiro de 2013

Resumo

Este documento apresenta a resolução do Trabalho Prático de Análise e Transformação de Software em que se definiu a linguagem i–, usando o AnTLR criou-se um compilador de forma a gerar código MSP.

*Email: hugoecfrade@gmail.com
†Email: miguelpintodacosta@gmail.com
‡Email: milton.nunes52@gmail.com

Conteúdo

1 Introdução **3**

2 Ambiente de Trabalho **3**

3 Descrição do problema **3**

4 Linguagem i– **4**

5 Definição e descrição formal da gramática **5**
5.1 Exemplos de frases válidas 7

6 AnTLR **8**
6.1 Gramática definida no AnTLR 8
6.2 Árvores de parser 12

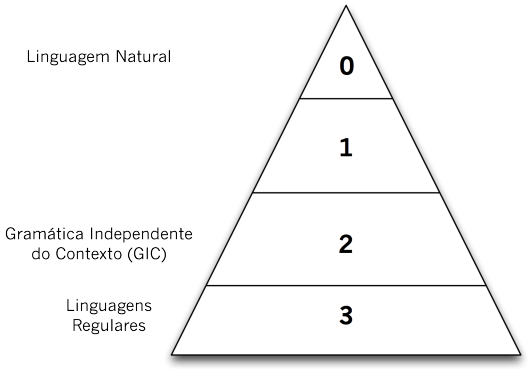
7 Tom **12**
7.1 Árvore de Sintaxe Abstrata (AST) 12

8 Conclusões **17**

9 Anexos **18**

1 Introdução

Tal como em maior parte das coisas no nosso dia à dia, as linguagens possuem uma hierarquia. No topo (0), encontra-se a linguagem natural, a mais difícil de decifrar devido à maior diversidade de termos e expressões que pudemos usar. Na base (3) encontram-se as linguagem regulares, que possuem um número muito limitado de termos, e por isso são bastante fáceis de descodificar e perceber.



Formalmente uma gramática independente do contexto é definida como uma gramática formal[1] por regras de produção da formalmente definidas como:`X -> x`, onde X é um símbolo não terminal e x é uma sequência de não terminais, ou até mesmo o vazio.

Depois de definida a gramática precisamos de um parser para o identificar. Como tal foi utilizado o AnTLR para criar esse parser. O AnTLR é uma ferramenta de reconhecimento de linguagem. Este aceita como input uma gramática que especifica a linguagem e gera o código fonte para o reconhecimento da linguagem. O AnTLR utiliza o algoritmo LL(*), algoritmo classificado como top-down.

Ler muitas linhas de código por vezes pode ser complicado, por isso interessa em muitos casos encontrar uma linguagem de fácil leitura e interpretação. Posto isto usamos a linguagem para difinir árvores de sintaxe abstrata (Gom). Desta forma tornou bastante simples perceber a estrutura de toda a linaguem criada.

2 Ambiente de Trabalho

Foi necessário usar um Gerador de Compiladores para gerar o nosso próprio compilador, por isso usamos o AnTLR que é também usado nas aulas. Para facilitar o processo de debugging durante a resolução do problema, usamos a ferramenta AnTLRWorks, que tem uma interface bastante agradável e simpática para ajudar a resolver problemas desta natureza.

Visto que era necessário gerar código para ser executado pelo MSP, foi utilizada a ferramenta que o professor disponibilizou.

3 Descrição do problema

O que é pretendido para este exercício é criar uma Gramática para frases idênticas a um programa escrito na Linguagem C. Depois de desenvolvida a linguagem é necessário criar em linguagem gom a árvore se sintaxe abstrata. Uma linguagem G é definida da seguinte forma:

```
G = <T, N, S, P>
```

em que:

[1] Objecto matemático que permite criação de linguagens através de um conjunto de regras de formação.

- **T** corresponde aos símbolos terminais;
- **N** corresponde aos símbolos não terminais;
- **S** indica o símbolo inicial;
- **P** as produções, com pi : X0 -> X1 ... Xi ... Xn.

Depois de definidos todos os símbolos e produções, é necessário escrever a gramática no AnTLR, introduzir algumas frases válidas para a linguagem e gerar as respectivas árvores de parser.

4 Linguagem i–

A linguagem i– é uma simplificação da linguagem C. Simplificada no sentido em que apenas reúne algumas das características presentes no C, nomeadamente, funções e os seus argumentos, declaração de variáveis, atribuições, expressões `if`, ciclos `while` e `for`, invocação de funções, retorno de variáveis numa função (`return`) e a utilização de expressões que utilizam operadores aritméticos, à exceção dos operadores de incremento (++) e decremento (–), operadores de comparação e operadores lógicos.

Neste momento, a nível de tipo de variáveis apenas irá suportar:

- `string`
- `char`
- `int`
- `float`
- `void`
- Quanto aos `arrays` ainda não é suportado.

Quanto às operações matemáticas suporta:

- soma
- subtração
- divisão
- multiplicação
- resto da divisão inteira

De referir que a gramática definida teve em conta as prioridades nas operações matemáticas.

Por fim, o tipo de instruções que se podem realizar num programa válido para esta linguagem são:

- Atribuição: `int i = 0; char a;`
- Condição `if`: `if(i > 0) i=i-1; else i=i+1;`
- Ciclo `while`: `while(i>0) {i= i-1;}`
- Ciclo `for`: `for(i=0 ; i < 10; i=i+1){a=a*i;}`
- Invocação de outras funções: `a = calcula(i, a);`
- Instrução `return`: `return 0;`

Por esta breve apresentação, pode-se então concluir que a gramática desenvolvida procurou ir de encontro ao que habitualmente compõe uma linguagem de programação e também com a preocupação de cumprir os requisitos pedidos.

5 Definição e descrição formal da gramática

O nosso programa pode incluir uma ou mais funções. Uma função é definida por um cabeçalho e respectivo corpo. No corpo podem ser feitas declarações, condições e ciclos. Estão ainda definidos os operadores matemáticos e lógicos, com as respectivas prioridades.

`G = <T, N, S, P>`

- T = { '{' , '}' , '(' , ')' , ';' , '=' , '-' , '+' , '*' , '/' , '%' , '||' , '&&' , '<' , '>' , '<=' , '>=' , '==' , '!=' , '!' , ID , STRING, CHAR , INT , TRUE , FALSE , RETURN , FOR , WHILE , IF , ELSE , TD_INT , TD_BOOL , TD_STRING , TD_CHAR , TD_FLOAT , TD_VOID }
- N = { programa, funcao, cabecalho, argumentos, corpo, corpo_funcao, declaracoes, declaracao, statements, statement, atribuicao, ifs, whiles, fors, condicao_for, invocacao, retorna, bloco, args, expr, orExpr, andExpr, equalExpr, addExpr, multExpr, notExpr, negationExpr, opAdd, opMult, opOr, opAnd, opRel, opNot, fator, constante }
- S = { `programa` }
- P =

```
programa     :   funcao
             |   programa funcao

funcao       :   cabecalho '{' corpo '}'
corpo        :   corpo_funcao
             |   corpo corpo_funcao

cabecalho    :   tipo ID '(' ')'
             |   tipo ID '(' argumentos ')'

argumentos   :   declaracao
             |   argumentos ',' declaracao

corpo_funcao:    statements
             |   declaracoes statements

declaracoes  :   E
             |   declaracoes declaracao ';'
             |   declaracoes declaracao '=' expr ';'

declaracao   :   tipo ID

statements   :   statement
             |   statements statement

statement    :   atribuicao ';'
             |   ifs
             |   whiles
             |   fors
             |   invocacao ';'
             |   retorna ';'

atribuicao   :   ID '=' expr

ifs          :   IF '(' expr ')' bloco
             |   IF '(' expr ')' bloco ELSE bloco
```

```
whiles      :   WHILE '(' expr ')' bloco

fors        :   FOR '(' (condicao_for) ';' expr ';' (condicao_for) ')' bloco

condicao_for:   expr
            |   atribuicao

invocacao   :   ID '(' args ')'

retorna     :   RETURN expr

bloco       :   '{' statements '}'
            |   statement

args        :   expr
            |   args ',' expr

expr        :   orExpr

orExpr      :   andExpr
            |   orExpr opOr andExpr

andExpr     :   equalExpr
            |   andExpr opAnd equalExpr

equalExpr   :   addExpr
            |   equalExpr opRel addExpr

addExpr     :   multExpr
            |   addExpr opAdd multExpr

multExpr    :   notExpr
            |   multExpr opMul notExpr

notExpr     :   negationExpr
            |   opNot negationExpr

negationExpr:   fator
            |   '-' fator

opAdd       :   '+'
            |   '-'

opMul       :   '*'
            |   '/'
            |   \'%'

opOr        :   '||'

opAnd       :   '&&'

opRel       :   '>'
            |   '<'
            |   '>='
            |   '<='
```

```
            |   '=='
            |   '!='

opNot       :   '!'

fator       :   ID
            |   constante
            |   invocacao

constante   :   STRING
            |   CHAR
            |   INT
            |   TRUE
            |   FALSE

tipo        :   TD_INT
            |   TD_BOOL
            |   TD_STRING
            |   TD_CHAR
            |   TD_FLOAT
            |   TD_VOID
```

5.1 Exemplos de frases válidas

De seguida são apresentados dois exemplos de frases válidas para a linguagem definida.

Listing 1: Exemplo de uma frase válida

```
int main (int args){
    int i = 10;
    i = i+1;
    int a;
    a = 20;

    while(i<a){
        i = i+1;
    }

    for(i+10; i < a; i=i+1){
        a = a-1;
        i = i-1;
    }
    return a;
}
```

Listing 2: Exemplo de uma frase válida

```
float main(int arg){
    int result;

    if (arg > 0)
        result = calc('p', arg);
    else
        result = calc('n', arg);
    return result;
}

```

```
float calc(char sinal, int arg){
    int v1 = 10;
    float v2 = 3;
    float result;
    if (sinal == 'p'){
        result = arg + v1 / v2;
    }else{
        result = -1*arg - v1 * v2;
    }
    return result;
}
```

6 AnTLR

6.1 Gramática definida no AnTLR

Listing 3: Toda a gramatica

```
grammar lingi;

// GAMMAR
programa: funcao*
    ;

funcao  :   cabecalho '{' corpo_funcao+ '}'
    ;

cabecalho
    :   tipo ID '(' argumentos? ')'
    ;

argumentos
    :   declaracao (',' declaracao)*
    ;

corpo_funcao
    :   (declaracoes)? statements
    ;

declaracoes
    :   declaracaoExpr+
    ;

declaracaoExpr
    :   declaracao ('=' expr)? ';'
    ;

declaracao
    :   tipo ID
    ;

statements
    :   statement+
    ;

statement
    :   atribuicao ';'
    |   ifs
```

```
41      |   whiles
42      |   fors
43      |   invocacao ';'
44      |   retorna ';'
45      ;
46
47  atribuicao
48      :   ID '=' expr
49      ;
50
51  ifs :   IF '(' expr ')' bloco ifsElse?
52      ;
53
54  ifsElse :   ELSE bloco
55      ;
56
57  whiles  :   WHILE '(' expr ')' bloco
58      ;
59
60  fors    :   FOR '(' forsexpr ';' expr ';' forsexpr ')' bloco
61      ;
62
63  forsexpr: expr
64      | atribuicao
65      ;
66
67  invocacao
68      :   ID '(' args ')'
69      ;
70
71  retorna :   RETURN expr
72      ;
73
74  bloco   :   '{' statements '}'
75      |   statement
76      ;
77
78  args    :   expr argsAux*
79      ;
80
81  argsAux :   ',' expr
82      ;
83
84  expr    :   orExpr
85      ;
86
87  orExpr  :   andExpr orExprAux*
88      ;
89
90  orExprAux:  opOr andExpr
91      ;
92
93  andExpr :   equalExpr andExprAux*
94      ;
95
96  andExprAux: opAnd equalExpr
97      ;
98
99  equalExpr
100     :   addExpr equalExprAux*
101     ;
```

```
102
103 equalExprAux: opRel addExpr
104     ;
105
106 addExpr :   multExpr addExprAux*
107     ;
108
109 addExprAux: opAdd multExpr
110     ;
111
112 multExpr:   notExpr multExprAux*
113     ;
114
115 multExprAux: opMul notExpr
116     ;
117
118 notExpr :   (opNot)? negationExpr
119     ;
120
121 negationExpr
122     :   ('-')? fator
123     ;
124
125 opAdd   :   '+'
126     |   '-'
127     ;
128
129 opMul   :   '*'
130     |   '/'
131     |   \'%'
132     ;
133
134 opOr    :   '||'
135     ;
136
137 opAnd   :   '&&'
138     ;
139
140 opRel   :   '>'
141     |   '<'
142     |   '>='
143     |   '<='
144     |   '=='
145     |   '!='
146     ;
147
148 opNot   :   '!'
149     ;
150
151 fator   :   ID
152     |   constante
153     |   invocacao
154     ;
155
156 constante
157     :   STRING
158     |   CHAR
159     |   INT
160     |   TRUE
161     |   FALSE
162     ;
```

```
163
164 tipo     :   TD_INT
165     |   TD_BOOL
166     |   TD_STRING
167     |   TD_CHAR
168     |   TD_FLOAT
169     |   TD_VOID
170     ;
171 // LEXER
172 TD_INT  :   'int';
173 TD_BOOL :   'bool';
174 TD_STRING:  'string';
175 TD_CHAR :   'char';
176 TD_FLOAT:   'float';
177 TD_VOID :   'void';
178 TRUE    :   'true';
179 FALSE   :   'false';
180
181 RETURN  :   'return';
182 IF  :   'if';
183 ELSE    :   'else';
184 WHILE   :   'while';
185 FOR :   'for';
186
187 ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
188     ;
189
190 INT :   '0'..'9'+
191     ;
192
193 FLOAT
194     :   ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
195     |   '.' ('0'..'9')+ EXPONENT?
196     |   ('0'..'9')+ EXPONENT
197     ;
198
199 COMMENT
200     :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
201     |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
202     ;
203
204 WS  :   ( ' '
205         | '\t'
206         | '\r'
207         | '\n'
208         ) {$channel=HIDDEN;}
209     ;
210
211 STRING
212     :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
213     ;
214
215 CHAR:  '\'' ( ESC_SEQ | ~('\''|'\\') ) '\''
216     ;
217
218 fragment
219 EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;
220
221 fragment
222 HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
223
```

```
fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
```

6.2 Árvores de parser

Em anexo estão dois exemplos de árvores de parser que foram gerados a partir das frases indicada anteriormente.

7 Tom

Como referido na página oficial do Tom (tom.loria.fr), tratase de uma extensão da linguagem gom projetada para manipular estruturas em árvore e documentos XML, permitindo fazer reconhecimento de padrões, facilitando a inspeção de objetos e na recuperação de valores.

Outras das grandes vantagens de usar Tom é que pode ser usada num vasto leque de linguagens, por exemplo: C, Java, Pythion, C++, C#...

O Tom cria uma árvore de sintaxe abstrata (AST) da linguagem que estiver a analisar e depois é possível, com alguma facilidade, aplicar métricas para analisar a qualidade dessas linguagens.

7.1 Árvore de Sintaxe Abstrata (AST)

O ficheiro Gom fornece uma sintaxe para definir de forma concisa e legível uma AST, cada ficheiro permite definir os módulos da linguagem e em cada módulo identificado os operadores e os seus tipos.

A seguir podemos ver o exemplo da AST criada para a nossa linguagem

Listing 4: AST (Gom)

```
module parser.lingi
imports String int
abstract syntax
    Programa     = Programa(Funcao*)
    Funcao       = Funcao(Cabecalho, Corpo_Funcao)
    Cabecalho    = Cabecalho1(tipo:String, ID:String)
                 | Cabecalho2(tipo:String, ID:String, Argumentos)
    Argumentos   = Argumentos(Declaracao)
                 | Argumentos(Argumentos, Declaracao)
    Declaracoes  = Declaracao*
                 | Delaracao, Declaracoes
    Declaracao   = Declaracao(tipo:String, ID:String)
                 | Declaracao(tipo:String, ID:String, Expr) APAGAR
    Corpo_Funcao= Corpo_Funcao(Statements)
```

```
                | Corpo_Funcao(Declaracoes, Statements)
    Statements  = Statements(Statement*)
    Statement   = StatementAtribuicao(Atribuicao)
                | StatementIfs(Ifs)
                | StatementWhiles(Whiles)
                | StatementFors(Fors)
                | StatementInvocacao(Invocacao)
                | StatementRetorna(Retorna)
    Atribuicao  = Atribuicao(ID:String, Expr)

    Whiles      = Whiles(Expr, Bloco)
    Fors        = Fors(ForExpr, Expr, Forexpr, Bloco)
    ForExpr     = ForExprExpr()
                | ForExprAtribuicao()
    ForExprExpr = Expr()
    ForExprAtribuicao = Atribuicao()
    Invocacao   = Invocacao(Args)
    Retorna     = Retorna(Expr)
    Bloco       = BlocoMore(Statements)
                | BlocoOne(Statement)
    Args        = Args1(Expr)
                | Args2(Expr Expr*)
    Expr        = Expr(OrExpr)
    OrExpr      = OrExpr1(AndExpr)
                | OrExpr2(AndExpr OrExprAux*)
    OrExprAux   = OrExprAux(OpOr AndExpr)
    AndExpr     = AndExpr1(EqualExpr)
                | AndExpr2(EqualExpr AndExprAux*)
    AndExprAux  = AndExprAux(OpAnd EqualExpr)
    EqualExpr   = AddExpr1(EqualExpr AddExpr)
                | AddExpr2(EqualExpr AddExpr EqualExprAux*)
    EqualExprAux= EqualExprAux(OpRel AddExpr)
    AddExpr     = AddExpr1(MultExpr)
                | AddExpr2(MultExpr AddExprAux*)
    AddExprAux  = AddExprAux(OpAdd MultExpr)
    MultExpr    = MultExpr1(NotExpr)
                | MultExpr2(NotExpr MultExprAux*)
    MultExprAux = MultExprAux(NotExpr)
    NotExpr     = NotExpr(NegationExpr)
    NegationExpr= NegationExpr(Fator)
    Fator       = Fator1(ID:String)
                | Fator2(Constante)
                | Fator3(Invocacao)
    Constante   = ConstanteString(s:String)
                | ConstanteChar(c:char)
                | ConstanteInt(i:int)
                | ConstanteBool(b:bool)
```

Para integrar esta AST com a nossa gramática é necessário nas produções colocar as instruções a dizer como vai ser gerada a árvore, por isso o ficheiro da gramática passa a ser o seguinte:

Listing 5: AST (Gom)

```
grammar lingi_tom;
options {
  output=AST;
  ASTLabelType=Tree;
  tokenVocab=lingi;
  k=2;
}

tokens {
```

```
    Programa;
    Funcao;
    Cabecalho1;
    Cabecalho2;
    Argumentos;
    Corpo_Funcao1;
    Corpo_Funcao2;
    Declaracoes;
    Declaracao;
    DeclaracaoExpr1;
    DeclaracaoExpr2;
    Statement;
    StatementAtribuicao;
    StatementIfs;
    StatementWhiles;
    StatementFors;
    StatementInvocacao;
    StatementRetorna;
    Atribuicao;
    Ifs1;
    Ifs2;
    IfsElse;
    Whiles;
    Fors;
    ForExprExpr;
    ForExprAtribuicao;
    Invocacao;
    Retorna;
    BlocoMore;
    BlocoOne;
    Args1;
    Args2;
    ArgsAux;
    Expr;
    AndExpr1;
    AndExpr2;
    OrExpr1;
    OrExpr2;
    OrExprAux;
    AndExprAux;
    EqualExpr1;
    EqualExpr2;
    EqualExprAux;
    AddExpr1;
    AddExpr2;
    AddExprAux;
    MultExpr1;
    MultExpr2;
    MultExprAux;
    NotExpr;
    NegationExpr;
    FatorID;
    FatorConstante;
    FatorInvocacao;
    ConstanteString;
    ConstanteChar;
    ConstanteInt;
    ConstanteBool;
}

programa: funcao*                    -> ^(Programa funcao*)
```

```
71          ;
72
73  funcao      :   c1=cabecalho '{' c2=corpo_funcao '}'
74                                  -> ^(Funcao $c1 $c2)
75          ;
76
77  cabecalho
78          :   tipo ID '(' a1=argumentos? ')'
79                                  -> {a1==null}? ^(Cabecalho1 tipo ID)
80                                  -> ^(Cabecalho2 tipo ID argumentos?)
81          ;
82
83  argumentos
84          :   declaracao (',' declaracao)*
85                                  -> ^(Argumentos declaracao+)
86          ;
87
88  corpo_funcao
89          :   d1=declaracoes? statements
90                                  -> {d1==null}? ^(Corpo_Funcao1 statements)
91                                  -> ^(Corpo_Funcao2 $d1 statements)
92          ;
93
94  declaracoes
95          :   declaracaoExpr+         -> ^(Declaracoes declaracaoExpr*)
96          ;
97
98  declaracaoExpr  :   declaracao ('=' expr)? ';'
99                                  -> {expr==null}? ^(DeclaracaoExpr1 declaracao)
100                                 -> ^(DeclaracaoExpr2 declaracao expr)
101         ;
102
103 declaracao
104         :   tipo ID                 -> ^(Declaracao tipo ID)
105         ;
106
107 statements
108         :   statement*              -> ^(Statement statement*)
109         ;
110
111 statement
112         :   atribuicao ';'          -> ^(StatementAtribuicao atribuicao)
113         |   ifs                     -> ^(StatementIfs ifs)
114         |   whiles                  -> ^(StatementWhiles whiles)
115         |   fors                    -> ^(StatementFors fors)
116         |   invocacao ';'           -> ^(StatementInvocacao invocacao)
117         |   retorna ';'             -> ^(StatementRetorna retorna)
118         ;
119
120 atribuicao
121         :   ID '=' expr             -> ^(Atribuicao ID expr)
122         ;
123
124 ifs     :   IF '(' expr ')' bloco ifsElse?
125                                 -> {ifsElse==null}? ^(Ifs1 expr bloco)
126                                 -> ^(Ifs2 expr bloco ifsElse)
127         ;
128
129 ifsElse     :   ELSE bloco          -> ^(IfsElse bloco)
130         ;
131
```

```
132 whiles  :   WHILE '(' expr ')' bloco-> ^(Whiles expr bloco)
133         ;
134
135 fors    :   FOR '(' f1=forsexpr ';' expr ';' f2=forsexpr ')' bloco
136                                     -> ^(Fors $f1 expr $f2 bloco)
137         ;
138
139 forsexpr: expr                      -> ^(ForExprExpr expr)
140         | atribuicao                -> ^(ForExprAtribuicao atribuicao)
141         ;
142
143 invocacao
144         :   ID '(' args ')'         -> ^(Invocacao args)
145         ;
146
147 retorna :   RETURN expr             -> ^(Retorna expr)
148         ;
149
150 bloco   :   '{' statements '}'      -> ^(BlocoMore statements)
151         |   statement               ->  ^(BlocoOne statement)
152         ;
153
154 args    :   e1=expr a1=argsAux*     -> {a1==null}? ^(Args1 $e1)
155                                     -> ^(Args2 $e1 $a1*)
156         ;
157 argsAux : ',' expr                  -> ^(ArgsAux expr)
158         ;
159
160 expr    :   orExpr                  -> ^(Expr orExpr)
161         ;
162
163 orExpr  :   andExpr orExprAux*      -> {orExprAux==null}? ^(OrExpr1 andExpr)
164                                     -> ^(OrExpr2 andExpr orExprAux*)
165         ;
166
167 orExprAux:  opOr andExpr            -> ^(OrExprAux andExpr)
168         ;
169
170 andExpr :   equalExpr (andExprAux)* -> {andExprAux == null}? ^(AndExpr1 equalExpr)
171                                     -> ^(AndExpr2 equalExpr andExprAux*)
172         ;
173
174 andExprAux: opAnd equalExpr         -> ^(AndExprAux equalExpr)
175         ;
176
177 equalExpr
178         :   addExpr (equalExprAux)* -> {equalExprAux==null}? ^(EqualExpr1 addExpr)
179                                     -> ^(EqualExpr2 addExpr equalExprAux*)
180         ;
181
182 equalExprAux: opRel addExpr         -> ^(EqualExprAux addExpr)
183         ;
184
185 addExpr :   multExpr (addExprAux)*  -> {addExprAux == null}? ^(AddExpr1 multExpr)
186                                     -> ^(AddExpr2 multExpr AddExprAux*)
187         ;
188
189 addExprAux: opAdd multExpr          -> ^(AddExprAux multExpr)
190         ;
191
192 multExpr:   notExpr (multExprAux)*  -> {multExprAux==null}? ^(MultExpr1 notExpr)
```

```
193                                     -> ^(MultExpr2 notExpr multExprAux*)
194         ;
195
196 multExprAux: opMul notExpr          -> ^(MultExprAux notExpr)
197         ;
198
199 notExpr :   (opNot)? negationExpr   -> ^(NotExpr negationExpr)
200         ;
201
202 negationExpr
203         :   ('-')? fator            -> ^(NegationExpr fator)
204         ;
205
206
207 fator   :   ID                      -> ^(FatorID ID)
208     |   constante                   -> ^(FatorConstante constante)
209     |   invocacao                   -> ^(FatorInvocacao invocacao)
210     ;
211
212 constante
213     :   STRING                      -> ^(ConstanteString STRING)
214     |   CHAR                        -> ^(ConstanteChar CHAR)
215     |   INT                         -> ^(ConstanteInt INT)
216     |   TRUE                        -> ^(ConstanteBool TRUE)
217     |   FALSE                       -> ^(ConstanteBool FALSE)
218     ;
219
220 tipo    :   TD_INT
221     |   TD_BOOL
222     |   TD_STRING
223     |   TD_CHAR
224     |   TD_FLOAT
225     |   TD_VOID
226     ;
```

8 Conclusões

A resolução deste exercício permitiu perceber melhor a forma como as linguagens podem ser úteis para gerar um programa, que dependendo do input que irá receber, o resultado final seja o esperado sem ter de estar a alterar o código do programa que é automaticamente gerado. Apesar de não termos qualquer tipo de output, as árvores geradas permitiram chegar a estas conclusões.

Umas das dificuldades foi perceber como o AnTLR fazia o parser das frases de forma a não haver ambiguidade nas produções.

9 Anexos