

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*

12 de Novembro de 2012

Resumo

Este documento apresenta a resolução do Trabalho Prático de Análise e Transformação de Software em que se definiu a linguagem i– e usando o AnTLR gerou-se um parser para esta linguagem.

*Email: hugoecfrade@gmail.com

†Email: miguelpintodacosta@gmail.com

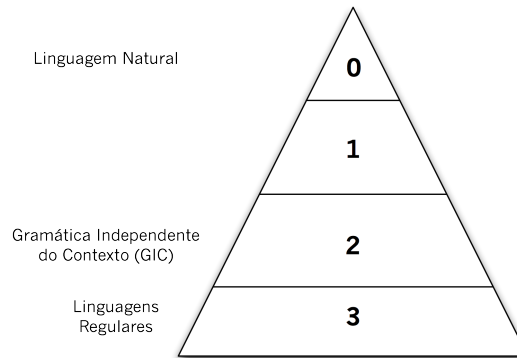
‡Email: milton.nunes52@gmail.com

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	Conclusões	12
8	Anexos	13

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 podemos usar. Na base (3) encontram-se as linguagens regulares, que possuem um número muito limitado de termos, e por isso são bastante fáceis de decodificar e perceber.



Formalmente uma gramática independente do contexto é definida como uma gramática formal¹ por regras de produção da formalmente definidas como: $X \rightarrow x$, 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.

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.

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. Uma linguagem G é definida da seguinte forma:

$$G = \langle T, N, S, P \rangle$$

em que:

- **T** corresponde aos símbolos terminais;
- **N** corresponde aos símbolos não terminais;
- **S** indica o símbolo inicial;
- **P** as produções, com $p_i : X_0 \rightarrow X_1 \dots X_i \dots X_n$.

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

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.

$G = \langle T, N, S, P \rangle$

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

```

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

```

1  int main (int args){
2      int i = 10;
3      i = i+1;
4      int a;
5      a = 20;
6
7      while(i<a){
8          i = i+1;
9      }
10
11     for(i+10; i < a; i=i+1){
12         a = a-1;
13         i = i-1;
14     }
15     return a;
16 }

```

Listing 2: Exemplo de uma frase válida

```

1  float main(int arg){
2      int result;
3
4      if (arg > 0)
5          result = calc('p', arg);
6      else
7          result = calc('n', arg);
8      return result;
9  }
10

```

```

11 float calc(char sinal, int arg){
12     int v1 = 10;
13     float v2 = 3;
14     float result;
15     if (sinal == 'p'){
16         result = arg + v1 / v2;
17     }else{
18         result = -1*arg - v1 * v2;
19     }
20     return result;
21 }

```

6 AnTLR

6.1 Gramática definida no AnTLR

Listing 3: Toda a gramatica

```

1 grammar lingi;
2
3 // GAMMAR
4 programa: funcao+
5     ;
6
7 funcao :   cabecalho '{' corpo_funcao+ '}',
8     ;
9
10 cabecalho
11     :   tipo ID '(' argumentos? ')',
12     ;
13
14 argumentos
15     :   declaracao (',' declaracao)*
16     ;
17
18 corpo_funcao
19     :   (declaracoes)? statements
20     ;
21
22 declaracoes
23     :   (declaracao ('=' expr)? ';'')+
24     ;
25
26 declaracao
27     :   tipo ID
28     ;
29
30 statements
31     :   statement+
32     ;
33
34 statement
35     :   atribuicao ';'
36     |   ifs
37     |   whiles
38     |   fors
39     |   invocacao ';'
40     |   retorna ';'

```



```

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

```

```

102     ;
103
104 opAdd    :    '+'
105         |    '-'
106         ;
107
108 opMul    :    '*'
109         |    '/'
110         |    \'%'
111         ;
112
113 opOr     :    '||'
114         ;
115
116 opAnd    :    '&&'
117         ;
118
119 opRel    :    '>'
120         |    '<'
121         |    '>='
122         |    '<='
123         |    '=='
124         |    '!='
125         ;
126
127 opNot    :    '!'
128         ;
129
130 fator
131     :    ID
132     |    constante
133     |    invocacao
134     ;
135
136 constante
137     :    STRING
138     |    CHAR
139     |    INT
140     |    TRUE
141     |    FALSE
142     ;
143
144 tipo
145     :    TD_INT
146     |    TD_BOOL
147     |    TD_STRING
148     |    TD_CHAR
149     |    TD_FLOAT
150     |    TD_VOID
151     ;
152
153 // LEXER
154 TD_INT   :    'int';
155 TD_BOOL  :    'bool';
156 TD_STRING:    'string';
157 TD_CHAR  :    'char';
158 TD_FLOAT:    'float';
159 TD_VOID  :    'void';
160 TRUE     :    'true';
161 FALSE    :    'false';
162

```

```

163 RETURN : 'return';
164 IF : 'if';
165 ELSE : 'else';
166 WHILE : 'while';
167 FOR : 'for';
168
169 ID : ('a'..'z'|'A'..'Z'|'_'') ('a'..'z'|'A'..'Z'|'0'..'9'|'_'')*
170 ;
171
172 INT : '0'..'9'+
173 ;
174
175 FLOAT
176 : ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
177 | '.' ('0'..'9')+ EXPONENT?
178 | ('0'..'9')+ EXPONENT
179 ;
180
181 COMMENT
182 : '//' ~('\n'|\r')* '\r'? '\n' {$channel=HIDDEN;}
183 | '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
184 ;
185
186 WS : ( ' '
187 | '\t'
188 | '\r'
189 | '\n'
190 ) {$channel=HIDDEN;}
191 ;
192
193 STRING
194 : '"' ( ESC_SEQ | ~('\\"'|'"') )* '"'
195 ;
196
197 CHAR: '\'' ( ESC_SEQ | ~('\''|'\\') ) '\''
198 ;
199
200 fragment
201 EXPONENT : ('e'|'E') ('+'|'-'?) ('0'..'9')+ ;
202
203 fragment
204 HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
205
206 fragment
207 ESC_SEQ
208 : '\\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
209 | UNICODE_ESC
210 | OCTAL_ESC
211 ;
212
213 fragment
214 OCTAL_ESC
215 : '\\\' ('0'..'3') ('0'..'7') ('0'..'7')
216 | '\\\' ('0'..'7') ('0'..'7')
217 | '\\\' ('0'..'7')
218 ;
219
220 fragment
221 UNICODE_ESC
222 : '\\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
223 ;

```

6.2 Árvores de parser

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

7 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.

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

8 Anexos