Linguagem i–

Miguel Costa*and Milton Nunes†

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

19 de Junho de 2013

Resumo

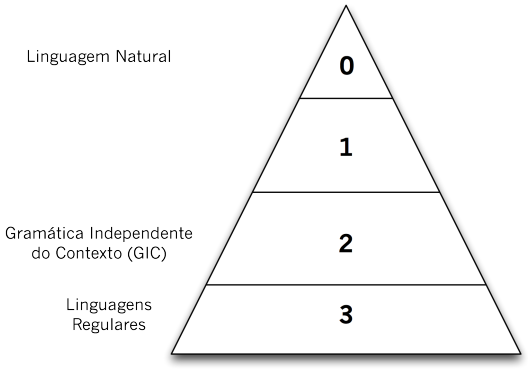
Este documento apresenta a resolução do Trabalho Prático de Análise e Transformação de Software em que se definiu a linguagem i– e usando o AnTLR criou-se um compilador de forma a gerar código para diferentes tarefas.

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

Conteúdo

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.

Quando se escrever um programa, o objetivo é que ele faça alguma tarefa automaticamente, para tentar simular essa execução, nesta fase usamos uma máquina virutal com o nome MSP (Mais Simples Possível)

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 máquina virtual que o professor disponibilizou para testes.

3 Descrição do problema

O que é pretendido para este exercício é criar uma Gramática para frases idênticas a um programa escrito numa Linguagem i– parecida ao C. Uma linguagem G é definida da seguinte forma:

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

em que:

- **T** corresponde aos símbolos terminais;

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

- **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.

O que é pretendido é usar a gramática criada nos trabalhos anteriores da disciplina e adaptar para gerar código para ser lido pela MSP. Depois de fazer o parser do código inserido, é necessário gerar código MSP, criar mecanismos para verificação de testes, injeção de falhas e ainda o cálculo de métricas do código introduzido.

Apesar de o professor ter aconselhado a usar TOM como representação intermédia depois do parser, adotamos antes a utilização de Tree Grammars como representação para a árvore 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 Gramáticas

Neste capítulo, iremos abordar a gramática concreta fornecida que define uma linguagem i– apresentada anteriormente. A transformação numa AST também foi fornecida e por isso abordaremos apenas o processo de criação associado. Por fim, mostraremos a implementação da cada um dos módulos (Tree Grammars) pedidos para este trabalho.

5.1 Gramática concreta

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, operações de I/O (scan e print), expressões if e 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.

Após criada a gramática, o próximo passo seria validar o texto de input. Mas como fazê-lo? Existem várias abordagens, uma delas e a que iremos utilizar neste trabalho é a geração de uma representação intermédia para que a partir dela se possam fazer várias tarefas sem ter de estar sempre a fazer parser ao código submetido.

5.1.1 AST

Uma Representação Intermédia (RI) é uma versão independente de qualquer linguagem ou máquina do código original. A utilização de uma RI traz algumas vantagens tais como o aumento do nível de abstração e uma separação mais limpa entre o produto inicial e o final.

Existem várias representações intermédias e a que iremos utilizar é a AST (Abstract Syntax Tree) que é uma representação em árvore da estrutura sintática abstrata do código fonte. A sintaxe é abstrata no sentido em que não representa cada detalhe que aparece na sintaxe real, ou seja, elementos como parênteses de agrupamento estão implícitos na estrutura da árvore e uma construção sintática como uma condição if e os seus blocos then e else pode ser representada através de um único nodo e dois ramos, e símbolos intermédios e palavras reservadas são tipicamente eliminados. Basicamente, mantém-se uma estrutura suficiente para realizar processos semânticos e geração de código.

Para realizar as tarefas pretendidas, temos de criar então a AST e para isso é necessário criar regras de reescrita sobre a gramática concreta, um mecanismo que o ANTLR oferece. Enquanto que uma gramática de parsing especifica como reconhecer input, as regras de reescrita são gramáticas geradoras, ou seja, especificam como gerar output. Estas regras de reescritas, já nos são fornecidas juntamente com a gramática concreta e a AST resultante terá, por cada, elemento que agrupa outros elementos, um token imaginário, ou seja, referências a tokens que não se encontram na produção original, elementos tais como ';', ou parênteses serão eliminados e elementos com o mesmo nome numa produção são agrupados numa única lista.

A gramática final com as regras de reescrita pode ser consultada em anexo.

Apesar de o professor ter aconselhado a utilização de TOM, devido a alguns problemas essencialmente de configuração em conseguir por tudo a funcionar, adotamos antes a utilização de AST porque já é uma RI que o Antlr oferece.

5.2 Tree Grammars

O próximo passo consiste na construção de um parser da AST gerada, que permitirá atravessá-la (tree walker) e manipulá-la, transformando-a gradualmente em diversas fases de tradução até que se obtenha uma forma final que satisfaça as nossas necessidades. Este parser será construído utilizando um mecanismo fornecido pelo ANTLR, uma Tree Grammar (TG). As ações numa TG possuem um contexto muito nítido e conseguem aceder a informação passada das regras invocadas.

A utilização de TGs, para além da utilização referida acima, também nos fornece algumas vantagens:

- uma especificação formal, concisa e independente de um sistema da estrutura da AST;
- as ações têm um contexto implícito graças à sua localização na gramática;
- os dados podem ser passados entre as ações de forma livre utilizando parâmetros (atributos), valores de retorno e variáveis locais.

Posto isto, o problema apresentado exige a construção de algumas TG (módulos), um módulo para gerar MSP, um módulo para intrudução de falhas, um módulo para gerar código para testes e um módulo para as métricas

6 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+
    ;

funcao
    : cabecalho '{' corpo_funcao '}'
    ;

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

argumentos
    : declaracao (',' declaracao)*
    ;
```

```
corpo_funcao
    : declaracoes statements
    ;

declaracoes
    : (declaracao ';')+
    ;

declaracao
    : tipo ID
    ;

tipo
    : TD_INT
    | TD_BOOL
    | TD_STRING
    | TD_VAZIO
    ;

statements
    : statement+
    ;


statement
    : atribuicao ';'
    | read ';'
    | write ';'
    | ifs
    | whiles
    | invocacao ';'
    | retorna ';'
    ;

retorna
    : RETURN expr
    ;

invocacao
    : ID '(' args? ')'
    ;

args
    : expr ( ',' expr )*
    ;

atribuicao
    : ID '=' expr
    ;

write
    : WRITE '(' expr ')'
    ;

read
```

```
    : READ '(' ID ')'
    ;


ifs
    : IF '(' expr ')' bloco (ELSE bloco)?
    ;

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

bloco
    : '{' statements '}'
    | statement
    ;

expr
    : orExpr
    ;

orExpr
    : andExpr (opOr^ andExpr )*
    ;

andExpr
    : equalityExpr(opAnd^ equalityExpr )*
    ;

equalityExpr
    : additiveExpr (opRel^ additiveExpr)*
    ;


additiveExpr
    : multiplicativeExpr (opAdd^ multiplicativeExpr)*
    ;

multiplicativeExpr
    : notExpr (opMul^ notExpr )*
    ;

notExpr
    : (opNot^)? negationExpr
    ;

negationExpr
    : ('-'^)? factor
    ;


opAdd
    :  '+'
    |  '-'
    ;
```

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

opOr: '||'
    ;

opAnd
    : '&&'
    ;

opRel
    : '>'
    |  '<'
    |  '>='
    |  '<='
    |  '=='
    | '!='
    ;

opNot
    : '!'
    ;

factor
    : ID
    | constante
    | invocacao
    ;

constante
    : STRING
    | INT
    | TRUE
    | FALSE
    ;
```

6.1 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  }
```

7 AnTLR

7.1 Gramática definida no AnTLR

A gramática defindа no Antlr de forma a gerar as AST, é criada basicamente através das instruções a seguir a "->", que indica qual o token que é enviado com os respetivos "parâmetros".

Listing 3: Toda a gramatica

```
1   grammar Cmb;
2
3   options{
4       backtrack = true;
5       output = AST;
6   }
7
8   tokens {
9       PROGRAMA;
10      DECLARACOES;
11      DECLARACAO;
12      STATEMENTS;
13      ATRIBUICAO;
14      THEN;
15      CORPO;
16      FUNCAO;
17      CAEBECALHO;
18      ARGUMENTOS;
```

```
    INVOCACAO;
    ARGS;
    RETURN;
    READ;
    WRITE;
}

//grammar

programa
    : funcao+
    -> ^(PROGRAMA funcao+)
    ;

funcao
    : cabecalho '{' corpo_funcao '}'
    -> ^(FUNCAO cabecalho corpo_funcao)
    ;

cabecalho
    : tipo ID '(' argumentos? ')'
    -> ^(CAEBECALHO tipo ID argumentos?)
    ;

argumentos
    : declaracao (',' declaracao)*
    -> ^(ARGUMENTOS declaracao+)
    ;

corpo_funcao
    :   declaracoes statements
    -> ^(CORPO declaracoes statements)
    ;


declaracoes
    :   (declaracao ';')+
    -> ^(DECLARACOES declaracao+)
    ;

declaracao
    :   tipo ID
    -> ^(DECLARACAO tipo ID)
    ;

tipo
    :   TD_INT
    |   TD_BOOL
    |   TD_STRING
    |   TD_VAZIO
    ;

statements
    :   statement+
    -> ^(STATEMENTS statement+)
    ;


statement
    :   atribuicao ';'  -> atribuicao
    |   read ';'    -> read
```

```
80      |   write ';'   -> write
81      |   ifs         -> ifs
82      |   whiles      -> whiles
83      |   invocacao ';'   -> invocacao
84      |   retorna ';' -> retorna
85      ;
86
87  retorna
88      : RETURN expr
89      -> ^(RETURN expr)
90      ;
91
92  invocacao
93      : ID '(' args? ')'
94      -> ^(INVOCACAO ID args?)
95      ;
96
97  args
98      : expr ( ',' expr )*
99      -> ^(ARGS expr+)
100     ;
101
102 atribuicao
103     :   ID '=' expr
104     -> ^('=' ID expr)
105     ;
106
107 write
108     :   WRITE '(' expr ')'
109     -> ^(WRITE expr)
110     ;
111
112 read
113     :   READ '(' ID ')'
114     -> ^(READ ID)
115     ;
116
117
118 ifs
119     :   IF '(' expr ')' bloco (ELSE bloco)?
120     -> ^(IF expr bloco (bloco)?)
121     ;
122
123 whiles
124     :   WHILE '(' expr ')' bloco
125     -> ^(WHILE expr bloco)
126     ;
127
128 bloco
129     :   '{' statements '}'          ->  statements
130     |   statement               ->  ^(STATEMENTS statement)
131     ;
132
133 expr
134     : orExpr
135     -> orExpr
136     ;
137
138 orExpr
139     :   andExpr (opOr^ andExpr )*
140     ;
```

```
141
142 andExpr
143     :   equalityExpr(opAnd^ equalityExpr )*
144     ;
145
146 equalityExpr
147     :   additiveExpr (opRel^ additiveExpr)*
148     ;
149
150
151 additiveExpr
152     :   multiplicativeExpr (opAdd^ multiplicativeExpr)*
153     ;
154
155 multiplicativeExpr
156     :   notExpr (opMul^ notExpr )*
157     ;
158
159 notExpr
160     :   (opNot^)? negationExpr
161     ;
162
163 negationExpr
164     :   ('-'^)? factor
165     ;
166
167
168 opAdd
169     :   '+'
170     |   '-'
171     ;
172
173 opMul
174     :   '*'
175     |   '/'
176     |   \'%'
177     ;
178
179 opOr:   '||'
180     ;
181
182 opAnd
183     :   '&&'
184     ;
185
186 opRel
187     :   '>'
188     |   '<'
189     |   '>='
190     |   '<='
191     |   '=='
192     |   '!='
193     ;
194
195 opNot
196     :   '!'
197     ;
198
199 factor
200     :   ID
201     | constante     ->  constante
```

```
202     | invocacao     ->  invocacao
203     ;
204
205 constante
206     :   STRING
207     |   INT
208     |   TRUE
209     |   FALSE
210     ;
211
212
213
214
215 //lexer
216
217 RETURN  :   'return';
218
219 IF  :   'if';
220
221 ELSE    :   'else';
222
223 TD_INT  :   'int';
224
225 TD_BOOL :   'bool';
226
227 TD_STRING   :   'string';
228
229 TD_VAZIO :  'void';
230
231 WHILE       :   'while';
232
233 TRUE        :   'true';
234
235 FALSE       :   'false';
236
237 WRITE       :   'print';
238
239 READ        :   'scan';
240
241 ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
242     ;
243
244 INT :   '0'..'9'+
245     ;
246
247 COMMENT
248     :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
249     |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
250     ;
251
252 WS  :   ( ' '
253         | '\t'
254         | '\r'
255         | '\n'
256         ) {$channel=HIDDEN;}
257     ;
258
259 STRING
260     :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
261     ;
262
```

```
263 CHAR:  '\'' ( ESC_SEQ | ~('\''|'\\') ) '\''
264     ;
265
266 fragment
267 HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
268
269 fragment
270 ESC_SEQ
271     :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
272     |   UNICODE_ESC
273     |   OCTAL_ESC
274     ;
275
276 fragment
277 OCTAL_ESC
278     :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
279     |   '\\' ('0'..'7') ('0'..'7')
280     |   '\\' ('0'..'7')
281     ;
282
283 fragment
284 UNICODE_ESC
285     :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
286     ;
```

8 MSP

8.1 Implementação

Para incluir a geração de código MSP na nossa linguagem, foi apenas Incluir a geração de código MSP na nossa linguagem foi apenas fazer instruções nas produções para guardar as respetivas instruções em MSP, tal como é mostrado a seguir.

Basicamente são criados 2 arrays, um array com as declarações e outro com as instruções.

Listing 4: Gerar MSP

```
1  tree grammar CmbTGMSP2;
2
3  options{
4      tokenVocab=Cmb;
5      ASTLabelType = CommonTree;
6      output = AST;
7      backtrack = true;
8  }
9
10 @header{
11     import java.util.ArrayList;
12 }
13
14 @members{
15     private int i = 0;
16     private boolean corpoFuncao = false;
17
18     private ArrayList<String> instrucoes = new ArrayList<String>();
19     private ArrayList<String> declaracoes = new ArrayList<String>();
20
21     void regCode(String codeStr){
22         instrucoes.add(codeStr);
23     }
```

```
24
25      void regDecl(String decl){
26          declaracoes.add(decl);
27      }
28
29  }
30
31  programa returns [ArrayList<String> msp_declaracoes, ArrayList<String> msp_instrucoes
        ]
32      :   ^(PROGRAMA (funcao)+
33      {
34          $programa.msp_declaracoes = declaracoes;
35          $programa.msp_instrucoes = instrucoes;
36      }
37      )
38      ;
39
40  funcao
41      :  ^(FUNCAO cabecalho corpo_funcao)
42      ;
43
44  cabecalho
45      :  ^(CAEBECALHO tipo ID argumentos?);
46
47  argumentos
48      :  ^(ARGUMENTOS declaracao+)
49      ;
50
51  corpo_funcao
52      : ^(CORPO {corpoFuncao = true;} declaracoes statements {corpoFuncao = false;} );
53
54  declaracoes
55      : ^(DECLARACOES declaracao+)
56      ;
57
58  declaracao
59      :   ^(DECLARACAO tipo ID {if(corpoFuncao) regDecl("Decl \"" + $ID.text + "\" " +
            i++ + " 1");})
60      ;
61
62  tipo
63      :   TD_INT
64      |   TD_BOOL
65      |   TD_STRING
66      |   TD_VAZIO
67      ;
68
69  statements
70      :    ^(STATEMENTS (statement)+)
71      ;
72
73
74  statement
75      :   atribuicao
76      |   read
77      |   write
78      |   ifs
79      |   whiles
80      |   invocacao
81      |   retorna
82      ;
```

```
83
84  retorna
85      : ^(RETURN expr)
86      ;
87
88  invocacao
89      : ^(INVOCACAO ID args?)
90      ;
91
92  args
93      : ^(ARGS (expr)+)
94      ;
95
96  atribuicao
97      : ^('=' ID {regCode("Pusha \"" + $ID.text +"\"") ;} expr {regCode("Store");})
98      ;
99
100 write    : ^(WRITE expr {regCode("IOut");})
101     ;
102
103 read     : ^(READ ID {regCode("Pusha \"" + $ID.text + "\",IIn,Store");})
104     ;
105
106
107 ifs      : ^(IF
108         expr {regCode("Jumpf \"senao"+ $IF.line +"\"");}
109         a=bloco {regCode("Jump \"fse"+ $IF.line +"\""); regCode("ALabel \"senao"+$IF.
                line+"\" ");}
110         (b=bloco)? {regCode("ALabel \"fse"+$IF.line+"\""); }  )
111     ;
112
113 whiles   : ^(WHILE {regCode("enq"+ $WHILE.line+ ": ");}
114     expr {regCode("JMPF fenq"+ $WHILE.line);}
115     bloco {regCode("fenq"+$WHILE.line+": ");})
116     ;
117
118 bloco    : statements
119     ;
120
121 expr returns [String instrucao]
122     :   ^('||' a=expr b=expr)   {regCode("Or"); $expr.instrucao = $a.instrucao + "||"
              + $b.instrucao;}
123     |   ^('&&' a=expr b=expr)   {regCode("And"); $expr.instrucao = $a.instrucao + "&&
            "  + $b.instrucao;}
124     |   ^('+' a=expr b=expr)    {$expr.instrucao = $a.instrucao + "+"   + $b.
            instrucao;}
125     |   ^('-' a=expr b=expr)    {$expr.instrucao = $a.instrucao + "-"   + $b.
            instrucao;}
126     |   ^('*' a=expr b=expr)    {regCode("Mul"); $expr.instrucao = $a.instrucao + "*"
              + $b.instrucao;}
127     |   ^('/' a=expr b=expr)    {regCode("Div");$expr.instrucao = $a.instrucao + "/"
              + $b.instrucao;}
128     |   ^(\'%' a=expr b=expr)   {regCode("Mod"); $expr.instrucao = $a.instrucao + "\%
            "  + $b.instrucao;}
129     |   ^('>' a=expr b=expr)    {regCode("Gt"); $expr.instrucao = $a.instrucao + ">"
              + $b.instrucao;}
130     |   ^('<' a=expr b=expr)    {regCode("Lt"); $expr.instrucao = $a.instrucao + "<"
              + $b.instrucao;}
131     |   ^('>=' a=expr b=expr)   {regCode("Ge"); $expr.instrucao = $a.instrucao + ">="
              + $b.instrucao;}
132     |   ^('<=' a=expr b=expr)   {regCode("Le"); $expr.instrucao = $a.instrucao + "<="
```

```
            + $b.instrucao;}
133     |   ^('==' a=expr b=expr)   {regCode("Eq"); $expr.instrucao = $a.instrucao + "=="
            + $b.instrucao;}
134     |   ^('!=' a=expr b=expr)   {regCode("Ne"); $expr.instrucao = $a.instrucao + "!="
            + $b.instrucao;}
135     |   ^('!' a=expr)           {$expr.instrucao = "!" + $a.instrucao;}
136     |   factor                  {$expr.instrucao = $factor.instrucao;}
137     ;
138
139 factor returns [String instrucao]
140     : ID        {regCode("Pusha \"" + $ID.text + "\",Load"); $factor.instrucao = $ID.
            text;}
141     | constante {System.out.println("PUSH " + $constante.text); $factor.instrucao =
            $constante.valor;}
142     | invocacao
143     ;
144
145 constante returns [String valor]
146     :   STRING  {$constante.valor = $STRING.text;}
147     |   INT {$constante.valor = $INT.text;}
148     |   TRUE    {$constante.valor = $TRUE.text;}
149     |   FALSE   {$constante.valor = $FALSE.text;}
150     ;
```

9 Compilação

Uma pergunta importante que ainda está por ser respondida é: ”Afinal como se liga isto tudo?”, ou seja, como é que se corre as árvores.

Para cada TG criada, existe uma thread correspondente que vai executar a tarefa correspondente ao output da TG, essas threads estão definidas na classe `MyThread`. Existe ainda a classe `Run` que basicamente criar uma AST que envia depois para as várias threads que irão executar tarefas diferentes sobre elas.

Para que isto seja possível é necessário no AntlWorks gerar o código de caga TG, depois na pasta output executar o comando `javac Run` para compilar todo o projeto. Depois de compilado, o comando para executar é `java Run ficheitoInput.i`

Parte da classe `Run.java`:

```
// gera a arvore
CharStream in = new ANTLRFileStream(args[0], "UTF8");
CmbLexer lexer = new CmbLexer(in);
CommonTokenStream tokens = new CommonTokenStream(lexer);
CmbParser parser = new CmbParser(tokens);
CmbParser.programa_return ret = parser.programa();

...
...

// Thread para gerar o MSP
CmbTGMSP2 walkerMSP2 = new CmbTGMSP2(new CommonTreeNodeStream(ret.getTree()));
Thread tMSP2 = new MyThread(walkerMSP2, 6);
tMSP2.start();
System.out.println("Começou MSP2");
```

Parte da classe `MyThread.java`:

```
public class MyThread extends Thread {
    ...
    private CmbTGMSP2 _walkerMSP2;
    private CmbTGMSP2.programa_return _walkerMSP2Ret;

    private int _tipo;

    public MyThread(Object walker, int tipo) {
        _tipo = tipo;

        switch (tipo) {
            ...
            case 6:
                _walkerMSP2 = (CmbTGMSP2) walker;
                break;
            ...
        }
    }

    public void run() {
        try {

            switch (_tipo) {
                ...
                case 6:
                    _walkerMSP2Ret = _walkerMSP2.programa();
                    //System.out.println("AQUI VAI SER O CODIGO MSP2");
                    toMSP2(_walkerMSP2Ret.msp_declaracoes, _walkerMSP2Ret.msp_instrucoes);
                    break;
                ...
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

...
...
...

    public void toMSP2(ArrayList<String> declaracoes, ArrayList<String> instrucoes) {
        //System.out.println("Declaracoes: " + declaracoes.size());
        //System.out.println("Instrucoes: " + instrucoes.size());

        String outS = "Msp ";

        outS += "[" + combine(declaracoes.toArray(new String[]{}), ",\n ") + "]";
        outS += "[" + combine(instrucoes.toArray(new String[]{}), ",\n ") + "]";

        //System.out.println(outS);

        try {
            FileWriter fstream = new FileWriter("msp2.txt");
            BufferedWriter out = new BufferedWriter(fstream);
            out.write(outS);
            out.close();
```

```
        } catch (Exception e) {// Catch exception if any
            System.err.println("Error: " + e.getMessage());
        }

    }

...
...
}
```

10 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.

A dificuldade neste trabalho foi encontrar a melhor forma de traduzir a linguagem i– em código para correr na máquina virtual MSP.