

# 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*

21 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

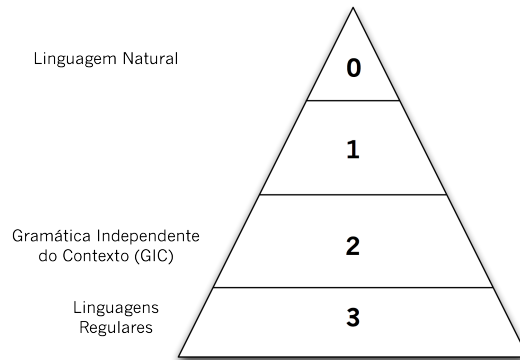
‡Email: milton.nunes52@gmail.com

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Ambiente de Trabalho</b>	<b>3</b>
<b>3</b>	<b>Descrição do problema</b>	<b>3</b>
<b>4</b>	<b>MSP</b>	<b>4</b>
4.1	Implementação . . . . .	4
4.1.1	Declarações . . . . .	4
4.1.2	Ler e Escrever . . . . .	4
4.1.3	if . . . . .	5
4.1.4	Operadores . . . . .	5
4.2	Exemplo . . . . .	6
<b>5</b>	<b>Conclusões</b>	<b>7</b>

# 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 formal<sup>1</sup> 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.

Quando se escrever um programa, o objetivo é que ele faça alguma tarefa automaticamente, para tentar simular essa execução, nesta fase usamos uma máquina virtual 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 é usar a gramática criada nos trabalhos anteriores da disciplina e adaptar para gerar código para ser lido pela MSP.

Das várias formas possíveis de fazer isso, a utilizada foi adicionar instruções às produções da gramática para dar como output o código pretendido.

---

<sup>1</sup>Objecto matemático que permite criação de linguagens através de um conjunto de regras de formação.

## 4 MSP

### 4.1 Implementação

Incluir a geração de código MSP na nossa linguagem foi apenas fazer instruções nas produções para imprimir no output, ou seja, basicamente criamos uma gramática tradutora.

Neste momento só suporta uma única função e variáveis do tipo inteiro.

Usando a gramática que foi criada nos projetos anteriores, adicionamos as instruções a seguir para gerar MSP.

#### 4.1.1 Declarações

As declarações devem ser feitas no início da função e são traduzidas da seguinte forma:

```
declaracoes
:      {System.out.println("MEMORIA DE DADOS");} declaracaoExpr+
;

declaracaoExpr
:      declaracao';'
;

declaracao
:      tipo ID {System.out.println($ID.text + " " + i++ + " TAM 1");}
;
```

Usando um exemplo, se tivermos declarado:

```
int in;
int y;
```

Temos como output:

```
MEMORIA DE DADOS
  in 0 TAM 1
  y  1 TAM 1
```

#### 4.1.2 Ler e Escrever

Relativamente às funções de leitura e escrita que o MSP implementa foi simples traduzir:

```
ler :      LER '(' ID ') ' {System.out.println("PSHA " + $ID.text + "\nIN\nSTORE");}
;

escrever:   ESCREVER '(' ID ') ' {System.out.println("PSHA " + $ID.text + "\nLOAD\nOUT");}
;
```

Se tivermos:

```
ler(in);
```

Obtemos:

```
PSHA in
IN
STORE
```

#### 4.1.3 if

Para controlar a existência de vários if's na mesma função, usamos uma variável global **fse** que indica o número do if no programa.

```
ifs :    IF '(' expr ')' {System.out.println("JMPF senao"+ ++senao);}
        bloco {System.out.println("JMP fse"+ ++fse);}
        ifsElse? {System.out.print("fse"+fse+": ");}
```

Usando como exemplo de entrada:

```
if(in > 10){
    escrever(in);
}else{
    escrever(y);
}
```

Temos como output MSP:

```
PSHA in
LOAD
PUSH 10
GT
JMPF senao1
PSHA in
LOAD
OUT
JMP fse1
```

#### 4.1.4 Operadores

Para usar nas condições dos if's e dos while's, é necessário converter os operadores lógicos e matemáticos. Isso é feito da seguinte forma:

```
orExprAux:    opOr andExpr {System.out.println("OR");}
;

andExpr :     equalExpr andExprAux*
;

andExprAux: opAnd equalExpr    {System.out.println("AND");}
;

equalExpr
:    addExpr equalExprAux*
;

equalExprAux: opRel addExpr
               {if($opRel.text.equals("==")) System.out.println("EQ");
                if($opRel.text.equals(">")) System.out.println("GT");}
```

```

        if($opRel.text.equals(">=")) System.out.println("GE");
        if($opRel.text.equals("<")) System.out.println("LT");
        if($opRel.text.equals("<=")) System.out.println("LE");
        if($opRel.text.equals("!=")) System.out.println("NE");}

;

addExpr :    multExpr addExprAux*
;

addExprAux: opAdd multExpr
;

multExpr: notExpr multExprAux*
;

multExprAux: opMul notExpr {if($opMul.text.equals("/")) System.out.println("DIV");
        if($opMul.text.equals("*")) System.out.println("MUL");
        if($opMul.text.equals("%")) System.out.println("MOD");}

;

```

## 4.2 Exemplo

O seguinte exemplo na linguagem i-:

```

int main(){
    int in;
    int y;

    ler(in);

    y = 0;

    if(in > 10){
        escrever(in);
    }else{
        escrever(y);
    }
    return 0;
}

```

Gera o seguinte código MSP:

```

MEMORIA DE DADOS
    in 0 TAM 1
    y 1 TAM 1
CODIGO
    PSHA in
    IN
    STORE
    PSHA y
    PUSH 0
    STORE
    PSHA in
    LOAD
    PUSH 10

```

```
GT
JMPF senao1
PSHA in
LOAD
OUT
JMP fse1
senao1: PSHA y
LOAD
OUT
fse1: PUSH 0
HALT
```

## 5 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.

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