# Trabalho Bimestral da Disciplina de Construção de Compiladores

**Alex F. Cordeiro**[1], **Pablo Mezzon Kintopp**[1]

[1]Universidade Técnologica Federal do Paraná (UTFPR)
Av. Brasil, 4232 - Independência, Medianeira - PR, 85884-000 – Medianeira – PR – Brazil

***Abstract.*** *Abstract.*

***Resumo.*** *Resumo.*

## 1. Compiladores

Compilador é um programa de computador que transforma ou traduz uma linguagem de programação em alto nível para uma linguagem em mais baixo nível. Isto é, ele converte uma linguagem de programação tal como C/C++, Java e Python em linguagem de máquina. Entretanto como aponta [Aikes 2016], o compilador proporciona vantagens além de sua principal função. Essas vantagens são: abstração do uso de linguagens de baixo nível, checagem de erros e vulnerabilidades, geração de código portável e otimização de código. O trabalho do compilador é subdividido em duas etapas: a primeira é chamada de *frontEnd* ou análise e a segunda é chamada síntese ou *backEnd*. Na etapa de análise ocorrem as análises léxica, sintática e semântica. Sendo que durante essas análises, diversos tipos de erros podem ser identificados e, as vezes, reparados. Nessa etapa também ocorre a criação de uma representação intermediaria do programa. Na etapa de síntese, as representações intermediarias criadas na etapa anterior são utilizadas para construir o programa de destino.

## 2. Análise Léxica

O analisador léxico é uma função que transforma sequências de caracteres em sequências de símbolos, palavras ou *tokens* [Aikes 2016]. Essa análise é a primeira etapa no processamento de programas. Ela faz a leitura sequencial dos caracteres que formam o programa fonte. Ao ler os caracteres é feita uma separação dos mesmo em *tokens*. E para cada um dessas palavras é feito um reconhecimento, isto é, o analisador verifica se os *tokens* gerados são válidos. Ainda durante essa análise, é usada a tabela de símbolos para guardar informações a respeito dos *tokens* criados. A tabela de símbolo é uma estrutura no formato *hash* usada para mapear um ID de um *token* com suas informações.

## 3. Análise Sintática

A análise sintática é a etapa que a partir dos *tokens* gerados pela análise léxica irá construir uma representação da estrutura que gramatical do programa. Essa estrutura é chamada árvore sintática e ela é fundamental na verificação das regras sintáticas da linguagem. Para fazer isso, os símbolos terminais são agrupados e verificados se formam uma frase sintaticamente válida. Em outras palavras, ao derivar os símbolos não terminais a estrutura no formato de árvore é construída. Se as folhas dessa arvore gerada forem compostas de símbolos terminais válidos, a sentença de entrada é aceita.

## 4. Análise Semântica

A análise semântica é a ultima etapa de análise antes do compilador gerar o programa intermediário. Essa fase procura possíveis erros semânticos e guarda informações contextuais adicionais. Isso é feito por meio da verificação de regras semânticas na linguagem e por cálculos de valores associados aos símbolos. Dessa maneira é possível obter-se o significado completo da frase.

## 5. Aplicação desenvolvida

Com base nas instruções recebidas em sala de aula, e nos requisitos do trabalho, foi desenvolvido um compilador contendo as etapas de análise léxica e sintática com um controle básico de erros. Para esse trabalho foi utilizada a linguagem Java, junto com a JavaCC, que é uma ferramenta de auxilio para criação de compiladores. A seguir é mostrada a gramática criada, utilizando para isso o formalismo de Backus-Naur:

<S> ::= <PROGRAM_BEGIN><BODY><PROGRAM_END>

<BODY> ::= (<DECLARATION> | <ATRIBUTION><SEMICOLON>
| <IF_BLOCK> | <FOR_BLOCK>)+

<DECLARATION> ::= <TYPE><ID>(<COMMA><ID>)*<SEMICOLON>

<ATRIBUTION> ::= <ID><ASSIGN> (<MATH_EXPRESSION>| <NUMBER>
|<STRING_DELIMITER><ID><STRING_DELIMITER>
| <ID>)

<DECLARATION_ATRIBUTION> ::= <TYPE><ID><ASSIGN> (<MATH_EXPRESSION>
| <NUMBER> | <STRING_DELIMITER><ID><STRING_DELIMITER>
| <ID>) <SEMICOLON>

<IF_BLOCK> ::= <IF><LOGIC_EXPRESSION><THEN><BODY>(<ELSE><BODY>)? <END_IF>

<FOR_BLOCK> ::= <FOR>((<TYPE><ATRIBUTION>)
| <NUMBER>) <TO> <LOGIC_EXPRESSION> <DOING>
<MATH_EXPRESSION> <MOREOVER> <BODY> <END_FOR>

<MATH_EXPRESSION> ::= (<ID>|<NUMBER>) (<MATH_OPERATOR> (<ID>|<NUMBER>))+

<LOGIC_EXPRESSION> ::= <LOGIC_EXPRESSION_SIMPLE>
((<AND> | <OR>) <LOGIC_EXPRESSION_SIMPLE>)*

<LOGIC_EXPRESSION_SIMPLE> ::= (<ID>|<NUMBER>)
<LOGIC_OPERATOR> (<ID>|<NUMBER>)

<MATH_OPERATOR> ::= <SUM>|<SUBTRACTION>
|<MULTIPLICATION>|<DIVISION>

<LOGIC_OPERATOR> ::= <BIGGER>|<MINOR>|<MINOR_EQUAL>
|<BIGGER_EQUAL>|<DIFERENTE>|<EQUAL>

<TYPE> ::= <INTEGER> | <FLOAT> | <STRING>

```
<NUMBER> ::= (<DIGIT>)+ (.(<DIGIT>)+)?
<ID> ::= <LETTER>(<LETTER><DIGIT>)*
<LETTER>::= { "_", "a"-"z", "A"-"Z"}
<DIGIT> ::= {"0"-"9"}
<SEMICOLON> ::= ;
<ASSIGN> ::= ::
<PROGRAM_BEGIN> ::= pogeando
<PROGRAM_END> ::= pogeou
<INTEGER> ::= numnatural
<FLOAT> ::= numreal
<STRING> ::= varditexto
<COMMA> ::= ,
<STRING_DELIMITER> ::= '
<IF> ::= umavezque
<THEN> ::= assimsendo
<ELSE> ::= porem
<END_IF> ::= umavezfeito
<FOR> ::= de
<TO> ::= ate
<DOING> ::= fazendo
<MOREOVER> ::= laceie
<END_FOR> ::= laceou
<AND> ::= ee
<OR> ::= ou
<SUM> ::= +
<SUBTRACTION> ::= -
<MULTIPLICATION> ::= *
<DIVISION> ::= /
<BIGGER> ::= >
<MINOR> ::= <
<MINOR_EQUAL> ::= <=
<BIGGER_EQUAL> ::= >=
<DIFERENTE> ::= <>
<EQUAL> ::= =
```

Como mostrado pelo formalismo de Backus-Naur, essa linguagem possui muitas construções com recursividade a direita. Esse tipo de recursividade pode ser vistas nas regras que geram os blocos fundamentais da linguagem, como o bloco condicional e o bloco de repetição. Em outras palavras, da maneira como a gramática foi criada a linguagem poderá ser escrita de maneira a aninhar estruturas, dando ao programador maior liberdade na hora de programar. Abaixo é mostrado um exemplo de uma linguagem válida gerada por essa gramática.

```
pogeando
        numnatural a :: 5 ;
        numreal b;
        numreal num :: b * 2 + 5.68;
        varditexto c, d, e;
        varditexto teste;
        d :: c;
        a :: 10 + D;
        b :: 25*2 + 89/2;
```

```
        numnatural f;
        f :: a;

        umavezque f <= 10 assimsendo
                f :: 50;
        umavezfeito

        umavezque a > f ee f > 0 ou a > 0 assimsendo
                numreal g :: 10.55;
                numreal h ;
                numreal i;
                h :: 5.555;
                g :: h;
                i :: g;
        porem
                numnatural g ;
                numnatural h;
                numnatural i;
                h :: 5;
                g :: h + 8;
                i :: g;
        umavezfeito

        numnatural j;

        de j :: i ate j >= 100 fazendo j + 1 laceie
                g :: i;
                umavezque a > f assimsendo
                        numreal k;
                        numreal l ;
                        numreal m;
                        k :: 5.555;
                        l :: h;
                        m :: g;
                porem
                        de l ate m > 500 fazendo m * l + 5 laceie
                                numnatural x;
                                x :: 5568;
                                x :: x - 500;
                        laceou
                umavezfeito
        laceou
pogeou
```

Entretanto uma linguagem inválida poderia ser dada simplesmente pela falta dos terminais obrigatórios *pogeando* e *pogeou* como mostrado abaixo:

```
numnatural g ;
numnatural h;
numnatural i;
h :: 5;
g :: h + 8;
i :: g;
```

Ou ainda, pela tentativa de uso de regras não definidas como pode ser visto no exemplo abaixo.

```
numnatural g, numnatural h ;

umavezque numnatural a = 0 > f assimsendo

de j :: i ate j >= 100 fazendo numnatural k = j + 1 laceie
```

## Referências

Aikes, J. (2016). Construcao de compiladores. http://moodle.utfpr.edu.br/.