*Ministério da Educação*
**Universidade Tecnológica Federal do Paraná**
*Pró-Reitoria de Pesquisa e Pós-Graduação*

# LINGUAGEM WAGNER

## Desenvolvimento de um compilador

**Helian Gustavo Bohm**
**Luiz Fernando da Costa Pereira**
**Martín Ávila Buitrón**

*CAMPUS TOLEDO*, 2022

**SUMÁRIO**

# 1. INTRODUÇÃO

Esse trabalho descreve um compilador desenvolvido para a matéria de Compiladores. O trabalho compreende as etapas de análise do programa fonte, sendo essas as análises léxica, sintática e semântica de um compilador, sem que o ocorra as etapas referentes à síntese. A linguagem desenvolvida é chamada de Wagner.

Wagner permite o uso de variáveis do tipo inteiro, ponto flutuante e cadeia de caracteres, recebendo os nomes de "*int"*, "*float"* e "*string"*, respectivamente, e é possível executar operações matemáticas de adição, subtração, multiplicação, divisão e resto de divisão com as variáveis numéricas. Permite também o uso de estruturas condicional e de repetição, através das funções "*if"* e "*while"*, que operam a partir de operações lógicas. "*read"* e *"write"* são as chamadas para leitura e escrita da linguagem. Pode-se também comentar o código. Não existe a função *main*.

Quando chamadas as estruturas condicional e de repetição, é necessário que o conteúdo referente a elas esteja entre chaves, sendo que a abertura dessas chaves deve ocorrer na mesma linha em que a estrutura é chamada e o fechamento em uma linha própria.

As principais limitações referentes ao compilador estão na impossibilidade de se utilizar vetores e criar estruturas e funções.

O compilador consegue receber códigos escritos pela *web* ou em arquivos de texto.

# 2. ANÁLISE LÉXICA

A análise léxica tem como objetivo varrer o código e determinar se o conteúdo do mesmo está gramaticalmente correto, com base nas regras da linguagem. Além disso, após a análise léxica o código é organizado em *tokens*, de modo a facilitar as etapas subsequentes do compilador.

## 2.1. Funcionamento

A análise léxica foi implementada através de um *parser* que percorre todo o código desenvolvido. As regras para a funcionalidade dessa etapa foram implementadas com o uso de um grafo, sendo que seus vértices representam os possíveis estados, e suas arestas os caracteres necessários para que ocorram as transições desses estados. A Figura 1 apresenta o autômato utilizado como base para implementação do grafo, é possível identificar os caracteres necessários para as transições de estados e os *tokens* gerados pelos estados finais.

Para o caso do estado 4 do autômato, uma cadeia de caracteres é gerada e é feita uma busca em uma tabela de símbolos como o objetivo de verificar se se está sendo utilizada alguma palavra reservada. Caso sejam identificadas as cadeias "*while*", "*if*", "*int*", "*float*", "*string*", "*write*" e "*read*" serão gerados, respectivamente, os *tokens* TK_While, TK_If, TK_Int, TK_Float, TK_String, TK_Write e TK_Read. Caso a cadeia gerada não corresponda a nenhuma dessas especificadas, será gerado um *token* TK_Identificador. Os identificadores são armazenados na tabela de símbolos e em uma tabela de identificadores.

Apesar de descrito no autômato como gerando um *token*, a implementação do estado 6, referente a comentários no código, faz com que não se considere o que aconteça na linha do comentário.

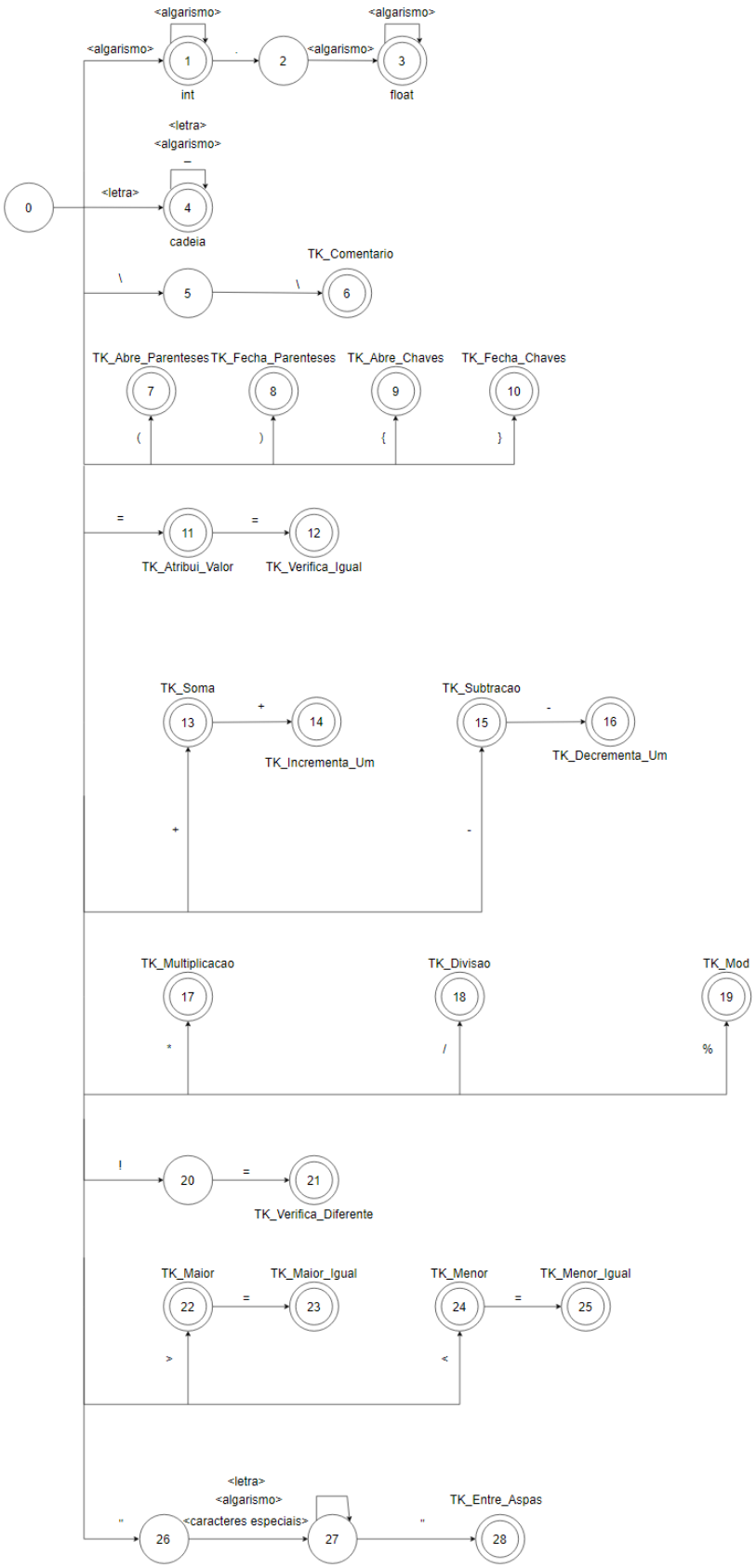


Figura 1. Autômato

## 2.2. Recuperação de Erros

Existem dois tipos de erros identificados na análise léxica. O primeiro é referente aos símbolos que não fazem parte do alfabeto da linguagem.

O segundo erro ocorre quando o *parser* não consegue percorrer alguma expressão e terminar em um estado descrito como final no autômato.

# 3. ANÁLISE SINTÁTICA

A análise sintática parte dos *tokens* gerados na análise léxica e, através de regras estabelecidas, verifica se a disposição deles em cada linha do código é válida.

## 3.1. Funcionamento

Como os *tokens* já vêm da análise léxica de modo que cada linha esteja armazenada em uma lista, o método utilizado para essa análise foi com uma *parser tree*, para que as folhas da árvore fossem comparadas com a lista de *tokens*. Para a criação das árvores, foram utilizadas diferentes EBNFs, de modo que a EBNF utilizada em cada linha analisada seria decidida com base no primeiro *token* da mesma. A gramatica é livre de contexto.

Após esse estado, os identificadores armazenados na tabela de identificadores passam a armazenar seus tipos e valores.

Seguem as respectivas EBNFs.

### 3.1.1. Caso primeiro *token* seja TK_While ou TK_If

<Lexema> := <While_If> <TK_Abre_Parenteses> <OP> <TK_Fecha_Parenteses> <TK_Abre_Chaves>

<While_If> := <TK_While> | <TK_If>

<OP> := <Operando> <LOG> <Operando>

<Operando> := <TK_Identificador> | <TK_Inteiro> | <TK_Flutuante>

<LOG> := <TK_Verifica_Igual> | <TK_Verifica_Diferente> | <TK_Maior> | <TK_Maior_Igual> | <TK_Menor> | <TK_Menor_Igual>

### 3.1.2. Caso primeiro *token* seja TK_Int

Nesse caso, assim como nos próximos dois, decide-se a EBNF utilizada com base na quantidade de *tokens* presentes na linha. Caso existam dois *tokens*:

<Lexema> := <TK_Int> <TK_Identificador>

Caso existam quatro *tokens*:

<Lexema := <TK_Int> <TK_Identificador> <TK_Atribui_Valor> <Operador>

<Operador> := <TK_Inteiro> | <TK_Identificador>

### 3.1.3. Caso primeiro *token* seja TK_Float

Caso existam dois *tokens*:

<Lexema> := <TK_Float> <TK_Identificador>

Caso existam quatro:

<Lexema> := <TK_Float> <TK_Identificador> <TK_Atribui_Valor> <Operador>

<Operador> := <TK_Flutuante> | <TK_Identificador>

### 3.1.4. Caso primeiro *token* seja TK_String

Caso existam dois *tokens*:

<Lexema> := <TK_String> <TK_Identificador>

Caso existam quatro:

<Lexema := <TK_String> <TK_Identificador> <TK_Atribui_Valor> <Operador>

<Operador> := <TK_Entre_Aspas> | <TK_Identificador>

### 3.1.5. Caso primeiro *token* seja TK_Write

<Lexema> := <TK_Write> <TK_Abre_Parenteses> <Conteudo_A_Ser_Printado> <TK_Fecha_Parenteses>

<Conteudo_A_Ser_Printado> := <TK_Identificador | <TK_Entre_Aspas>

### 3.1.6. Caso primeiro *token* seja TK_Read

<Lexema> := <TK_Read> <TK_Abre_Parenteses> <TK_Identificador> <TK_Fecha_Parenteses>

### 3.1.7. Caso primeiro *token* seja TK_Fecha_Chaves

<Lexema> := <TK_Fecha_Chaves>

### 3.1.8. Caso primeiro *token* seja TK_Identificador

Nessa situação, existem diferentes EBNFs para os casos de dois, três e cinco *tokens* presentes na lista. Caso existam dois *tokens*:

<Lexema> := <TK_Identificador> <OP>

<OP> := <TK_Incrementa_Um> | <TK_Decrementa_Um>

Caso existam três *tokens*:

<Lexema> := <TK_Identificador> <TK_Atribui_Valor> <Operando>
<Operando> := <TK_Flutuante> | <TK_Inteiro> | <TK_Identificador> | <TK_Entre_Aspas>

Por fim, caso existam cinco *tokens*:

<Lexema> := <TK_Identificador> <TK_Atribui_Valor> <Operação_Mat>

<Operação_Mat> := <Operando> <Simbolo_Mat> <Operando2>

<Operando> := <TK_Flutuante> | <TK_Inteiro> | < TK_Identificador>

<Operando2> := <TK_Flutuante> | <TK_Inteiro>

<Simbolo_Mat> := <TK_Soma> | <TK_Subtracao> | <TK_Multiplicacao> | <TK_Divisao> | <TK_Mod>

### 3.2. Recuperação de Erros

Com base nas regras estabelecidas, é possível retornar erros sintáticos ao usuário. Atribuição de valores de tipos inválidos a uma variável e não abrir ou fechar parênteses ao utilizar *write* ou *read* são alguns dos exemplos de erros identificados por essa etapa. Contudo, qualquer expressão que não esteja formatada de modo a ser validada pelas regras apresentadas retornará erros.

## 4. ANÁLISE SEMÂNTICA

A última etapa do compilador desenvolvido é a análise semântica. Nela é necessário identificar diferentes problemas que podem ocorrer e que não são resolvidos nas etapas anteriores.

O erro talvez mais simples de se identificar é o não fechamento de uma chave no código. É resolvido verificando se, para cada função que abre uma chave existe uma que fecha também.

Em operações matemáticas não se pode utilizar dados incompatíveis e realizar divisão por 0.

Outros potenciais erros são referentes a variáveis, o compilador não permite que uma mesma variável seja inicializada mais de uma vez, ou que seja utilizada sem ter sido inicializada. Além disso, variáveis do tipo inteiro só podem receber dados do tipo inteiro, as variáveis do tipo ponto flutuante podem receber dados inteiros e de ponto flutuante, já as cadeias de caracteres podem receber tanto os dados que as demais recebem quanto cadeias de caracteres. Em operações matemáticas que utilizem variáveis do tipo *string* é feita uma conversão, inicialmente para hexadecimal e depois para decimal.

## 5. PRÓXIMAS ETAPAS

A primeira questão a ser desenvolvida é identificar erros de forma mais específica para cada uma das etapas de análise do compilador, e a verificação do escopo do código. Após isso, é necessário o desenvolvimento de toda a etapa de síntese.

## 6. ACESSO AO CÓDIGO

O código pode ser acessado pelo seguinte *link*:

https://github.com/mab0205/COMPILER-

## 7. CONCLUSÃO

O desenvolvimento desse compilador permitiu a aplicação prática de conceitos aprendidos nas matérias de Teoria da Computação e de Compiladores, e permitiu um entendimento mais aprofundado do funcionamento dos compiladores.