



**UNIVERSIDADE FEDERAL DE ALAGOAS - UFAL  
INSTITUTO DE COMPUTAÇÃO - IC  
CAMPUS A. C. SIMÕES**

**CIÊNCIA DA COMPUTAÇÃO**

**ESPECIFICAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO GF  
2018.2**

**EDUARDO GABRIEL NUNES DE FARIAS  
FILIPE FALCÃO BATISTA DOS SANTOS**

**MACEIÓ - AL  
MARÇO DE 2019**

# SUMÁRIO

<b>1. Estrutura Geral de um Programa</b>	<b>4</b>
1.1. Se Admite Escopo Global e Como Variáveis São Declaradas	4
1.2. Como é Identificado o Ponto de Início de Execução	4
1.3. Onde e Como São Admitidas Definições de Funções	4
1.4. Onde e Como São Feitas Definições Locais	4
<b>2. Nomes</b>	<b>5</b>
2.1. Sensibilidade à Caixa	6
2.2. Limite de Tamanho	6
2.3. Formato: Expressão Regular (ER)	6
<b>3. Tipos e Estruturas de Dados</b>	<b>7</b>
3.1. Forma de Declaração	7
3.2. Tipos de Dados Primitivos	7
3.2.1. Declaração	7
3.2.2. Constantes Literais (ERs)	7
3.2.3. Operações	8
3.3. Cadeias de caracteres	8
3.3.1 Declaração/formato	8
3.3.2. Operações	8
3.4. Arranjos	8
3.4.1. Declaração	8
3.4.2. Referência a elemento	9
3.4.3. Forma de armazenamento em memória	9
3.5. Equivalência de tipos	9
3.5.1. Forma de equivalência	9
3.5.2. Coerções Admitidas	9
3.5.3. Conversão de tipo explícita (cast)	9
3.6. Se admite constantes com nome	9
<b>4. Atribuição e Expressões</b>	<b>10</b>
4.1. Atribuição	10
4.1.1. Símbolo	10
4.1.2. Operador ou Comando	10
4.2. Expressões Aritméticas, Relacionais e Lógicas	10
4.2.1. Operadores	10
4.2.2. Precedência	11
4.2.3. Associatividade	11

4.2.4. Se tipo das operações são definidos por expressão ou operador a operador	12
4.2.5. Aplicabilidade de avaliação em curto-circuito	12
<b>5. Sintaxe e exemplo de estruturas de controle</b>	<b>13</b>
5.1. Comandos de seleção	13
5.2. Comandos de iteração	14
5.2.1. Controle por contador	14
5.2.2. Controle lógico	14
5.3. Desvios incondicionais	14
<b>6. Subprogramas</b>	<b>15</b>
6.1. Procedimentos e funções	15
6.2. Métodos de passagem de parâmetros	15
6.3. Subprogramas como parâmetros	15
<b>7. Exemplos de código</b>	<b>16</b>
7.1. Alô Mundo	16
7.2. Série de Fibonacci	16
7.3. Shell Sort	17
<b>8. Tokens</b>	<b>19</b>
8.1. Introdução	19
8.2. Enumeração com as Categorias dos Tokens	20
8.3. Descrição dos Tokens	20

# 1. Estrutura Geral de um Programa

A linguagem GF é uma linguagem criada com o objetivo de implementar algoritmos de maneira simples e legível, a fim de auxiliar na aprendizagem de conceitos básicos de programação.

## 1.1. Se Admite Escopo Global e Como Variáveis São Declaradas

O programa escrito em GF, aceita escopo global e suas variáveis são declaradas com a utilização da palavra reservada “**var**” seguido de um espaço.

## 1.2. Como é Identificado o Ponto de Início de Execução

O ponto de início de execução é definido pela função “**main**”. Essa função tem por padrão o tipo de retorno “**int**” (similar a como é feito em C ou C++). A razão do tipo de retorno ser por padrão um inteiro é a verificação de erros relativos ao programa. Seu escopo é delimitado pelas palavras reservadas “**begin**” e “**end**”.

```
main() int
begin
...
return 0;
end
```

## 1.3. Onde e Como São Admitidas Definições de Funções

As funções iniciam com a palavra reservada “**function**”, em seguida um espaço e o nome da função, dentro de parênteses temos os parâmetros, em seguida outro espaço e o tipo de retorno da função, por fim, na linha abaixo, a palavra reservada “**begin**”. Na última linha da função é escrita a palavra reservada “**end**”. Nesse contexto, “**begin**” e “**end**” delimitam o escopo da função. Exemplo:

```
function nomeDaFuncao(int a) int
begin
...
end
```

## 1.4. Onde e Como São Feitas Definições Locais

A delimitação de escopo na linguagem é feita com as palavras reservadas “**begin**” e “**end**”, usados para iniciar e encerrar o escopo, respectivamente. Os comentários são definidos usando o símbolo hash (#) e o ponto e vírgula (;) é utilizado para terminar uma instrução.

## 2. Nomes

Os nomes que podem ser definidos de acordo com a nossa linguagem são compostos por uma sequência de caracteres, onde não são permitidos a utilização de caracteres especiais e números. Exemplo: “@gua”, “b0la” e “\$aída” não são nomes válidos, enquanto "agua", "bola", e "saida" são nomes válidos. Os nomes podem ser usados para nomear variáveis e funções, sendo chamados de identificadores.

Palavra-chave	Definição
begin	Definição de início.
end	Definição de final.
function	Definição de função.
int	Definição de tipo primitivo inteiro.
float	Definição de tipo primitivo de ponto flutuante.
char	Definição de tipo primitivo caractere.
string	Definição de tipo primitivo para uma cadeia de caracteres.
boolean	Definição de tipo primitivo lógico.
true	Valor lógico para “verdadeiro”.
false	Valor lógico para “falso”.
if	Definição de estrutura condicional "se".
elif	Definição de estrutura condicional "se não, se".
else	Definição de estrutura condicional "se não".
read	Instrução para receber informação de entrada.
print	Instrução para imprimir informação na saída (tela).
while	Definição de estrutura de repetição com controle lógico.
repeat	Definição de estrutura de repetição controlada por contador.
in	Componente da estrutura de repetição controlada por contador.
main	Definição da função de entrada do programa.
null	Definição de valor inicial nulo.

empty	Definição de retorno vazio.
var	Define a instanciação de uma variável.
return	Retorno de uma função.

### **2.1. Sensibilidade à Caixa**

Os nomes são case-sensitive, ou seja, os nomes "bola" e "Bola" são identificadores de duas entidades diferentes.

### **2.2. Limite de Tamanho**

Os nomes são limitados a 32 caracteres, como adotado por várias outras linguagens de programação.

### **2.3. Formato: Expressão Regular (ER)**

A expressão regular utilizada é: "[a-z\_A-Z)][[:alpha:]]\*".

## 3. Tipos e Estruturas de Dados

Na linguagem GF, a vinculação é fortemente tipada e estática. Dessa forma, quando um tipo é atribuído a uma variável ele permanece vinculado a ela durante toda a sua existência.

### 3.1. Forma de Declaração

A declaração é feita utilizando a palavra reservada “**var**”, seguida de um espaço, o tipo de dados (definidos abaixo), mais um espaço, e por fim um nome válido de uma variável.

### 3.2. Tipos de Dados Primitivos

São definidos os seguintes tipos de dados primitivos: **char**, **string**, **int**, **float**, **boolean**.

#### 3.2.1. Declaração

- **Caractere**: tem como base a tabela ASCII e utiliza a palavra-chave “**char**”;
- **String**: uma cadeia de caracteres (char), que é definida pela palavra-chave “**string**”;
- **Inteiro**: tipo numérico e inteiro, representado em 32 bits, e definido pela palavra-chave “**int**”;
- **Ponto flutuante**: tipo numérico e real de 32 bits, seguindo o padrão IEEE 754, com precisão de 6 casas decimais, sendo definido pela palavra-chave “**float**”;
- **Booleano**: assume somente os valores “**true**” e “**false**”. É definido pela palavra-chave “**boolean**”;
- **Array unidimensional**: definido pelo seu tipo que pode ser int, string, char, boolean, ou float, em seguida um espaço e o nome da variável. O tamanho é definido dentro de um par de colchetes, sendo este obrigatoriamente um número natural maior que zero. Caso o par de colchetes não possua nenhum número entre eles, teremos a declaração de um array sem tamanho definido.

#### 3.2.2. Constantes Literais (ERs)

- **Caractere**: um caractere literal somente é válido estando entre aspas simples. Expressão regular: “[[:alpha:]]\\n\\t”;
- **String**: uma string literal pode ter qualquer conjunto de caracteres ASCII dentro de entre aspas duplas. Expressão regular: “([[:alpha:]]\\n\\t)\*”;
- **Inteiro**: um literal inteiro aceita somente dígitos de 0 a 9. Expressão regular: “[[:digit:]]\*”;

- **Ponto flutuante:** as condições de literais inteiros se aplicam, e além disso, deve-se utilizar um ponto (.) para indicar as casas decimais após os dígitos. Expressão regular: "**[[:digit:]]\*\.[[:digit:]]+**";
- **Booleano:** um literal booleano assume somente os valores "true" ou "false". Expressão regular: "**true|false**";
- **Valores nulos:** para iniciar um tipo com um valor nulo é utilizado a palavra-chave **null**. Expressão regular: "**null**".

### 3.2.3. Operações

- **Atribuição:** Todos os tipos possuem esta operação;
- **Concatenação:** Os tipos de dados caracter, string, e array (somente do mesmo tipo) possuem esta operação;
- **Relacional:** Os tipos de dados caracter, string, inteiro, ponto flutuante, e booleano possuem esta operação;
- **Aritmética:** Os tipos de dados inteiro e ponto flutuante possuem esta operação;
- **Lógico:** Apenas o tipo de dados booleano possui esta operação.

## 3.3. Cadeias de caracteres

É definido como cadeia de caracteres as variáveis que são do tipo **string**.

### 3.3.1 Declaração/formato

**String:** uma cadeia de caracteres (char), que é definida pela palavra-chave **string**;

```
var string nome = "Gabriel";
```

### 3.3.2. Operações

As variáveis do tipo **string** aceitam as operações dos tipos: **atribuição, concatenação e relacional** (apenas igualdade entre operandos e diferença entre operandos).

## 3.4. Arranjos

É definido como arranjo um array unidimensional.

### 3.4.1. Declaração

**Array unidimensional:** em um array literal, seus valores devem ser colocados dentro de chaves e devem seguir, unicamente, o tipo definido no array. Ao ser especificado o tamanho do array, deve ser inicializado com a quantidade de dados equivalente ao valor entre os colchetes.



```
var int array[5];  
array = {1, 2, 3, 4, 5};
```

### 3.4.2. Referência a elemento

A referência a um elemento é feita ao utilizar a declaração do nome do elemento, em seguida um abre colchete, o número referente a posição do elemento desejado no array e por fim um fecha colchetes. O início do índice de um array é em 1.

```
print(array[5]); # Exibe o quinto elemento do array
```

### 3.4.3. Forma de armazenamento em memória

A linguagem GF não suporta a definição de arrays multidimensionais. Logo, todos os armazenamentos de arrays são feitos por linhas.

## 3.5. Equivalência de tipos

A linguagem GF suporta coerção automática e admite cast.

### 3.5.1. Forma de equivalência

Na linguagem GF, a equivalência de tipos é feita por nome, isso significa que duas variáveis são equivalentes se elas são definidas na mesma declaração, ou se em suas declarações são utilizados o mesmo nome de tipo.

### 3.5.2. Coerções Admitidas

A coerção acontece de maneira automática entre do tipo **string** para o tipo **char** e, e entre os tipos **int** e **float**. No caso de float para int, apenas a parte inteira de um número real é atribuída a variável inteira. No caso de int para float, uma parte decimal ".0" é adicionada. A coerção também é automática na concatenação de int ou float com string ou char.

### 3.5.3. Conversão de tipo explícita (cast)

A linguagem GF admite cast. São admitidas apenas as mesmas coerções que podem ser automaticamente realizadas (definidas no item acima).

## 3.6. Se admite constantes com nome

A linguagem GF não admite constantes com nome.

## 4. Atribuição e Expressões

### 4.1. Atribuição

As atribuições são realizadas ao utilizar um comando com o operador “=”. No lado direito da expressão fica o valor que irá ser atribuído e no lado esquerdo fica a variável. Somente são realizadas atribuições para tipos iguais, com exceção das coerções automáticas e casts (apresentadas nos itens 3.5.2 e 3.5.3). Dessa forma, caso o tipo de uma variável seja diferente do valor atribuído a ela, acontecerá um erro de compilação.

#### 4.1.1. Símbolo

O símbolo utilizado para a operação de atribuição é o “=”.

#### 4.1.2. Operador ou Comando

A atribuição é feita através de um comando de atribuição, como definido no início da sessão.

```
var int a = 1;
var int a = 'a'; # Erro de compilação
```

## 4.2. Expressões Aritméticas, Relacionais e Lógicas

### 4.2.1. Operadores

#### Aritméticos

- “+” : soma de operandos;
- “-” : subtração de operandos (binário) e negação (unário);
- “\*” : multiplicação de operandos;
- “/” : divisão de operandos;
- “^” : exponenciação de operandos;
- “%” : resto da divisão inteira.

#### Relacionais

- “>” : maior que;
- “<” : menor que;
- “>=” : maior ou igual que;
- “<=” : menor ou igual que;
- “==” : igualdade entre operandos;
- “!=” : diferença entre operandos.

## Lógicos

- “not” : negação lógica;
- “and” : conjunção (“e” lógico);
- “or” : disjunção (“ou” lógico).

## Concatenação

- “++” : concatenação.

### 4.2.2. Precedência

A precedência é dada pela tabela abaixo, da precedência mais alta até a mais baixa. Os operadores seguem as regras de associatividade especificadas na tabela:

Operador	Associatividade
$\wedge$	Direita para a esquerda
- (unário negativo) not	Direita para a esquerda
* /	Esquerda para direita
+ -	Esquerda para direita
++	Esquerda para direita
< > <= >=	Esquerda para direita
== !=	Esquerda para direita
and	Esquerda para direita
or	Esquerda para direita
=	Direita para a esquerda

É possível utilizar parênteses para alterar a precedência dos operadores. Como por exemplo:

```
1 + 2 * 2 # Resultado: 4
(1 + 2) * 2 # Resultado: 6
```

Neste segundo caso, temos o resultado igual a 6 graças a alteração de precedência com a utilização dos parênteses.

### **4.2.3. Associatividade**

Na linguagem GF, existe associatividade da esquerda para direita e da direita para a esquerda. Os operadores que possuem cada tipo de associatividade estão descritos na tabela do item 4.2.2.

### **4.2.4. Se o tipo das operações são definidos por expressão ou operador a operador**

O tipo das operações são definidos por expressão.

### **4.2.5. Aplicabilidade de avaliação em curto-circuito**

A linguagem GF não implementa a avaliação curto-circuito em operadores lógicos. Sempre que há uma operação lógica envolvendo dois operandos, independente do valor do primeiro operando, o segundo operando sempre é avaliado para determinar o resultado da operação lógica.

## 5. Sintaxe e exemplo de estruturas de controle

### 5.1. Comandos de seleção

O comando **"if"** é uma instrução condicional simples que requer uma expressão lógica. A lista de comandos será especificada dentro do bloco delimitado pela palavras reservadas **"begin"** e **"end"**, e é executada apenas se o valor da expressão lógica que vem entre parênteses for verdadeiro. Exemplo da sintaxe da instrução:

```
if (expressao_logica)
begin
...
end
```

O comando **"elif"** é uma estrutura condicional composta e funciona de forma similar. Se o valor da primeira expressão lógica for falso e o valor da segunda for verdadeiro, a lista de comandos que será executada é aquela definida entre os delimitadores que vêm após a palavra reservada **"elif"**. Exemplo da sintaxe da instrução:

```
if (expressao_logica)
begin
...
end
elif (expressao_logica_2)
begin
...
end
```

O comando **"else"** é uma estrutura condicional composta e funciona de forma similar. Se o valor da expressão lógica for falso, a lista de comandos que será executada é aquela definida entre os delimitadores que vêm após a palavra reservada **"else"**. Exemplo da sintaxe da instrução:

```
if (expressao_logica)
begin
...
end
else
begin
...
end
```

## 5.2. Comandos de iteração

Na linguagem GF, temos duas opções de comandos de iteração, o **"repeat"** e o **"while"**.

### 5.2.1. Controle por contador

O comando **"repeat"** permite repetir um conjunto de instruções por um número específico de vezes. O comando requer a definição de dois elementos:

1. Um valor inicial para o contador;
2. Um valor final para o contador.

Um valor de incremento do contador do laço é opcional. Seu valor padrão é de 1. A sintaxe da instrução **"repeat"** é similar ao laço for de python com o uso da função range, e é definida da seguinte maneira:

```
repeat i in (val_inicial, val_final, incremento)
begin
...
end
```

Se um valor para o incremento não for passado, como por exemplo:

```
repeat i in (val_inicial, val_final)
```

então, nesse caso, o incremento inferido pela linguagem será 1.

### 5.2.2. Controle lógico

O comando **"while"** permite que você repita uma lista de comandos até que uma expressão lógica especificada seja falsa. A sintaxe desta instrução é definida da seguinte forma:

```
while (expressao_logica)
begin
...
end
```

## 5.3. Desvios incondicionais

A Linguagem GF não possui nenhum comando de desvio incondicional.

## 6. Subprogramas

### 6.1. Procedimentos e funções

Na linguagem GF toda função deve ser declarada antes de ser usada, essa declaração tem que ser feita e implementada antes da função **"main"**, que é a responsável pelo ponto onde irá iniciar a execução do código. Nossa linguagem não permite definir funções aninhadas, ou seja, cada função tem que ser escrita separadamente uma da outra. Na declaração de uma função todos os argumentos devem ser declarados de acordo com a sintaxe abaixo:

```
function nomeDaFuncao(tipo param1, ..., tipo paramN) tipo
begin
...
end
```

Somente é admitido uma saída em uma chamada de função, ou seja, o modelo semântico de chamada de função é o modo de entrada. Abaixo temos um exemplo de chamada de função:

```
function exemplo(int a) empty
begin
    print(a);
end

main() empty
begin
    var int a;
    a = 1;
    exemplo(a);
    return 0;
end
```

### 6.2. Métodos de passagem de parâmetros

Só é admitido passar como parâmetro de entrada para uma função da linguagem GF, valores de tipos primitivos. A linguagem GF não suporta que seja passado como parâmetro um ponteiro ou algo similar.

### 6.3. Subprogramas como parâmetros

A linguagem GF não admite passagem de subprogramas como parâmetro para uma função.

## 7. Exemplos de código

### 7.1. Hello World!

```
main() int
begin
    print("Hello World!");
    return 0;
end
```

### 7.2. Série de Fibonacci

```
function fibonacci(int limit) int
begin
    var int count;
    var int fib1;
    var int fib2;
    var int fib3;

    count = 0;
    fib1 = fib2 = 1;

    if (limit == 0)
    begin
        print("0");
    end
    elif (limit >= 1)
    begin
        print("0,");
    end

    while (fib1 + fib2 <= limit)
    begin
        if (count < 2)
        begin
            print("1,");
        end
        else
        begin
            fib3 = fib2 + fib1;
            fib1 = fib2;
```



```

        fib2 = fib3;

        if (fib3 < limit)
        begin
            print(fib3 ++ ",");
        end
        elif (fib3 = limit)
        begin
            print(fib3);
        end
    end

    count = count + 1;
end

return 0;
end

main() int
begin
    var int limit;
    read(limit);
    fibonacci(limit);
    return 0;
end

```

### 7.3. Shell Sort

```

function shellSort(int array[], int size) int[]
begin
    var int i;
    var int j;
    var int gap;
    gap = size / 2;

    while(gap > 0)
    begin
        repeat i in (gap, size, 1)
        begin
            var int temp;
            var int j;

```

```

        j = 1;
        temp = array[i];

        while((j >= gap) and (array[j - gap] > temp))
        begin
            array[j] = array[j - gap];
            j = j - temp;
        end

        array[j] = temp;
    end

    gap = gap / 2;
end

return 0;
end

function printArray(int array[], int size) int
begin
    var int i;

    repeat i in (1, size, 1)
    begin
        print(array[i] ++ " ");
    end
    print("\n");

    return 0;
end

function readArray(int size) int[]
begin
    var int array[size];
    var int i;

    repeat i in (1, size, 1)
    begin
        read(array[i]);
    end
end

```

```
        return array;
    end

    main() int
    begin
        var int unsorted[10];
        printArray(readArray(10));
        printArray(shellSort(unsorted, 10));

        return 0;
    end
```

## 8. Tokens

### 8.1. Introdução

A linguagem de programação que irá ser utilizada para produção dos analisadores sintático e léxico será o Java.

### 8.2. Enumeração com as Categorias dos Tokens

As categorias de tokens definidas na linguagem GF são:

```
package lexical;

public enum Tokens {

    // Id
    id,

    // Keywords
    main,
    typeInt,
    typeFloat,
    typeBool,
    typeChar,
    typeString,
    typeEmpty,
    typeNull,
    funDecl,
    varDecl,
    cmdPrint,
    cmdRead,
    cmdIf,
    cmdElif,
    cmdElse,
    cmdWhile,
    cmdRepeat,
    cmdReturn,

    // Delimiters
    paramBeg,
    paramEnd,
    scopeBeg,
    scopeEnd,
    arrayBeg,
    arrayEnd,
    arrayValBeg,
    arrayValEnd,
    endLine,
    commaSep,
```

```
    // Constants
    constNumInt,
    constNumFloat,
    constBool,
    constChar,
    constString,

    // Operators
    opAssign,
    opEquals,
    opRel,
    opAditiv,
    opMult,
    opUnaryNeg,
    opConcat,
    opNot,
    opAnd,
    opOr,

    // Unknown
    unknown
```

```
}
```

### 8.3. Descrição dos Tokens

Token	Expressão Regular	Descrição
id	"[a-z_A-Z)][[:alpha:]]**"	Identificador
main	"main"	Definição da função principal
typeInt	"int"	Tipo int
typeFloat	"float"	Tipo float
typeBool	"boolean"	Tipo booleano
typeChar	"char"	Tipo char
typeString	"string"	Tipo string
typeEmpty	"empty"	Tipo de retorno empty
typeNull	"null"	Tipo null
funDecl	"function"	Declaração de função
varDecl	"var"	Declaração de variável
cmdPrint	"print"	Comando de exibição de dados
cmdRead	"read"	Comando de leitura de dados
cmdIf	"if"	Estrutura condicional de uma via
cmdElif	"elif"	Estrutura condicional de duas vias (else if)
cmdElse	"else"	Estrutura condicional de duas vias (else)
cmdWhile	"while"	Estrutura de repetição controlada por expressão
cmdRepeat	"repeat"	Estrutura de repetição controlada por contador
cmdIn	"in"	Componente da estrutura de repetição por contador
cmdReturn	"return"	Retorno de uma função
paramBeg	"\""	Abertura de parênteses

paramEnd	"\"	Fechamento de parênteses
scopeBeg	"begin"	Definição de início de escopo
scopeEnd	"end"	Definição de fim de escopo
arrayBeg	"["	Abertura de colchetes
arrayEnd	"]"	Fechamento de colchetes
arrayValBeg	"{"	Abertura de chaves
arrayValEnd	"}"	Fechamento de chaves
endLine	","	Ponto e vírgula
commaSep	","	Vírgula
constNumInt	"[:digit:]]**"	Constante inteira
constNumFloat	"[:digit:]]*\.[:digit:]]+\"	Constante float
constBool	"true false"	Constante booleana
constChar	"[:alpha:]] \\n \\t\"	Constante caractere
constString	"([[:alpha:]] \\n \\t)*\"	Constante cadeia de caracteres
opAssign	"="	Atribuição
opEquals	"== !="	Operador de igualdade
opRel	"> >= < <="	Operador relacional comparativo
opAditiv	"\+ -\"	Operador aditivo
opMult	"* \\ \\^ %"	Operador multiplicativo
opUnaryNeg	"-\"	Operador unário negativo
opConcat	"\\+ \\+\"	Operador de concatenação
opNot	"not"	Operador lógico NOT
opAnd	"and"	Operador lógico AND
opOr	"or"	Operador lógico OR