

## Especificação da Linguagem Do::Block – v1.5

---

### Notação BNF Estendida

Nas regras léxicas e sintáticas descritas abaixo os caracteres da notação BNF estão grifados em verde.

- Alternativas são separadas por barras verticais, ou seja, ' $a \mid b$ ' significa "*a ou b*".
  - Colchetes indicam ocorrência opcional: ' $[a]$ ' significa um *a* opcional, ou seja, equivale a " $a \mid \varepsilon$ " ( $\varepsilon$  refere-se à cadeia vazia).
  - Chaves indicam repetição: ' $\{a\}$ ' significa " $\varepsilon \mid a \mid aa \mid aaa \mid \dots$ ".
  - Parênteses indicam ocorrência de uma das alternativas: ' $(a \mid b \mid c)$ ' significa obrigatoriedade de escolha de *a* ou *b* ou *c*.
- 

### 1. Regras Léxicas

<i>letra</i>	::= a   b   ...   z   A   B   ...   Z
<i>digito</i>	::= 0   1   ...   9
<b>id</b>	::= { <i>_</i> } <i>letra</i> { <i>letra</i>   <i>digito</i>   <i>_</i> }
<b>intcon</b>	::= <i>digito</i> { <i>digito</i> }
<b>realcon</b>	::= <b>intcon</b> . <b>intcon</b>
<b>charcon</b>	::= ' <i>ch</i> '   '\n'   '\0', onde <i>ch</i> denota qualquer caractere imprimível da tabela ASCII, como especificado pela função <b>isprint()</b> da linguagem C, diferente de \ (barra invertida) e ' (apóstrofo).
<b>stringcon</b>	::= " <i>{ch}</i> ", onde <i>ch</i> denota qualquer caractere imprimível da tabela ASCII, como especificado pela função <b>isprint()</b> da linguagem C, diferente de " (aspas) e do caractere <i>newline</i> .
<b>idconst</b>	::= indica a ocorrência de uma constante inteira declarada e inicializada previamente com " <b>const int</b> "

*comentário*      Comentários iniciam sempre com o sufixo // (dupla barra comum) e se estendem até a ocorrência do caracter de fim de linha.

---

## 2. Regras Sintáticas

Símbolos *não-terminais* são apresentados em itálico; símbolos **terminais** são apresentados em negrito e, eventualmente, entre aspas por questões de clareza.

Declarações e comandos da linguagem têm estrutura própria, que indica como inicia e quando termina um comando. Sendo assim, declarações e comandos devem estar estruturalmente completos em uma única linha.

### 2.1 Regras de produção da gramática

```
prog                ::= {decl_list_var} {decl_block_prot} block_main {block_def}  
decl_list_var      ::= [const] tipo decl_var { , decl_var }  
decl_block_prot ::= block id [with [&] tipo { [ ] } { , [&] tipo { [ ] } } ]  
block_main        ::= block main {decl_list_var} { cmd } endblock  
decl_var           ::= id { [ intcon | idconst ] } [ = ( intcon | realcon | charcon | stringcon |  
                              { ( intcon | realcon | charcon | stringcon ) { , ( intcon | realcon |  
                              charcon | stringcon ) } } ) ]  
  
tipo                ::= char  
                      | int  
                      | real  
                      | bool  
  
block_def          ::= block id [with tipo id1 { [ intcon1 | idconst1 ] } { , tipo id2 { [ intcon2  
                              | idconst2 ] } } ] {decl_list_var} { cmd } endblock  
  
cmd                ::= do ( id [with id1 { , id2 } ] | cmd ) varying idx from expr1 (to |  
                              downto) expr2  
                      | do ( id [with id1 { , id2 } ] | cmd ) while ( expr )  
                      | do ( id [with id1 { , id2 } ] | cmd ) [for expr times ]  
                      | if ( expr ) { cmd } { elseif ( expr ) { cmd } } [ else { cmd } ] endif  
                      | while ( expr ) { cmd } endwhile  
                      | atrib
```

	goback
	getint id
	getreal id
	getchar id
	putint id
	putreal id
	putchar id
<i>atrib</i>	::= <b>id</b> { [ <i>expr</i> ] } = <i>expr</i>
<i>expr</i>	::= <i>expr_simp</i> [ <i>op_rel</i> <i>expr_simp</i> ]
<i>expr_simp</i>	::= [ +   - ] <i>termo</i> { ( +   -      ) <i>termo</i> }
<i>termo</i>	::= <i>fator</i> { ( *   /   && ) <i>fator</i> }
<i>fator</i>	::= <b>id</b> { [ <i>expr</i> ] }   <b>intcon</b>   <b>realcon</b>   <b>charcon</b>   ( <i>expr</i> )   ! <i>fator</i>
<i>op_rel</i>	::= ==
	!=
	<=
	<
	>=
	>

## 2.2. Associatividade e Precedência de Operadores

A tabela a seguir apresenta a associatividade de diversos operadores, assim como, as regras de precedência de cada um deles. A operação de um operador de maior precedência deve ser executada antes da operação associada a um operador de menor precedência. A precedência dos operadores diminui, à medida que avançamos da primeira linha para a última linha da tabela.

<u>Operador</u>	<u>Associatividade</u>
!, +, - (unário)	direita para esquerda
*, /	esquerda para direita
+, - (binário)	esquerda para direita
<, <=, >, >=	esquerda para direita
==, !=	esquerda para direita

&&	esquerda para direita
	esquerda para direita

---

## 3. Regras Semânticas

### 3.1. Declarações

As regras a seguir indicam como o processamento das declarações deve ser feito. Aqui a *definição* de uma função refere-se à especificação de seus parâmetros formais, variáveis locais e ao próprio corpo da função.

1. Um identificador pode ser declarado no máximo uma vez como global e no máximo uma vez como local em um bloco particular qualquer. Contudo, um mesmo identificador pode aparecer como local em vários blocos distintos. Caso ocorra um identificador local a um bloco possuir o mesmo lexema de um identificador global já declarado, o escopo local tem precedência enquanto ele estiver em execução.
2. Declarações de identificadores como variáveis e nomes de blocos (protótipo ou definição) podem ocorrer globalmente (escopo mais externo) segundo a ordem estabelecida na gramática da linguagem, mas a referência a qualquer desses elementos só pode ser feita após a sua declaração.
3. A declaração de todos os identificadores locais (no corpo da definição de um bloco) devem ocorrer sempre antes de qualquer comando (instrução de programa).
4. No escopo global, um bloco pode ter no máximo um nome e uma assinatura associada (protótipo); um bloco global pode ser definido no máximo uma vez. A linguagem não aceita blocos declarados e definidos localmente, ou seja, no escopo interno de outro bloco.
5. Se um bloco possuir um protótipo (ou assinatura) declarado, então os tipos dos parâmetros formais declarados na definição da função devem ser iguais, em número, tipificação e ordem, com os parâmetros presentes no protótipo. Blocos não possuem valor de retorno; caso seja necessário retornar valores processados ou gerados no bloco, usa-se parâmetros passados por referência. O protótipo de um bloco, se presente, deve anteceder a definição do bloco principal (**block main**). As definições dos blocos (com os respectivos corpos de comandos) devem suceder a definição do bloco principal. Toda definição de bloco deve possuir um protótipo associado declarado.
6. Um identificador pode aparecer no máximo uma vez na lista dos parâmetros formais na definição de um bloco.
7. Os parâmetros formais de um bloco possuem escopo local a esse bloco.
8. A execução de todo bloco (a exceção do bloco principal) deve obrigatoriamente ser feita mediante o uso do comando “do”

9. Nesta versão, a linguagem não suporta tipos apontadores, nem programas que não possuam bloco principal (main).
10. Outras regras semânticas não tratadas neste documento podem existir e devem ser elucidadas à medida que questionamentos associados forem surgindo ao longo do desenvolvimento do compilador

### 3.2. Requisitos de Consistência de Tipos

Variáveis devem ser declaradas antes de serem usadas. Blocos devem ter os seus tipos de parâmetros (via assinatura), antes deles serem executados via comando “do” no corpo de comandos de outros blocos. Se um identificador é declarado como possuindo escopo local a um bloco (variável local), então todos os usos daquele identificador se referem a essa instância local do identificador. Se um identificador não é declarado localmente a um bloco, mas é declarado globalmente, então qualquer uso daquele identificador, dentro dos blocos, se refere à instância com escopo global do identificador. As regras a seguir indicam como deve ser feita a checagem da consistência de tipos. A noção de compatibilidade de tipos é definida como se segue:

1. inteiro é compatível com inteiro, e caracter é compatível com caracter;
2. inteiro é compatível com caracter, e vice-versa;
3. O tipo implícito de uma expressão relacional (ex.:  $a \geq b$ ) é booleano, que é compatível com o tipo inteiro; em uma variável do tipo booleano, um valor igual a 0 (zero) indica falso lógico e um valor inteiro diferente de zero indica verdadeiro lógico;
4. Qualquer par de tipos não coberto por uma das regras acima não é compatível.

#### 3.2.1. Declaração e definição de Blocos

1. Blocos só podem ter sua execução iniciada via comando “do”; portanto, blocos não podem ocorrer como parte de uma expressão, por exemplo. Para usar valores resultantes do processamento do bloco deve-se fazer uso auxiliar de parâmetros passados por referência.
2. A presença do marcador especial “&” antecedendo o tipo de um parâmetro na declaração de um bloco (em sua assinatura) indica que aquele parâmetro será transferido por referência para o bloco, podendo, portanto, sofrer alteração no código do bloco que será refletida no escopo externo; caso esse marcador especial não ocorra, a transferência da variável do escopo externo para o escopo interno do bloco será por valor (cópia). O marcador especial “&”, quando necessário só deve aparecer na assinatura ou protótipo do bloco (ou seja, em sua declaração) – sua ocorrência é inválida, portanto, tanto na definição quanto no uso do bloco (via comando “do”).
3. Todo bloco declarado deve ser definido e vice-versa, de acordo com o especificado na gramática da linguagem.
4. Blocos podem possuir variáveis declaradas internamente. Nesse caso, elas terão escopo local e os espaços de memória alocados para elas, assim como a liberação desses espaços, ocorrerá a cada execução do bloco.

5. Um bloco não pode ser declarado ou definido internamente a outro bloco.

### 3.2.2. Expressões

O tipo de uma expressão *expr* é estabelecido como se segue:

1. Se *expr* é uma constante inteira, então seu tipo é inteiro.
2. Se *expr* é um identificador, então o tipo de *expr* é o tipo daquele identificador.
3. Se *expr* é uma expressão na forma  $e_1 + e_2$ ,  $e_1 - e_2$ ,  $e_1 * e_2$ ,  $e_1 / e_2$ , ou  $-e_1$ , então o tipo de *expr* é compatível com os tipos dos elementos da expressão, restritos a inteiro, caracter e real (**real**), ou seja, em " $e_1 + e_2$ " se  $e_1$  for caracter e  $e_2$  for inteiro, a operação é possível porque estes tipos possuem compatibilidade entre si e o tipo da expressão fica sendo inteiro; por outro lado, se  $e_1$  for um inteiro e  $e_2$  for um real um conflito de tipos é estabelecido e uma mensagem de erro deve ser emitida;
4. Se *expr* é uma expressão na forma  $e_1 \geq e_2$ ,  $e_1 \leq e_2$ ,  $e_1 > e_2$ ,  $e_1 < e_2$ ,  $e_1 == e_2$ , ou  $e_1 != e_2$  então o tipo de *expr* é booleano.
5. Se *expr* é uma expressão na forma  $e_1 \&\& e_2$ ,  $e_1 || e_2$ , ou  $!e_1$ , então o tipo de *expr* é booleano.

As regras para checagem de tipos em uma expressão, além daquelas estabelecidas acima, são as seguintes:

1. Cada argumento passado em uma execução de bloco deve ser compatível com o parâmetro formal correspondente declarado na assinatura ou definição daquele bloco.
2. As subexpressões associadas com os operadores  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\leq$ ,  $\geq$ ,  $<$ ,  $>$ ,  $==$ , e  $!=$  devem ser compatíveis com os tipos inteiro, caracter ou real. As subexpressões associadas com os operadores  $\&\&$ ,  $||$ , e  $!$  devem possuir tipos compatíveis com booleano.
3. Uma expressão usada como índice (entre colchetes) no acesso aos elementos de um *array* deve ser do tipo inteira.

### 3.3.3. Comandos

1. Apenas variáveis dos tipos básicos (inteiro, caracter, real e booleano) podem receber atribuições; o tipo associado ao lado direito de um comando de atribuição deve ser compatível com o tipo do lado esquerdo daquele comando de atribuição (o que está à direita e à esquerda do sinal de "=").
2. O tipo da expressão condicional em um comando "**do .. while**", "**if**", "**elseif**" e "**while**" deve ser booleano. Em todos os casos, se a avaliação da expressão condicional for falsa, logo na primeira avaliação, nenhuma execução do bloco ou dos respectivos comandos internos deve ocorrer. Caso ocorra a execução do bloco, ela se repetirá (nos comandos **do .. while** e **while**) enquanto a condição for verdadeira; nos comandos condicionais (**if** e **elseif**), o respectivo bloco só será executado se a expressão condicional associada resultar verdadeira.
3. Em um comando "**do .. varying**",  $expr_1$  deve ser menor que  $expr_2$  caso a cláusula "**to**" seja usada; similarmente,  $expr_1$  deve ser maior que  $expr_2$  caso a cláusula

“**downto**” seja usada. Em todos os outros casos, a execução do bloco ou o conjunto de comandos internos associados não deve ser iniciada.

---

## 4. Características Operacionais

A linguagem Do::Block possui as mesmas características de execução de uma linguagem de programação estruturada em blocos como o C. A descrição abaixo trata de alguns pontos específicos que devem ser de interesse. **Para outros aspectos não tratados explicitamente neste documento, deve-se discutir o projeto da linguagem buscando encontrar uma solução de consenso entre a especificação e filosofia da linguagem e a sua implementação.**

### 4.1. Dados

#### 4.1.1. Escalares

Variáveis do tipo escalar (inteiro, real, caracter ou booleano) ocupam uma posição de memória na máquina virtual do compilador.

Valores do tipo caracter são considerados sinalizados e havendo necessidade de converter um caracter em um inteiro por questões de compatibilidade, essa conversão deverá estender o sinal.

#### 4.1.2. Constantes Cadeias

Uma cadeia de characters (*string*) “ $a_1a_2a_3\dots a_n$ ” é um vetor de caracteres contendo  $n+1$  elementos, cujos primeiros  $n$  elementos são os caracteres da respectiva cadeia de caracteres, e cujo último elemento é o character `NULL` ou ‘\0’.

#### 4.1.3. Inicialização

Identificadores constantes (declarados com **const**) devem ser *explicitamente* inicializados na sua declaração (e apenas neste ponto). Identificadores não constantes podem ou não ser inicializados *explicitamente* na sua declaração – caso eles não sejam inicializados explicitamente pelo programador Do::Block, seus valores devem *implicitamente* inicializados com 0, se o identificador for do tipo inteiro, 0.0, se for do tipo real, ‘\0’ (caracter em branco), se for do tipo caracter e 0 (falso) se for do tipo booleano.

Identificadores do tipo *array* (vetores e matrizes) podem ser inicializados na sua declaração com o uso dos caracteres ‘{’ e ‘}’ indicando o início e o fim dos valores de inicialização de cada elemento (separados por vírgula). Caso o número de valores de inicialização seja inferior ao número de elementos do vetor ou da matriz, o último valor especificado pelo programador deve ser usado para preencher o restante dos elementos da estrutura. Caso

esse número de valores de inicialização seja maior que o número de elementos da estrutura, uma mensagem de erro deve ser emitida.

#### 4.1.4 Estruturados

Vetores e matrizes são dados estruturados de um mesmo tipo para Do::Block e seus elementos escalares são acessíveis pelo uso do operador de indexação '[']. Internamente, no código gerado, tanto vetores quanto matrizes são armazenados em posições contíguas de memória (no caso das matrizes as linhas são armazenadas de forma contígua).

### 4.2. Expressões

#### 4.2.1. Ordem de avaliação

- **Expressões Aritméticas** : os operandos de uma expressão (uma adição entre parênteses, por exemplo) devem ser calculados antes do cálculo da própria expressão. As regras sintáticas da linguagem descritas anteriormente garantem tanto a precedência quanto a associatividade dos operadores aritméticos e lógicos em uma expressão qualquer; operações de adição e subtração só são executadas após a execução das multiplicações e divisões a menos que as primeiras ocorram entre parênteses.
- **Expressões booleanas** : novamente, as regras sintáticas da linguagem estabelecem a ordem de avaliação dos operandos de uma operação de comparação envolvendo os operadores relacionais `>=`, `>`, `<=`, `<`, `==`, `!=`; da mesma forma acontece para os operadores lógicos `&&` (and), `||` (or) e `!` (not); as expressões envolvendo esses conectores lógicos devem ser avaliadas segundo a técnica do “circuito mais curto”.

#### 4.2.2. Conversão de tipos

Se um objeto do tipo caracter é parte de uma expressão, seu valor deve ser convertido (estendendo o sinal) para um valor do tipo inteiro antes que a expressão possa ser avaliada.

### 4.3. Comandos de Atribuição

#### Ordem de Avaliação

A ordem em que a expressão do lado direito de um comando de atribuição (após o sinal de “=”) será avaliada deverá respeitar as regras de precedência dos operadores estabelecidas na gramática.

### 4.4. Blocos

#### 4.4.1. Execução de um bloco



A execução de um bloco é sempre iniciada pelo emprego do comando “**do**”. Caso o comando possua cláusulas (“**varying**”, “**to/downto**”, “**for**”), elas devem ser avaliadas antes da primeira execução do bloco.

Um comando “**do**” sem cláusulas executa o respectivo bloco uma única vez.

#### 4.4.2. Avaliação de argumentos de blocos

A ordem em que os argumentos de um bloco deverão ser avaliados antes do início da execução do bloco obedecerá a ordem de ocorrência, ou seja, da esquerda para a direita.

#### 4.4.3. Passagem de parâmetros

Argumentos são sempre passados por valor (cópia), a exceção dos declarados com o marcador especial “**&**” antecedendo o tipo na declaração de um bloco (seu protótipo ou assinatura); nesse caso, a passagem do argumento se dá por referência – ou seja, alterar o valor do argumento dentro do bloco implica em perenizar essa alteração quando o bloco terminar sua execução.

Um argumento do tipo constante não pode ser associado a um parâmetro declarado com passagem por referência – apenas variáveis podem ser passadas por referência.

Por definição, parâmetros do tipo *array* (vetores e matrizes) são sempre passados por referência (não demandam, nem devem, ser declarados com o marcado “**&**” antecedendo seu tipo).

Parâmetros do tipo *array* (vetores e matrizes) devem possuir suas respectivas dimensões máximas (limites) declarados na **definição** do bloco. Na sua declaração, deve ocorrer apenas pares de colchetes ([ ]), tantos quantos forem as dimensões do *array*. Na execução do bloco, via comando “**do**”, apenas o nome do respectivo *array* (sem suas dimensões) deve ser especificado na lista de argumentos, posto que essas dimensões já foram declaradas no protótipo do bloco e estabelecidas na sua definição.

#### 4.4.4. Fim da execução de um bloco

Um bloco encerra sua execução e o fluxo de execução retorna para o ponto onde a execução do bloco foi iniciada ao final da execução regular de um bloco ou quando da ocorrência da execução do comando **goback** (inserido entre os comando que compõe o bloco). O comando **goback** não é necessário no final de um bloco – está implícito.

### 4.5. Execução do Programa

A execução do programa inicia no bloco especial nomeado **main**. Todo programa deve obrigatoriamente possuir um e um único bloco nomeado **main**, que corresponde ao ponto de entrada de execução do programa.