

# Descrição da linguagem Ç para a turma COMPILADORES/2019.1-1

Esta linguagem é case sensitive tanto para palavras reservadas quanto para identificadores. Logo, “uai” é uma palavra reservada, mas “UAI” não é. Além disso, “abcd”, “ABCD” e “aBCd” representam identificadores diferentes.

O compilador só deve aceitar arquivos contendo caracteres com valores na tabela ASCII entre 9 e 10 ou entre 32 e 126 (caracteres imprimíveis da tabela ASCII). Portanto, caracteres fora desta escala não são considerados nesta especificação.

## 1. Palavras e símbolos reservados:

e escreve int le letra mainha nada nao ou real se senao tome uai vaza  
, ; = + - \* / % ( ) [ ] { } < > <= >= == != ' "

### 1.1. Separadores

Os tradicionais espaço em branco, tab e quebra de linha são caracteres que indicam o fim de uma possível sequência de caracteres que formam um único elemento, a menos que apareçam em uma constante dos tipos caractere ou string (ver itens 2.3 e 2.4).

## 2. Constantes e identificadores

### 2.1. Identificador:

$$[a-zA-Z\_][a-zA-Z0-9\_]*$$

Todo identificador é iniciado por uma letra maiúscula ou minúscula ou por um underline seguido por zero ou mais letras maiúsculas/minúsculas, dígitos ou underlines. O tamanho máximo é de 128 caracteres.

Exemplos: a abCD x0 a0B\_1c7 \_111

### 2.2. Constante numérica:

$$[0-9]^+(\.[0-9]^*)$$

Um ou mais dígitos, que podem ser seguidos por ponto e zero ou mais dígitos para representar um número real. O tamanho máximo é de 128 caracteres. Uma constante numérica não pode ser imediatamente seguida por letras maiúsculas/minúsculas, underlines ou pontos.

Exemplos: 0123 1234567890 123. 123.456

### 2.3. Constante do tipo caractere:

Um caractere imprimível da tabela ASCII (exceto aspas simples) entre aspas simples. O tamanho mínimo e máximo é de 1 caractere entre as aspas simples.

Ex: 'a' 'A' '-' ' ' ' '

### 2.4. Constante do tipo string:

Sequência de caracteres imprimíveis da tabela ASCII (exceto aspas duplas) entre aspas duplas. O tamanho mínimo é de 0 e tamanho máximo é de 128 caracteres entre as aspas duplas.

Ex: "abc" "A saída esperada eh: " "\o/"

## 3. Estrutura de um programa

Todo programa deve ser a seguinte estrutura:

```
* protótipos de função aqui *  
  
* funções aqui *  
  
int mainha() {  
    * declaração de variáveis aqui *  
  
    * bloco de código aqui *  
}
```

Todos os protótipos de função devem ser listados antes do código de qualquer função. Após os protótipos, segue-se o código de todas as funções, e então o programa principal.

Um programa principal é separado em duas partes, a primeira contendo todas as declarações de variáveis, e a segunda composta por qualquer quantidade dos demais comandos abaixo, em qualquer ordem. Embora não haja um separador entre as duas partes, não é permitida a declaração de variáveis após um ou mais comandos de um outro tipo.

### 3.1. Declarações:

As declarações devem ser feitas da seguinte forma:

```
$tipo$ $id$;  
$tipo$ $id$ = $expr_arit$;  
$tipo$ $id$[$expr_arit$];
```

\$tipo\$ pode receber os valores **int** (inteiro de 64 bits em complemento de dois), **real** (ponto flutuante de 64 bits seguindo o padrão IEEE-754) e **letra** (inteiro de 8 bits em complemento de dois). \$id\$ é um identificador (ver item 2.1). \$expr\_arit\$ é uma expressão aritmética (ver item 3.3).

Exemplos:

```
int a;  
double pi= 3.14159      ;  
letra nome[256];
```

### 3.2. Atribuição

Segue o padrão para atribuição:

```
$id$ = $expr_arit$;  
$id$[$expr_arit$] = $expr_arit$;
```

No primeiro caso, \$id\$ é o identificador da variável de destino, e \$expr\_arit\$ é uma expressão aritmética que definirá o valor da variável. O segundo caso mostra a variável de destino sendo indexado caso seja um vetor.

Exemplos:

```
pi      = 3.14159      ;  
nome[233]='a' ;
```

### 3.3. Expressões aritméticas

Expressões aritméticas são zero ou mais operações sobre variáveis dos tipos **int**, **real** e/ou **letra**, posições de vetores, constantes numéricas e constantes do tipo caractere utilizando-se operadores aritméticos (“+”, “-”, “\*”, “/” e “%”) e parêntesis para alterar a precedência. Entre os operadores, “\*”, “/” e “%” possuem a mesma precedência, que por sua vez é superior a precedência dos operadores “+” e “-”, que também possuem a mesma precedência. O operador “-” pode ainda ser utilizado para mudar o sinal de expressões aritméticas, nesse caso com precedência superior a todos os demais operadores. No caso de operações com a mesma precedência, a ordem de resolução segue da esquerda para a direita. Por fim, o operador “%” só funciona quando ambos os operandos forem inteiros. Chamadas de função (ver item 5) podem ser utilizadas em expressões aritméticas. O tipo **real** tem precedência maior do que o tipo **int** que por sua vez tem precedência maior do que o tipo **letra** na resolução de expressões aritméticas. **IMPORTANTE:** Expressões aritméticas não são um comando por si só, elas apenas fazem parte de outros comandos (ex: Atribuição).

Exemplos:

```
a  
-a[0]  
(a)-b[c+d]  
'a'+345  
-(1)  
a+b*c  
(a+-b)%c(5,23)  
a/10*'b'
```

### 3.4. Expressões relacionais

`$expr_rel$` é uma expressão relacional, que compara duas expressões aritméticas através de um operador relacional (“<”, “>”, “<=”, “>=”, “==” e “!=”). **IMPORTANTE:** Expressões relacionais não são um comando por si só, elas apenas fazem parte de outros comandos (ex: Desvio condicional).

Exemplos:

```
$expr_arit$ == $expr_arit$  
$expr_arit$ != $expr_arit$
```

### 3.5. Expressões relacionais compostas

As palavras reservadas **e** e **ou** podem ser utilizadas para unir uma ou mais expressões relacionais formando uma expressão composta `$expr_comp$`, sendo que **e** tem maior precedência do que **ou**, parêntesis podem ser utilizados para alterar a precedência, e em operações com a mesma precedência, a ordem de resolução segue da esquerda para a direita. O operador **nao** pode ser usado para inverter o resultado de uma `$expr_comp$`. **nao** tem maior precedência do que **e** e **ou**.

Exemplos:

```
$expr_rel$ ou $expr_rel$ e $expr_rel$  
( $expr_rel$ ou $expr_rel$ ) e nao $expr_rel$
```

### 3.6. Desvio condicional

Um desvio condicional segue o seguinte padrão:

```
se ( $expr_comp$ ) {  
    * bloco de código aqui *  
}  
senao {  
    * bloco de código aqui *  
}
```

Um desvio condicional sempre possui um bloco **se**, seguido ou não por um bloco **senao**. Cada bloco tem 0 ou mais comandos e sempre é delimitado por chaves.

Exemplo:

```
se ( a > b ) {  
    b = a;  
}
```

### 3.7. Repetição

A estrutura de repetição segue o padrão:

```
uai ( $expr_comp$ ) {  
    * bloco de código aqui *  
}
```

Nela, o bloco de comandos é executado enquanto \$expr\_comp\$ for verdadeira.

Exemplo:

```
a=0;
uai(a < 10) {
    a = a+1;
}
```

O comando **vaza** pode ser utilizado para parar a execução de uma repetição.

### 3.8. Leitura e escrita

A leitura é realizada da seguinte forma:

```
le ( $id$ );
le ( $id$[$exp_arit$] );
```

A função **le** é inteligente o suficiente para ler um único caractere caso \$id\$ seja do tipo **letra**, um único inteiro caso \$id\$ seja do tipo **int**, ou um número com ponto caso \$id\$ seja um do tipo **real**. Pode-se ler uma posição de um vetor com essa mesma função. Para impressão, segue-se o padrão:

```
escreve ( $exp_arit$ );
escreve ( $string$ );
```

A função **escreve** imprime o resultado de uma expressão aritmética. Caso o resultado seja do tipo **letra**, o caractere correspondente na tabela ASCII é impresso. Ela também pode ser utilizada para imprimir constantes do tipo string.

Exemplos:

```
le(a);
le(v[11]);
escreve(a+b);
escreve('a');
escreve("abc");
```

## 4. Protótipos de função

O cabeçalho de todo código é composto por protótipos de função que seguem o seguinte modelo:

```
$tipo$ $id$ ( $lista_vars$ );
```

\$lista\_vars\$ é a lista de parâmetros da função, sendo vários pares de \$tipo\$ e \$id\$ separados por vírgula. O \$id\$ pode ser seguido por colchetes caso o parâmetro seja um vetor. \$tipo\$, além de **int**, **real** e **letra**, também pode ser **nada** no tipo de retorno de funções. Parâmetros são cópias dos valores passados na chamada de função, a menos que sejam vetores. Vetores são passados por referência, logo a função alterará o conteúdo original do vetor.

Exemplos:

```
real sqrt(real n);  
nada ordena(int v[], int n);
```

## 5. Funções

Toda função deve ser a seguinte estrutura:

```
$tipo$ $id$ ( $lista_vars$ ) {  
    * declaração de variáveis aqui *  
  
    * bloco de código aqui *  
}
```

Uma função segue a mesma estrutura do programa principal, com separação de declaração de variáveis e demais comandos. O retorno de uma função é dado pelo comando **tome**:

```
tome $expr_arit$;  
tome;
```

Caso o tipo de retorno seja **nada**, pode-se usar **tome** sem valor para encerrar a execução da função. **tome** também pode ser utilizado no programa principal como mensagem de erro para o sistema operacional.

Uma chamada de função pode ser um comando por si só, onde o retorno é ignorado caso ele não seja **nada**. Os parâmetros de uma chamada de função devem ser consistentes com a declaração da mesma.

Exemplo:

```
ordena(idades,num_individuos);
```