UNIVERSIDADE FEDERAL DE ALAGOAS
Instituto de Computação
Ciência da Computação

Especificação da linguagem C+-

Compiladores

Daniel Humberto Cavalcante Vassalo
Gabriel Barbosa Pereira

2018.1

Sumário

Características Gerais

Estrutura geral

Funções só podem ser declaradas em escopo global. **Variáveis** podem ser declaradas tanto globalmente quanto dentro do escopo de funções. Ao declarar uma variável, se ela não for inicializada ela terá um valor padrão (citado abaixo no tópico **tipos de dados**). Os **identificadores** (tanto de variáveis quanto de funções) só podem conter caracteres alfanuméricos, são sensíveis à caixa, não possuem limitação de tamanho e devem começar obrigatoriamente com uma letra. As instruções devem ser feitas no escopo de funções e devem ser finalizadas com **ponto e vírgula**. Além disso, blocos de instruções são delimitados por **chaves**.

O ponto inicial da execução é a função “main”.

Tipos de dados

A linguagem admite os seguintes tipos de dados:

- inteiro: **int**
  - valor padrão: **0**
- ponto flutuante: **float**
  - valor padrão: **0.00**
  - a representação padrão é feita com 1 casa de unidade e 2 casas decimais, sem nenhum tipo de arredondamento. Caso seja necessária uma precisão maior na representação (o valor sendo muito grande ou muito pequeno), a linguagem irá decidir entre colocar mais casas a direita/esquerda da representação ou utilizar notação científica, levando em conta o critério da “menor representação”
- booleano: **bool**
  - valor padrão: **false**
- caractere: **char**
  - valor padrão: **‘\0’ (caractere nulo)**
- cadeia de caracteres: **string(tamanho)**
  - valor padrão: **“” (string vazia)**
  - O tamanho informado entre parênteses é o tamanho máximo que a string pode alcançar, ou seja, é a quantidade máxima de caracteres que podem estar presentes na string
  - A linguagem disponibiliza a função **length(string)** que informa o tamanho atual da **string**
- arranjos unidimensionais: **tipo[tamanho]**
  - valor padrão: **depende do tipo**

- O tamanho informado entre colchetes é a quantidade máxima de elementos no arranjo
- A linguagem disponibiliza a função **length(arranjo)** que informa o tamanho máximo do **arranjo**
- O índice inicial de todo arranjo é o valor 0 e o índice final é o tamanho máximo menos 1
- O acesso a um elemento do arranjo é feito informando-se o índice entre colchetes (ex.: arranjo[2] acessa o 3º elemento do arranjo)

A especificação de tipos é estática e as variáveis são declaradas da seguinte forma: identificador seguido de “dois pontos”, tipo e “ponto e vírgula”:

```
identificador: tipo;
```

Exemplos:

```
identificador: int;
identificador: float;
identificador: bool;
identificador: char;
identificador: string(tamanho);
```

No caso de arranjos unidimensionais após o tipo deve vir o tamanho do arranjo entre colchetes:

```
identificador: tipo[tamanho];
```

Exemplos:

```
identificador: int[tamanho];
identificador: float[tamanho];
identificador: bool[tamanho];
identificador: char[tamanho];
identificador: string(tamanhoString)[tamanho];
```

E para atribuir um valor a um arranjo unidimensional basta envolver os elementos entre colchetes e separá-los por vírgulas, além disso, a **quantidade de elementos** deve ser **igual** ao **tamanho** especificado na declaração.

Ex.:

```
a: int[4] = [1, 2, 3, 4];
```

ou

```
a: int[4];
a = [1, 2, 3, 4];
```

Tipos de retorno

A linguagem aceita como retorno das funções todos os tipos de dados listados acima, inclusive os arranjos unidimensionais como pode ser visto no Shell Sort implementado no tópico “Programas Exemplo”. Além desses tipos de retorno, a linguagem também aceita o tipo **void** como um tipo válido de retorno, o qual denota a ausência de retorno da função.

Constantes literais

- **Inteiro**
  Valores inteiros que podem ser representados em 32 bits.
  Intervalo: [-2,147,483,647 ; +2,147,483,647]
- **Ponto flutuante**
  Ponto flutuante de precisão simples representado em 32 bits.
  Ex.: 2.34, 6.90, 1.11.
- **Booleano**

Assume apenas os valores **true** e **false**, representados com todas as letras minúsculas.

- **Caractere**
  Caracteres de 8 bits (ASCII), representados entre aspas simples.
  Ex.: ‘a’, ‘b’, ‘c’.
- **Cadeia de caracteres**
  Cadeia de caracteres de 8 bits (ASCII) representados entre aspas duplas.
  Ex.: “abc”, “hello”.
- **Arranjos unidimensionais**

Conjunto de elementos do tipo escolhido, envolvidos entre colchetes e separados por vírgulas.

Ex.: [1, 2, 3, 4], [‘a’, ‘b’, ‘c’], [“hello”, “oie”].

Operações de cada tipo

- **Inteiro e Ponto flutuante**
  Admite operações aritméticas e relacionais.
- **Booleano**

Admite operações relacionais de igualdade e diferença, além das operações lógicas.

- **Caractere e Cadeia de caracteres**
  Admite operações relacionais e a operação de concatenação.
- **Arranjos unidimensionais**
  Admite operações relacionais de igualdade e diferença.

Concatenação

Comentários de múltiplas linhas não são aceitos pela linguagem, apenas comentário de linha única são aceitos e são feitos através do símbolo #.

Ex.:

```
# Variáveis do sistema:
largura: float, comprimento: float, altura: float;
```

Operadores

Aritméticos

Para tipos numéricos.

+	Adição
-	Subtração
*	Multiplicação
/	Divisão
^	Exponencial
-	Unário negativo

Caso um dos operandos seja do tipo **float**, o outro será convertido para **float** caso já não seja.

Relacionais

Para tipos numéricos, caracteres e cadeias de caracteres, igualdade e desigualdade para os tipos booleanos.

>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual

Caso um dos operandos seja do tipo **string** o outro será convertido para **string**, caso já não seja.

Senão caso um dos operandos for do tipo **float** o outro será convertido para **float**, caso já não seja.

Senão caso um dos operandos seja do tipo **int**, o outro será convertido para **int**, caso já não seja.

Igualdade e diferença

==	Igual
!=	Diferente

Não assumem coerção.

Lógicos

Para tipos booleanos.

!	Negação
&&	Conjunção
||	Disjunção

Concatenação

Gera cadeias de caracteres.

+	Concatenação

A operação de concatenação sempre deve possuir um argumento do tipo **char** ou **string** e resultará em uma cadeia de caracteres. Exemplos:

- Concatenação de **string** com **int**:
  “a” + 2 => “a2”
- Concatenação de **bool** com **string** :
  false + “ane” => “falseane”
- Concatenação de **string** com **float**:
  “a” + 2.3 => “a2.30”

Precedência e associatividade

A tabela a seguir descreve a precedência dos operadores (da ordem mais alta - mais acima na tabela, para a ordem mais baixa - mais abaixo na tabela). A ordem de associatividade é descrita na segunda coluna da tabela.

! (negação)	Direita para esquerda
- (unário negativo)	Direita para esquerda
^ (exponencial)	Direita para esquerda
* (Multiplicação) e / (Divisão)	Esquerda para direita
+ (Soma e Concatenação) e - (Subtração)	Esquerda para direita
< <= >= >	Não há associatividade, pois esses operadores não podem aparecer lado a lado, já que eles retornam um valor **bool** e não há coerção para esse tipo de operando
== e != (Diferente de)	Esquerda para direita

&&	Esquerda para direita
||	Esquerda para direita

Uma observação importante é que os parênteses podem ser usados para alterar a precedência dos operadores.

Instruções

Estrutura condicional de uma e duas vias

A sintaxe da estrutura condicional é da seguinte forma:

```
if(condição1) {
     blocoDeInstruções;
} elif(condição2) {
     blocoDeInstruções;
} else {
     blocoDeInstruções;
}
```

Sempre começando com a instrução **if**, seguida de 0 ou mais **elif**s, seguidos de 0 ou 1 **else**. Nas instruções **if** e **elif** as **condições** devem ser estritamente de tipo **bool**, senão ocorrerá um erro de tipo. Caso a **condição** seja igual a **true**, o **blocoDeInstruções** será executado, caso seja **false**, será verificada a condição do **elif** (ou será executado o bloco do **else,** se for o caso).

Estrutura Iterativa com Controle Lógico

A sintaxe da estrutura iterativa com controle lógico é a seguinte:

```
while(condição) {
     blocoDeInstruções;
}
```

O **blocoDeInstruções** será executado sempre que **condição** for **true**, caso seja **false** o laço será interrompido. E caso **condição** não seja do tipo **bool**, ocorrerá um erro de tipo.

Estrutura iterativa controlada por contador com passo igual a um caso omitido

A sintaxe da estrutura iterativa com contador é a seguinte:

```
for(contador = valor; limite; passo) {
      blocoDeInstruções;
}
```

Aqui o "**; passo**" pode ser omitido, e nesse caso o **contador** será incrementado em **1** a cada passo.

O **contador** deve ser obrigatoriamente inicializado com um **valor** do tipo **int**, devendo ser previamente declarado.

O **limite** também deve ser do tipo **int**, sendo este o valor que o **contador** irá assumir na iteração final, pois a **cada passo** será verificado se o **contador** excedeu o valor do **limite**.

Entrada e saída

A entrada de dados é feita pela função **read(stringFormatação, variáveis)**, onde as variáveis a serem lidas devem ser passadas separadas por vírgulas, e na entrada os valores devem ser digitados seguindo a formatação presente na **stringFormatação**.

A **stringFormatação** deve conter uma string informando a formatação e os tipos a serem lidos, e as variáveis devem ser de tipos correspondentes aos que aparecem na string e devem estar na mesma ordem da string. As opções de tipos que podem ser colocadas na stringFormatação são:

- %f, para ponto flutuante
- %i, para inteiro
- %c, para caractere
- %s, para string
- %b, para booleano

Segue um exemplo:

```
var1: int, var2: string(15), var3: bool, var4: char;
read("%i %s %b\n%c", var1, var2, var3, var4);
```

Neste exemplo os valores deveriam ser entrados da seguinte forma:

```
15 "oi tudo bem?" true
'A'
```

A saída de dados é feita pela função **print(stringFormatação, variáveis)**, onde os parâmetros são passados da mesma forma que na função **read**. A **stringFormatação** segue o mesmo padrão mostrado acima.

Segue um exemplo:

```
var1: int = 1, var2: string(2) = "oi";
var3: bool = true, var4: char = 'k';
print("inteiro: %i, cadeia de caracteres: %s\n", var1, var2);
print("booleano: %b, caractere: %c", var3, var4);
```

Neste exemplo, a saída seria mostrada da seguinte maneira:

```
inteiro: 1, cadeia de caracteres: oi
booleano: true, caractere: k
```

Como mostrado no exemplo acima, para que cada chamada da função print possua uma quebra de linha no final do seu conteúdo mostrado, esta quebra deve aparecer na **stringFormatação**.

Para inteiros e strings, caso não seja especificado a quantidade de unidades e caracteres a serem mostrados na função print (colocando um **valor inteiro** depois da **%** e **antes** da **letra que indica o tipo**), a função print tentará mostrar todo o conteúdo da variável; caso seja especificado um tamanho menor do que a representação padrão, os caracteres excedentes serão truncados; e caso o tamanho seja maior do que a representação padrão, os caracteres restantes serão espaços. Abaixo seguem alguns exemplos.

Para os tipos bool e char, a função print sempre mostrará a representação padrão da variável.

Ex.:

```
inteiro: int = 854761, booleano: bool = true;
str: string = “my string”, caractere: char = ‘g’;
print(“%i %s %b %c”, inteiro, str, booleano, caractere);
# 854761 my string true g
print(“%3i %6s %b %c”, inteiro, str, booleano, caractere);
# 854 my str true g
print(“%10i %5s %b %c”, inteiro, str, booleano, caractere);
# 854761     my str true g
```

No caso de um ponto flutuante, o padrão de casas decimais que serão apresentados é 2, porém o usuário pode definir a quantidade de casas decimais que serão apresentadas no print colocando um **.** e o número de casas decimais requeridas, por exemplo:

```
pi: float = 3.14159265359;
print(“O número pi é %f”, pi); # O número pi é 3.14
print(“O número pi é %.4f”, pi); # O número pi é 3.1415
```

A função print irá truncar o restante dos números, apresentando apenas aqueles ou definidos pelo programador ou pelo padrão da linguagem (duas casas decimais).

Atribuição

A atribuição é feita por meio do **comando** representado pelo símbolo **=**.

O tipo da expressão é ditada pela variável alvo não admitindo nenhum tipo de coerção.

Funções

A declaração de funções é da seguinte forma: identificador seguido dos parâmetros entre parênteses, seguido do tipo de retorno e bloco de instruções entre chaves:

```
identificador(parâmetros): tipoDeRetorno {
	blocoDeInstruções;
}
```

Toda função deve ter obrigatoriamente um **return** que retorna um valor/variável do mesmo tipo especificado no cabeçalho da função. Com exceção quando o tipo de retorno da função é **void**, neste caso o **return** não pode estar presente na função.

A linguagem não permite o mascaramento de variáveis dentro dos blocos.

A passagem de parâmetros é feita utilizando o modelo de transferência de dados "**passagem por valor**", onde os parâmetros formais serão inicializados por **cópia**, com exceção no caso do parâmetro real ser um arranjo, daí o valor será passado por **caminho de acesso**.

A declaração dos **parâmetros formais** segue o mesmo padrão de declaração de variáveis:

```
soma(param1: int, param2: float): float {
	return param1 + param2;
}
```

Programas Exemplo

Hello world

```
main(): void {
	print(“Alô mundo!”);
}
```

Fibonacci

```
fibonacci(n: int): void {
	a: int = 0, b: int = 1, i: int = 0;
	limit: bool = n >= 0;

	while(limit) {
		if (i > 0) {
			print(", ");
		}

		if (i <= 1) {
			print("%i", i);
		} else {
			print(“%i”, a + b);
			b = a + b;
			a = b - a;
		}
		i = i + 1;
		limit = (a+b) <= n;
	}
}

main(): void {
	n: int;
	read(“%i”, n);
	fibonacci(n);
}
```

Shell sort

```
shellSort(vet: int[]): int[] {
      i: int, j: int, value: int;
      gap: int = 1;
      while(gap < length(vet)) {
            gap = 3*gap+1;
      }
      while(gap > 1) {
            gap = gap / 3;
            for(i = gap; length(vet) - 1) {
                  value = vet[i];
                  j = i;
                  while(j >= gap && value < vet[j - gap]) {
                        vet[j] = vet [j - gap];
                        j = j - gap;
                  }
                  vet[j] = value;
            }
      }
      return vet;
}

main(): void {
      i: int, aux: int;
      v: int[5];

      for(i = 0; length(v)) {
            read("%i", aux);
            v[i] = aux;
      }

      v = shellSort(v);


      for(i = 0; length(v)) {
            print("%i ", v[i]);
      }
}
```

Especificação dos tokens

Especificação da linguagem de implementação

A linguagem que será utilizada para implementação do analisador léxico e sintático da linguagem C+- será **JAVA**.

Enumeração com as categorias dos tokens

```
public enum TokenCategory {
    main, id, tInt, tFloat, tBool, tChar, tString, scopeBeg, scopeEnd, paramBeg, paramEnd, arrayBeg, arrayEnd, lineEnd, commaSep, consNumInt, consNumFlo, consBool, consChar, consString, rwPrint, rwRead, rwIf, rwElif, rwElse, rwFor, rwWhile, rwReturn, opAssign, opLogic, opNot, opAditiv, opConc, opUnMinus, opMult, opRel, opEquals
}
```

Especificação dos tokens da linguagem

Categoria simbólica/Enum	Expressão regular do lexema
main	‘main’
id	‘[[:alpha:]][[:alnum:]]*’
tInt	‘int’
tFloat	‘float’
tBool	‘bool’
tChar	‘char’
tString	‘string’
scopeBeg	‘{‘
scopeEnd	‘}’
paramBeg	‘(‘
paramEnd	‘)’
arrayBeg	‘[‘
arrayEnd	‘]’
lineEnd	‘;’

commaSep	‘,’
consNumInt	‘[[:digit:]]+’
consNumFlo	‘{consNumInt}\.{consNumInt}([eE][-+]?[[:digit:]]+)?’
consBool	‘true|false’
consChar	‘’[[:alpha:][:digit:][:graph:][:space:]]’’
consString	‘“[[:alpha:][:digit:][:graph:][:space:]]*”’
rwPrint	‘print’
rwRead	‘read’
rwIf	‘if’
rwElif	‘elif’
rwElse	‘else’
rwFor	‘for’
rwWhile	‘while’
rwReturn	‘return’
opAssign	‘=’
opLogic	‘||||&&’
opNot	‘!’
opAditiv	‘\+|-’
opConc	‘+’
opUnMinus	‘-’
opMult	‘*|\/|\^’
opRel	‘<=|>=|<|>’
opEquals	‘==|!=’