Universidade Federal de Alagoas
Instituto de computação

Compiladores
GD
Especificação da Linguagem

Alunos: Davi José de Melo Silva, Gustavo Augusto Calazans Lopes
Professor: Alcino Dall'Igna Jr.

2020.2

1 Introdução **4**

2 Estrutura geral do programa **4**

2.1 Ponto inicial do execução **4**
2.2 Definições de funções **5**

3 Conjunto de tipos de dados e nomes **5**

3.1 Identificadores **6**
3.2 Palavras reservadas **6**
3.3 Coerção **6**
3.4 Inteiro **7**
3.5 Ponto Flutuante **7**
3.6 Caracteres e cadeias de caracteres **8**
3.7 Booleanos **9**
3.8 Arranjos Unidimensionais **9**
3.9 Valores padrão **10**

4 Conjunto de operadores **10**

4.1 Aritméticos **10**
4.2 Relacionais **10**
4.3 Lógicos **11**
4.4 Concatenação de cadeias de caracteres **11**
4.5 Precedência e associatividade **12**

5 Instruções **12**

5.1 Atribuição **12**
5.2 Estruturas condicionais **13**
5.3 Estrutura de repetição por controle lógico **14**
5.4 Estrutura de repetição por controle de contador **14**
5.5 Entrada e saída **15**
5.6 Funções **16**

6 Programas de exemplos **17**

6.1 Alô mundo! **17**
6.2 Fibonacci **17**
6.3 Shell sort **18**

1 Introdução

A linguagem GD foi criada baseada nas linguagens C, Python e JavaScript. Não é uma linguagem orientada a objetos, é case-sensitive, possui palavras reservadas e possui suporte a coerção. Pode ser usada para criação de algoritmos simples. Possui estrutura de dados simples como o arranjo unidimensional. Possui tipagem estática, estruturas condicionais e de repetições.

2 Estrutura geral do programa

Um programa feito em GD deve possuir:

- Todas as funções possuem escopos delimitados por chaves de abertura e fechamento, possuindo um tipo de retorno.
- Deve possuir uma função main com tipo e retorno obrigatório sendo um inteiro, para o retorno deve ser 0 pois só assim o sistema operacional irá reconhecer que o programa executou com sucesso.
- Todas as linhas devem terminar com ponto e vírgula (“;”).
- Em qualquer bloco o escopo vai ser delimitado por abertura e fechamento de chaves.

2.1 Ponto inicial do execução

O ponto inicial do programa é identificado através da criação de uma função principal chamada **main** que possui como tipo de retorno obrigatório sendo inteiro e retorno igual a 0 (zero) através da palavra reservada **return**. O escopo de uma função é determinado por chaves de abertura e de fechamento e o uso de parênteses para abertura e fechamento de parâmetros de função. No final de cada linha deve possuir o caractere “;” . Segue um exemplo:

```
int function main() {
        ...
        return 0;
}
```

2.2 Definições de funções

Para criação de novas funções, deve-se utilizar a palavra reservada **function** sendo que antes dela deve possuir o tipo de retorno da função podendo ser **int, float, bool (ou um array de qualquer tipo citado anteriormente), string, char, void,** sendo void utilizado para quando não houver nenhum tipo de retorno, em seguida o nome da função que é um identificador único seguido de uma lista de parâmetros delimitados por parênteses. A lista de parâmetros deve ser composta pelo tipo seguido do identificador. Cada parâmetro é separado por vírgulas (“,”), e por fim, para definir o escopo dessa função utiliza-se abertura e fechamento de chaves. Exemplos:

- Exemplo 1:

<tipo de retorno> function <identificador da função>(<tipo do parâmetro 1> <identificador 1>, ...) {}

```
void function funcao1() {
    ...
}
```

- Exemplo 2 (retornando array):

<tipo de retorno> [] function <identificador da função>(<tipo do parâmetro 1> <identificador 1>, ...) {}

```
int [] function funcao2() {
    ...
}
```

- Exemplo 3 (array como parâmetro da função):

<tipo de retorno> function <identificador da função>(<tipo do parâmetro 1> <identificador 1> , <tipo do parâmetro 2> <identificador 2> [<tamanho do vetor>] , ...) {}

Obs: no exemplo acima o array foi colocado como segundo parâmetro porquê a variável que possui o seu tamanho precisa estar declarada num parâmetro anterior ao array.

```
int function funcao2(int size, int array[size]) {
    ...
}
```

3 Conjunto de tipos de dados e nomes

3.1 Identificadores

Os identificadores possuem algumas regras. De modo geral são case-sensitive e não devem possuir certos caracteres especiais.

- Podem começar somente com letras maiúsculas ou minúsculas;
- Não podem começar com números ou qualquer caractere especial;
- O único caractere especial permitido é o underline (“_”) ;
- Não podem ser palavras reservadas;
- Não é permitido o uso de espaço;

3.2 Palavras reservadas

As palavras reservadas são:
main, if, elif, else, for, while, function, int, float, bool, string, char, print, input, False, True.

3.3 Coerção

É possível realizar coerções em certos casos. Ao fazer uma operação com um número inteiro e ponto flutuante o resultado será um número de ponto flutuante. Ao concatenar algum número, caractere ou booleano a uma cadeia de caracteres o resultado será uma string. É possível concatenar 2 ou mais elementos de uma vez. É possível também concatenar strings diretamente, sem que estejam atribuídas à alguma variável.
Supondo que temos quatro variáveis, uma string, uma booleana, uma ponto flutuante e outra inteira, de nomes string1, booleano1, float1 e int1, com seus respectivos valores “Teste”, **True**, 23.578, 17. Se um dos valores envolvidos for

string, deverá ser usado ‘.’ para realizar a operação, caso sejam só números usar ‘+’. Exemplos:

- Exemplo 1:
  int1 = int1 + float1;
  O novo valor de int1 será 40.578.

- Exemplo 2:
  string1 = string1 . booleano1;
  O novo valor de string1 será “TesteTrue”.

- Exemplo 3:
  string1 = float1 . string1 . booleano1;
  O novo valor de string1 será “23.578TesteTrue”.

- Exemplo 4:
  string1 = float1 . “ “ . string1 . “ OK “ . booleano1;
  O novo valor de string1 será “23.578 Teste OK True”.

Independente do tipo a ser concatenado a uma string, o valor literal desse tipo será transformado em string.

3.4 Inteiro

int identifica a variável como um número inteiro de 32 bits, sendo seu valor literal uma cadeia de números inteiros entre 0 e 9.
Ex de declaração de um inteiro:

```
int  inteiro = 10;
```

Caso não seja inicializado com algum valor, seu valor padrão será 0.

3.5 Ponto Flutuante

float identifica variáveis como um número com ponto flutuante de 64 bits, sendo seu valor literal uma sequência de dígitos entre 0 e 9 seguido de um

ponto (“.”) e demais dígitos (com precisão de 8 dígitos). Ao estourar esse limite é desconsiderado a parte que passou do limite.
Ex de declaração de um float:

```
float pontoFlutuante = 15.47;
```

Caso não seja inicializado com algum valor, seu valor padrão será 0.0

3.6 Caracteres e cadeias de caracteres

Para armazenar apenas um caractere utilizamos o tipo de variável char podendo salvar números, letras maiúsculas e minúsculas ou caracteres especiais. Utiliza-se a palavra reservada **char** seguida do identificador. Seu valor é delimitado por apóstrofos. A seguir temos um exemplo de declaração com inicialização de valor seguido de um que não é inicializado e depois é atribuído um valor a essa variável.
Ex:

```
char caractere = ‘a’;
```

Nesse caso temos uma variável chamada “caractere” que ao ser declarada já foi atribuída o valor “a” a ela.

Se a variável não for inicializada com algum valor, seu valor padrão será vazio. É possível inicializá-la com o valor vazio também.

Ex2:

```
char caractere;
caractere = ‘a’;
```

Ex3: Inicializando com valor vazio

```
char caractere = ‘’;
```

Já uma cadeia de caracteres aceita os mesmos valores que um char pode possuir, porém seu valor é delimitado por aspas e identificados através da palavra reservada **string**. Da mesma forma é possível inicializar a variável já com um valor ou com o valor vazio. Seu valor padrão ao não ser atribuído nenhum valor será vazio.
Ex:

```
string string1 = “Esta é uma sequência de caracteres.”
```

3.7 Booleanos

Booleanos são identificados através da palavra reservada **bool** e possuem somente dois resultados possíveis, sendo eles através das palavras reservadas **True** e **False**. Caso essa variável não seja inicializada com algum valor, seu valor padrão será **False**. A seguir é possível ver um exemplo de declaração de variável com inicialização e um exemplo só de declaração.
Ex:

```
bool teste = True;
```

Ex2:

```
bool valor_booleano;
```

3.8 Arranjos Unidimensionais

Um Arranjo Unidimensional ou Array é declarado como na linguagem C, onde o tamanho do arranjo é gerenciado pelo programador e não pode ser mudado durante a execução do programa. Definimos o seu formato como sendo:
<tipo> identificador[Tamanho]
Ex:

```
int array1[9];
```

3.9 Valores padrão

TIPO	VALOR PADRÃO
int	0
float	0.0
char	‘’ (vazio)
string	“” (vazio)
bool	False

4 Conjunto de operadores

4.1 Aritméticos

- “+” Soma de dois operandos
- “-” Subtração de dois operandos
- “-” Unário Negativo
- “/” Divisão de dois operandos
- “*” Multiplicação de dois operandos
- “%” Retorna um número inteiro restante da divisão dos dois operandos.

Operações de divisão por zero como na maioria das linguagens será retornado um NaN (Not a Number).

4.2 Relacionais

- “==”: Operador que irá retornar um resultado verdadeiro ao verificar dois operandos, um a esquerda do operador e o outro a direita, caso sejam iguais.
- “!=”: Operador que irá retornar um resultado verdadeiro ao verificar dois operandos, um a esquerda do operador e o outro a direita, caso sejam diferentes.
- “<”: Operador “menor que” que irá retornar um booleano de valor True ao verificar dois operandos, um a esquerda do operador e o outro a

direita, caso o da esquerda seja menor que o da direita, caso contrário retornará False.

- “>”: Operador “maior que” que irá retornar um booleano de valor True ao verificar dois operandos, um a esquerda do operador e o outro a direita, caso o da esquerda seja maior que o da direita, caso contrário retornará False.
- “<=”: Operador “menor ou igual que” que irá retornar um booleano de valor True ao verificar dois operandos, um a esquerda do operador e o outro a direita, caso o da esquerda seja menor ou igual que o da direita, caso contrário retornará False.
- “>=”: Operador “maior ou igual que” que irá retornar um booleano de valor True ao verificar dois operandos, um a esquerda do operador e o outro a direita, caso o da esquerda seja maior ou igual que o da direita, caso contrário retornará False.

4.3 Lógicos

- “!”: Operador “negação”, é um operador unário que retorna a negação de seu operando. Fica a esquerda do operando.
- “&&” Operador "conjunção" (E lógico), é um operador que irá retornar um valor booleano, caso ambos operandos sejam verdadeiros retorna True, caso contrário, retorna False.
- “||” Operador “disjunção” (OU lógico), é um operador que irá retornar um valor booleano, caso um dos operandos for verdadeiro será retornado True, e se ambos forem falsos, retorna False.

4.4 Concatenação de cadeias de caracteres

Para concatenar duas cadeias de caracteres utiliza-se **“.”,** a concatenação de dois valores string, retorna outra string que é a união dos dois operandos. Sendo sua associatividade da esquerda para a direita. Supondo que tenhamos duas variáveis, string1 e string2 com respectivamente os valores, “Testando ” e “concatenação.” que será atribuída a uma nova variável chamada string3, o resultado da concatenação dessas duas variáveis resultaria em:

string string1 = “Testando ”;

string string2 = “concatenação”;
string string3 = string1 . string2;

Valor de string3: “Testando concatenação”.

4.5 Precedência e associatividade

Quando menor o valor, menor a precedência.

Operação	Precedência/Associatividade
“-” (unário negativo)	1/Da direita para esquerda
“*” “/” (multiplicação e divisão)	2/Da esquerda para direita
“+” “-” (soma e subtração)	3/Da esquerda para direita
“==” e “!=”	4/Da direita para esquerda
“<”, “<=”, “>” e “>=” (comparação)	5/Da esquerda para direita
“!” (negação)	6/Da esquerda para direita
“&&” “||”	7/Da esquerda para direita
“.” (concatenação)	8/Da esquerda para direita

Obs: Parênteses terão precedência sobre todos os operadores. No caso de haver parênteses aninhados, o parênteses mais interno terá maior precedência, e assim consecutivamente.

5 Instruções

Cada bloco de instruções na linguagem seu escopo são delimitados por chaves de abertura e fechamento **“{}”** e cada linha é finalizada com o caractere **“;”**.

5.1 Atribuição

Uma atribuição é feita utilizando o caractere **“=”** sendo que do lado esquerdo deve possuir o nome do identificador da variável e após a igualdade valor a ser atribuído, que deve ser do mesmo tipo do identificador. Exemplo de uma atribuição:

```
identificador = valor;
```

5.2 Estruturas condicionais

Em GD usamos a palavra reservada **if** para executar uma ação caso uma condição lógica seja verdadeira, podemos usar a cláusula **elif** para obter várias condições que serão testadas em sequência. Ainda possui uma cláusula **else** que será executada caso todas as condições anteriores sejam falsas, onde condição pode ser qualquer expressão que seja avaliada como verdadeira ou falsa. Para cada declaração de **if** podemos ter várias cláusulas **elif** em sequência e apenas uma cláusula **else** que deverá ser a última. A ordem de avaliação é em sequência. As condicionais só serão avaliadas até uma ser verdadeira, a partir dela nenhuma mais será avaliada. Caso nenhum os if e elif em sequência forem verdadeiros, o else ao final será executado, caso exista. A seguir temos alguns exemplos:

Ex1: Estrutura **if** com apenas uma via

```
if(<condição>) {
        ...
}
```

Ex2: Estrutura **if** seguido de um **elif**

```
if(<condição>) {
        ...
} elif(<condição>) {
        ...
}
```

Ex3: Estrutura **if** seguido de um **else**

```
if(<condição>) {
        ...
} else {
        ...
}
```

Ex4: Estruturas combinadas

```
if(<condição>) {
        ...
} elif(<condição>) {
        ...
} else {
        ...
}
```

Obs: No Ex4 poderia ter vários elif's entre o primeiro elif e o else.

5.3 Estrutura de repetição por controle lógico

A repetição ocorre enquanto a condição for verdadeira, quando for falsa o loop irá parar sua execução. O teste da condição ocorre antes que o laço seja executado. Desta forma se a condição for verdadeira o laço executará e testará a condição novamente. Declaramos usando a palavra reservada **while** seguida de parênteses com a condição, onde condição pode ser qualquer expressão que seja avaliada como verdadeira ou falsa. O escopo deve ser delimitado por chaves de abertura e fechamento. Exemplo:

```
while(<condição>) {
        ....
}
```

5.4 Estrutura de repetição por controle de contador

Para utilizar a estrutura de repetição por contador utiliza-se o comando **for** seguido do identificador que irá fazer o controle. Passamos 3 informações por parâmetro, sendo eles o valor inicial que o loop irá começar, valor final para o loop parar de executar e por fim o valor de seu incremento. Seu escopo

também é delimitado por abertura e fechamento de chaves, a seguir temos uma demonstração de declaração:

for identificador = (valor inicial do identificador, valor final do identificador, incremento) {

...

}

Quando o identificador atingir o valor final especificado, o loop irá parar.

5.5 Entrada e saída

As funções de entrada e saída são **input** para entrada e **print** para saída.

Para exibir uma informação na tela, utilizamos o comando **print**. Em GD podemos passar uma saída formatada, sempre será retornada uma string. Para isso deve incluir especificadores de formato (começando com: %), e em seguida os argumentos separados por vírgulas, para assim então ser formatado e inserido na string resultante substituindo seus respectivos especificadores.

Para a função de leitura utilizamos o comando **input,** devemos declarar as variáveis que forem usadas para receber os valores previamente. Podemos receber mais de um valor, para isso separamos os parâmetros da função por “,”. Os valores são identificados através do espaço entre uma e outra. Se a quantidade de variáveis for diferente da quantidade passada para receber, o programa irá desconsiderar. Os parâmetros são passados entre parênteses.

A seguir podemos ver exemplos de leitura e escrita:

Entrada:

- Exemplo 1:
  int variavel1;
  input(variavel1);

- Exemplo 2:

int variavel1;
int variavel2;
input(variavel1, variavel2);

Obs: se houver o caso de executar a função input() no fim do arquivo, ela receberá EOF.

Saída:

- Exemplo 1:
  int parametro1 = 1;
  float parametro2 = 12.766;
  print(“Isto é um print com parâmetro número %d, e um float com 2 casas decimais de precisão %2f”, parametro1, parametro2);

- Exemplo 2:
  print(“Este é um print simples”);

A formatação deve seguir as seguintes regras:

Tipo	Formatação	Formatação de Tamanho	*
int	%d	%*d	Quantidade de dígitos desejada na saída
float	%f	%*f	Quantidade de casas decimais
char	%c	-	-
string	%s	-	-
boolean	%b	-	-

Obs: Para imprimir um ‘%’ na saída é preciso colocar %% dentro do print();

5.6 Funções

As funções funcionam baseada na linguagem C, onde primeiro definimos o tipo de retorno, seguido da palavra reservada **function** seguido de:

- Nome da Função.
- Lista de argumentos para a função, entre parênteses e separados por vírgulas. Deve possuir o tipo esperado do argumento a que será recebido seguido do nome dele.
- Escopo delimitado entre chaves { }.

Para retornar algum valor utiliza-se a palavra reservada **return** que deve retornar um valor do mesmo tipo que foi usado na assinatura da função. Caso essa função não tenha nenhum tipo de retorno, deve usar a palavra reservada **void**. A seguir temos um exemplo de como funciona a assinatura de uma função.

<tipo> **function** <nome da função> (<tipo> parametro1, <tipo> parametro2, ...)
{
    ....
    return <valor>;
}

6 Programas de exemplos

6.1 Alô mundo!

```
int function main() {
   print("Hello World");
   return 0;
}
```

6.2 Fibonacci

```
void function fibonacci(int num) {
  if(num == 0) {
    return;
```

```
  } elif(num == 1) {
    print("0");
    return;
  } elif(num == 2) {
    print("0, 1, 1");
    return;
  }

  int prev2number = 1;
  int prev1number = 1;
  int numberToPrint = prev2number + prev1number;

  while(numberToPrint < num) {
    print(",%d", numberToPrint);
    prev2number = prev1number;
    prev1number = numberToPrint;
    numberToPrint = prev1number + prev2number;
  }
}

int function main() {
  int num;
  input(num);
  fibonacci(num);
  return 0;
}
```

6.3 Shell sort

```
int [] function shellSort(int size, int vetor[size]) {
    int i;
    int j;
    int value;
    int h = 1;
    while(h < size) {
        for i = (h,  size; i = i + 1) {
```

```
            value = vetor[i];
            j = i;
            while(j > h - 1 && value <= vetor[j - h]) {
                vetor[j] = vetor [j - h];
                j = j - h;
            }
            vetor[j] = value;
        }
        h = h / 3;
    }
    return vetor;
}

int function main() {
  int vetor[1000];
  int value;
  int i = 0;
  while(input(value) != EOF) {
      vetor[i] = value;
      i = i + 1;
  }
  int size = i;
  vetor = shellSort(size, vetor);
  for i = (0, size - 1, i = i + 1) {
      print("%d ", vetor[i]);
  }
  print("%d", vetor[size - 1]);
  return 0;
}
```