Universidade Federal de Alagoas - UFAL
Compiladores 2019.2

LM Language

Professor: Alcino.
Alunos: Manoela, Luan.

06 de Dezembro de 2019, Maceió - AL

Sumário

1. Estrutura do código

O programa deve conter sua função principal, denominada como main, palavra-reservada. A função main do programa é responsável por dar início a sua execução. O void na função main sempre retorna 0, caso não consiga executar e 1 caso contrário.

Escopo

O escopo é delimitado com os símbolos “{“, para iniciar, e “}”, para encerrar o escopo.

Comentários

A linguagem não admite comentários.

Término de uma instrução

O término de uma instrução é indicada com o símbolo “;”.

Abaixo temos um exemplo da estrutura:

```
<declaração de funções antes da main>
main() void {
         <declaração de variáveis>
         <instruções>;
};
```

1.1. Nomes

1.1.1. Palavras reservadas

Palavra-chave	**Ação**
void	Omissão de tipo
int	Definição do tipo inteiro
float	Definição do tipo ponto flutuante
char	Definição do topo caractere
string	Definição do tipo string

bool	Definição do tipo lógico
true	Valor lógico “verdadeiro”
false	Valor lógico “falso”
null	Definição de valor inicial nulo
if	Definição de estrutura condicional
elif	Definição de estrutura condicional mais abrangente
else	Definição de estrutura condicional com segunda via
when	Definição de estrutura de repetição com controle lógico
repeater	Definição de estrutura de repetição controlada por contador
return	Retorno de uma função
main	Definição da função principal do programa
read	Instrução para receber dados de entrada
print	Instrução para imprimir informações de saída

1.2. Tipos de dados

Na linguagem, a vinculação de tipo é estática.

- **String:** cadeia de caracteres definida pela palavra-chave string;
- **Caractere:** definido pela palavra-chave char, de acordo com a tabela ASCII;
- **Ponto flutuante:** Segue o padrão IEEE 754, e é definido pela palavra-chave float;
- **Inteiro:** tipo de 32 bits, definido pela palavra-chave int;
- **Booleano:** definido pela palavra-chave bool, assumindo valores true ou false;
- **Array unidimensional:** definido pelo tipo seguido do nome da variável.

Os tamanhos dos arrays são especificados dentro de colchetes. O valor do tamanho, dentro dos colchetes, é positivo ou omitido, para definir um array sem um tamanho fixo.

1.2.1. Constantes literais

- **Inteiro**: um literal inteiro aceita dígitos de 0 a 9

- **Ponto flutuante:** as condições de literais inteiros se aplicam, devendo indicar as casas decimais após os dígitos, utilizando o ponto (.). A notação científica, possui um número de ponto flutuante, seguido do caractere maiúsculo E, seguido de um número inteiro representando o expoente;
- **Caractere:** um caractere literal é válido entre aspas simples;
- **String:** uma string literal pode ter qualquer símbolo dentro de aspas duplas;
- **Booleano:** um literal booleano assume os valores true ou false;
- **Array**: num array literal seus valores devem ser colocados dentro de colchetes e devem seguir, unicamente, o tipo definido no array. Caso especificado o tamanho, deve ser inicializado com a quantidade de dados equivalente.
- Nos casos onde queira inicializar um tipo com valor nulo, usasse a constante **null**.

1.2.2. Valores padrão

- **Inteiro:** inicializado com valor 0 (zero);
- **Ponto flutuante:** inicializado com valor 0.0 (zero);
- **Caractere:** inicializado com o valor ‘ (vazio no ASCII);
- **String:** inicializado com o valor “” (palavra vazia);
- **Booleano:** inicializado com o valor false (falso);
- **Array:** inicializado com o valor padrão do tipo usado.

1.2.3. Operações com Tipos

- **Inteiro:** atribuição, aritmética, relacional;
- **Ponto flutuante:** atribuição, aritmética, relacional;
- **Caractere:** atribuição, concatenação, relacional;
- **String:** atribuição, concatenação, relacional;
- **Booleano:** atribuição, lógico, relacional de igualdade;
- **Array:** atribuição, concatenação de mesmo tipo.

1.2.4. Coerção

A linguagem não admite coerção.

1.3. Conjunto de Operadores

1.3.1. Aritmético

- + : operador de soma;
- - : operador de subtração;
- * : operador de multiplicação;
- / : operador de divisão;
- ^ : operador de exponenciação;
- - : operador unário.

1.3.2. Relacional

- > : maior que;
- < : menor que;

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

1.3.3. Lógico

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

1.3.4. Concatenação

- ++ : concatenação entre operandos.

Na tabela abaixo, temos a ordem de precedência (decrescente) e associatividade dos operadores acima. Se uma expressão estiver entre parênteses, esta será avaliada primeiro.

Operador	**Associatividade**
^	à direita
-(unário negativo)	à direita
* /	à esquerda
+ -	à esquerda
++	à esquerda
< > <= >=	não associativo
== !=	não associativo
and	à esquerda
or	à esquerda

1.4. Escopo

- Na linguagem, o escopo é léxico, o que significa que tudo que for definido dentro de um escopo ficará visível para os escopos que está contido. O tempo de vida de uma variável definida será até o fim do escopo que a definiu.

1.5. Ambiente de referenciamento

- Como o escopo é léxico, todas as variáveis definidas em um escopo, estão disponíveis em todos os outros escopos que contêm.

2. Instruções

2.1. Atribuição

- As atribuições são feitas através de um comando utilizando o operador =.
- Do lado esquerdo fica a variável alvo e do lado direito o valor a ser atribuído a ela.
- As atribuições só poderão ser feitas para tipos iguais, caso contrário ocorrerá um erro de compilação. Exemplo:

```
float atr = 1.5;
atr = "teste";
```

2.2. Instruções condicionais e iterativas

Nessas instruções, as condições avaliadas sempre terão um resultado booleano, indicando qual direcionamento o programa deve ter.

2.2.1. Condicional de uma via

- Para realizar uma instrução condicional de uma via a linguagem usa a palavra-chave **if** seguida de parênteses, com a condição desejada dentro dos parênteses.
- Pode haver mais de uma condição envolvida, para isso é necessário o uso dos operadores lógicos para combinar as expressões. De forma geral, a instrução condicional de uma via tem a seguinte forma:

`if (<condição>){`
`  <instruções>;`
`};`
	`if (valor > 10) {`
	`  print("valor maior que 10");`
	`};`

2.2.2. Condicional de duas vias

- Seguindo a mesma lógica do tópico anterior, podemos usar instrução condicional de duas vias utilizando a palavra-chave **if** com a palavra-chave **else** em seguida. De forma geral, teremos uma instrução condicional de duas vias tem a seguinte forma:

`if (<condição>) {`
`  <instruções>;`
`} else {`
	`if (valor >= 10) {`
	`  print("valor maior ou igual a 10");`
	`} else {`

`    <instruções>;` `};`	`    print("valor menor que 10");` `};`

2.2.3. Condicional generalizado

- Caso a verificação precise ser mais abrangente ou exija várias validações, podemos usar a palavra-chave **elif** , e aninhar as validações a serem feitas. De forma geral, teremos uma instrução condicional mais abrangente da seguinte forma

`if (<condição>) {`
`    <instruções>;`
`} elif (<condição>) {`
`    <instruções>;`
`} else {`
`    <instruções>;`
`};`
	`int valor = 8;`
	`if (valor == 10) {`
	`    print("valor é igual a 10");`
	`} elif (valor > 10) {`
	`    print("valor maior que 10");`
	`} else {`
	`    print("valor é menor que 10");`
	`};`

2.2.4. Instrução iterativa com controle lógico

- Para instrução iterativa com controle lógico usamos a palavra chave when, que recebe uma expressão lógica e a executa até ela tornar-se falsa. De forma geral, uma instrução iterativa com controle lógico tem a seguinte forma:

`when (<condição>) {`
`    <instruções>;`
`};`
	`int limite = 98;`
	`when (limite < 100) {`
	`  print("limite menor que 100");`
	`  limite = limite + 1;`
	`};`

2.2.5. Instrução iterativa controlada por contador

- Para a instrução iterativa controlada por contador utilizamos a palavra-chave **repeater** seguida de parênteses que contém o contador, o limite do contador, e o passo. O incremento deve ser especificado. De forma geral, uma instrução iterativa controlada por contador tem a seguinte forma:

```
repeater (<contador>; <limite>; <incremento>) {
    <instruções>;
```

```
};
```

```
int contador;
repeater (contador = 0; 100; 1){
     print("contador menor que 100");
};
```

2.3. Formas de Controle

- Não há operadores como break e continue como na linguagem C, para controlar a estrutura de repetição.

3. Entrada

Para entrada de dados utilizamos a palavra-chave read. Uma instrução de entrada de dados tem a seguinte forma:

```
read(<variáveis>);
```

```
int x;
float y;
read(x, y);
```

O último valor lido pela **read** fica disponível em uma função chamada lastValueRead().

4. Saída

- Para saída de dados utilizamos a palavra-chave **print** .
- A instrução de saída deve conter como parâmetro uma string literal ou uma variável, ou ainda conter uma variável e um string literal.
- Para exibir um valor de uma ou mais variáveis com uma string literal pode-se formatar a saída especificando o tipo de dado esperado na string literal. Usamos **%d** para **int** , **%f** para **float** , **%c** para **char** e **%s** para **string** .
- No caso do tipo **float** , pode-se dizer quantas casa decimais após o ponto devem ser exibidas, usando **%._f**, onde em _ deve-se colocar a quantidade desejada. Também há a opção de usar o operador de concatenação no caso de char e string . Abaixo temos alguns exemplos:

```
print(<variáveis> ou <string
literal>);
```

```
int v1 = 1;
float v2 = 2.00000;
string s1 = 'e';
string s2 = " um exemplo!";
print(v1, v2, s1, s2);
print("%d %.2f %c %s",v1, v2, s1,
```

	s2) ; print ("aqui vai ter um inteiro %d", v1)

5. Funções

- Na linguagem toda função deve ser declarada antes de ser usada, essa declaração tem que ser feita e implementada antes da função principal main.
- Não é possível criar funções aninhadas.
- Entrada da função os parâmetros são opcionais e são passados por referência.
- Saída da função pode ter um valor de retorno ou não, quando uma função não tem tipo de retorno definido então é assumido que não há retorno na função, não sendo necessário usar a palavra-chave **return** no fim da função. De forma geral, a definição de uma função tem o seguinte formato:

```
<nome_da_funcao> (<parâmetros (opcionais)>) <tipo_do_retorno> {
     <instrucoes_da_funcao>;
     return <tipo_do_retorno>;
};
```

5.1. Alo Mundo

```
main() void {
      print("Alo mundo");
};
```

5.2. Fibonacci

```
fibonacci(int limit) void {
    int count;
    int fib1;
    int fib2;
    int result;

    fib1 = 1;
    fib2 = 1;
```

```
    if(limit == 0) {
        print("0");
    }else{
        when(result <= limit) {
             if(count == 0){
                  print("0");
             }elif(count < 2) {
                  print(",1");
                  result = 1;
             }else {
                  result = fib1 + fib2;
                  fib1 = fib2;
                  fib2 = result;
                  if(result<=limit){
                      print("," ++ result);
                  };
             };
             count = count + 1;
         };
    };
};

main() void {
    int limit;
    read(limit);
    fibonacci(limit);
};
```

5.3. Shell sort

```
shellSort(int values[10], int size) int[] {
      int h;
      h = 1;

      if(h < size) {
           h = h * 3 + 1;
      };

      h = h / 3;
      int c;
      int j;
```

```
      if(h > 0) {
            int i;
            i = h;
            repeater(i; size; 1) {
                  c = values[i];
                  j = i;
                  if(j >= h and values[j - h] > c) {
                        values[j] = values[j - h];
                        j = j - h;
                  };
                  values[j] = c;
            };
            h = h / 2;
      };
      return values;
};

main() void {
      int i;
      int unsortedValues[10];
      i = 0;
      repeater(i; 10; 1) {
            unsortedValues[i] = lastValueRead();
      };
      i = 0;
      repeater(i; 10; 1) {
            print(unsortedValues[i]);
            print("\n");
      };
      int sortedValues[10] = shellsort(unsortedValues, 10);
      i = 0;
      repeater(i; 10; 1) {
            print(sortedValues[i]);
            print("\n");
      };
};
```

Especificações dos tokens

A linguagem de programação escolhida para a implementação do analisador léxico e sintático é JAVA.

1. Categoria dos tokens

```
public enum TokensCategories {

MAIN, ID, VOID, INT, BOOL, CHAR, STRING, FLOAT, OK, CK, OP, CP,
OB, CB, SCO, SPTR, CTEINT, CTEFLOAT, CTEBOOL, CTECHAR, CTESTRING,
IF, ELSE, ELIF, REPEATER, WHEN, PRINT, READ, RETURN, AND, OR, NOT,
OPA, OPM, OPE, OPU, ORC, ORE, ATR, CONCAT, NULL;
}
```

2. Descrição dos tokens

Token	Descrição	Token	Descrição
ID	token para um identificador;	**SPTR**	token para um separador;
VOID	token para o tipo vazio;	**NULL**	token para definição de valor inicial nulo;
INT	token para o tipo int;	**SCO**	token para ponto e vírgula;
FLOAT	token para o tipo float;	**OK**	token para definição de início de escopo;
CHAR	token para o tipo char;	**CK**	token para definição de fim de escopo;
STRING	token para o tipo string;	**OP**	token para abertura de parênteses;
BOOL	token para o tipo booleano;	**CP**	token para fechamento de parênteses;
RETURN	token para a um retorno de uma função;	**OB**	token para abertura de colchetes;
MAIN	token para a definição da	**CB**	token para fechamento de

	função principal;		colchetes;
READ	token para comando de leitura;	**OPA**	token para operador aritmético;
PRINT	token para comando de exibição de dados;	**OPM**	token para operador multiplicativo;
IF	token para uma estrutura condicional de uma via;	**OPE**	token para o operador exponencial;
ELIF	token para uma estrutura condicional mais abrangente;	**OPU**	token para um operador unário;
ELSE	token para uma estrutura condicional de duas vias;	**ATR**	token para uma atribuição;
WHEN	token para uma estrutura de repetição com controle lógico	**ORC**	token para um operador relacional comparativo;
REPEATER	token para uma estrutura de repetição controlada por contador;	**ORE**	token para um operador relacional de igualdade;
CTEINT	token para uma constante inteira;	**AND**	token para o operador lógico AND;
CTEFLOAT	token para uma constante float;	**OR**	token para o operador lógico OR;
CTEBOOL	Token para uma constante booleana;	**NOT**	token para o operador lógico NOT;
CTECHAR	token para uma constante caractere;	**CONCAT**	token para o operador de concatenação;

CTESTRING	token para uma constante cadeia de caractere;		

3. Expressões regulares auxiliar

digit = [0-9];
letter = [a-z]|[A-Z];
symbol = ‘ ’ | ‘\.’ | ‘\,’ | ‘\:’ | ‘\;’ | ‘\!’ | ‘\?’ | ‘\+‘ | ‘\-’ | ‘*‘ | ‘\/’ | ‘\\’ | ‘_’ | ‘\<‘ | ‘\>’ | ‘\=’ | ‘\(‘ | ‘\)’ | ‘\[‘ | ‘\]’ | ‘\{‘ | ‘\}’ | ‘\’’ | ‘\“‘ | ‘\@’ | ‘\%’ | ‘\&’ | ‘\$’ | ‘\^’ | ‘\\|’;

4. Expressões para os lexemas

ID	(‘letter’)((‘letter’|’digit’)*)
VOID	‘void’
INT	‘int’
FLOAT	‘float’
CHAR	‘char’
STRING	‘string’
BOOL	‘bool’
RETURN	‘return’
MAIN	‘main’
READ	‘read’
PRINT	‘print’
IF	‘if’
ELIF	‘elif’
ELSE	‘else’
WHEN	‘when’
REPEATER	‘repeater’

CTEINT	((‘digit’)+)
CTEFLOAT	((‘digit’)+)(’\.’)((‘digit’)+)
CTEBOOL	(‘true’ | ‘false’)
CTECHAR	(’\’’)((‘letter’ ’digit’ ’symbol’)?)(’\’’)
CTESTRING	(‘\”’)(((‘letter’ ‘digit’ ‘symbol’)?)*)(‘\”’)
SPTR	‘\,’
SCO	‘\;’
OK	‘\{’
CK	‘\}’
OP	‘\(’
CP	‘\)’
OB	‘\[‘
CB	‘\]’
OPA	(‘\+’ | \‘-’)
OPM	(‘*’ | ‘\/’)
OPE	‘\^’
OPU	‘\-’
ATR	‘\=’
ORC	(‘\<’ | ‘\>’ | ‘\<=’ | ‘\>=’)
ORE	(‘\==’ | ‘\!=’)
AND	‘and’
OR	‘or’
NOT	‘not’
CONCAT	‘\++’
NULL	‘null’