UNIVERSIDADE FEDERAL DE ALAGOAS – UFAL
CAMPUS A. C. SIMÕES

CIÊNCIA DA COMPUTAÇÃO – COMP263

MAXWELL ESDRA ACIOLI SILVA
LUÍS GUSTAVO DE ARAÚJO ROCHA

ESPECIFICAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO 2M

MACEIÓ - AL
JANEIRO DE 2016

UNIVERSIDADE FEDERAL DE ALAGOAS – UFAL
CAMPUS A. C. SIMÕES

CIÊNCIA DA COMPUTAÇÃO – COMP263

MAXWELL ESDRA ACIOLI SILVA
LUÍS GUSTAVO DE ARAÚJO ROCHA

ESPECIFICAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO 2M

Trabalho desenvolvido sob a orientação do professor Alcino Dall'Igna, como requisito parcial para obtenção de nota na disciplina Compiladores do curso de Ciência da Computação da Universidade Federal de Alagoas.

MACEIÓ - AL
JANEIRO DE 2016

ESPECIFICAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO 2M

1. Introdução

A linguagem de programação 2M foi criada para implementação de algoritmos simples, além de ser útil na aprendizagem de conceitos básicos de programação, como as principais estruturas de controle e alguns tipos de dados.

2. Características Gerais

- Domínio da aplicação: Aplicações simples, implementação de algoritmos simples
- Paradigma de linguagem: Imperativa.
- Método de implementação: Compilação.

2.1. Forma Geral de um Programa em 2M

Para se criar um programa na linguagem de programação 2M é preciso escrever um código, obedecendo as regras da linguagem, o qual deve ser feito em um arquivo de extensão *.2m*.

Um programa em 2M é composto por um conjunto de funções. A função pela qual o programa começa a ser executado chama-se *major*.

A função *major* não retorna nenhum valor e é definida pela seguinte sintaxe:

major() **empty** [

]#

Após cada comando em 2M deve-se colocar um
#, que é definido como terminador de um comando na linguagem descrita.

Os delimitadores de escopo na linguagem 2M são "[" para iniciar e "]" para finalizar.

Um programa em 2M deve ser identado para que possa ser lido com mais facilidade.

A linguagem 2M permite também fazer comentários por bloco, o compilador elimina tudo o que se encontra entre a sequência inicial /$ e final $/.

3. Características Léxicas

Quando se programa em 2M, as variáveis e funções podem ser nomeadas através de um identificador. O identificador (ou nome) deve ser iniciado por uma letra ou underscore "_". A partir do segundo caractere, pode conter letras, números ou underscore.

2M é uma linguagem *case-sensitive*, ou seja, faz diferença entre nomes com letras minúsculas e nomes com letras maiúsculas. Assim, *compilador*, *Compilador* e *COMPILADOR* funcionam como identificadores diferentes.

Os identificadores devem ter no máximo 16 caracteres, não admitem caracteres especiais (por exemplo, %, $, {, ^) e não podem coincidir com as palavras reservadas da linguagem.

As palavras reservadas são nomes exclusivos para:

- **Especificadores de tipo:**

char
cchar
dec
int
logic

- **Função principal:**

major

- **Comandos de iteração ou seleção:**

if
else
elseif
iterator
do
while

- **Operadores lógicos:**

truth
false
not
and
or

- **Tipo de função sem retorno:**

empty

4. Especificação de tipos

A linguagem 2M é estaticamente tipada, logo todas as variáveis em 2M tem um tipo definido no momento de sua declaração. Esse tipo fica vinculado à variável durante toda sua vida.

São definidos os seguintes tipos de variáveis na linguagem 2M:

- ***int*** - inteiro de 16 bits;
- ***dec*** - ponto flutuante de 32 bits;
- ***char*** - caractere;
- ***cchar*** - cadeia de caracteres, que tem como limite um número máximo de 32.767 caracteres;
- ***logic*** - booleano;
- ***empty*** - tipo de função sem retorno;
- *tipo_de_variavel identificador_do_array*{*tamanho*} - arranjos unidimensionais.

4.1. Operações suportadas

Todos os tipos suportam as operações descritas na tabela abaixo com valores correspondentes ao seu respectivo tipo, ou seguindo regras de coerção.

Tipo	**Operações Suportadas**
int	*atribuição, aritméticos, relacionais*
dec	*atribuição, aritméticos, relacionais*
char	*atribuição, relacionais*
cchar	*atribuição, relacionais, concatenação*
logic	*atribuição, relacionais, lógicos*
array	*atribuição*

4.2 Coerção

A linguagem 2M provê conversão automática entre valores do tipo inteiro e do real. Caso um número ponto flutuante seja atribuído a uma variável do tipo inteiro, o valor dela

será apenas a parte inteira do número, já no caso inverso a parte inteira do ponto flutuante será o número inteiro e a parte decimal será zero.

5. Variáveis

5.1. Declaração

A declaração de uma variável tem a seguinte forma geral:

tipo *identificador_da_variavel#*

5.2. Inicialização

A inicialização deve ser feita em um comando separado do comando de declaração da variável, através de uma atribuição de valor, por exemplo:

variavel = 10#

Enquanto um valor não for atribuído a uma variável, a mesma possuirá um valor padrão definido como segue:

- ***int***: inicializado com o valor 0 (zero);
- ***dec***: inicializado com o valor 0.0 (zero);
- ***char***: inicializado com o valor ‘ ’ (espaço em branco, 32 no código ASCII);
- ***cchar***: inicializado com o valor “” (cadeia de caracteres vazia);
- ***logic***: inicializado com o valor false;
- ***array***: inicializado com o valor padrão de seu tipo definido.

5.3 Escopo

Na linguagem 2M todas as variáveis são globais, consequentemente são visíveis em todo o programa.

6. Operadores

Temos os seguintes operadores na linguagem 2M:

Aritméticos:

+	adição
-	subtração
*	multiplicação
/	divisão
^	exponencial
-	unário negativo

Relacionais:

>	maior que
<	menor que
==	igual
~=	diferente de
>=	maior ou igual
<=	menor ou igual

Lógicos:

truth	verdade
false	falso
not	negação
and	conjunção
or	disjunção

Concatenação de cadeias de caracteres:

++	concatenação

6.1. Precedência e Associatividade

A precedência é dada pela tabela abaixo, da ordem mais alta a mais baixa. Os operadores seguem as regras de associatividade especificadas na tabela:

not	Direita para esquerda
- (unário negativo)	Direita para esquerda
* /	Esquerda para direita
+ -	Esquerda para direita
< <= > =>	Esquerda para direita
== ~=	Esquerda para direita
and	Esquerda para direita
or	Esquerda para direita

Pode-se utilizar parenteses para alterar a precedência dos operadores. Como por exemplo em:

```
3 + 4 * 5
```

A expressão deve ser avaliada como *23*, pois a multiplicação será operada primeiro. No entanto, se organizarmos a expressão da seguinte forma:

```
( 3 + 4 ) * 5
```

Teremos um resultado diferente, pois os parenteses determinam que inicialmente deve ser operada a adição, e só então a multiplicação. Resultando nesse caso com a expressão sendo avaliada como *35*.

7. Instruções

7.1. Estrutura Condicional de Uma e Duas Vias

if
if/else
if/elseif/else

A instrução **if** controla a ramificação condicional. O corpo de uma instrução **if** é executado se o valor de expressão for diferente de zero. A sintaxe da instrução **if** tem dois formatos. A expressão condicional que controla a instrução é uma expressão lógica.

Sintaxe:

if (*expressao_logica*) [
    comandos#
]#

if (*expressao_logica*) [
    comandos#
] **else** [
    comandos#
]#

Caso a verificação precise ser mais abrangente ou exija várias validações temos a opção de usar **if**'s **aninhados**, usando a palavra reservada **elseif**. Assim verificamos, se a primeira opção não é satisfeita, verificamos a segunda e assim por diante.

Sintaxe:

if (*expressao_logica*) [
    comandos#
] **elseif** [
    comandos#
] **else** [
    comandos#
]#

7.2. Estrutura Iterativa com Controle Lógico

while
do while

A instrução **while** permite que você repita uma instrução até que uma expressão especificada seja falsa.

Sintaxe:

while(*expressao_logica*) [
comandos#
]#

A instrução **do while** permite que a instrução seja executada ao menos uma vez mesmo que a expressão especificada seja falsa.

Sintaxe:

do [
comandos#
] **while**(*expressao_logica*)**#**

7.3. Estrutura Iterativa Controlada por Contador

iterator

A instrução **iterator** permite repetir uma instrução ou uma instrução composta por um número especificado de vezes. O corpo de uma instrução **iterator** é executado zero ou mais vezes até que uma condição opcional se torne falsa. Você pode usar expressões opcionais na instrução **iterator** para inicializar e alterar valores durante a execução da instrução **iterator**.

Sintaxe:

iterator (*inicializacao* ; *condicao* ; *incremento*) [
comandos#
]#

7.4. Entrada e Saída

printout
readin

A instrução de saída **printout** deverá conter como parâmetro a mensagem que será impressa na tela. Esta mensagem pode ser uma cadeia de caracteres (que deverá ficar delimitada entre aspas duplas) ou uma variável (nesse caso será impresso seu valor). Pode ser usado o operador de concatenação para imprimir strings e variáveis numa mesma mensagem.

Já na instrução de entrada **readin** deve conter como parâmetros o tipo da variável que será lida, seguido do nome identificador da variável que será lida como entrada.

Sintaxe:

printout(*mensagem_a_ser_escrita_na_tela*)#
readin(**tipo**, *identificador_variável*)#

8. Atribuição

A atribuição é feita com operador = , com associatividade sempre da direita para a esquerda, por exemplo:

a = *b*# (O valor da variável *b* será atribuído à variável *a*)

9. Funções

Na linguagem 2M toda função deve ser declarada antes de ser usada. Na declaração de uma função não poderão ser omitidos quaisquer argumentos da mesma.

As funções são definidas de acordo com a seguinte sintaxe:

nome_da_funcao(**tipo** *param1 ;* **tipo** *param2 ; ...*) ***tipo_de_retorno*** [
    comandos#
]#

10. Exemplos

10.1 Hello World

```
major() empty [
        printout("Hello World")#
]#
```

10.2 Sequência de Fibonacci

```
major() empty [
        fibonacci( int limit ) empty#

        int limit#
        readin(int, limit)#

        fibonacci(limit)#

]#

fibonacci( int limit ) empty [

        int count#

        int fib1#
        int fib2#
        int fib3#

        fib1 = 1#
        fib2 = 1#

        if(limit == 0) [
                printout("0")#
        ]#

        while(count < limit)  [

                if(count <2) [
                      printout("1 ")#
                ] else [
                        fib3 = fib1 + fib2#
                        fib1 = fib2#
                        fib2 = fib3#

                        printout(fib3 ++ " ")#
```

```
            ]#
            count = count +1#
        ]#

]#
```

10.3 Shell Sort

```
major() empty [
        shell_sort( int array{10} ) int { }#

        int unsorted{10}#
        int sorted{10}#

        unsorted= {9, 8, 3, 2, 5, 1, 4, 7, 6, 0}#

        sorted = shell_sort(unsorted)#

]#

shell_sort( int array{10} ) int { }[

        int i#
        int j#
        int h#
        int size#
        int value#

        size = arraySize(array)#
        iterator(h = 1; h < size; h = h * 3 + 1) [

        ]#

        iterator( h = h/3 ; h < 1 ; ) [

                iterator (i = h ; i < size ; i = i + 1) [

                        value = array{i}#

                        iterator( j = i - h ; j >= 0 and value < array{j} ; j = j - h) [
                                array{j + h} = array{j}#
                        ]#

                        array{j + h} = value#

                ]#
        ]#
]#
```