GuaráScript

Especificação da Linguagem

Versão 2.0.0

Roberto Luiz Souza Monteiro
Marcelo A. Moret
Hernane Borges de Barros Pereira

Salvador, fevereiro 2015

Sumário

Operadores

A linguagem GuaráScript suporta vinte e dois operadores, para operações matemáticas e lógicas, envolvendo números inteiros, reais, complexos, cadeias de caracteres, vetores associativos e matrizes. Estes operadores têm as mesmas propriedades daqueles encontrados na linguagem C , exceto pelo operador potenciação(**) que tem as mesmas propriedades que no Matlab, embora utilize um lexema diferente.

A tabela a seguir apresenta cada um dos operadores suportados, assim como exemplos de sua utilização:

Operador	Descrição	Exemplo
$	Macro	$a, $(a)
@	Indireção	@a
!	Negação	!(a && b)
~	Inversão de bits	~a
**	Potenciação	a ** 2
*	Multiplicação	a * b
/	Divisão	a / b
%	Resto da divisão	a % b
+	Adição	a + b
−	Subtração	a − b
<<	Deslocamento de bits à esquerda	a << 2
>>	Deslocamento de bits à direita	a >> 2
<	Menor que	a < b
< =	Menor ou igual que	a < = b
>	Maior que	a > b
> =	Maior ou igual que	a > = b
==	Igual a	a = = b
!=	Diferente de	a != b
&|	Multiplicação booleana	a &| b
&	E bit a bit	a & b
∧	Ou exclusivo bit a bit	a ∧ b
|	Ou bit a bit	a | b
&&	E lógico	a && b
||	Ou lógico	a || b
=	Atribuição	a = b + 1

Para estes operadores, está definida a regra de precedência e associatividade apresentada na tabela seguinte:

Operadores	**Associatividade**
$	Direita para esquerda.
@	Direita para esquerda.
!, ~	Direita para esquerda.
**, *, /, %	Esquerda para direita.
+, −	Esquerda para direita.
<< , >>	Esquerda para direita.
< , < =, > , > =	Esquerda para direita.
==, !=	Esquerda para direita.
&|	Esquerda para direita.
&	Esquerda para direita.
∧	Esquerda para direita.
|	Esquerda para direita.
&&	Esquerda para direita.
||	Esquerda para direita.
=	Direita para esquerda.

Tipos de dados

São suportados sete tipos de dados, definidos automaticamente. O programador não precisa informar o tipo de dado que uma variável irá conter. O analisador léxico se encarrega de descobrir o tipo de dado mais adequado e alocar os recursos necessários para armazená-lo.

A tabela a seguir apresenta os tipos de dados suportados pelo interpretador GuaráScript:

Tipo	Descrição
Inteiro	Valor inteiro entre −2147483648 e 2147483647, para uma arquitetura de 32 bits. Pode ser expresso em base octal, decimal ou hexadecimal.
Real	Valor em ponto flutuante entre −1.0E+308 e 1.0E+308 para uma arquitetura de 32 bits.
Complexo	Número complexo em ponto flutuante entre −1.0E+308 e 1.0E+308, para ambas as partes, real e imaginária, para uma arquitetura de 32 bits.
Cadeia de caracteres	Cadeia de caracteres, delimitada por aspas, sem limite de tamanho (depende da arquitetura).
Vetor Associativo	Vetor cujos índices podem ser valores inteiros ou cadeias de caracteres, e cujos elementos podem conter valores inteiros, reais, complexos ou cadeias de caracteres.
Matriz	Matriz sem limite de dimensões. Cada elemento pode conter um valor inteiro, real, complexo ou cadeias de caracteres.
Manipulador	Manipulador de uso geral. Pode ser usado com ponteiros de memória. O sistema de arquivos virtual do GuaráScript utiliza manipuladores como ponteiros de arquivos.
Nulo	Um valor nulo. Não é inteiro, real, complexo, cadeia de caracteres, vetor ou matriz e não pode ser utilizado em nenhum tipo de operação exceto comparações e atribuições.

Inteiros

Valores inteiros podem ser representados em base octal, decimal ou hexadecimal, do mesmo modo que em Tcl.

Um valor octal é qualquer valor inteiro, não zero, cujo primeiro dígito seja o caractere zero(0).

Por exemplo:

```
x=017;
```

Um valor decimal é qualquer valor inteiro, não zero, cujo primeiro dígito não seja o caractere zero(0).

Por exemplo:

```
x=15;
```

Um valor hexadecimal é qualquer valor inteiro começando pelo prefixo 0x.
Por exemplo:

```
x=0xf;
```

Reais

Números reais podem ser expressos em notação decimal ou notação científica.
Por exemplo:

```
x=0.1;y=1.0e-3;
```

Complexos

Números complexos podem ser expressos como a soma de dois valores numéricos sendo a primeira parcela a parte real e a segunda, a parte imaginária que deve estar multiplicada da constante *i*.

Por exemplo:

```
x=1+2*i;
y=2.0+sin(0.5)*i;
```

A simbologia adotada para os números complexo é semelhante àquela utilizada no Matlab exceto pelo fato do GuaráScript não suportar o símbolo "j" como sinônimo da constante *i* e pelo fato de *i* não poder ser simplesmente pós-fixado da parte imaginária do número complexo, uma vez que i é uma constante e não um operador.

Um número complexo também pode ser criado através da função *complex(real, imaginário)*, ou calculando-se a raiz quadrada de −1:

```
a=sqrt(-1) #A variável a contém o número complexo i.
```

Cadeias de caracteres

Cadeias de caracteres, ou strings, podem ter qualquer tamanho, mas precisam estar delimitadas entre aspas, ou aspas simples.

Por exemplo:

```
nome="Albert Einstein";
email='albert@einstein.org';
```

Para concatenar cadeias de caracteres é utilizado o operador concatenação(+).

Por exemplo:

```
nome="Albert "+"Einstein";
```

Em uma operação de concatenação, se um dos operandos não for do tipo string ele será convertido para string antes que a concatenação seja realizada. Caso o operando seja inteiro, ele será representado em base decimal. Caso seja real, será representado utilizando a notação que apresente a menor perda de significado.

Por exemplo:

```
p=10.0;
economia="Foram economizados "+p+"%de energia no último ano";
```

É possível realizar comparações lexicográficas entre strings utilizando-se os operadores < , < =, > ,

> =, = =, !=. Caso um dos operadores não seja string, ele será convertido para string antes que a comparação seja realizada.

Para saber o comprimento de uma string pode-se utilizar a função *length(string)*.

Por exemplo:

```
a="Alo Mundo!";
l=length(a); #A variável l contém o valor 10.
```

É possível acessar caracteres em strings como se estas fossem matrizes lineares.

Por exemplo:

```
a="Alo Mundo!";
b=a[4]; #A variável b contém o valor "M".
```

Vetores associativos

Vetores associativos podem ter índices inteiros ou strings, chamados "chaves". Cada elemento do vetor pode conter um valor inteiro, real, complexo ou string. O uso de vetores associativos em GuaráScript é semelhante a Tcl , porém com alguns recursos a mais.

Para criar um vetor associativo pode-se utilizar a função *array(elemento1, elemento2, ...)* ou a estrutura *{elemento1, elemento2, ...}*. Em qualquer um dos casos, as chaves para os elementos serão números inteiros, começando com zero(0).

Por exemplo:

```
#Cria um vetor com chaves 0,1,2 e elementos 10,40,50,
#respectivamente.
a={10,40,100};
#Cria um vetor idêntico ao vetor a acima.
b=array(10,40,100);
```

Um elemento de um vetor associativo pode ser acessado indicando-se a chave para o elemento entre colchetes(*[]*).

Por exemplo:

```
a={10,40,100};
print(a[1]); #Exibe o valor 40 no console.
```

Para conhecer os valores das chaves de um vetor deve-se utilizar a função *keys(vetor)*. Esta função retorna um vetor associativo com chaves 0, 1, 2, ..., e valores iguais às chaves do vetor especificado. Vetores associativos também podem ser criados dinamicamente, como no exemplo a seguir:

```
#Cria o vetor associativo "cliente" e insere um novo elemento
#indexado pela chave "nome".
cliente["nome"]="Albert Einstein";
#Cria um novo elemento indexa do pela chave "e_mail".
cliente["e_mail"]="albert@einstein.org";
```

Pode-se excluir um elemento de um vetor atribuindo-se a ele o valor *NULL*:

```
a[1]=10;
a[2]=3.14;
a[3]=2*PI;

a[2]=NULL; #Remove o elemento indexado pela chave 2 do vetor.

#Para excluir o vetor, basta atribuir a ele o valor NULL:
a[1]=10;
a[2]=3.14;
a[3]=2*PI;

a=NULL;#Exclui o vetor;
```

Vetores associativos são internamento armazenados como tabelas hash, implementadas como listas duplamente encadeadas. Sendo a procura pelas chaves realizadas de forma seqüencial. Esta abordagem torna este tipo de vetor inadequado para cálculos matemáticos. Para esta finalidade recomenda-se o uso de matrizes, descritas a seguir.

Matrizes

Matrizes são vetores multi-dimensionais, criadas especialmente para realização de cálculos matemáticos. A implementação de matrizes no interpretador GuaráScript, foi tanto quanto possível, idealizada para prover as mesmas facilidades encontradas no Matlab.

Existem três formas para se criar uma matriz: através da função *matrix(valor, dimensão1, dimensão2, ...)*, *matrix2D(dimensão1, dimensão2, elemento1, elemento2, ...)* ou através da estrutura *[elemento, elemento, ...; elemento, elemento, ...; ...]*. No primeiro caso, a matriz será criada com as dimensões informadas e seus elementos inicializados com o valor especificado. No segundo caso será criada uma matrix bidimensional, com as dimensões especificadas, e preenchida com os elementos indicados. No último caso, a matriz criada será bidimensional e conterá os valores especificados na estrutura. As vírgula separam as colunas enquanto os ponto-e-vírgulas separam as linhas da matriz.

Por exemplo:

```
#Cria uma matriz com 2x2 com todos os elementos 0.
a=matrix(0,2,2);
#Cria uma matriz 2x2 contendo os valores 1,2,3 e 4.
b=[1,2;3,4];
```

A função *length(matriz)* retorna a dimensão linear da matriz especificada. Para obter as dimensão cartesianas da matriz, deve-se utilizar a função *dim(matriz)*. Esta função retorna um vetor associativo, onde cada elemento contém uma das dimensões da matriz.

Por exemplo:

```
#Cria uma matriz com 2x2 com todos os elementos 1.
a=matrix(1,2,2);
#Cria uma matriz 3x2 contendo os valores 1,2,3,4,5 e 6.
b=[1,2,3;4,5,6];

dim_a=dim(a); #Retorna o vetor{2,2}.
dim_b=dim(b); #Retorna o vetor{3,2}.
```

```
l=length(b); #Retorna o valor 6.
```

Um elemento de uma matriz pode ser acessado indicando-se os índices para o elemento entre colchetes(*[]*).

Por exemplo:

```
#Cria uma matriz 2x2 contendo os valores 1,2,3 e 4.
a=[1,2;3,4];
a[0,1]=2;

print(a);

#Exibe no console:
#[1,7,3,4]
```

Diferentemente dos vetores associativos, atribuir o valor *NULL* a um elemento de uma matiz não elimina o elemento, mas torna seu valor igual a *NULL*. Por outro lado, atribuir o valor *NULL* a uma matriz destrói a matriz.

Por exemplo:

```
#Cria uma matriz 2x2 contendo os valores 1,2,3 e 4.
a=[1,2;3,4];
a=NULL; #Destrói a matriz.
```

É possível comparar duas matrizes através dos operadores ==, !=, < , < =, > , > =. O operadores == e != comparam o conteúdo das matrizes, enquanto os operadores < , < =, > e > =, comparam as dimensões lineares das matrizes.

As operações adição(+), subtração(−) e multiplicação(*) são suportada apenas para matrizes bidimensionais. Embora a operação de divisão de matrizes não esteja definida na matemática, o GuaráScript a implementa como o produto da inversa da segunda matriz pela primeira, a fim de permitir a fácil solução de sistemas de equações lineares do tipo A * X = B . Neste caso a matriz de soluções do sistema pode ser obtida calculando-se X = B / A, que em GuaráScript é equivalente a X = A^{-1} * B.

Por exemplo:

```
a=[1,2;3,4]; #Cria uma matriz 2x2 contendo os valores1,2,3 e 4.
b=[5,6;7,8]; #Cria uma matriz 2x2 contendo os valores 5,6,7 e 8.

c=a+b; #A matriz c contém os valores 6,8,10 e 12.
d=2*a; #A matriz d contém os valores 2,4,6 e 8.
e=a*b; #A matriz e contém os valores 19,22,43,50.

x=[5;3;-1]/[2,3,-1;4,4,-3;2,-3,1];#x=[1;2;3]
```

Arquivos

Arquivos são abstrações do sistema operacional que permitem acessar dados armazenados em dispositivos de diversos tipos, de modo transparente para o programador. Alguns sistemas operacionais, como o UNIX permitem acessar dispositivos como câmeras de vídeo e *scanners* do mesmo modo como se realiza acesso a um arquivo contendo textos. Um ponteiro de arquivo é uma variável especial que permite acessar dados contidos em arquivos, utilizando-se as funções de manipulação de arquivos. Ponteiros para arquivos são criados através da função *fopen(arquivo, modo)*, descrita na seção **Bibliotecas de funções**, sub-seção **system**.

Por exemplo:

```
fd = fopen(“notas.txt”, "r");
```

Exemplo:

```
#!/usr/local/bin/guash

#Este programa lê dados de um arquivo fonte, salva os dados no
#arquivo de destino e, em seguida, exibe o arquivo fonte, seguido
#do arquivo de destino.

#Verifica se foram passados o nome do arquivo de origem e o nome
#do arquivo de destino...
if (argc < 3) {
    println("Use: source_text_file target_text_file.")
    exit
}

#Abre o arquivo de origem e o arquivo de destino...
fs = fopen(argv[2], "r")
ft = fopen(argv[3], "w")

#Copia dados do arquivo de origem para o arquivo de destino...
while (!feof(fs)) {
    if((line = fgets(fs)) == NULL) {
        break
    }

    fputs(line, ft)
}

#Fecha os arquivo...
fs = fclose(fs)
ft = fclose(ft)

#Exibe o arquivo de origem...
println("Source file...")
fs = fopen(argv[2], "r")

while (!feof(fs)) {
    if((line = fgets(fs)) == NULL) {
```

```
        break
    }

    print(line)
}

fs = fclose(fs)

#Exibe o arquivo de destino...
println("Target file...")
ft = fopen(argv[3], "r")

while (!feof(ft)) {
    if((line = fgets(ft)) == NULL) {
        break
    }

    print(line)
}

ft = fclose(ft)
```

Identificação do tipo de um dado

Em muitos casos pode ser necessário determinar o tipo de um dado antes que se possa trabalhar com ele. Para esta tarefa deve-se utilizar a função *type(dado)*. A função type retorna o tipo do dado como um valor inteiro, de acordo com a tabela a seguir.

Tipo	**Valor numérico**	**Constante**
Inteiro	0	*GUA_INTEGER.*
Real	1	*GUA_REAL.*
Complexo	2	*GUA_COMPLEX.*
Cadeia de caracteres	3	*GUA_STRING.*
Vetor associativo	4	*GUA_ARRAY.*
Matriz	5	*GUA_MATRIX.*
Manipulador	6	*GUA_HANDLE.*

Obtenção do tamanho de um dado

A função *length(valor)* retorna o tamanho de um dado do tipo *string*, *array* ou *matrix*.

Variáveis

Variáveis são *containers* para diferentes tipos de dados e têm o mesmo significado que na matemática. Pode-se associar qualquer nome a uma variável. A única restrição é que um nome não deve começar com os caracteres de 0 a 9, ponto(.) ou cifra(*$*) ou conter os caracteres '#' e '$', operadores matemáticos e lógicos e qualquer *token* definido na gramática da linguagem.

Atribuição de variáveis

Uma variável é criada automaticamente ao se atribuir um valor a ela. Dependendo do valor atribuído, o parser irá armazená-lo internamente, utilizando um dos tipos de dados suportados pelo interpretador.

Por exemplo:

```
a=1 #Cria uma variável inteira.
B=1.0 #Cria uma variável real.
c="Alô Mundo!" #Cria uma variável string.
D=[1,2;3,4] #Cria uma variável matriz.
```

Destruição de variáveis

Uma variável é destruída atribuindo-se a ela um valor NULL.

Por exemplo:

```
a=1 #Cria uma variável inteira.
B=1.0 #Cria uma variável real.
c="Alô Mundo!" #Cria uma variável string.
D=[1,2;3,4] #Cria uma variável matriz.
a=NULL
b=NULL
c=NULL
d=NULL
```

Identificação do tipo de dado de uma variável

É possível saber se uma variável está definida através da função *exists(nome_da_variável)*. Para conhecer o seu tipo de dado deve-se utilizar a função *type(nome_da_variável)*. Esta função retorna o tipo de dado como um valor inteiro, de acordo com a tabela .

Escopo de variáveis

Variáveis definidas fora de funções e procedimentos têm escopo global e são visíveis em todo o programa. Variáveis criadas no interior de procedimentos e funções têm escopo local e são destruídas quando o fluxo de execução deixa o interior da função. Para se acessar o conteúdo de uma variável global deve-se preceder o nome da variável do caractere cifra(*$*).

Por exemplo:

```
a=1 #Atribui o valor 1 à variável local a.
$a=2 #Atribui o valor 2 à variável global a.
b=a+1 #bcontémovalor2.
c=$a+1 #ccontémovalor3.
a=NULL #Destrói a variável local a.
$a=NULL #Destrói a variável global a.
$b=NULL #Causa um erro pois a variável global b não está definida.
```

Macro-substituição de nomes de variáveis

É possível realizar macro substituição de nomes de variáveis utilizando-se o operador macro(*$*). Para tanto, a expressão a ser substituída deve ser colocada entre parênteses e pós-fixada ao operador macro.

Por exemplo:

```
a=1 #Atribui o valor 1 à variável local a.
b="a" #b contém o nome da variável.
c=$(b) #c contém o valor armazenado na variável a.
a1=2
d=$(b+1) #d contém o valor da variável a 1.
```

No exemplo anterior, a variável c conterá o valor da variável a, pois a expressão entre parênteses será avaliada como "a" e o operador macro retornará o valor contido na variável denominada "a". Por outro lado, a variável d conterá o valor da variável a1, pois a expressão b + 1 será interpretada como a concatenação da string "a" com a string "1", resultando em "a1".

Indireção

É possível realizar indireção de nomes de variáveis utilizando-se o operador indireção(*@*). A indireção consiste em acessar uma variável, cujo nome encontra-se armazenado em outra variável.

Por exemplo:

```
a=1 #Atribui o valor 1 à variável local a.
b="a" #b contém o nome da variável.
c=@b #c contém o valor armazenado na variável a(1).
```

No exemplo anterior, a variável c conterá o valor da variável a, pois o operador indireção fará o interpretador ler o conteúdo da variável b(“a”) e acessar o conteúdo da variável com este nome(“a”).

É possível ler e escrever em variáveis utilizando o operador indireção.

Constantes

Constantes são variáveis criadas na inicialização do interpretador. A seguir são apresentadas as constantes suportadas pelo interpretador GuaráScript:

Valor	Constante	Descrição
	INTERP_VERSION	Número da versão do interpretador.
	GUA_VERSION	Número da versão do avaliador de expressões.
0	*GUA_INTEGER*	Número inteiro.
1	*GUA_REAL*	Número real.
2	*GUA_COMPLEX*	Número complexo.
3	*GUA_STRING*	Cadeia de caracteres.
4	*GUA_ARRAY*	Vetor associativo.
5	*GUA_MATRIX*	Matriz.
6	*GUA_HANDLE*	Manipulador.
1	*TRUE*	Verdeiro.
0	*FALSE*	Falso.
	NULL	Nulo.
$\sqrt{(-1)}$	*i*	Parte imaginária de um número complexo.

Variáveis internas

O interpretador mantém algumas variáveis globais utilizadas para configuração do ambiente e recuperação de informações especiais pelo programador. A seguir são apresentadas as variáveis internas definidas pelo interpretador GuaráScript:

Variável	Descrição
GUA_AVG	Valor médio obtido durante a avaliação da última declaração *test.*
GUA_DESIRED	Valor desejado durante a avaliação da última declaração *test.*
GUA_DEVIATION	Desvio em torno do valor esperado, obtido durante a avaliação da última declaração *test.*
GUA_ERROR	Mensagem referente ao último erro ocorrido.
GUA_RESULT	Resultado da avaliação do último bloco de código executado dentro de uma estrutura *try...catch* ou *test.*
GUA_TIME	Tempo consumido durante a avaliação da última declaração *test.*
GUA_TOLERANCE	Tolerância para o valor obtido durante a avaliação da última declaração *test.*
GUA_TRIES	Numero de tentativas realizadas durante a avaliação da última declaração *test.*

Comentários

Qualquer texto precedido pelo caractere sustenido(#) é um comentário e será ignorado pelo avaliador de expressões.

Por exemplo:

```
#Esta linha é um comentário.
A=1 #Este comentário será ignorado pelo avaliador de expressões.
if(a>0 #Este comentário é válido) #Mas este não é.{
    print("a>0\n");
}
```

Separação entre comandos

É possível colocar vários comandos em uma mesma linha, separando-os com o caractere ponto-e-vírgula(*;*).

Por exemplo:

```
a=b=1;c=2;printf("a=%d,b=%d\n",a,b);
```

Estruturas de decisão

Estruturas de decisão são mecanismos que permitem a realização condicional de partes do código de um programa. O GuaráScript permite a execução condicional através da declaração if.

if

A estrutura *if...elseif...else...* oferece suporte ao controle de fluxo de execução baseado em condições determinadas. A sintaxe desta estrutura é semelhante àquela oferecida pela linguagem C, exceto pelas chaves({}) que são obrigatórias:

```
if (condição 1) {
    código 1...
} elseif (condição 2) {
    código 2...
} elseif (condição N) {
    código N...
} else {
    código alternativo...
}
```

Caso a condição apresentada pela declaração *if* seja avaliada como verdadeira, o bloco de código 1 será avaliado, caso contrário, as condições apresentadas pelas declarações *elseif* serão testadas. Se uma delas for avaliada como verdadeira, o bloco de código correspondente será avaliado. Caso nenhuma das condições apresentadas seja avaliada como verdadeira, o bloco de código determinado pela declaração *else* será avaliado. As declarações *elseif* e *else* são opcionais.
A estrutura *if...elseif...else...* considera os valores FALSE, zero(0) e NULL como falso e qualquer outro valor como verdadeiro.

Exemplo:

```
x=1;

if (x<1) {
    printf("x é menor que 1\n");
} elseif (x==1) {
    printf("x é igual a 1\n");
} else {
    printf("x é maior que 1\n");
}
```

Estruturas de repetição

Estruturas de repetição são utilizadas quando se deseja que uma porção de código de programa seja executada um determinado número de vezes, ou até que uma dada condição seja satisfeita. São oferecidas quatro estruras de repetição: *while*, *do*, *for* e *foreach*.

while

A estrutura *while* executa o bloco de código especificado enquanto uma condição dada for avaliada como verdadeira. A sua sintaxe é apresentada a seguir:

```
while (condição) {
    código...
```

```
}
```

Exemplo:

```
j=0;

while (j<10) {
    printf("j=%d\n",j);
    j=j+1;
}
```

A palavra chave *break* interrompe a execução do laço de repetição sem completar a interação corrente, enquanto a palavra chave *continue* salta imediatamente para a próxima interação.

do

A estrutura *do...while* é semelhante à estrutura *while*, exceto que enquanto *while* testa a condição antes de executar o bloco de código, *do* executa o bloco de código especificado e depois testa a condição dada. A sintaxe da declaração *do* é apresentada a seguir:

```
do {
    código...
} while (condição)
```

Exemplo:

```
j=0;

do {
    printf("j=%d\n",j);
    j=j+1;
} while (j<10);
```

A palavra chave *break* interrompe a execução do laço de repetição sem completar a interação corrente, enquanto a palavra chave *continue* salta imediatamente para a próxima interação.

for

A estrutura *for* executa o bloco de código especificado enquanto uma condição dada for avaliada como verdadeira, opcionalmente, avaliando uma expressão antes de iniciar a primeira interação e uma expressão especificada, a cada interação. A sua sintaxe é apresentada a seguir:

```
for (inicialização;condição;incremento) {
    código...
}
```

Exemplo:

```
for (j=0;j<10;j=j+1) {
    printf("j=%d\n",j);
}
```

No exemplo anterior, a expressão j = 0 será executada antes que ocorra a primeira interação do laço de repetição e, a cada interação, será avaliada a expressão j = j + 1. O laço continuará sendo executado até que a expressão j < 10 seja avaliada como falsa.

A palavra chave *break* interrompe a execução do laço de repetição sem completar a interação corrente, enquanto a palavra chave *continue* salta imediatamente para a próxima interação.

foreach

A estrutura *foreach* permite percorrer um vetor associativo do primeiro ao último elemento, automaticamente, na ordem em que os elementos foram inseridos no vetor. Esta estrutura é muito semelhante àquela de mesmo nome encontrada na linguagem Tcl .

A sintaxe da estrutura *foreach* é apresentada a seguir:

```
foreach (vetor;chave;valor) {
    código...
}
```

O primeiro argumento da estrutura *foreach* é o vetor associativo que se deseja percorrer, enquanto que o segundo argumento é o nome de uma variável que receberá a chave do elemento no vetor e terceiro elemento é o nome de uma variável que receberá o valor armazenado na célula do vetor.

Exemplo:

```
a={10,20,30}

foreach (a;chave;valor) {
    printf("chave=%s,valor=%d\n",chave,valor);
}
```

No exemplo anterior a variável chave receberá os valores 0, 1 e 2, um de cada vez, a cada interação, enquanto a variável valor receberá os valores 10, 20 e 30.

A palavra chave *break* interrompe a execução do laço de repetição sem completar a interação corrente, enquanto a palavra chave continue salta imediatamente para a próxima interação.

Sub-rotinas

Procedimentos e funções podem ser criados utilizando-se a palavra chave *function*. A sintaxe desta declaração é apresentada a seguir:

```
function nome_da_função (argumento 1,argumento 2,argumento N) {
    código...
    return(valor)
}
```

Uma função pode ter qualquer número de argumentos formais e pode receber qualquer número de argumentos ao ser invocada. O programador deve cuidar de verificar se o número de argumentos passados para a função corresponde ao esperado e tratar eventuais erros.
Pode-se atribuir qualquer nome a uma função. A única restrição é que um nome não deve começar com os caracteres de 0 a 9, ponto(.) ou cifra(*$*) ou conter os caracteres '#' e '$', operadores matemáticos e lógicos e qualquer *token* definido na gramática da linguagem.
Para se definir um valor *default* a um argumento de uma função deve-se utilizar o operador atribuição(=).

Por exemplo:

```
function fatorial (n=0) {
    if ((n==0)||(n==1)) {
        return(1);
    } else {
        return(n*fatorial(n-1));
    }
}

fatorial(5) #Retorna 120.
fatorial; #Retorna 1;
```

No exemplo anterior o argumento n recebe o valor 0, como valor default, quando nenhum valor é passado para a função.

A palavra chave *return* retorna imediatamente da função, utilizando o valor especificado como valor de retorno. Se *return* for utilizado sem argumentos, a subrotina será um procedimento e retornará NULL.
Para destruir uma função basta atribuir a ela o valor NULL:

```
fatorial=NULL #Destrói a função fatorial.
```

O interpretador GuaráScript oferece uma facilidade quando se deseja criar funções de apenas um comando ou simples funções matemáticas: o operador atribuição(=) (somente a partir da versão 2.0).

Por exemplo:

```
f(x)=x**2+2*x+1;

y=f(2);
```

```
printf("f(2)=%f\n",y);
```

Tratamento de erros

Quando ocorre um erro em um programa, o interpretador GuaráScript interrompe imediatamente a execução do script e exibe uma mensagem de erro no console. Este comportamento pode não ser o mais adequado, pois alguns erros de tempo de execução podem ser tratados e recuperados utilizando-se algoritmos elaborados. Para estes casos, é fornecida a estrutura *try...catch* que captura as mensagens de erro, evitando a finalização prematura do script em execução. Esta estrutura é muito semelhante àquela encontrada no Matlab.

try

A sintaxe da estrutura *try...catch* é apresentada a seguir:

```
try {
    código...
} catch {
    código de tratamento de erro....
}
```

A declaração *try* avalia o bloco de código especificado e caso ocorra um erro executa o código de tratamento de erro indicado pela cláusula *catch*. Caso nenhum erro ocorra, a variável *GUA_RESULT* conterá o resultado da avaliação do bloco de código dado.

A cláusula *catch* é opcional, e a mensagem de erro gerada pelo interpretador também pode ser recuperada através da variável global *GUA_ERROR*.

Testes

Algumas vezes é preciso medir o tempo de execução de um script a fim de comparar sua performance com scripts pré-existentes. A estrutura *test* permite realizar testes de performance em scripts.

test

A sintaxe da estrutura *test* é apresentada a seguir:

```
test (número de repetições[;valor[;tolerância]]) {
    código...
} catch {
    código de tratamento de erro...
}
```

A declaração *test* avalia o bloco de código especificado, o número de vezes indicado e reporta o tempo consumido através da variável global *GUA_TIME*. Caso nenhum erro ocorra, a variável *GUA_RESULT* conterá o resultado da avaliação do bloco de código dado. Opcionalmente, pode-se definir um valor esperado e a tolerância em relação a este valor. As variáveis *GUA_AVG* e *GUA_DEVIATION* conterão respectivamente, o valor médio obtido e o desvio em relação ao valor esperado.

A cláusula *catch* é opcional e será executada caso ocorra algum erro durante a execução do script ou o teste resulte em um valor fora da tolerância esperada.

Funções embutidas no interpretador

O interpretador GuaráScript oferece algumas funções internas, disponíveis em todos os ambientes operacionais suportados pela linguagem, e diversas funções de biblioteca que podem ou não estar disponíveis, dependendo dos recursos oferecidos pelo sistema operacional para o qual o interpretado tenha sido portado.

A seguir são apresentadas as funções internas atualmente fornecidas pelo interpretador.

error

A função *error* causa uma exceção, utilizando o argumento passado como mensagem de erro. Sua sintaxe é apresentada a seguir:

```
try {
    error("mensagem de erro");
} catch {
    println("Erro: " + GUA_ERROR);
}
```

expr

A função *expr* avalia uma expressão lógica, matemática ou envolvendo strings ou vetores e retorna o resultado desta avaliação. Sua sintaxe é apresentada a seguir:

```
r=5;
a=expr("2*PI*r");
```

eval

A função *eval* avalia uma expressão dada e retorna o resultado desta avaliação. A sintaxe da função *eval* é apresentada no exemplo a seguir:

```
a="
function fatorial (n=0) {
    if ((n==0)||(n==1)) {
        return(1);
    } else {
        return(n*fatorial(n-1));
    }
}";

print(a);

eval(a);

fatorial(5)#Retorna 120.
```

exit

A função *exit* encerra o programa em execução, retornando ao sistema operacional o valor especificado. Sua sintaxe é apresentada a seguir:

```
exit(valor)
```

Exemplo:

```
exit(0);
```

type

A função *type* retorna o tipo de um dado como um número inteiro. A tabela a seguir apresenta os valores e constantes retornados por esta função.

Valor	Constante	Descrição
0	*GUA_INTEGER*	Número inteiro.
1	*GUA_REAL*	Número real.
2	*GUA_COMPLEX*	Número complexo.
3	*GUA_STRING*	Cadeia de caracteres.
4	*GUA_ARRAY*	Vetor associativo.
5	*GUA_MATRIX*	Matriz.
6	*GUA_HANDLE*	Manipulador.

Exemplo:

```
a=1;
b=1.2;
c="Alo Mundo!";
d={1,2,3};
e=[1,2;3,4];
type(a) #Retorna 0.
type(b) #Retorna 1.
type(c) #Retorna 3.
type(d) #Retorna 4.
type(e) #Retorna 5.
```

length

A função *length* retorna o tamanho de uma string, vetor ou matriz. No caso de matrizes *length* retorna a dimensão linear da matriz.

Exemplo:

```
a="Alo Mundo!";
l=length(a); #Avariável l contém o valor 10.
```

exists

A função *exists* retorna verdadeiro(TRUE) se a variável ou função especificada existir, e falso(FALSE) caso contrário.

Exemplo:

```
a=1;
exists("a") #Retorna TRUE;
exists("b") #Retorna FALSE;
```

complex

A função *complex* cria um número complexo dada as suas partes real e imaginária. A sua sintaxe é exibida a seguir:

```
complex(real,imaginário)
```

Exemplo:

```
a=complex(1,2);
print(a) #Será exibido "1+2*i"
```

array

A função *array* cria um novo vetor associativo contendo os elementos passados como argumentos. A sua sintaxe é exibida a seguir:

```
array(elemento 1,elemento 2,...,elemento N)
```

Exemplo:

```
a=array(10,40,100);
b=arrayToString(a);b contém a string "{10,40,100}"
print(b);
```

As chaves correspondentes aos elementos criados são números inteiros, seqüênciais, começando em zero(0).

arrayToString

A função *arrayToString* converte um vetor associativo em uma representação string que pode ser utilizada pela função *expr* para recriar o vetor. A sua sintaxe é a seguinte:

```
arrayToString(vetor)
```

Exemplo:

```
a={10,40,100};
b=arrayToString(a);b contém a string "{10,40,100}"
print(b);
```

keys

A função *keys* retorna as chaves de um vetor associativo como um novo vetor associativo. A sua sintaxe é a seguinte:

```
keys(vetor)
```

```
Exemplo:
cliente["nome"]="Albert Einstein";
cliente["e-mail"]="albert@einstein.org";

b=keys(cliente) #Retorna o vetor{"nome","e-mail"}.
```

matrix

A função *matrix* cria uma matriz, dado um valor inicial para preencher suas células e suas dimensões. A sua sintaxe é dada a seguir:

matrix(valor,dimensão 1,dimensão 2,...,dimensão N)

Exemplo:

```
#Cria uma matriz com 2x2 com todos os elementos 0.
a=matrix(0,2,2);
```

matrix2D

A função *matrix2D* cria uma matriz bidimensional, dadas as suas dimensões e os valores de seus elementos. A sintaxe da função *matrix2D* é apresentada a seguir:

matrix2D(dimensão 1,dimensão 2,elemento 1,elemento 2,...,elemento N)

Exemplo:

```
a=matrix2D(2,3,1,2,3,4,5,6) #Cria a matriz [1,2;3,4;5,6.
```

dim

A função *dim* retorna um vetor associativo contendo as dimensões de uma matriz. Sua sintaxe é apresentada a seguir:

dim(matriz)

Exemplo:

```
#Cria uma matriz com 2x2 com todos os elementos 1.
a=matrix(1,2,2);
#Cria uma matriz 3x2 contendo os valores 1,2,3,4,5 e 6.
b=[1,2,3;4,5,6];

dim_a=dim(a); #Retorna o vetor {2,2}.
dim_b=dim(b); #Retorna o vetor {3,2}.
```

matrixToString

A função *matrixToString* converte uma matriz em uma representação string que pode ser utilizada pela função *expr* para recriar a matriz. A sua sintaxe é a seguinte:

matrizToString(matriz)

Exemplo:

```
a=[1,2;3,4];
b=matrixToString(a);b contém as tring "[1,2;3,4]"
print(b);
```

toString

A função *toString* converte um valor qualquer em uma representação *string* que pode ser utilizada pela função *expr* para recriar o dado. A sua sintaxe é a seguinte:

```
toString(dado)
```

Exemplo:

```
a=[1,2;3,4];
b=toString(a);b contém as tring "[1,2;3,4]"
print(b);
```

after

A função *after* agenda a avaliação de uma expressão. (somente a partir da versão 2.0) Esta função é semelhante ao comando *after* fornecido pela linguagem Tcl e sua sintaxe é apresentada a seguir:

```
after(tempo,script);
```

Onde o tempo deve ser especificado em milisegundos, enquanto o script pode ser qualquer expressão válida em GuaráScript. A função *after* retorna um identificador que pode ser utilizado pela função *cancel* para cancelar o evento temporal agendado.

Exemplo:

```
#Agenda a execução para daqui a 2 segundos.
timeId=after(2000,"print(\"Passaram 2 segundos!\")");
#A execução continua até que o corra o evento temporal agendado.
x=2;
b=2*x;
```

cancel

A função *cancel* cancela um evento agendado pela função *after*. (somente a partir da versão 2.0) A sua sintaxe é apresentada a seguir:

```
cancel(identificador);
```

Exemplo:

```
#Agenda a execução para daqui a 2 segundos.
timeId=after(2000,"print(\"Passaram 2 segundos!\")");
#A execução continua até que ocorra o evento temporal agendado.
x=2;
calcel(timeId); #Cancela o evento temporal.
b=2*x;
```

print

A função *print* imprime uma expressão no console (movida para a biblioteca *system* para melhorar a portabilidade).

Exemplo:

```
a=1;
print("a="+a);
```

println

A função *println* imprime uma expressão no console e avança o cursor para a próxima linha (movida para a biblioteca *system* para melhorar a portabilidade).

Exemplo:

```
a=1;
println("a="+a);
```

Bibliotecas de funções

O interpretador GuaráScript fornece diversos recursos de programação através de bibliotecas de funções. A seguir são apresentadas as bibliotecas padrão, fornecidas com o sistema, suas funções e constantes internas.

array

A biblioteca *array* fornece funções para manipulação de vetores associativos.

As tabelas a seguir apresentam as constantes e funções providas por esta biblioteca.

Constante	Valor	Descrição
ARRAY_VERSION		Versão da biblioteca.

Função	Descrição
intersection(array1, array2)	Retorna um novo *array* contendo a interseção entre dois *arrays* fornecidos.
search(array, valor)	Retorna a chave para a primeira ocorrência de *valor* no vetor dado.

Exemplos:

```
a={4,1,3,2};
sort(a,ARRAY_INC) #Cria um novo vetor  contendo {1,2,3,4}.
c=search(a,3)#Retorna 2.
```

complex

A biblioteca *complex* fornece funções básicas para manipulação de números complexos. A tabela a seguir apresenta as funções oferecidas por esta biblioteca. As funções desta biblioteca fornecem os mesmos recursos que as funções de mesmo nome no Matlab.

As tabelas a seguir apresentam as constantes e funções providas por esta biblioteca.

Constante	Valor	Descrição
COMPLEX_VERSION		Versão da biblioteca.

Função	Descrição
abs(valor)	Retorna o módulo de um número complexo.
arg(valor)	Retorna o argumento de um número complexo.
conj(valor)	Retorna o número complexo conjugado.
imag(valor)	Retorna a parte imaginária de um número complexo.
real(valor)	Retorna a parte real de um número complexo.

Exemplos:

```
a=1.0+2.0*i;
```

```
b=real(a) #b contém o valor 1.0.
c=imag(a) #c contém o valor2.0.
```

file

A biblioteca *file* contém funções que permitem ao aplicativo manipular arquivos no sistema de arquivos do dispositivo. Esta biblioteca é baseada nas funções de mesmo nome oferecidas pela linguagem C e, quando o ANSI C não for suficiente, em recursos oferecidos pela linguagem Tcl.

As tabelas a seguir apresentam as constantes e funções atualmente suportadas pela biblioteca *file*.

Constante	**Descrição**
FILE_VERSION	Versão da biblioteca.
stdin	Entrada padrão.
stdout	Saída padrão.
stderr	Saída de mensagens de erro padrão.
SEEK_SET	Referência ao início do arquivo.
SEEK_CUR	Referência à posição atual no arquivo.
SEEK_END	Referência ao final do arquivo.
EOF	Final do arquivo.

Função	Descrição
clearerr(apontador_de_arquivos)	Limpa a variável interna de controle de erros de manipulação de arquivos.
fclose(apontador_de_arquivos)	Fecha o *arquivo* especificado.
feof(apontador_de_arquivos)	Retorna TRUE se o final do arquivo foi alcançado.
ferror(apontador_de_arquivos)	Retorna o código do erro ocorrido ao acessar o arquivo indicado pelo *apontador de arquivos* especificado.
fflush(apontador_de_arquivos)	Descarrega o *buffer* do arquivo especificado.
fgets(n, apontador_de_arquivos)	Lê *n* caracteres do arquivo, ou até encontrar um caractere de *nova linha*. O caractere de nova linha é dependente do sistema operacional adjacente: UNIX, *LF*, Windows, *CR+LF*, MacOS, *CR*.
fileno(apontador_de_arquivos)	Retorna o indicativo de arquivo para o apontador de arquivo especificado.
fopen(arquivo, modo)	Abre o *arquivo* especificado, no modo de acesso determinado, e retorna um *apontador de arquivos* para ele. O *modo* de acesso pode ser determinado por qualquer combinação dos modos leitura(*"r"*), escrita(*"w"*) ou anexação(*"a"*). Também são suportados os modos de atualização leitura(*"r+"*), escrita(*"w+"*) ou anexação(*"a+"*). Para abrir o arquivo em modo binário deve-se utilizar o sufixo *"b"*, assim os modos leitura(*"rb"*), escrita(*"wb"*) ou anexação(*"ab"*) também são suportados.
fputs(string, apontador_de_arquivos)	Grava a *string* especificada no arquivo e inclui um caractere de *nova linha* ao final dela.
fread(n, apontador_de_arquivos)	Lê *n* caracteres do arquivo especificado.
fseek(apontador_de_arquivos, posição, referencial)	Posiciona o ponteiro de leitura e gravação do arquivo, na posição especificada, relativa ao referencial indicado. O Referencial pode ser: *SEEK_SET*, para o início do arquivo, *SEEK_CUR*, para a posição atual no arquivo e *SEEK_END*, para o final do arquivo.
ftell(apontador_de_arquivos)	Retorna a posição atual no arquivo.
fwrite(string, apontador_de_arquivos)	Grava a *string* especificada no arquivo.
getchar()	Lê o próximo caractere do console.
gets()	Lê uma string a partir do console até encontrar um caractere de *nova linha*. O caractere de nova linha é dependente do sistema operacional adjacente: UNIX, *LF*, Windows, *CR+LF*, MacOS, *CR*.
putchar(caractere)	Escreve um caractere no console.
puts(string)	Exibe a *string* especificada no console e inclui um caractere de *nova linha* ao final dela.

Função	Descrição
rewind(apontador_de_arquivos)	Retorna o ponteiro de leitura e gravação do arquivo para o início do arquivo.

Exemplos:

```
#Funções de manipulação de arquivos.
h=fopen("test.txt","w") #Abre o arquivo no modo de escrita.
fputs("Alo Mundo!",h); #Escreve no arquivo.
fclose(h);#Fecha o arquivo.
```

math

A biblioteca *math* fornece as funções e constantes matemáticas mais utilizadas em aplicações científicas. Estas funções são idênticas àquelas de mesmo nome providas pela biblioteca ANSI C.

As tabelas a seguir apresentam as constantes e funções providas por esta biblioteca.

Constante	Valor	Descrição
MATH_VERSION		Versão da biblioteca.
E	2.7182818284590452354	Número neperiano (e).
PI	3.14159265358979323846	Número PI (π).

Função	Descrição
acos(x)	Calcula o arco cosseno de x.
asin(x)	Calcula o arco seno de x.
atan(x)	Calcula o arco tangente de x.
atan2(y, x)	Calcula o arco tangente de y/x e utiliza os sinais de y e x para determinar o quadrante do resultado.
ceil(x)	Retorna o resultado da aproximação de x para maior.
cos(x)	Calcula o cosseno de x.
cosh(x)	Calcula o cosseno hiperbólico de x.
deg(x)	Converte um ângulo x em radianos para graus.
dist(x1, y1, x2, y2) ou *dist(x1, y1, z1, x2, y2, z2)*	Calcula a distância entre dois pontos no plano(p1(x1,y1) e p2(x2, y2)) ou no espaço(p1(x1, y1, z1) e p2(x2, y2, z2)).
exp(x)	Calcula e^x.
fabs(x)	Retorna o módulo de x.
factorial(x)	Calcula o fatorial de x.
floor(x)	Retorna o resultado da aproximação de x para menor.
fmax(x1, x2)	Retorna o maior de dois números, x1 e x2.
fmin(x1, x2)	Retorna o menor de dois números, x1 e x2.
fmod(x, y)	Retorna o resto da divisão de x por y.
ldexp(x, y)	Calcula $x \cdot 2^y$.
log(x)	Calcula lnx
log10(x)	Calcula $\log_{10}x$.
pow(x, y)	Calcula x^y.
rad(x)	Converte um ângulo x em graus para radianos.
random()	Retorna um número aleatório entre 0 e 1.
round(x)	Retorna o resultado da aproximação de x para o inteiro mais próximo (como um valor real).
roundl(x)	Retorna o resultado da aproximação de x para o inteiro mais próximo (como um valor inteiro).
sin(x)	Calcula o seno de x.
sinh(x)	Calcula o seno hiperbólico de x.
sqrt(x)	Calcula $\sqrt{x}$.
srandom(x)	Inicializa a semente de números aleatórios utilizando o valor fornecido.
tan(x)	Calcula a tangente de x.
tanh(x)	Calcula a tangente hiperbólica de x.

matrix

A biblioteca *matrix* fornece funções básicas para manipulação de matrizes. As tabelas a seguir apresentam as constantes e funções providas por esta biblioteca.

Constante	Valor	Descrição
MATRIX_VERSION		Versão da biblioteca.

Função	Descrição
avg(matriz[,l1,c1,l2,c2])	Retorna a média dos valores contidos nas células da *matriz*. Os valores *l1*, *c1* e *l2*, *c2*, são opcionais e correspondem, respectivamente, ao número da *linha* e da *coluna* do *canto superior esquerdo* e ao número da *linha* e da *coluna* do *canto inferior direito* de um *retângulo de células* da *matriz* especificada.
count(matriz[,l1,c1,l2,c2])	Retorna o número de valores não nulos contidos nas células da *matriz*. Os valores *l1*, *c1* e *l2*, *c2*, são opcionais e correspondem, respectivamente, ao número da *linha* e da *coluna* do *canto superior esquerdo* e ao número da *linha* e da *coluna* do *canto inferior direito* de um *retângulo de células* da *matriz* especificada.
cross(vetor1, vetor2)	Retorna o produto vetorial dos vetores dados.
delcol(matriz, linha)	Remove a *coluna* indicada, da *matriz* especificada.
delrow(matriz, linha)	Remove a *linha* indicada, da *matriz* especificada.
det(matriz)	Retorna o determinante da matriz dada.
diag(dimensão1, dimensão2, valor)	Retorna uma matriz diagonal, com os elementos da diagonal principal iniciando com o valor dado.
dot(vetor1, vetor2)	Retorna o produto escalar dos vetores dados.
gauss(matriz)	Retorna a matriz triangular equivalente da matriz dada.
ident(dimensão1, dimensão2)	Retorna a matriz identidade (movido para o núcleo do interpretador).
inv(matriz)	Retorna a matriz inversa da matriz dada (movido para o núcleo do interpretador).
jordan(matriz)	Retorna a matriz diagonal equivalente da matriz dada.
max(matriz[,l1,c1,l2,c2])	Retorna o maior dos valores contidos nas células da *matriz*. Os valores *l1*, *c1* e *l2*, *c2*, são opcionais e correspondem, respectivamente, ao número da *linha* e da *coluna* do *canto superior esquerdo* e ao número da *linha* e da *coluna* do *canto inferior direito* de um *retângulo de células* da *matriz* especificada.
min(matriz[,l1,c1,l2,c2])	Retorna o menor dos valores contidos nas células da *matriz*. Os valores *l1*, *c1* e *l2*, *c2*, são opcionais e correspondem, respectivamente, ao número da *linha* e da *coluna* do *canto*

Função	Descrição
	superior esquerdo e ao número da *linha* e da *coluna* do *canto inferior direito* de um *retângulo de células* da *matriz* especificada.
one(dimensão1, dimensão2)	Retorna uma matriz onde todos os elementos possuem o valor 1.
rand(dimensão1, dimensão2)	Retorna uma matriz com valores aleatórios.
sum(matriz[,l1,c1,l2,c2])	Retorna a soma dos valores contidos nas células da *matriz*. Os valores *l1*, *c1* e *l2*, *c2*, são opcionais e correspondem, respectivamente, ao número da *linha* e da *coluna* do *canto superior esquerdo* e ao número da *linha* e da *coluna* do *canto inferior direito* de um *retângulo de células* da *matriz* especificada.
trans(matriz)	Retorna a matriz transposta da matriz dada.
zero(dimensão1, dimensão2)	Retorna uma matriz onde todos os elementos possuem o valor 0.

Exemplos:

```
a=[1,2;3,4]
b=trans(a) #b contém a transposta de a.
c=inv(a) #c contém a inversa de a.
d=sum([1,2,3;4,5,6;7,8,9]) #d contém o valor 45.
e=sum([1,2,3;4,5,6;7,8,9], 1, 1, 2, 2) #e contém o valor 28.
f=avg([1,2,3;4,5,6;7,8,9]) #f contém o valor 5.
g=avg([1,2,3;4,5,6;7,8,9], 1, 1, 2, 2) #g contém o valor 7.
h=count([1,2,3;4,5,6;7,8,9]) #h contém o valor 9.
i=count([1,2,3;4,5,6;7,8,9], 1, 1, 2, 2) #i contém o valor 4.
j=delrow([1,2,3;4,5,6;7,8,9], 1) #j contém a matriz [1,2,3;7,8,9].
k=delcol([1,2,3;4,5,6;7,8,9], 1) #k contém a matriz [1,3;4,6;7,9].
```

numeric

A biblioteca *numeric* fornece funções para computação numérica.

As tabelas a seguir apresentam as constantes e variáveis globais e funções suportadas pela biblioteca *numeric*.

Variável	**Valor**	**Descrição**
NUM_VERSION		Versão da biblioteca.
GOLDEN_NUMBER	1.6180	Número de ouro.
NUM_X	"x"	Variável independente padrão.
NUM_XYZ	{"x","y","z"}	Variáveis padrão, x, y e z .

Função	Descrição
bisection(função, a, b, toler, maxinter)	A função *bisection* utiliza o método da bisseção para encontrar uma raíz de uma função dada, em um intervalo [a, b], no qual a função seja contínua. Onde *função* é a função para a qual se deseja encontrar o valor da raíz, *a* e *b* são os extremos do intervalo, *toler* é o valor da tolerância e *maxinter* é o número máximo de interações. A cada interação, é atribuído à variável global apontada por *NUM_X* o valor de x = (a + b) / 2, e avaliada a expressão *função*. Isto permite que a função seja especificada na forma algébrica, por exemplo: x**2+2*x+1. Este método é útil quando não se conhece a derivada da função. (somente a partir da versão 2.0)
gaussInt(função, a, b, n)	A função *gaussInt* utiliza o método da *quadratura gaussiana* para calcular a integral definida I = $\int_a^b$ f(x)dx. Onde *função* é a função para a qual se deseja calcular o valor da integral, *a* e *b* são os limites inferior e superior do intervalo de integração e *n* é o número de pontos a considerar. A função utiliza a variável indicada por *NUM_X*, por *default* "x", no cálculo da f(x). (somente a partir da versão 2.0)
gaussInt2D(função, ax, bx, nx, ay, by, ny)	A função *gaussInt2D* utiliza o método de *Gauss-Legendre* para calcular a integral dupla definida I = $\int_a^b$dx $\int_c^d$ f(x,y)dy. Onde *função* é a função para a qual se deseja calcular o valor da integral, *ax* e *bx* são os limites inferior e superior em x, *nx* é o número de pontos a considerar em x, *ay* e *by* são os limites inferior e superior em y, e *ny* é o número de pontos a considerar em y. A função utiliza as variáveis indicadas por *NUM_XYZ*, por *default* {"x", ÿ", "z"}, no cálculo da f(x,y,z). (somente a partir da versão 2.0)
gaussInt3D(função, ax, bx, nx, ay, by, ny, az, bz, nz)	A função *gaussInt3D* utiliza o método de *Gauss-Legendre* para calcular a integral tripla definida I = $\int_a^b$dx $\int_c^d$dy $\int_e^f$ f(x,y,z)dz. Onde *função* é a função para a qual se deseja calcular o valor da integral, *ax* e *bx* são os limites inferior e superior em x, *nx* é o número de pontos a considerar em x, *ay* e *by* são os limites inferior e superior em y, *ny* é o número de pontos a considerar em y, *az* e *bz* são os limites inferior e superior em z, e *nz* é o número de pontos a considerar em z. A função utiliza as variáveis indicadas por *NUM_XYZ*, por *default* {"x", ÿ", "z"}, no cálculo da f(x,y,z). (somente a partir da versão 2.0)
gaussLSS(a, b)	A função *gaussLSS* utiliza o método de Gauss para resolver um sistema de equações lineares na forma B = A * X. Onde *a* e *b* são as matrizes do sistema. Esta função retorna uma matriz contendo os valores de x que resolvem o sistema dado.
gaussSeidelLSS(a, b, toler, maxinter)	A função *gaussSeidelLSS* utiliza o método de Gauss-Seidel para resolver um sistema de equações lineares na forma B = A

Função	Descrição
	* X. Onde *a* e *b* são as matrizes do sistema, *toler* é o valor da tolerância e *maxinter* é o número máximo de interações. Esta função retorna uma matriz contendo os valores de x que resolvem o sistema dado. (somente a partir da versão 2.0)
jacobiLSS(a, b, toler, maxinter)	A função *jacobiLSS* utiliza o método de Jacobi para resolver um sistema de equações lineares na forma B = A * X. Onde *a* e *b* são as matrizes do sistema, *toler* é o valor da tolerância e *maxinter* é o número máximo de interações. Esta função retorna uma matriz contendo os valores de x que resolvem o sistema dado. (somente a partir da versão 2.0)
lsquares(x, y)	A função *lsquares* utiliza o método dos mínimos quadrados para ajustar a curva y = ax + b a um conjunto de pares de valores (x, y) fornecidos. Onde *x* e *y* são as matrizes contendo os valores conhecidos de *x* e *y*. Esta função retorna uma matriz contendo os valores de a e b que ajustam a curva y = ax + b ao conjunto de pontos dado. (somente a partir da versão 2.0)
newton(função, derivada, x0, toler, maxinter)	A função *newton* utiliza o método de Newton para encontrar uma raíz de uma função dada. Onde *função* é a função para a qual se deseja encontrar o valor da raíz, *derivada* é a derivada da função, *x0* é um valor aproximado da raíz, *toler* é o valor da tolerância e *maxinter* é o número máximo de interações. A cada interação, é atribuído à variável global apontada por *NUM_X* o valor de x = x + Δx, e avaliadas as expressões *função* e *derivada*. Isto permite que as funções sejam especificada na forma algébrica, por exemplo: x**2+2*x+1, para a função e 2*x+2, para a derivada. Este método é útil somente quando se conhece a derivada da função. (somente a partir da versão 2.0)

Exemplos:

```
x=gaussLSS([10,3,-2;2,8,-1;1,1,5],[57;20;-4]);

print("x="+x);
```

```
x=gaussSeidelLSS([10,3,-2;2,8,-1;1,1,5],[57;20;-4],0.00001,50);

print("x="+x);
```

```
x=jacobiLSS([10,3,-2;2,8,-1;1,1,5],[57;20;-4],0.00001,50);

print("x="+x);
```

```
r=lsquares([0.3,2.7,4.5,5.9,7.8],[1.8,1.9,3.1,3.9,3.3]);

printf("y=%f*x+%f\n",r[0],r[1]);
```

```
r=bisection("x**2+2*x+1",-5.0,5.0,0.005,100);

printf("Raiz de x**2+2*x+1:%6.3f\n",r);
```

```
r=newton("x**2+2*x+1","2*x+2",1,0.005,100);

printf("Raiz de x**2+2*x+1:%6.3f\n",r);
```

```
I=gaussInt("sen(x)",0,PI,6);

print("I="+I);
```

```
I=gaussInt2D("2*x*y*sen(x*y**2)",0,PI/2,PI/2,PI);

print("I="+I);
```

printf

A biblioteca *printf* fornece duas funções *printf* e *sprintf*. Ambas as funções têm funcionalidades idênticas àquelas encontradas na linguagem C . De fato, a função *sprintf* foi implementada de modo a utilizar a função de mesmo nome da biblioteca ANSI C, para executar seu trabalho.

A função *printf* escreve no console dados formatados, de acordo com um formato especificado, enquanto *sprintf* atribui a uma variável uma string contendo dados formatados segundo as mesmas especificações de formato que *printf*. As sintaxes das funções *printf* e *sprintf* são apresentadas a seguir:

```
printf(formato,valor 1,valor 2,valor N)
sprintf(formato,valor 1,valor 2,valor N)
```

onde *valor 1*, *valor 2*, *valor N* são valores(inteiro, real ou string) que se deseja exibir, e formato é uma string contendo a especificação a ser utilizada na formatação dos dados.

A tabela a seguir apresenta os especificadores de formato suportados pelas funções *printf* e *sprintf*.

Especificador	**Descrição**
%	Caractere *%*.
#	Seleciona um modo alternativo de formatação. Para o formatador o será sempre exibido um número 0, para x e X, o resultado será prefixado de *0x* ou *0X*, e para *e*, *E*, *f*, *g* e *G*, o resultado será sempre exibido contendo um ponto decimal.
-	Determina que o valor correspondente seja justificado pela esquerda.
+	Determina que o sinal do número seja sempre exibido, mesmo que seja um número positivo.
0	Justifica o número com zeros(0) à esquerda.
Espaço	Adiciona espaço à esquerda do número se o primeiro caractere não for um sinal.

Especificador	Descrição
d ou *i*	Converte um valor inteiro sinalizado em uma string.
u	Converte um valor inteiro não sinalizado em uma string.
o	Converte um valor inteiro em uma string octal não sinalizada.
x ou *X*	Converte um valor inteiro em uma string hexadecimal não sinalizada.
c	Converte um valor inteiro no caractere ASCII que ele representa.
s	Insere uma string.
f	Converte um número em ponto flutuante em uma string na forma xx.yyy, onde yyy é dado pela precisão especificada na string de formatação(o valor default é 6).
e ou *E*	Converte um número em ponto flutuante em uma string na forma xx.yyye± zz ou xx.yyyE± zz, onde yyy é dado pela precisão especificada na string de formatação(o valor default é 6).
g ou *G*	Decide automaticamente se deve usar o formato determinado por *f*, *e* ou *E*.

Uma especificação de formato deve sempre começar com o caractere *%* seguido de um modificador -, +, 0 ou espaço, o tamanho máximo do campo, um separador(.), a precisão e um especificador de conversão *u*, *i*, *o*, *x*, *X*, *f*, *e*, *E*, *g* ou *G*. Qualquer outro caracter, será exibido da forma como aparecer na string de formatação.

Exemplos:

```
printf("x=%06.3f\n",1) #Exibe "x=01.000" no console.
printf("x=%02X\n",65) #Exibe "x=41" no console.
printf("x=%#2x\n",65) #Exibe "x=0x41" no console.
#Atribui o valor "x=0000000001" à variável a.
a=sprintf("x=%010.10s\n",1)
```

string

A biblioteca *string* fornece funções para manipulação de cadeias de caracteres. As tabelas a seguir apresentam as constantes e funções providas por esta biblioteca.

Constante	Valor	Descrição
STRING_VERSION		Versão da biblioteca.

Função	Descrição
at(string, posicao)	Retorna o caractere presente na *string* especificada, na posição indicada.
isalnum(caractere)	Retorna VERDADEIRO se o caractere indicado for alfanumérico.
isalpha(caractere)	Retorna VERDADEIRO se o caractere indicado for alfabético.
isascii(caractere)	Retorna VERDADEIRO se o caractere indicado for ASCII de 7 bits.
isblank(caractere)	Retorna VERDADEIRO se o caractere indicado for branco(ESPAÇO ou TAB).
iscntrl(caractere)	Retorna VERDADEIRO se o caractere indicado for um caractere de controle.
isdigit(caractere)	Retorna VERDADEIRO se o caractere indicado for um dígito de 0 a 9.
isgraph(caractere)	Retorna VERDADEIRO se o caractere indicado for um caractere imprimível, exceto ESPAÇO.
islower(caractere)	Retorna VERDADEIRO se o caractere indicado estiver em CAIXA BAIXA.
isprint(caractere)	Retorna VERDADEIRO se o caractere indicado for um caractere imprimível, incluindo ESPAÇO.
ispunct(caractere)	Retorna VERDADEIRO se o caractere indicado não for alfanumérico ou ESPAÇO.
isspace(caractere)	Retorna VERDADEIRO se o caractere indicado for um caractere de formatação: ESPAÇO, FORM-FEED('\f'), NEWLINE('\n'), CARRIAGE RETUR('\r'), HORIZONTAL TAB('\t') e VERTICAL TAB ('\v').
isupper(caractere)	Retorna VERDADEIRO se o caractere indicado estiver em CAIXA ALTA.
isxdigit	Retorna VERDADEIRO se o caractere indicado for hexadecimal.
ltrim(string, caracteres)	Remove caracteres à esquerda da *string* especificada. O *caractere default* é o caractere espaço(“ “).
range(string, inicio, fim)	Retorna a cadeia de caracteres entre *inicio* e *fim*, presente na *string* especificada.

replace(string, caractere, novo)	Substitui todas as ocorrências do *caractere* especificado, por um *novo* caractere indicado.
rtrim(string, caracteres)	Remove caracteres à direita da *string* especificada. O *caractere default* é o caractere espaço(“ “).
split(string, separadores)	Quebra uma *string* em partes, utilizando os *separadores* de partes especificados. O resultado é um vetor associativo, cujos elementos são as partes da *string* e os índices, números inteiros começando em zero(0). O *separador default* é o caractere espaço(“ “).
string(tamanho, caracteres)	Cria uma *string* com o *tamanho* especificado, preenchendo-a com os *caracteres* indicados.
strpos(string1, string2)	Retorna a posição da primeira ocorrência de *string2* em *string1*.
tolower(string)	Converte uma *string* para caixa baixa(letras minúsculas).
toupper(string)	Converte uma *string* para caixa alta(letras maiúsculas).
trim(string, caracteres)	Remove caracteres à esquerda e à direita da *string* especificada. O *caractere default* é o caractere espaço(“ “).

Exemplos:

```
a="Alô Mundo!";
b=index(a,4) #Retorna "M".
c=range(a,4,8) #Retorna "Mundo".
d=split(a)#Retorna {“Alô”, “Mundo!”}.
e=split("a,b;c d", ",; ") #Retorna {"a","b","c","d"}.
f=rtrim("abc.; ", ".; ") #Retorna “abc”.
```

system

A biblioteca *system* contém funções que permitem ao aplicativo interagir com o sistema operacional adjacente. Esta biblioteca é baseada nas funções de mesmo nome oferecidas pela linguagem C e, quando o ANSI C não for suficiente, em recursos oferecidos pela linguagem Tcl.

As tabelas a seguir apresentam as constantes e funções atualmente suportadas pela biblioteca *system* e os caracteres de formatação suportados pela função *strftime*, respectivamente.

Constante	**Descrição**
SYS_VERSION	Versão da biblioteca.
SYS_HOST	Nome do sistema operacional para o qual o interpretador foi compilado(*“linux”*, *“windows”* ou *“macos”*).
SYS_ARCH	Nome da arquitetura para a qual o interpretador foi compilado(atualmente apenas *“i386”*).
argc	Número de argumentos passados na linha de comando.
argv	Vetor associativo contendo os argumentos passados na linha de comando.
env	Vetor associativo contendo as variáveis de ambiente.

Função	Descrição
exec(comando)	Executa o *comando* do sistema operacional, ou programa especificado, retornando o valor de retorno do comando, em caso de sucesso, ou −1, caso ocorra algum erro. Esta função não existe em ANSI C, mas é semelhante ao comando de mesmo nome encontrado na linguagem Tcl.
load(biblioteca)	Carrega a *biblioteca* especificada. O formato da biblioteca é dependente do sistema operacional adjacente. Esta função é semelhante ao comando de mesmo nome encontrado na linguagem Tcl (somente a partir da versão 1.2 da biblioteca *system*). (somente a partir da versão 2.0)
print(string)	Exibe a *string* especificada no console sem incluir um caractere de *nova linha* ao final dela.
println(string)	Exibe a *string* especificada no console e inclui um caractere de *nova linha* ao final dela.
source(arquivo)	Avalia o conteúdo de um *arquivo* como um *script*. Esta função é semelhante ao comando de mesmo nome encontrado na linguagem Tcl.
strftime(formato, hora)	Retorna a data e hora informados pelo argumento *hora*, formatados de acordo com o padrão *formato*. Esta função possui os mesmos especificadores de formato que a função de mesmo nome da biblioteca ANSI C.
time()	Retorna a hora do sistema como o número de segundos e micro-segundos passados desde *01/01/1970.*

Especificador	Descrição
a	Nome da semana abreviado(Mon, Tue, etc.).
A	Nome da semana completo(Monday, Tuesday, etc.).
b	Nome do mês abreviado(Jan, Feb, etc.).
B	Nome do mês completo.
c	Data e hora localizados. No UNIX e no MacOS, "%a %b %d %H:%M:%S %Y", no Windows, depende das configurações no Painel de Controle.
d	Dia do mês(01 – 31).
h	Nome do mês abreviado.
H	Hora no formato 24 horas(00 – 23).
j	Dia do ano(001 – 366).
m	Número do mês(01 – 12).
M	Minuto(01 – 12).
n	Insere uma nova linha.
p	Exibe AM/PM.
U	Dia da semana(00 - 52). Domingo é o primeiro dia da semana.
W	Dia da semana(00 - 52). Domingo é o primeiro dia da semana.
x	Data no formato local. No UNIX e no MacOS, "%m/%d/%y", no Windows, depende das configurações no Painel de Controle.
X	Hora no formato local. No UNIX e no MacOS, "%H:%M:%S", no Windows, depende das configurações no Painel de Controle.
y	Ano no formato de dois dígitos(00 – 99).
Y	Ano no formato de quatro dígitos.
Z	Fuso horário.
%	Exibe um caractere *%*.

Exemplos:

```
#Ambiente.
for(j=0;j<argc;j++){
    print("argv["+j+"]="+argv[j]);
}

foreach(env,nome,valor){
    printf("%s=%s\n",nome,valor);
}

#Data e hora.
printf("%s\n",strftime("%H:%M:%S%d/%m/%Y",time()));
```

tui

A biblioteca *tui(Terminal User Interface)* fornece funções para interface homem máquina através de terminais ANSI.

As tabelas a seguir apresentam as constantes e variáveis globais e funções suportadas pela biblioteca *tui*.

Variável	**Valor**	**Descrição**
TUI_VERSION		Versão da biblioteca.
BLACK	0	Cor preto.
RED	1	Cor vermelho.
GREEN	2	Cor verde.
YELLOW	3	Cor amarelo.
BLUE	4	Cor azul.
MAGENTA	5	Cor magenta.
CYAN	6	Cor ciano.
WHITE	7	Cor branco.

Função	**Descrição**
box(x1, y1, x2, y2, cor_da_borda, cor_de_fundo)	Desenha um retângulo na tela, dadas as coordenadas do *canto superior esquerdo*(*x1*, *y1*) e do *canto inferior direito*(*x2*, *y2*), utilizando as cores da *borda* e de *fundo* especificadas.
clreol	Limpa a linha onde se encontra o cursor na tela.
clrscr	Limpa a tela.
entry(x, y, tamanho, cor_do_texto, cor_do_fundo)	Exibe uma caixa para digitação, nas coordenadas *x*, *y* especificadas, com o *tamanho*(em caracteres) indicado, utilizando as cores informadas para o primeiro(*texto*) e o segundo(*fundo*) planos, respectivamente.
getch	Retorna um caractere, a partir do teclado, sem efetuar *buffer* e sem exibi-lo na tela.
getche	Retorna um caractere, a partir do teclado, sem efetuar *buffer* e exibindo-o na tela.
getkey	Retorna um caractere, a partir do teclado, como um valor inteiro, sem efetuar *buffer* e sem exibi-lo na tela.
gotoxy(x, y)	Posiciona o cursor na *linha*(x) e *coluna*(y) especificadas.
textbackground(cor)	Configura a cor do segundo plano(*fundo*) tela.
textcolor(cor)	Configura a cor do primeiro plano(*texto*) da tela.

Exemplo:

O programa a seguir, ilustra como utilizar não apenas a biblioteca *tui* para construção de interfaces de usuário baseadas em terminal ANSI, mas também como utilizar as funções da biblioteca *file* para acesso a arquivos de dados.

```
#!/usr/local/bin/guash
#
# phone.gua
#
#     This file implements a simple phone book.
#
# Copyright (C) 2005 Roberto Luiz Souza Monteiro
#
# This program is free software; you can redistribute it
# and/or modify it under the terms of the GNU General
# Public License as published by the Free Software
# Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General
# Public License along with this program; if not, write
# to the Free Software Foundation, Inc., 59 Temple Place,
# Suite 330, Boston, MA  02111-1307  USA
#

file = "/tmp/phone.txt";

record["name"] = "";
record["phone"] = "";

title = "PHONE BOOK";
copyright = "Copyright(C) 2005 by Roberto Luiz Souza Monteiro.";

boxtitle = "";

#
# Edit a record.
#
function editRecord(getAll = TRUE) {
    clrscr;

    box(1, 1, 80, 24, RED, BLACK);
    box(5, 8, 76, 17, WHITE, CYAN);

    textcolor(RED);
```

```
    textbackground(BLACK);

    gotoxy((80 - length($title)) / 2, 3);
    printf("%s", $title);

    textcolor(BLUE);
    textbackground(BLACK);

    gotoxy((80 - length($copyright)) / 2, 5);
    printf("%s", $copyright);

    textcolor(BLACK);
    textbackground(WHITE);

    gotoxy((80 - length($boxtitle)) / 2, 8);
    printf("%s", $boxtitle);

    textcolor(BLACK);
    textbackground(CYAN);

    gotoxy(10,12);
    printf("Name:");
    gotoxy(10,13);
    printf("Phone:");

    $record["name"] = entry(17, 12, 50, BLACK, WHITE);

    if (!getAll) {
        return(TRUE);
    }

    textcolor(WHITE);
    textbackground(BLACK);

    $record["phone"] = entry(17, 13, 20, BLACK, WHITE);

    textcolor(WHITE);
    textbackground(BLACK);

    gotoxy(10,22);
    printf("Ok( [Y] Yes, [N] No )?");

    while (TRUE) {
        option = getch();

        if ((toupper(option)=="Y") || (toupper(option)=="N")) {
            break;
        }
    }

    if (toupper(option) == "Y") {
```

```
        return(TRUE);
    } else {
        return(FALSE);
    }
}

#
# Show a record.
#
function showRecord() {
    clrscr;

    box(1, 1, 80, 24, RED, BLACK);
    box(5, 8, 76, 17, WHITE, CYAN);

    textcolor(RED);
    textbackground(BLACK);

    gotoxy((80 - length($title)) / 2, 3);
    printf("%s", $title);

    textcolor(BLUE);
    textbackground(BLACK);

    gotoxy((80 - length($copyright)) / 2, 5);
    printf("%s", $copyright);

    textcolor(BLACK);
    textbackground(WHITE);

    gotoxy((80 - length($boxtitle)) / 2, 8);
    printf("%s", $boxtitle);

    textcolor(BLACK);
    textbackground(CYAN);

    gotoxy(10,12);
    printf("Name:");
    gotoxy(10,13);
    printf("Phone:");

    textcolor(RED);

    gotoxy(17,12);
    printf($record["name"]);
    gotoxy(17,13);
    printf($record["phone"]);
}

#
# Insert a new record.
```

```
#
function insertRecord() {
    $record["name"] = "";
    $record["phone"] = "";

    $boxtitle = "New Record";

    if (editRecord()) {
        fseek($fp, 0, SEEK_END);

 fputs(sprintf("%-50.50s%-20.20s\n", $record["name"],
$record["phone"]), $fp);

        fflush($fp);
    }
}

#
# Update a record.
#
function updateRecord() {
    $boxtitle = "Edit Record";

    editRecord(FALSE);

    found = FALSE;

    rewind($fp);

    while (!feof($fp)) {
        data = fread(71, $fp);

        if (length(data) == 71) {
            if (strpos(toupper(range(data, 0, 49)),
toupper($record["name"])) >= 0) {
                $record["name"] = range(data, 0, 49);
                $record["phone"] = range(data, 50, 69);
                found = TRUE;
                break;
            }
        }
    }

    if (found) {
        showRecord;

        textcolor(WHITE);
        textbackground(BLACK);

        gotoxy(10,22);
        printf("Ok( [Y] Yes, [N] No )?");
```

```
        while (TRUE) {
            option = getch();

            if ((toupper(option)=="Y")||(toupper(option)=="N")) {
                break;
            }
        }

        if (toupper(option) == "Y") {
            if (editRecord()) {
                fseek($fp, -71, SEEK_CUR);

                fputs(sprintf("%-50.50s%-20.20s\n",
$record["name"], $record["phone"]), $fp);

                fflush($fp);
            }
        } else {
            return;
        }
    } else {
        textcolor(WHITE);
        textbackground(BLACK);

        gotoxy(10,20);
        printf("Record not found!");
        gotoxy(10,22);
        printf("Press any key to continue...");
        getch;
    }
}

#
# Delete a record.
#
function deleteRecord() {
    $boxtitle = "Delete Record";

    editRecord(FALSE);

    found = FALSE;

    rewind($fp);

    while (!feof($fp)) {
        data = fread(71, $fp);

        if (length(data) == 71) {
            if (strpos(toupper(range(data, 0, 49)),
toupper($record["name"])) >= 0) {
```

```
                $record["name"] = range(data, 0, 49);
                $record["phone"] = range(data, 50, 69);
                found = TRUE;
                break;
            }
        }
    }

    if (found) {
        showRecord;

        textcolor(WHITE);
        textbackground(BLACK);

        gotoxy(10,22);
        printf("Ok( [Y] Yes, [N] No )?");

        while (TRUE) {
            option = getch();

            if ((toupper(option)=="Y")||(toupper(option)=="N")) {
                break;
            }
        }

        if (toupper(option) == "Y") {
            $record["name"] = "";
            $record["phone"] = "";

            fseek($fp, -71, SEEK_CUR);

            fputs(sprintf("%-50.50s%-20.20s\n", $record["name"],
$record["phone"]), $fp);

            fflush($fp);
        } else {
            return;
        }
    } else {
        textcolor(WHITE);
        textbackground(BLACK);

        gotoxy(10,20);
        printf("Record not found!");
        gotoxy(10,22);
        printf("Press any key to continue...");
        getch;
    }
}

#
```

```
# Search record.
#
function searchRecord() {
    $boxtitle = "Search Record";

    editRecord(FALSE);

    found = FALSE;

    rewind($fp);

    while (!feof($fp)) {
        data = fread(71, $fp);

        if (length(data) == 71) {
            if (strpos(toupper(range(data, 0, 49)),
toupper($record["name"])) >= 0) {
                $record["name"] = range(data, 0, 49);
                $record["phone"] = range(data, 50, 69);
                found = TRUE;
                break;
            }
        }
    }

    if (found) {
        showRecord;
    } else {
        textcolor(WHITE);
        textbackground(BLACK);

        gotoxy(10,20);
        printf("Record not found!");
    }

    textcolor(WHITE);
    textbackground(BLACK);

    gotoxy(10,22);
    printf("Press any key to continue...");
    getch;
}

#
# The main program.
#

# Open the table.
try {
    fp = fopen(file, "r+");
} catch {
```

```
    fp = fopen(file, "w+");
}

option = "";

# Show the main menu.
while (TRUE) {
    menutitle = "General Options";

    clrscr;

    box(1, 1, 80, 24, RED, BLACK);
    box(29, 9, 49, 17, WHITE, CYAN);

    textcolor(RED);
    textbackground(BLACK);

    gotoxy((80 - length(title)) / 2, 3);
    printf("%s", title);

    textcolor(BLUE);
    textbackground(BLACK);

    gotoxy((80 - length(copyright)) / 2, 5);
    printf("%s", copyright);

    textcolor(BLACK);
    textbackground(WHITE);

    gotoxy((80 - length(menutitle)) / 2, 9);
    printf("%s", menutitle);

    textcolor(BLACK);
    textbackground(CYAN);

    gotoxy(33, 11);
    printf("[1] NEW");
    gotoxy(33, 12);
    printf("[2] EDIT");
    gotoxy(33, 13);
    printf("[3] DELETE");
    gotoxy(33, 14);
    printf("[4] SEARCH");
    gotoxy(33, 15);
    printf("[5] EXIT");

    textcolor(GREEN);
    textbackground(BLACK);

    gotoxy(10,22);
    printf("Select an option.");
```

```
    option = getch();

    if (option == "1") {
        insertRecord;
    } elseif (option == "2") {
        updateRecord;
    } elseif (option == "3") {
        deleteRecord;
    } elseif (option == "4") {
        searchRecord;
    } elseif (option == "5") {
        # Close the table and exit.
        fclose(fp);

        textcolor(WHITE);
        textbackground(BLACK);

        clrscr;

        break;
    }
}

exit(0);
```