Universidade Federal de Alagoas
Campus A. C. Simões

Ciência da Computação
Compiladores - 2018.1
Carlos Walter de Lima Barbosa Neto
Lucas dos Santos Sales

Especificação da Linguagem de programação CLPL

Maceió - AL
Julho de 2018

Sumário

Estrutura do Programa

A linguagem CLPL é composta por um conjunto de funções, o programa começa na função Init, os programas na linguagem dever ser salvos com extensão .clpl. A estrutura sintática inicial é a seguinte:

```
Init() : int{
    return 0;
}
```

A função Init retorna obrigatoriamente um inteiro 0 caso o programa tenha sido executado com sucesso, se qualquer outra coisa diferente de 0 for retornada houve algum erro ao longo do programa . Seu escopo é definido por '{' inicio do escopo '}' fim do escopo, utilizando ';' como separador. Comentários de linha são definidos apenas por //.

Convenções Léxicas

Em CLPL os identificadores são cadeias de caracteres que devem obrigatoriamente começar com uma letra e podem ser seguidos por letras e números, variáveis podem ser declaradas dentro do escopo de uma função, CLPL é uma linguagem *case sensitive* e nenhum identificador deve ser igual a uma palavra reservada.

As palavras reservadas são:

int float char bool string void true false and or not if else elseif while do for input output return const bitor bitand to rate

Especificações de Tipos

A linguagem CLPL é estaticamente tipada e possui os seguintes tipos:

Tipo	Descrição	Declaração
int	Inteiro de 64 bits, caso nenhum valor inicial seja atribuído o inteiro vai receber o valor 0 por padrão.	int myInt = 12;
float	Ponto Flutuante de dupla precisão de 64 bits, caso nenhum valor inicial seja	float myFloat = 5.6;

	atribuído o ponto flutuante recebe 0 por padrão	
char	Caracter definido por um caractere entre aspas simples da tabela ASCII, caso nenhum valor inicial seja atribuído o tipo char recebe *null* (0 na tabela ASCII) por padrão	char myChar = ‘a’;
bool	Tipo booleano que pode assumir apenas *true* ou *false* como valor ocupa um espaço de 8 bits, caso nenhum valor inicial seja atribuído o booleano recebe *false* por padrão	bool myBool = true;
string	Cadeia de caracteres definidas por aspas duplas, caso nenhum valor inicial seja atribuído a *string*, ela será vazia por padrão	string myString = “CLPL”;
Vetor	Sequência de elementos que possuem o mesmo tipo, pode ser delimitado por chaves ou um valor pode ser atribuído especificando o índice do elemento.	int myVector[3] = {1, 2, 3}; myVector[1] = 2;

Em caso de *overflow* o se o valor transbordar pelo máximo permitido ele irá retornar a contagem no mínimo permitido, em um ciclo, o mesmo acontece em caso de *overflow* pelo menor valor permitido.

Tipos	**Operações suportadas**
int	Atribuição, relacionais, aritméticos
float	Atribuição, relacionais, aritméticos
char	Atribuição, concatenação, relacionais
bool	Atribuição, lógicas, relacionais: ==, !=
string	Atribuição, concatenação, relacionais

Em CLPL só é permitido a coerção de tipo de um inteiro para ponto flutuante, de ponto flutuante para inteiro, no caso desta última o as casas decimais do número são eliminadas, e também do tipo *char* para o tipo *string* caso seja feita uma concatenação entre caracteres.

As constantes são identificadas pela palavra reservada const antes do tipo de dado e a atribuição inicial de um valor é obrigatória. Como na forma abaixo:

```
const int a = 6;
```

CLPL também admite variáveis globais que devem ser declaradas acima de todas as funções declaradas no programa, podem ser declaradas como uma variável normal, entretanto na hora de usar uma variável global ao longo do programa é necessário usar um @ antes da variável para explicitar que essa é uma variável global, dessa forma, também será possível declarar uma variável local com mesmo nome sem que haja conflitos.

```
const float pi = 3.14;

Função():void{
     Comandos;
}

Init() : int {
    int k = @pi;
    return 0;
}
```

Operadores

Os operadores de CLPL são:

Aritméticos	
+	*Adição*
-	*Subtração*
*	*Multiplicação*
/	*Divisão*
%	*Resto da divisão*
- *(unário)*	*Negativo*
^	*Exponenciação*

Relacionais	
>	*Maior que*
>=	*Maior igual que*
<	*Menor que*
<=	*Menor igual que*
!=	*Diferente*
==	*igual*

Lógicos	
true	*Verdadeiro*
false	*Falso*
and	*conjunção*
or	*disjunção*
not	*Negação*
bitor	*Disjunção bit a bit*
bitand	*conjunção bit a bit*

++	Concatenação de cadeia de caracteres
=	Atribuição

Nas operações aritméticas o que define o valor de retorno é a presença ou não de um ponto flutuante. Quando as operações aritméticas estão sendo avaliadas uma operação entre um inteiro e um ponto flutuante se transforma numa operação entre dois pontos flutuantes.

A precedência e associatividade dos operadores é dada abaixo, sendo os primeiros os de ordem mais alta e os últimos os de ordem mais baixa:

()	

not, -(unário)	*Direita para Esquerda*
^	*Esquerda para Direita*
*, /, %	
+, -	
<, <=, =>, >	
==, !=	
and, bitand	
or, bitor	
=	*Direita para Esquerda*

Instruções

Controle Lógico

if
if/else
if/elseif/else

A instrução *if* controla o condicional. Os comandos que formam o corpo da estrutura executadas caso sua expressão lógica seja verdadeira, caso contrário os comandos no corpo de *else* são executadas (caso exista). As sua estrutura sintática é:

```
if(expressão_logica)
{
        Comandos
}
else
{
        Comandos
}
```

Caso seja necessário também é possível usar *if*'s aninhados onde cada estrutura é testada para saber os comandos de quais corpos devem ser executados. Com a sintaxe:

```
if(expressão_logica)
{
        Comandos
}
elseif(expressão_logica)
{
        Comandos
}
[...]
else
{
        Comandos
}
```

A expressão lógica deve obrigatoriamente ser um tipo booleano.

Estruturas de Interação Por Controle lógico:

while

O comando *while* permite executar uma série de comando do seu corpo enquanto sua expressão lógica for verdadeira. Tem a seguinte forma sintática:

```
while(expressão_logica)
{
        Comandos
}
```

O comando também pode ser executado no mínimo uma vez, com a seguinte estrutura sintática:

```
do
{
        Comandos;
```

```
}while(expressão_logica);
```

A expressão lógica deve obrigatoriamente ser um tipo booleano.

Estrutura de Interação Controlada Por Passo:

A estrutura *for* permite que uma instrução seja executada um número de vezes através de um iterador, dado pela seguinte sintaxe:

```
for (var = var1 to var2  rate valor(opcional) )
{
        Comandos;
}
```

Onde "var" é a variável de controle que deve ser declarada previamente antes de ser usada na instrução, "var1 to var2" delimita o intervalo que é fechado entre os dois valores, e o *rate* representa o passo. Se o valor do passo não for especificado ele será tomado como 1.

Em ambos as estruturas de interação variáveis criadas dentro do corpo da instrução são destruídas ao final de cada *loop*. Os valores de var2 e rate são avaliados a cada iteração do loop.

Entrada de dados:

A entrada é pega numa função *input* que retorna o número de caracteres inseridos na entrada, dado pela seguinte sintaxe:

```
input("%var %var...", var1,  var2, … );
```

A *string* do *input* refere-se ao modo como se espera receber os dados, o %var é usado para indicar que naquela posição é esperada uma variável que deve ter o tipo da variável correspondente na sequência de variáveis dadas depois da *string*, o valor do campo será associado na ordem a variável correspondente.

Saída de Dados:

A saída de dados é da dada pela seguinte sintaxe:

```
output(“String %tipo_de_dado ...”, var, ...);
```

A *string* corresponde a formatação na qual a saída vai ser escrita, o % é usado como caractere de escape seguido pelo tipo da variável que é associada na ordem a uma das variáveis passadas para a função.

Os ponto flutuantes, por padrão, terão 2 casas decimais, mas as casas podem ser especificadas colocando um ponto seguindo pelo número de casas no fim do tipo, como a seguir:

```
output(“String %float.3 ...”, var, ...);
```

Caso seja desejado desejado também é possível imprimir os pontos flutuantes em notação científica, a saída irá deixar apenas o primeiro inteiro visível e a quantidade de casas decimais através de uma exponenciação:

```
float var = 0.000000000001
output(“%floatCient ...”, var, ...);
Saida: 1*10^-12
```

No caso dos inteiros é possível especificar quantas casas são desejadas, caso deseje imprimir mais casas do que o número possui as restantes serão preenchidas com 0, caso seja menor que a quantidade de casas do número este será impresso normalmente.

```
int a = 333;
output(“String %.4int ...”, var, ...);
saída: 0333
```

Essa notação também é válida para ponto flutuante, definindo a quantidade de casas na parte inteira da notação.

No caso das *strings* é possível imprimir apenas um “pedaço” dela especificando o intervalo que deseja ser impresso:

```
output(“%String[3-10] ...”, var, ...);
```

Atribuição:

A atribuição é dada pela seguinte sintaxe:

```
a = b;
```

Onde a variável a recebe o valor de b, o retorno de uma atribuição é o valor da variável destino, a atribuição é tanto uma instrução quanto uma operação aritmética.

Funções:

A declaração das funções é dada pela seguinte sintaxe:

```
Nome_da_função(tipo : nome, tipo : nome… ) : Tipo_de_retorno{
    Comandos;
    return valor_retornado (caso necessário);
}
```

As funções devem retornar o tipo de retorno especificado, caso possível o valor de retorno vai ser convertido no tipo de retorno especificado na função, caso não se deseje retornar nada, o tipo de retorno deve ser especificado como *void*. As variáveis são passadas através de cópias para a função.

Declaração

A declaração das variáveis é dada pelo tipo da variável seguida pelo seu nome, em vetores também é necessário especificar o tamanho do vetor, como abaixo:

```
int a;
int a[tamanho_do_vetor];
```

Exemplo de Código

Alô Mundo

```
Init():int{
        output("Alô mundo!\n");
```

```
	return 0;
}
```

Fibonacci

```
fib(int : limite):int{
	int f = 1;
	int anterior = 1;
	int contador = 2;
	if(limite == 0){
		output("0");
		return 1;
	}
	if(limite == 1){
		output(", 1");
		return 2;
	}
	while(f <= limite){
		output(", %int", f);
		contador = contador + 1;
		int temp = f;
		f = f + anterior;
		anterior = temp;
	}
	return contador;
}

Init():int{
	output("Digite o limite:");
	int limite;
	input("%var", limite);
	fib(limite); //retorna quantos elementos da foram impressos
	return 0;
}
```

ShellSort

```
ShellSort(int[] : a, int : tam): int[] {
```

```
        int h = tam/2;
        int i, j;
        while(h >= 1){
                for(i = 0 to  h){
                        j = i;
                        while(j+h < tam){
                                if(a[j] > a[j+h]){
                                        int temp = a[j];
                                        a[j] = a[j+h];
                                        a[j+h] = temp;
                                }
                                j = j+h;
                        }
                }
                h = h-1;
        }
        return a;
}

printArray(int : a[], int : tam):void{
        int i;
        for(i = 0 to tam-1){
                output("%int ", a[i]);
        }
        output("\n");
}

Init():int{
        output("Digite o tamanho: ");
        int tam;
        input(“%var”, tam);
        int a[tam];
        output("Digite os elementos:\n");
        for(i = 0 to tam-1){
                input(“%var”, a[i]);
        }
        output(“Array antes de ordenar:\n”);
        printArray(a, tam);
        output(“Array após ordenar:\n”);
        a = ShellSort(a, tam);
        printArray(a, tam);
```

```
        return 0;
}
```

Especificação dos tokens

A linguagem será implementada na linguagem Python e é composta pelos tokens:

class TokenCategory(Enum):

Id, Init, TypeInt, TypeFloat, TypeBool, TypeChar, TypeString, TypeConst, OpArAd, OpArMult, OpArdiv, OpArMod, OpArExp, OpReD, OpReI, OpLogAnd, OpLogOr, OpLogNot, OpLogBand, OpLogBor, OpConcac, OpAtr, InsSIf, InsSElseif, InsSElse, InsInWh, InsInDo, InsInfor, BeginP, EndP, BeginC, EndC, BeginCh, EndCh, ConstInt, ConstFlaot, ConstBool, ConstChar, ConstString, SepV, SepPV, void, IntTo, IntRate, Out, In, Global, SepPont, Return, EOF = list(range(50))

Auxiliares

ASCII	.

Lexemas

Id	[[:alpha:]]([[:alnum:]])*
Init	“Init”
TypeInt	“int”
TypeFloat	“float”
TypeBool	“bool”
TyperChar	“char”
TypeString	“String”
TypeConst	“const”
opArAd	“+” | “-”
OpArMult	“*”
OpArDiv	“/”
OpArMod	“%”
OpArExp	“^”
OpReD	“>” | ” >=” | ” <” | “<=”

OpReI	“==” | “!=”
OpLogAnd	“and”
OpLogOr	“or”
OpLogNot	“not”
OpLogBand	“bitand”
OpLogBor	“bitor”
OpConcac	“==”
OpAtr	“=”
InsSIf	“if”
InsSElseif	“elseif”
InsSElse	“else”
InsInWh	“while”
InsInDo	“do”
InsInfor	“for”
BeginP	“(“
EndP	“)”
BeginC	“[“
EndC	“]”
BeginCh	“{“
EndCh	“}”
ConstInt	[[:digit:]]+
ConstFloat	[[:digit:]]+’.’[[:digit:]]*
ConstBool	‘true’ | ‘false’
ConstChar	‘‘(ASCII)‘’’
ConstString	‘“’(ASCII)*‘”’

SepV	“,”
SepPv	“;”
void	“void”
IntTo	“to”
IntRate	“rate”
Global	@[[:alpha:]]([[:alnum:]])*
SepPont	‘:’
In	“input”
Out	“output”
Return	“return”
EOF	“”