Linguagem CGVLang

CGVLang é uma linguagem de programação simples de paradigma imperativo desenvolvida pelos acadêmicos Caio Hideki Gomes Shimohiro, Gustavo Alberto Ohse Hanke e Vinícius Visconsini Diniz como trabalho da disciplina de Compiladores no curso de Ciência da Computação da Universidade Estadual do Oeste do Paraná (Unioeste) no ano letivo de 2024.

Este documento se trata de uma especificação/documentação da citada linguagem.

---

1. Tipos de Dados

A linguagem trabalha com tipos de dados estáticos, ou seja, uma vez declarada a variável com um tipo, esse não se modifica no decorrer do código.

A seguir, uma tabela com os tipos presentes na linguagem e suas devidas características.

Tipo	Tamanho	Descrição	Representividade
byte	1 byte	Representa um número inteiro não sinalizado ou um caractere.	Números entre 0 e 255 ou caracteres codificados dentro dessa faixa.
int	8 bytes	Representa um número inteiro sinalizado.	Números inteiros entre 2^{-63} e 2^{63}.
real	8 bytes	Representa um número real sinalizado em notação de ponto flutuante.	Números reais entre $-\ 1.8\ X\ 10^{308}$ e $1.8\ X\ 10^{308}$ com intervalos de no mínimo $4.94\ X\ 10^{-324}$.

---

2. Nome de variáveis

Os identificadores das variáveis devem ser representados por uma sequência de caracteres de a-z, A-Z, números de 0-9 e o símbolo _ (underline). É necessário que o primeiro caractere da sequência do nome seja alfabético para não haver erro com a identificação de um possível número.

---

3. Operadores

3.1. Aritméticos

Símbolo	Descrição
+	Adição
-	Subtração
*****	Multiplicação
/	Divisão
****	Exponenciação
=	Atribuição

Se os operadores forem utilizados entre tipos diferentes, como padrão os dados de tipo de tamanho menor são convertidos para o tipo de tamanho maior. Há uma excessão para a atribuição (=). Para ela, no caso de atribuição de maior para menor, será enviado um warning (aviso), mas o código ainda compilará. Em uma conversão de tipo real para int, enviado um warning de possível perda de precisão, mas também compilará.

Com relação a ordem de precedência dos operadores, é utilizado o padrão PEMDAS (Parênteses -> Exponenciação -> Multiplicação -> Divisão -> Adição -> Subtração)

3.2. Relacionais

Símbolo	Descrição
==	Igualdade
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a

Não existe um tipo booleano, então 0 é considerado como valor **falso** e qualquer outro número é considerado como valor **verdadeiro**.

A ordem de precedência é como segue a tabela de cima para baixo.

A conversão de dados é sempre do menor para o maior.

3.3. Lógicos

Símbolo	Descrição

!	not (Negação)
&&	and (E lógico)
||	or (Ou lógico)

Precedência vale da esquerda para a direita.

Regras gerais dos operadores:

- Para todas as operações, os parênteses têm prioridade na precedência.
- Se misturados operadores lógicos e aritméticos, os aritméticos são executados primeiro

---

4. Entrada e saída de dados

4.1. Entrada

O comando padrão de entrada da linguagem é o get().

Como parâmetro ele recebe um identificador de variável para a qual o valor entrado será atribuído.

Exemplo:

```
int a
get(a)
```

4.2. Saída

O comando padrão de entrada da linguagem é o put().

Como parâmetro ele recebe o dado a ser impresso na tela. Esse dado pode ser de tipo byte ou um int caso seja indicado na frente do número um sinal b para a conversão do tipo int para tipo byte. Só é aceito um caractere por vez.

Exemplo:

```
put('A')
put(10b)
```

---

5. Estruturas de controle

5.1. Estrutura de decisão

A estrutura de decisão padrão da linguagem é o doif(), que recebe como parâmetro a condição de decisão dos blocos. Caso a condição retorne valor verdadeiro, o código procede ao bloco

subsequente ao comando. Caso contrário, pula para o bloco especificado pelo comando maybe_not.

Ambos os blocos de comando devem estar delimitados por chaves {}

Exemplo:

```
int a
a = 10
doif(a < 20)
{
# Comando caso a < 20 for verdadeiro
}

maybe_not
{
# Comando caso a < 20 for falso
}
```

5.2. Laço de repetição

O laço de repetição padrão da linguagem é representado pelo comando do_it_again(), que recebe como parâmetro a condição de parada do laço. Então a repetição dos comandos para apenas quando a condição retornar valor falso.

O bloco de comandos a serem repetidos também é delimitado por chaves.

Exemplo:

```
int a
a = 0
do_it_again(a < 5)
{
# Bloco de comandos a serem repetidos
a = a + 1
}
```

5.3. Laço contado

O laço contado repete os comandos enquanto não for alcançado o valor que condiciona a parada do laço e é representado pelo comando do_it_again_until(). Como parâmetros, recebe um identificador que armazena o valor da contagem, o valor inicial do contador, o valor final e o tamanho do salto. Todos os valores com exceção do identificador são números inteiros.

Exemplo:

```
do_it_again_until(a, 0, 5, 1)
{
# Bloco de comandos a serem repetidos
}
```

6. Palavras reservadas

As palavras reservadas para a linguagem são:

byte	int	real

---

7. Sintaxe

Um pequeno trecho de código para exemplificar melhor como a estrutura da linguagem funciona.

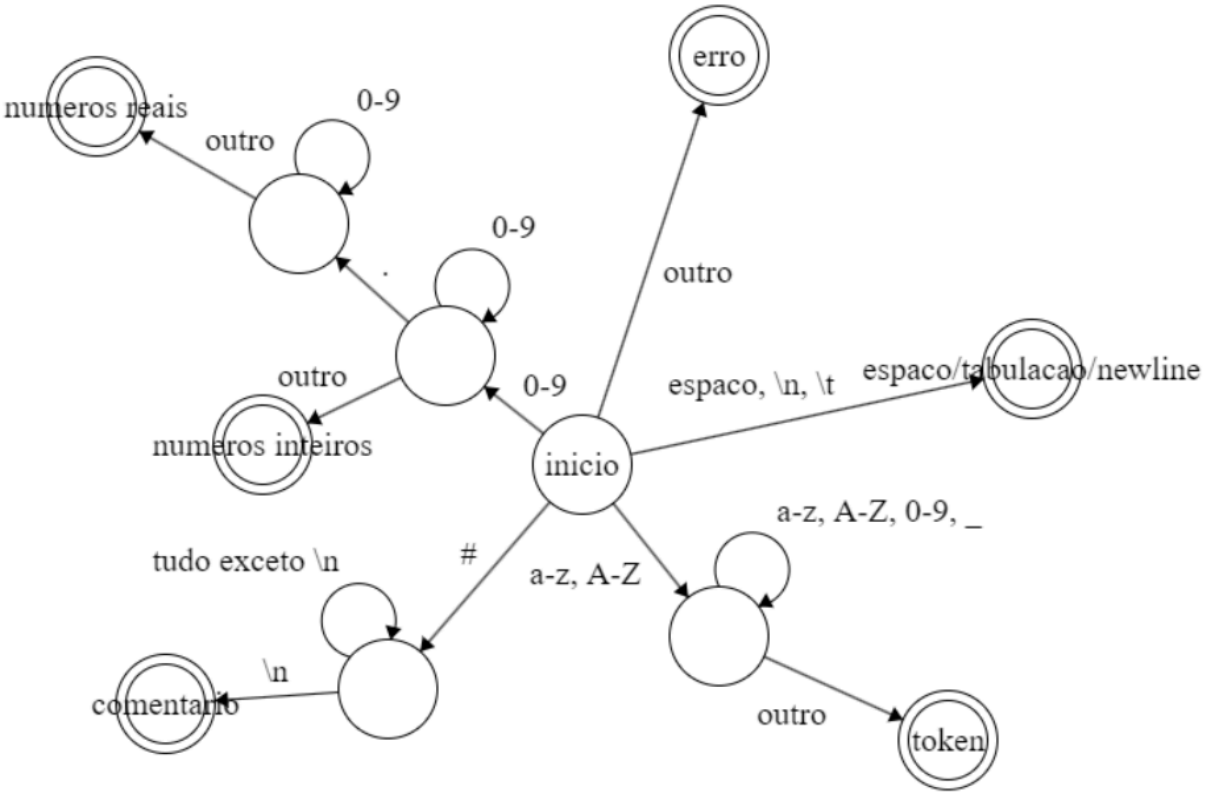
Um detalhe que comentários, marcados pelo símbolo # fazem com que o compilador ignore o que vier após ele até o final da linha.

```
int a
real b

get(a)

b = a + 3.0

doif(b < 10.0)
{
put(‘M’)
put(‘e’)
put(‘n’)
put(‘o’)
put(‘r’)

do_it_again_until(e, 0, 2, 1)
{
put(‘!’)
}
}

maybe_not
{
put(‘N’)
put(‘a’)
put(‘o’)

int f
f = 0
do_it_again(!(f == 2))
{
put(‘!’)
}
```

```
}
```

---

8. Diagrama de autômato



---

9. Aspectos Semânticos

9.1. S

No estado S, ele aceita todos os tokens de funções, abertura de parênteses, id de variável, valor numérico, tipo e fechamento de escopo (fechamento de chave). Apenas no caso das últimas duas que a produção está relacionada com ações sintáticas. Na primeira, a ação armazena o tipo definido no token. Já na segunda, ele exclui o escopo, também retirando-o da tabela de símbolos.

S ->	type TYPEDCL S |	{set_type}
	eof	{end_scope}

9.2. Expr

É a produção do topo das expressões, sendo apenas o ponto inicial para as expressões seguintes em ordem de precedência. Basicamente só consome tokens dos operadores, mas não tem nenhuma ação semântica ligada.

9.3. Expr lógicos, comparativos e aritméticos (exceção not)

Produções geradas uma pela outra seguindo a ordem de precedência dos operadores na sequência de or -> and -> maior/igual -> menor/igual -> maior -> menor -> igual -> soma -> subtração -> multiplicação -> divisão -> potenciação. Cada vez que um valor, id, abre parênteses e um not são lidos, ele executa a ação semântica que cria um nó abaixo de precedência maior até chegar na produção do not e empilha o símbolo em uma stack específica para a expressão.

9.4. Expr lógicos, comparativos e aritméticos (exceção not) rest

São geradas na produção padrão da expressão do operador específico (expr_non_lo_or produz expr_non_lo_or_rest). Eles executam a ação semântica que propaga a expressão do nó. Caso leiam o operador específico desse estado, solucionam a operação.

9.5. Expr not

Lê o not e executa a ação semântica de solucionar a expressão, no caso sendo apenas o not.

9.6. Expr final

Os tokens de valor e id são consumidos e acionam a ação de definir os operandos da expressão.

9.7. Funções

Generalizando para todas as funções, quando consomem o token, acionam a ação específica de cada uma das funções. Como exemplo, get chama a função que define os atributos da variável do id passado.