UNIVERSIDADE FEDERAL DE ALAGOAS

INSTITUTO DE COMPUTAÇÃO

COMPILADORES

# TCNPL

## v6

## Especificação

Lucas Amorim e Nelson Gomes Neto

Professor Alcino Dall'Igna

2018.1

# Sumário

# 1. Especificação da Linguagem

## 1.1 Introdução

A TCNPL é uma linguagem que visa ter alta legibilidade, para isso sacrificando um pouco da facilidade de escrita. Nessa linguagem tentamos tornar o entendimento mais fácil, fazendo com que sentenças sejam legíveis em termos de linguagem natural. Isso, no entanto, não nos impediu de manter uma sintaxe relativamente próxima a C, e por esse motivo, pode-se notar várias referências nessa especificação à essa. Nas seções seguintes especificaremos as funcionalidades dessa linguagem. Para que o entendimento fique mais fácil, fizemos questão de estabelecer algumas convenções para serem usadas durante toda essa especificação.

Convencionamos algumas notações para facilitar o entendimento. Tais convenções são rapidamente descritas a seguir:

- Tipos são sempre descritos pela cor azul;
- Nomes, que não sejam palavras especiais, são sempre descritos pela cor laranja;
- Comentários são descritos pela cor verde;
- Palavras especiais não tem destaque.

A escolha por não destacar palavras especiais vem do fato que a TCNPL não suporta palavras-chave, somente palavras reservadas.

TCNPL é uma linguagem com escopo estático e não suporta o uso de variáveis globais.

## 1.2 Estrutura geral de um programa

Um programa em TCNPL tem a seguinte estrutura geral:

```
function anyFunction() {
      textOut("Any function…")
}


@isEntryPoint
function main() is int {
      a is int
      a = 2
      if (a) {
            anyFunction()
      }
      return 0
}
```

Instruções e declaração de variáveis devem todos serem feitos dentro de blocos de código, ou seja, dentro de trechos envolvidos por chaves. Fora de um bloco, só é permitido declarar funções e utilizar a anotação `@isEntryPoint` que indica qual o ponto inicial de execução, a função de entrada deve obrigatoriamente retornar um valor do tipo inteiro. Mais detalhes sobre funções podem ser encontrados na seção 1.14.

Para comentar uma linha, basta adicionar `//` onde deseja-se iniciar o comentário, exemplos serão mostrados em seções posteriores.

## 1.3 Estruturas de dados

- Inteiro: `int`
- Ponto flutuante: `real`
- Caractere: `char`
- Booleano: `bool`
- Cadeia de caracteres: `string`
- Arranjos unidimensionais: `array`

## 1.4 Constantes literais

- `int`
  - Inteiros de 64 bits (de -9223372036854775808 até 9223372036854775807);
  - Os valores são cíclicos, então em caso de overflow do maior valor inteiro, a contagem continua do menor e vice-versa.
- `real`
  - Ponto flutuante de precisão dupla de 64 bits;
  - Casas decimais separadas por ponto;
  - Em caso de valor muito grande/pequeno, será adotado +infinity/-infinity respectivamente como valores especiais. Esses valores não mudam com nenhum tipo de operação aritmética.
- `char`
  - Inteiros de 8 bits;
  - Podem ser usados caracteres ASCII para representar seus valores;
  - Devem ser representados entre aspas simples;
  - Tratamento de overflow igual ao tipo inteiro.
- `bool`
  - Ocupando um inteiro de 64 bits, e representado por: false (0) e true (diferente de 0).
- `string`
  - Conjunto de caracteres entre aspas duplas.
- `array`
  - Conjunto de elementos de um tipo, separados por vírgula, e com início e fim demarcados por chaves.

### Notação científica

TCNPL suporta representação de inteiros e valores de ponto flutuante através de notação científica. Constantes são representadas dessa forma utilizando um valor inteiro ou real, seguindo imediatamente uma letra ‘e’ e imediatamente um inteiro representando o expoente, conforme o exemplo:

```
var1 is real
var1 = 1.5e20 // var1 = 1.5 x 10^20
```

## 1.5 Tabela de precedência e associatividade

| | | | |
|---|---|---|---|
| 1 | `[]` | Acesso de arranjo | Esquerda para direita |
| | `as` | Conversão de tipo | |
| | `is` | Especificação de tipo | |
| | `of` | Tipo para arranjo | |
| | `()` | Chamada de função / Expressão | |
| 2 | `-` | Subtração unário | Direita para esquerda |
| | `!` | NOT lógico | |
| | `~` | NOT bit a bit | |
| 3 | `**` | Exponenciação | |
| | `*/` | Radiciação | Esquerda para direita |
| 4 | `*` | Multiplicação | |
| | `/` | Divisão | |
| | `%` | Resto de divisão | |
| 5 | `+` | Soma | |
| | `-` | Subtração | |
| 6 | `<<` | Deslocamento bit a bit para esquerda | |
| | `>>` | Deslocamento bit a bit para direita | |
| 7 | `<` | Menor | |
| | `<=` | Menor ou igual | |
| | `>` | Maior | |
| | `>=` | Maior ou igual | |
| 8 | `==` | Igualdade | |
| | `!=` | Desigualdade | |
| 9 | `&` | AND bit a bit | |
| 10 | `^` | XOR bit a bit | |
| 11 | `\|` | OR bit a bit | |
| 12 | `&&` | AND lógico | |
| 13 | `\|\|` | OR lógico | |
| 14 | `=` | Atribuição | Direita para esquerda |
| 15 | `,` | Vírgula | Esquerda para direita |

## 1.6 Operações

| Tipo | Operador | Compatibilidade |
|---|---|---|
| Aditivas | + | Inteiro, ponto flutuante, caractere e cadeia de caracteres, para os dois últimos o operador realiza uma concatenação |
| | - | Inteiro e ponto flutuante |
| Multiplicativas | * | Inteiro e ponto flutuante |
| | / | |
| | % | Inteiro somente |
| Exponenciais | ** | Inteiro e ponto flutuante |
| | */ | |
| Lógicas | != | Inteiro, ponto flutuante, booleano, caractere e cadeia de caracteres |
| | == | |
| | && | Booleano somente |
| | \|\| | |
| | ! | |
| Relacionais | < | Inteiro, ponto flutuante e caractere |
| | <= | |
| | > | |
| | >= | |
| Atribuição | = | Todos os tipos, lado esquerdo recebe o valor do lado direito |

Note que a concatenação de caracteres ou cadeia de caracteres com tipos booleanos ou numéricos faz com que estes últimos sejam convertidos para cadeia de caracteres.

## 1.7 Coerções

TCNPL suporta diversos tipos de coerção, a seguir especificamos e exemplificamos algumas das coerções possíveis:

### Inteiros

TCNPL suporta coerção do tipo inteiro para ponto flutuante, cadeia de caracteres e valores booleanos.

- Ponto flutuante
  TCNPL suporta coerção do tipo inteiro para ponto flutuante, da mesma forma que C:

```
var1 is int
var1 = 1
var2 is real
var2 = var1 // var2 = 1.0
```

- Cadeia de caracteres
  O suporte à coerção para cadeia de caracteres se dá da mesma forma que Java:

```
var1 is int
var1 = 1
var2 is string
var2 = “oi” + var1 // var2 = “oi1”
```

- Valores booleanos
  Mais uma vez, o tratamento de coerções de inteiro para booleano é o mesmo que em C:

```
var1 is int
var1 = 2
var2 is bool
var2 = var1 // var2 = true
var1 = 0
var2 = var1 // var2 = false
```

### Cadeia de caracteres

TCNPL suporta somente um tipo de coerção a partir de cadeia de caracteres: para booleano. Cadeias de caracteres vazias são avaliadas como `false`, cadeias com um ou mais caracteres são avaliadas como `true`:

```
var1 is string
var1 = “something”
var2 is bool
var2 = var1 // var2 = true
var1 = “”
var2 = var1 // var2 = false
```

### Outros tipos

Os outros tipos apresentados podem ser todos convertidos para cadeias de caractere por coerção durante uma concatenação ou atribuição. A seguir uma tabela especificando o que acontece com cada valor ao ser feita coerção para string:

| Tipo a ser feita coerção | Resultado |
|---|---|
| real | O mesmo que utilizar a função `format` com formato “%real” |
| char | string de tamanho um representando aquele caractere |
| bool | string contendo as cadeias “true” ou “false” para os valores `true` ou `false` respectivamente |
| array | string contendo a coleção completa separando cada item por espaço, será feita coerção de cada um dos elementos do arranjo individualmente. |

## 1.8 Declaração de variáveis

Variáveis devem ser declaradas da seguinte forma:

```
name is type
```

Essa forma de declaração visa aumentar a legibilidade, pois fica claro o tipo da variável até para quem não conhece a linguagem.

É possível declarar mais de uma variável de um mesmo tipo na mesma sentença, bastando separar os nomes por vírgula, da seguinte forma:

```
name, name2 is type
```

A seguir alguns exemplos para os tipos mostrados nas seções passadas:

- Inteiro

  name is int

- Cadeia de caracteres

  name is string

- Arranjos

  name is array[size] of type

  Para acessar um índice específico do arranjo, utilizasse a notação name[i]. Para atribuição, faz-se name[i] = var1.

## 1.9 *Type-casting*

TCNPL suporta *type-casting* através do operador `as`. Pode-se usar *type-casting* em qualquer uma das já mencionadas coerções. Por exemplo:

```
var1 is int
var2 is real
var1 = 1 / 2 // var1 = 0
var2 = 1 / (2 as real) // var2 = 0.5
```

## 1.10 Instruções

A seguir serão mostradas as instruções suportadas pela TCNPL, onde lê-se `logical_exp`, entende-se qualquer operação ou atribuição que possa ser realizada coerção para tipo booleano, baseado nas coerções especificadas na seção 1.7. Visto que grande parte das instruções têm funcionamento idêntico a linguagem C, em várias das seções é mostrada uma tabela de correspondência entre a instrução em TCNPL e a correspondente em C.

## Estrutura condicional de uma e duas vias

Estruturas condicionais seguem uma sintaxe e funcionamento semelhante a C. As palavras reservadas são as seguintes:

```
if        elif      else
```

E a sintaxe é a seguinte:

```
if (logical_exp) {
      //code block
} elif (logical_exp) {
      //code block
} else {
      //code block
}
```

A seguir a tabela de correspondência destas instruções:

| Sintaxe na TCNPL | Equivalente em C |
|---|---|
| if | if |
| elif | else if |
| else | else |

## Estrutura iterativa com controle lógico

### *Palavras reservadas*

```
repeat while
```

### *Sintaxe*

A seguir a estrutura iterativa com pré-teste:

```
repeat while (logical_exp) {
      //code block
}
```

E com pós-teste:

```
repeat {
      //code block
} while (logical_exp)
```

E a tabela de correspondência:

| Sintaxe na TCNPL | Equivalente em C |
|---|---|
| repeat while | while |
| repeat {} while | do {} while |

### Estrutura iterativa controlada por contador (com passo igual a um, caso omitido)

*Palavras reservadas*

```
repeat to at
```

*Sintaxe*

```
repeat (name = var to var2) {
      //code block
}
```

Onde `name` é o contador, que deve ser declarado antes da instrução `repeat` e deve ser obrigatoriamente do tipo inteiro. Como o passo não está especificado, ele é igual a 1.

Especificando o passo:

```
repeat (name = var to var2 at rate) {
      //code block
}
```

Onde `rate` é o passo, seguindo as mesmas regras de `name`. A seguir uma breve tabela de correspondência com C:

| Sintaxe na TCNPL | Equivalente em C |
|---|---|
| `repeat (name = var to var2)` | `for (name = var; name <= var2; name++)` |
| `repeat (name = var to var2 at rate)` | `for (name = var; name <= var2; name+=rate)` |

## 1.11 Entrada

Utiliza-se a função `in` para entrada:

```
in("%type1 %type2...", name1, name2, …)
```

O primeiro argumento é a cadeia de formatação, nela o símbolo '%' é utilizado para marcar o começo da especificação de um tipo, que serão recebidos na entrada. Quando a entrada corresponde ao tipo especificado, ele é lido e colocado na variável correspondente. As variáveis são passadas depois da cadeia de formatação e a ordem de leitura é a ordem que essas foram passadas. Portanto, o tipo deve ser sempre compatível com o especificado na cadeia de formatação.

## 1.12 Saída

Utiliza-se a função `textOut` para saída:

```
textOut("string")
```

A função suporta apenas cadeias de caractere como primeiro argumento. Para obter saída formatada, utiliza-se a função `format`, detalhada na subseção seguinte.

## 1.13 A função `format`

A função format funciona passando-se uma cadeia de formatação como primeiro argumento e em seguida os valores, da mesma forma que a função `in`:

```
format(formatstring, var1, var2,…)
```

O retorno da função é uma cadeia de caracteres com a formatação e valores especificados.

Na tabela a seguir são representadas as diversas opções que podem ser utilizadas na cadeia de formatação:

| Formato | Significado | Exemplos |
|---|---|---|
| %tipo | Imprime o valor sem tratamento especial de espaço ou tamanho. Para ponto flutuante, o valor é formatado com duas casas decimais. | Formatando uma cadeia de caracteres, sem tratamento especial:<br><br>format(“string: %string”, “oi”)<br><br>Que retorna: “string: oi” |
| %tipo[inteiro][caractere] | Especifica o valor da variável no tamanho do inteiro especificado, onde o espaço que falta é completado pelo caractere especificado. | Para imprimir uma cadeia de caracteres com espaços como preenchimento:<br><br>format(“string: %string[5][ ]”, “oi”)<br>Que resulta em: “string:  oi”<br><br>Ou seja, 3 espaços antes da cadeia para completar o tamanho requisitado de 5. |
| %.[inteiro]real | Especifica o número máximo de casas decimais a serem impressas para um valor de ponto flutuante. | format(“out: %.[3]real”, 5.12345)<br>Que resulta em: “out: 5.123” |
| %[sci] | Representa o valor em notação científica. Pode ser usado com todos os outros formatos acima, com valores de tipos numéricos (int ou real). No caso em que for usado com um dos dois últimos formatos, este será aplicado à mantissa. | format(“out: %[sci]int”, 12345678)<br>Que resulta em “out: 1.2345678e7” |

Pode-se também passar um parâmetro extra para a função `textOut` para delimitar o número de caracteres por linha. Ao especificar tal número, a função insere uma quebra de linha cada vez que o número especificado é atingido durante a impressão, a seguir um exemplo dessa funcionalidade, note que TCNPL não suporta uma única instrução separada em múltiplas linhas, isso se deu devido ao espaço insuficiente nesta especificação para mostrar a instrução em uma linha somente:

```
textOut(“Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Mauris non nisi id augue tristique aliquam.”, 80)
```

Que limita o número de caracteres a 80 por linha. O resultado da impressão ficaria da seguinte forma:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris non nisi id augu
e tristique aliquam.
```

## 1.14 Funções

Funções na TCNPL têm uma estrutura muito semelhante a C, as diferenças ficam com o suporte a retorno múltiplo e algumas poucas diferenças de sintaxe herdadas dos métodos de declaração já apresentados. As palavras reservadas no âmbito de funções que ainda não foram apresentadas são as seguintes:

```
function return @isEntryPoint
```

Funções são especificadas conforme trecho de exemplo a seguir:

```
function name(args…) is type {
      //code block example
      var1 is type
      return var1
}
```

Onde "type" é o tipo de retorno da função. O bloco de código deve conter obrigatoriamente a instrução `return` seguida de um valor compatível com o tipo especificado. Funções de entrada podem ser declaradas anotando-as com "@isEntryPoint".

Caso não haja necessidade de retorno, pode-se omitir por completo a declaração de tipo, além disso, não há necessidade da instrução return:

```
function name(args…) {
      textOut("Ok")
}
```

Funções em TCNPL suportam o modelo semântico de entrada e saída através da abordagem de passagem de parâmetros por valor-resultado.

## 1.15 Exemplos de código

### Alô Mundo!

```
@isEntryPoint
function helloWorld() is int {
    textOut("Alô mundo!")
    return 0
}
```

### Série de Fibonacci

```
function fibonacci(limit is int) {
    if (limit < 0) {
        return
    }
    a, b, aux is int
    a = 0
    b = 1
    textOut(format("%int", a))
    repeat while (a <= limit) {
        aux = a
        a = b
        b = aux + b
        if (a <= limit) {
            textOut(format(", %int", a))
        }
    }
}

@isEntryPoint
function main() is int {
    limit is int
    in(“%int”, limit)
    fibonacci(limit)
    return 0
}
```

Shell Sort

```
function shellSort(size is int, arr is array[] of int) {
    h is int
    h = size / 3
    repeat while (h > 0) {
        i is int
        repeat (i = h to (size - 1)) {
            temp, j is int
            temp = arr[i]
            repeat while (j >= h && arr[j - h] > temp) {
                arr[j] = a[j - h]
                j = j - h
            }
            arr[j] = temp
        }
        h = h / 2
    }
}

function printArray(size is int, arr is array[] of int) {
    i is int
    repeat (i = 0 to size - 1) {
        textOut(format("%int", arr[i]))
    }
}

@isEntryPoint
function main() is int {
    i, size is int
    in("%int", size)
    arr is array[size] of int
    repeat (i = 0 to size - 1) {
        in("%int", arr[i])
    }
    printArray(size, arr)
    shellSort(size, arr)
    printArray(size, arr)
    return 0
}
```

# 2. Especificação dos tokens

Os analisadores léxico e sintático serão implementados na linguagem Python.

## 2.1 Enumeração

```
class TokenCategory(Enum):
  id, typeBool, typeInt, typeReal, typeChar, typeString, \
  typeArray, asCast, isType, of, bool, int, real, scynot, char, \
  string, repeat, whileLoop, to, at, ifSel, \
  elifSel, elseSel, opParen, clParen, function, returnFun, \
  entryPoint, opBraces, clBraces, opBrackets, clBrackets, \
  unary,  exp, mult, plus, minus, bitShift, relational, eqOrDiff,\
  bitAnd, bitOr, logicAnd, logicOr, attrib, comma, unknown =
list(range(47))
```

A definição e significado de cada símbolo são especificados na seção 2.2.

## 2.2 Terminais

| | |
|---|---|
| id | [[:alpha:]](_\|[[:alnum:]])* |
| Tipos | |
| typeBool | "bool" |
| typeInt | "int" |
| typeReal | "real" |
| typeChar | "char" |
| typeString | "string" |
| typeArray | "array" |
| Especificadores de Tipos | |
| asCast | "as" |
| isType | "is" |
| of | "of" |
| Constantes Literais | |
| bool | "true"\|"false" |
| int | [[:digit:]]+ |
| real | [[:digit:]]+"."[[:digit:]]* |
| scynot | {real}e{int} |
| char | """(\\.\|[^"\\])?""" |
| string | \"(\\.\|[^"\\])*\" |
| Iteração | |
| repeat | "repeat" |
| whileLoop | "while" |
| to | "to" |
| at | "at" |
| Seleção | |
| ifSel | "if" |
| elifSel | "elif" |
| elseSel | "else" |
| Função / Expressão | |

| | |
|---|---|
| opParen | "(" |
| clParen | ")" |
| Função | |
| function | "function" |
| returnFun | "return" |
| entryPoint | "@isEntryPoint" |
| Bloco | |
| opBraces | "{" |
| clBraces | "}" |
| Acesso de arranjos | |
| opBrackets | "[" |
| clBrackets | "]" |
| Operadores | |
| unary | "!"\|"~" |
| exp | "**"\|"*/" |
| mult | "*"\|"/"\|"%" |
| plus | "+" |
| minus | "-" |
| bitShift | "<<"\|">>" |
| relational | "<"\|"<="\|">="\|">" |
| eqOrDiff | "=="\|"!=" |
| bitAnd | "&" |
| bitOr | "\|" |
| logicAnd | "&&" |
| logicOr | "\|\|" |
| attrib | "=" |
| Separador | |
| comma | "," |