UNIVERSIDADE FEDERAL DE ALAGOAS

INSTITUTO DE COMPUTAÇÃO

COMPILADORES

TCNPL

v5

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;
  - Notação científica é permitida, bastando escrever "real"e"int", assim, o número real será multiplicado por 10 elevado ao int especificado.
  - 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`
  - Toda a tabela ASCII, ocupando assim 8 bits;
  - 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	Esquerda para direita
	`*/`	Radiciação	
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

- Aditivas: +, -
  - Inteiro, ponto flutuante, booleano, caractere e cadeia de caracteres;
  - No caso de uma cadeia de caracteres: + (concatenação)
    - E “qualquer tipo” + “cadeia de caracteres” = cadeia de caracteres.
- Multiplicativas: *, /, %
  - Inteiro, ponto flutuante, booleano e caractere;
  - % é exclusivo para inteiro e caractere.
- Exponenciais: **, */
  - Inteiro, ponto flutuante, booleano e caractere.
- Lógicas: !=, ==, &&, ||, !
  - Inteiro, ponto flutuante, booleano, caractere e cadeia de caractere.
- Relacionais: <, <=, >, >=
  - Inteiro, ponto flutuante, caractere e cadeira de caractere.
- Bit a bit: &, |, ~, ^, <<, >>
  - Inteiro, ponto flutuante, booleano e caractere.
- Atribuição: =
  - Qualquer tipo suporta atribuição;
  - Também é possível fazer múltiplas atribuições, bastando separar por vírgula os nomes e valores do lado esquerdo e direito;
  - O lado esquerdo recebe o(s) valor(es) do lado direito.

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, com o diferencial de funcionar mesmo em atribuições:

```
var1 is int
var1 = 1
var2 is string
var2 = var1 // var2 = “1”
```

- 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 chamar `format(number)`
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, seguindo todas as regras da seção 1.13

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
```

Ou mesmo declarar numa mesma sentença múltiplas variáveis de tipos diferentes:

```
name, name2 is type1, type2
```

Essa notação também é usada para funções de retorno múltiplo. Mais detalhes serão dados em sua respectiva seção.

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, var2 is int, 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 `lineIn` para entrada:

```
lineIn()
```

A função retorna a próxima linha da entrada em uma `string`, ou seja, todos os caracteres até o primeiro "\n" do *stream* de entrada.

Frequentemente existe a necessidade de obter entrada formatada, para isso TCNPL disponibiliza a função `split`, que aceita uma `string` e um caractere como argumento, o retorno será um `array` da `string` passada, segmentada toda vez que o caractere passado ocorre na `string`, por exemplo:

```
ent is string
splitted is array[3] of string
ent = lineIn() // ent = "Uma string qualquer"
splitted = split(ent, ' ')
// splitted = ["Uma", "string", "qualquer"]
```

1.12 Saída

Utiliza-se a função `textOut` para saída literal:

```
textOut(arg is string)
```

Para obter uma saída formatada para ponto flutuante, pode-se usar a função `format`, passando como argumentos o valor a ser formatado e o número de casas decimais.

Para ponto flutuante, por exemplo, pode-se fazer da seguinte forma para imprimir o valor de "num" formatado em 3 casas decimais.

```
textOut("Out: " + format(num, 3))
```

Não passando o segundo argumento para a função `format`, o valor padrão 2 é utilizado.

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.
```

1.13 As funções `format` e `textOut` em casos não ideais

Nesta seção descrevemos um pouco mais a respeito do funcionamento da função `format` e `textOut`.

Como já mostrado na seção 1.12, a função `format` tem o objetivo de formatar um determinado valor de ponto flutuante para uma `string` seguindo o formato especificado.

Em condições ideais, ou seja, onde o número formatado não excede 8 dígitos totais e não for menor que o valor mínimo representável pelo número de casas decimais solicitadas (underflow), o valor formatado é exatamente como descrito na seção anterior.

As condições para o caso de passar a variável diretamente para a função `textOut`, sem intermédio da função `format`, são descritas a seguir:

Tipo de dado	Condição
`int`	Não exceder 8 dígitos totais
`real`	Mesmas condições da função `format`

Em casos que violem uma das condições acima, serão tomadas medidas para tornar o número apresentável. Tais medidas são descritas a seguir:

Violação	Ação
excedeu 8 dígitos totais	Valor será convertido para notação científica
underflow (somente real)	Será mostrado o valor mais próximo dado as limitações de casas decimais.

A primeira violação pode ser demonstrada pelo seguinte exemplo:

```
var1 is int
var1 = 100000000
textOut(var1) //exibe 1e9
```

Já a segunda violação, pelo seguinte exemplo:

```
var1 is real
var1 = 0.000001
textOut(var1) //exibe 0.00, pois é a melhor aproximação
//usando duas casas decimais para o valor de var1
```

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”)
}
```

Caso se queira fazer retorno múltiplo, basta especificar os diferentes tipos a serem retornados na declaração da função, assim como na declaração múltipla:

```
function name(args…) is type1, type2 {
      //code block example
      var1 is type1
      var2 is type2

      return var1, var2
}
```

O retorno múltiplo deve ser associado a um recebimento múltiplo por parte do chamador:

```
name1, name2 is int
name1, name2 = name() // função que retorna 2 valores do tipo inteiro
```

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(a)
	repeat while (a < limit) {
		aux = a
		a = b
		b = aux + b
		textOut(", " + a)
	}
}

@isEntryPoint
function main() is int {
	limit is int
	limit = lineIn() as int
	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
  }
}

@isEntryPoint
function main() is int {
      i, size is int
      size = lineIn() as int
      arr is array[size] of int
      line is array[size] of string
      line = split(lineIn(), ' ')
      repeat (i = 0 to size - 1) {
            arr[i] = line[i] as int
      }
      shellSort(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, additive, bitShift, relational, eqOrDiff,\
  bitAnd, bitOr, logicAnd, logicOr, attrib, comma = list(range(45))
```

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	"*"|"/"|"%"
additive	"+"|"-"
bitShift	"<<"|">>"
relational	"<"|"<="|">="|">"
eqOrDiff	"=="|"!="
bitAnd	"&"
bitOr	"|"
logicAnd	"&&"
logicOr	"||"
attrib	"="
Separador	
comma	","