Descrição da linguagem para a turma COMPILADORES/2019.1-2

Esta linguagem é case sensitive tanto para palavras reservadas quanto para identificadores. Logo, “RECEBA” é uma palavra reservada, mas “receba” não é. Além disso, “abcd”, “ABCD” e “aBCd” representam identificadores diferentes.

O compilador só deve aceitar arquivos contendo caracteres com valores na tabela ASCII entre 9 e 10 ou entre 32 e 126 (caracteres imprimíveis da tabela ASCII). Portanto, caracteres fora desta escala não são considerados nesta especificação.

1. Palavras e símbolos reservados:

```
ATEH BIT DE ENQUANTO ESCREVA FIM FUNCAO INICIO INTEIRO LEIA NULO PARA
PARE REAL RECEBA SE SENAO VAR VET . : ; <- + - * / % ** ( ) [ ] < >
<= >= = <> " & | !
```

1.1. Separadores

Os tradicionais espaço em branco, tab e quebra de linha são caracteres que indicam o fim de uma possível sequência de caracteres que formam um único elemento, a menos que apareçam em uma constante do tipo string (ver item 2.3).

2. Constantes e identificadores

2.1. Identificador:

$$[a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^*$$

Todo identificador é iniciado por uma letra maiúscula ou minúscula seguida por zero ou mais letras maiúsculas/minúsculas ou dígitos. O tamanho máximo é de 512 caracteres.

Exemplos: `a abCD x0 a0B1c7 c111`

2.2. Constante numérica:

$$[0\text{-}9]^+$$
$$[0\text{-}9]^+,[0\text{-}9]^*$$

Um ou mais dígitos formam uma constante numérica inteira. Um ou mais dígitos seguidos por vírgula e zero ou mais dígitos formam uma constante numérica real. O tamanho máximo é de 512 caracteres. Uma constante númerica não pode ser imediatamente seguida por letras maiúsculas/minúsculas ou vírgulas.

Exemplos: `0123 1234567890 123, 123,456`

2.3. Constante do tipo string:

Sequência de caracteres imprimíveis da tabela ASCII (exceto aspas duplas) entre aspas duplas. O

tamanho mínimo é de 0 e tamanho máximo é de 512 caracteres entre as aspas duplas.

Ex: `"abc" "A saida esperada eh: " "\o/"`

3. Estrutura de um programa

Todo programa deve ser a seguinte estrutura:

```
* declaração de variáveis globais aqui *

* protótipos de função aqui *

* funções aqui *

VAR
  * declaração de variáveis aqui *
INICIO
  * bloco de código aqui *
FIM
```

Todas as variáveis globais devem ser declaradas antes dos protótipos de função. Todos os protótipos de função devem ser listados antes do código de qualquer função. Após os protótipos, segue-se o código de todas as funções, e então o programa principal.

Um programa principal é separado em duas partes, a primeira contendo todas as declarações de variáveis, e a segunda composta por qualquer quantidade dos demais comandos abaixo, em qualquer ordem.

3.1. Declarações:

As declarações devem ser feitas da seguinte forma:

```
$tipo$ : $lista_ids$ .
VET $tipo$ : $id$ $tam$ .
```

$tipo$ pode receber os valores **INTEIRO** (inteiro de 64 bits em complemento de dois), **REAL** (ponto flutuante de 64 bits seguindo o padrão IEEE-754) e **BIT** (1 bit representando verdadeiro e falso). $lista_ids$ é uma lista de identificadores (ver item 2.1) separados por ponto-e-vírgula. id é um identificador. tam é uma constante numérica inteira (ver item 2.2). Vetores devem ser declarados um de cada vez. $lista_ids$ deve conter ao menos um item.

Exemplos:

```
INTEIRO : a; b;c  ; d .
VET REAL:v 100.
BIT:please.
```

3.2. Atribuição

Segue o padrão para atribuição:

```
$id$ <- $expr_arit$ .
$id$[$expr_arit$] <- $expr_arit$ .
```

No primeiro caso, id é o identificador da variável de destino, e $expr_arit$ é uma expressão aritmética que definirá o valor da variável. O segundo caso mostra a variável de destino sendo indexado caso seja um vetor.

Exemplos:

```
Pi      <-   3,14159     .
bit[233]<-0.
```

3.3. Expressões aritméticas

Expressões aritméticas são zero ou mais operações sobre variáveis dos tipos **INTEIRO** e/ou **REAL**, posições de vetores desses tipos e constantes numéricas utilizando-se operadores aritméticos (“+”, “-”, “*”, “/”, “%” e “**”) e parêntesis para alterar a precedência. Entre os operadores, “**” é o que tem maior precedência, seguido por “*”, “/” e “%” que possuem a mesma precedência, que por sua vez é superior à precedência dos operadores “+” e “-”, que possuem a mesma precedência. O operador “-” pode ainda ser utilizado para mudar o sinal de expressões aritméticas (com precedência do operador “**”). No caso de operações com a mesma precedência, a ordem de resolução segue da direita para a esquerda. Por fim, o operador “%” só funciona quando ambos os operandos forem inteiros, e o operador “**” só funciona quando o segundo operando for inteiro. Chamadas de função (ver item 5) podem ser utilizadas em expressões aritméticas. O tipo REAL tem precedência maior do que o tipo INTEIRO na resolução de expressões aritméticas. IMPORTANTE: Expressões aritméticas não são um comando por si só, elas apenas fazem parte de outros comandos (ex: Atribuição).

Exemplos:

```
a
-a[0]
(a)-b[c+d]
a+345/b
-(1)
a+b*c
a**(-b%c)
a/10**b(5;23)
```

3.4. Expressões relacionais

$expr_rel$ é uma expressão relacional, que compara duas expressões aritméticas através de um operador relacional (“<”, “>”, “<=”, “>=”, “=” e “<>”). Uma variável do tipo BIT também é considerada uma expressão relacional. IMPORTANTE: Expressões relacionais não não um comando por si só, elas apenas fazem parte de outros comandos (ex: Desvio condicional).

Exemplos:
```
$expr_arit$ = $expr_arit$
$expr_arit$ <> $expr_arit$
```

3.5. Expressões relacionais compostas

Os símbolos **&** e **|** podem ser utilizados para unir uma ou mais expressões relacionais formando uma expressão composta $expr_comp$, sendo que **&** tem maior precedência do que **|**, parêntesis podem ser utilizados para alterar a precedência, e em operações com a mesma precedência, a ordem de resolução segue da direita para a esquerda. O operador **!** pode ser usado para inverter o resultado de uma $expr_comp$. **!** tem maior precedência do que **&** e **|**.

Exemplos:
```
$expr_rel$ | $expr_rel$ & $expr_rel$
( $expr_rel$ | $expr_rel$ ) & ! $expr_rel$
```

3.6. Desvio condicional

Um desvio condicional segue o seguinte padrão:

```
SE $expr_comp$
INICIO
  * bloco de código aqui *
FIM
SENAO
INICIO
  * bloco de código aqui *
FIM
```

Um desvio condicional sempre possui um bloco **SE**, seguido ou não por um bloco **SENAO**. Cada bloco tem 0 ou mais comandos e sempre é delimitado por **INICIO** e **FIM**.

Exemplo:
```
SE a > b
INICIO
  b <- a.
FIM
```

3.7. Repetição

A primeira estrutura de repetição segue o padrão:

```
ENQUANTO $expr_comp$
INICIO
  * bloco de código aqui *
FIM
```

Nela, o bloco de comandos é executado enquanto $expr_comp$ for verdadeira. A segunda estrutura

segue o seguinte padrão:

```
PARA $id$ DE $expr_arit$ ATEH $expr_arit$
INICIO
  * bloco de código aqui *
FIM
```

Nesse caso, a estrutura recebe uma variável inteira, seguida de um valor inicial e um valor final (ambos inclusos no intervalo) para a contagem, que é realizada em incrementos de um.

Exemplos:

```
a=0;
ENQUANTO a < 10
INICIO
  b <- a.
  a <- a+1.
FIM

PARA a DE 0 ATEH 9
INICIO
  b <- a.
FIM
```

O comando **PARE** pode ser utilizado para parar a execução de uma repetição.

3.8. Leitura e escrita

A leitura é realizada da seguinte forma:

```
LEIA $lista_ids$ .
```

A função **LEIA** é inteligente o suficiente para ler um único inteiro caso a variável seja do tipo **INTEIRO**, um número com vírgula caso a variável seja um do tipo **REAL**, ou 0 ou 1 caso a variável seja do tipo **BIT**. Pode-se ler uma posição de um vetor com essa mesma função. $lista_ids$ deve conter ao menos um item. Para impressão, segue-se o padrão:

```
ESCREVA $lista_elems$ .
```

A função **ESCREVA** imprime o resultado de expressões aritméticas, variáveis e constantes separadas por ponto-e-vírgula. $lista_elems$ deve conter ao menos um item.

Exemplos:

```
LEIA a.
LEIA a; b; c.
ESCREVA a+b.
ESCREVA "abc";a  ;  123,5;abc.
```

4. Protótipos de função

O cabeçalho de todo código é composto por protótipos de função que seguem o seguinte modelo:

```
FUNCAO $id$ ( $lista_vars$ ) : $tipo$ .
```

$lista_vars$ é a lista de parâmetros da função, sendo vários pares ao estilo $tipo$:id separados por ponto-e-vírgula. O $tipo$ pode ser seguido por **VET** caso o parâmetro seja um vetor. $tipo$, além de **INTEIRO**, **REAL** e **BIT**, também pode ser **NULO** no tipo de retorno de funções. Parâmetros são cópias dos valores passados na chamada de função, a menos que sejam vetores. Vetores são passados por referência, logo a função alterará o conteúdo original do vetor.

Exemplos:

```
FUNCAO sqrt(REAL:n):REAL.
FUNCAO ordena(VET INTEIRO:v; INTEIRO:n):NULO.
```

5. Funções

Toda função deve ser a seguinte estrutura:

```
FUNCAO $id$ ( $lista_vars$ ) : $tipo$
VAR
  * declaração de variáveis aqui *
INICIO
  * bloco de código aqui *
FIM
```

Uma função segue a mesma estrutura do programa principal, com separação de declaração de variáveis e demais comandos. O retorno de uma função é dado pelo comando **RECEBA**:

```
RECEBA ( $expr_arit$ ) .
RECEBA .
```

Caso o tipo de retorno seja **NULO**, pode-se usar **RECEBA** sem valor para encerrar a execução da função. **RECEBA** não pode ser utilizado no programa principal. A presença do comando RECEBA no programa principal deve ser considerada um erro semântico.

Uma chamada de função pode ser um comando por si só, onde o retorno é ignorado caso ele não seja **NULO**. Os parâmetros de uma chamada de função devem ser consistentes com a declaração da mesma.

Exemplo:

```
ordena(idades;num_individuos).
```