INSTITUTO DE INFORMÁTICA UFG

# SOFTWARE BÁSICO
# TRABALHO EM GRUPO
# 2021/2

UFG

O trabalho se baseia na implementação de um tradutor de uma linguagem simples (chamada de *BPL – Bruno's Programming Language*) para Assembly.

## 1. Regras do Trabalho

- A data de entrega do trabalho será no dia **27 de março de 2022, 23h59**.
- O trabalho deve ser feito em grupo de 2 alunos.
- Qualquer plágio (total ou parcial) implicará em nota zero para todos os envolvidos.
- Os grupos deverão apresentar o trabalho para o professor, em horários agendados. Caso o grupo não apresente, o trabalho terá nota zero.
- O grupo deve entregar um **arquivo .zip** (não .rar, .tgz, etc.) contendo o código fonte.
- O tradutor deve ser escrito na linguagem C e será compilado e testado no ambiente Linux.
- Não é permitido o uso de bibliotecas de terceiros para reconhecimento de padrões ou gramáticas.
- A tradução deve seguir as regras da plataforma AMD64 no sistema Linux, como estudado na disciplina:
  - Regras de tradução de funções, condicionais, operações aritméticas e arrays.
  - As variáveis locais devem ser alocadas obrigatoriamente na pilha (incluindo os arrays).
  - As variáveis locais e a pilha devem estar alinhadas.
  - Deve-se seguir as regras de passagem de parâmetros e retorno.
  - Deve-se seguir as regras de salvamento de registradores (*caller-saved* e *callee-saved*).
- O tradutor deve ler um arquivo em BPL da entrada padrão (e.g., usando o *scanf*) e imprimir a tradução desse programa em Assembly na saída padrão (e.g., usando o *printf*). Pode-se criar um arquivo com a linguagem e utilizar o redirecionamento para testar:

```
$ ./tradutor < prog.blp
```

- Caso omissos devem ser tratados com o professor. Não tente assumir qualquer coisa se estiver com dúvida.

# 2. Descrição da Linguagem

A linguagem é baseada na definição de funções (uma ou várias). As funções sempre retornam algum valor.

## 2.1. Definição de Função

A definição de função inicia com a palavra-chave **`function`**, seguido pelo nome da função e depois de zero a três parâmetros. O nome da função segue o padrão **`fN`**, onde ***`N`*** é um índice único começando de 1 (1, 2, 3, etc.).

Os parâmetros podem ser um valor inteiro (*int*) ou um ponteiro para um array de inteiro (*int**). O nome do parâmetro inteiro segue o padrão **`piN`**, onde ***`N`*** é um índice que indica se é o primeiro (1), segundo (2) ou terceiro (3) parâmetro. O nome do parâmetro array inteiro segue o padrão **`paN`**, onde ***`N`*** é um índice que indica se é o primeiro (1), segundo (2) ou terceiro (3) parâmetro.

Assuma que os índices dos parâmetros sempre estarão corretos.

Exemplo:

```
function f1
def
enddef
return ci0
end

function f2 pi1
def
enddef
return ci0
end

function f3 pa1 pi2 pa3
def
enddef
return ci0
end
```

## 2.2. Variáveis Locais

Uma função pode ter até cinco variáveis locais, as quais podem ser variáveis inteiras (*int*) ou arrays de inteiros (*int*[]). Tanto as variáveis inteiras ou os arrays inteiros devem ser alocados na pilha e não possuem valor inicial (têm "lixo" de memória).

As variáveis são definidas uma por linha, dentro de um bloco **`def-enddef:`**

```
def
  ...
enddef
```

O bloco de definição de variável é obrigatório, mas pode ser vazio se não houver variáveis locais a serem definidas.

A definição das variáveis inteiras iniciam com a palavra-chave **`var,`** seguida do nome da variável. O nome segue o padrão **`viN`**, onde ***`N`*** é um índice de identificação, por exemplo, **`vi1`** ou **`vi3`**.

A definição dos arrays inteiros iniciam com palavra-chave **`vet`**, seguida do nome do array, seguido da palavra-chave **`size`** e então uma constante inteira informando o tamanho do array. O nome do array segue o padrão **`vaN`** (por exemplo, **`va3`** ou **`va5`**).

Uma constante inteira tem o formato **`civ`**, onde ***`v`*** é o valor da constante, por exemplo, **`ci5`** (5), **`ci-15`** (-15), **`ci1024`** (1024), **`ci-8273`** (-8273), etc. No caso do tamanho dos arrays, sempre deve ser uma constante positiva não nula (> 0).

Não haverá variáveis com o mesmo índice dentro de uma mesma função, não importando que as variáveis sejam inteiras ou arrays. Os índices serão incrementais iniciando de 1 para cada função.

Exemplo:

```
function f1
def
enddef
return ci0
end

function f2 pi1
def
var vi1
vet va2 size ci30
var vi3
enddef
return ci-1
end

function f3 pa1, pi2
def
vet va1 size ci10
vet va2 size ci20
var vi3
enddef
return ci5
end
```

### 2.3. Corpo da Função

O corpo da função é um conjunto de comandos que inicia depois da definição das variáveis. Um comando pode ser (i) atribuição de variável inteira, (ii) alteração de uma posição do array, (iii) recuperação de valor de uma posição de um array, (iv) condicional **`if`** ou (v) returno de um valor.

#### 2.3.1. Atribuição

Uma atribuição de variável inteira pode ser uma atribuição simples, uma expressão ou o retorno de uma chamada de função. Uma atribuição simples pode ser uma variável inteira recebendo o valor de outra variável, um parâmetro inteiro ou uma constante. Uma expressão pode ser as operações de soma, subtração, multiplicação ou divisão, sendo que os operandos podem ser variáveis inteiras, parâmetros inteiros ou constantes (não pode ter chamada de função ou arrays na expressão). Por fim, uma variável inteira pode receber o retorno de uma chamada de função.

Exemplo:

```
function f1 pi1
def
var vi1
var vi2
enddef
vi1 = ci1                               # vi1 = 1
vi2 = vi1                               # vi2 = vi1
vi1 = pi1 + vi2                         # vi1 = pi1 + vi2
vi2 = vi1 * ci-5                        # vi2 = vi1 * -5
return vi1
end
```

Obs: A linguagem não tem comentários, eles foram colocados no exemplo como forma de esclarecimento.

#### 2.3.2. Chamada de Função

As chamadas de função são feitas utilizando a palavra-chave **`call`** seguida do nome da função. Depois do nome da função são passados os parâmetros para função a ser chamada (até três parâmetros). Se a função recebe um parâmetro inteiro, pode-se passar o valor de uma variável inteira, um parâmetro inteiro ou uma constante inteira. Se o parâmetro for um ponteiro para array,

pode-se passar um array local (o tradutor deve obter o ponteiro do array local e passar para a função) ou um parâmetro array (que já é um ponteiro, passado por outra função).

Exemplo:

```
function f1 pi1, pa2
def
enddef
return pi1
end

function f2 pa1
def
var vi1
var vi2
vet va3 size ci30
enddef
vi1 = ci1
vi2 = call f1 vi1 va3                  # vi2 = f1(vi1, &va3)
vi2 = call f1 ci5 pa1                  # vi2 = f1(5, pa1)
return vi2
end
```

Obs: A linguagem não tem comentários, eles foram colocados no exemplo como forma de esclarecimento.

### 2.3.3. Acesso ao Array

A recuperação de um valor de um array utiliza o comando **`get`**, no seguinte formato:

```
get array index índice to destino
```

Onde:

- *array:* um array local ou um parâmetro array.
- *índice:* uma constante inteira não-negativa (índice do vetor – inicia em 0).
- *destino:* uma variável local inteira ou um parâmetro inteiro.

Para modificar uma posição de um array, utiliza-se o comando **`set`**:

```
set array index índice with valor
```

Onde:

- *array:* um array local ou um parâmetro array.
- *índice:* uma constante inteira não-negativa (índice do vetor – inicia em 0).
- *valor:* uma variável local inteira, um parâmetro inteiro ou uma constante inteira.

Exemplo:

```
function f1 pi1, pa2
def
var vi1
vet va2 size ci10
enddef
vi1 = pi1 + ci1
set va2 index ci5 with ci2             # va2[5] = 2
set pa2 index ci0 with vi1             # pa2[0] = vi1
get va2 index ci8 to vi1               # vi1 = va2[8]
return pi1
end
```

Obs: A linguagem não tem comentários, eles foram colocados no exemplo como forma de esclarecimento.

### 2.3.4. Condicional

O condicional **`if`** possui um único valor de teste que pode ser uma variável, um parâmetro ou uma constante. Ela segue a mesma lógica de C, onde zero (0) é falso e qualquer valor não zero (positivo ou negativo) é verdadeiro. O corpo do condicional possui apenas um único comando. O formato do **`if`** possui é:

```
if condição
comando
endif
```

Onde:

- *condição*: uma variável, um parâmetro ou uma constante inteira.
- *comando*: atribuição, acesso a array (*get/set*) ou retorno.

Exemplo:

```
function f1
def
var vi1
vet va2 size ci10
enddef
vi1 = ci3
if vi1                      # if (vi1 != 0) → vi1 = 0
vi1 = ci0
endif
if vi1                      # if (vi1 != 0) → vi1 = va2[8]
get va2 index ci8 to vi1
end
if vi1                      # if (vi1 != 0) → return vi1
return vi1
endif
return ci-1
end
```

Obs: A linguagem não tem comentários, eles foram colocados no exemplo como forma de esclarecimento.

### 2.3.5. Retorno da Função

O comando **`return`** só poderá aparecer como último comando do corpo da função ou no corpo do **`if`**. Toda função terá obrigatoriamente um **`return`** como último comando do corpo. O formato do comando é:

```
return valor
```

Onde:

- *valor:* uma variável, um parâmetro ou uma constante inteiros.

## 3. Integração com a Linguagem C

Como o seu tradutor vai gerar código Assembly seguindo as regras de C, é possível escrever uma função main() em C e chamar as funções em BPL.

Para isso é necessário criar as assinaturas das funções BPL em C. Toda função BPL retorna *int* e os parâmetros podem ser *int* (quando temos pi) ou *int** (quando temos pa).

prog.bpl

```
function f1
def
enddef
return ci1024
end

function f2 pi1
def
var vi1
enddef
vi1 = pi1 + ci1
return vi1
end

function f3 pa1 pi2
def
var vi1
enddef
get pa1 index ci1 to vi1
vi1 = vi1 + ci1
return vi1
end
```

main.c

```
#include <stdio.h>

// Funções definidas em BPL
int f1();
int f2(int);
int f3(int*, int);

int main()
{
   int vet[] = {100, 200, 300};

   int x;
   x = f1();
   printf("f1 = %d\n", x);

   x = f2(1);
   printf("f2 = %d\n", x);

   x = f3(vet, 1);
   printf("f3 = %d\n", x);

   return 0;
}
```

Para executar o teste:

```
$ ./tradutor < prog.bpl > prog.S
$ gcc -o prog prog.S main.c
$ ./prog
f1 = 1024
f2 = 2
f3 = 201
```

Note que é possível também chamar funções em C utilizando BPL, desde que a função em C siga o padrão de nome "**fN**", receba os parâmetros dos tipos correspondentes e retorno *int*.

prog.bpl

```
function f1
def
var vi1
enddef
vi1 = call f100 ci1024
vi1 = call f100 vi1
return vi1
end
```

main.c

```
#include <stdio.h>

// Função definidas em BPL
int f1();

// Função chamada por BPL
int f100(int v) {
   printf("Valor vindo de BPL: %d\n", v);
   return 2048;
}

int main()
{
   f1();
   return 0;
}
```

Para executar o teste:

```
$ ./tradutor < prog.bpl > prog.S
$ gcc -o prog prog.S main.c
$ ./prog
Valor vindo de BPL: 1024
Valor vindo de BPL: 2048
```

## 4. BNF da Linguagem

```
<prog>     → <func>
           | <func> <prog>

<func>     → <header> <defs> <cmds> <ret> '\n' 'end' '\n'

<header>   → 'function' <fname> <params> '\n'
<fname>    → 'f'<num>
<params>   → ε
           | <param> <params>
<param>    → <parint>
           | <pararr>
<parint>   → 'pi'<num>
<pararr>   → 'pa'<num>

<defs>     → 'def' '\n' <vardef> 'enddef' '\n'
<vardef>   → 'var' <varint> '\n'
           | 'vet' <vararr> 'size' <const> '\n'
<varint>   → 'vi'<num>
<vararr>   → 'va'<num>
<const>    → 'ci'<snum>

<cmds>     → <cmd> '\n'
           | <cmd> '\n' <cmds>
<cmd>      → <attr>
           | <arrayget>
           | <arrayset>
           | <if>

<attr>     → <varint>  '=' <expr>
<expr>     → <valint>
           | <oper>
           | <call>

<valint>   → <varint>
           | <parint>
           | <const>

<oper>     → <valint> <op> <valint>
<op>       → '+' | '-' | '*' | '/'

<call>     → 'call' <fname> <args>
<args>     → ε
           | <arg> <args>
<arg>      → <valint>
           | <array>
<array>    → <vararr>
           | <pararr>

<arrayget> → 'get' <array> 'index' <const> 'to'   <varint>
<arrayset> → 'set' <array> 'index' <const> 'with' <valint>
```

```
<if>   → 'if' <valint> '\n' <body> '\n' 'endif'

<body> → <attr>
       | <arrayget>
       | <arrayset>
       | <ret>

<ret>  → 'return' <valint>

<num>  → <digit>
       | <digit> <num>

<snum> → <num>
       | '-'<num>

<digit> → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'| '8' | '9'
```