SBVCONC: Construção de Compiladores

Aula 13: Subprogramas

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Subprogramas

- Usaremos o termo subprogramas para indicar tanto procedimentos quanto funções;
- Durante o curso já lidamos com subprogramas e questões relacionadas à escopo dentro do *scanner*, do *parser* e da tabela de identificadores, sendo assim, o maior esforço que teremos que empregar para viabilizar a implementação de subprogramas envolvem modificações nas classes da AST.

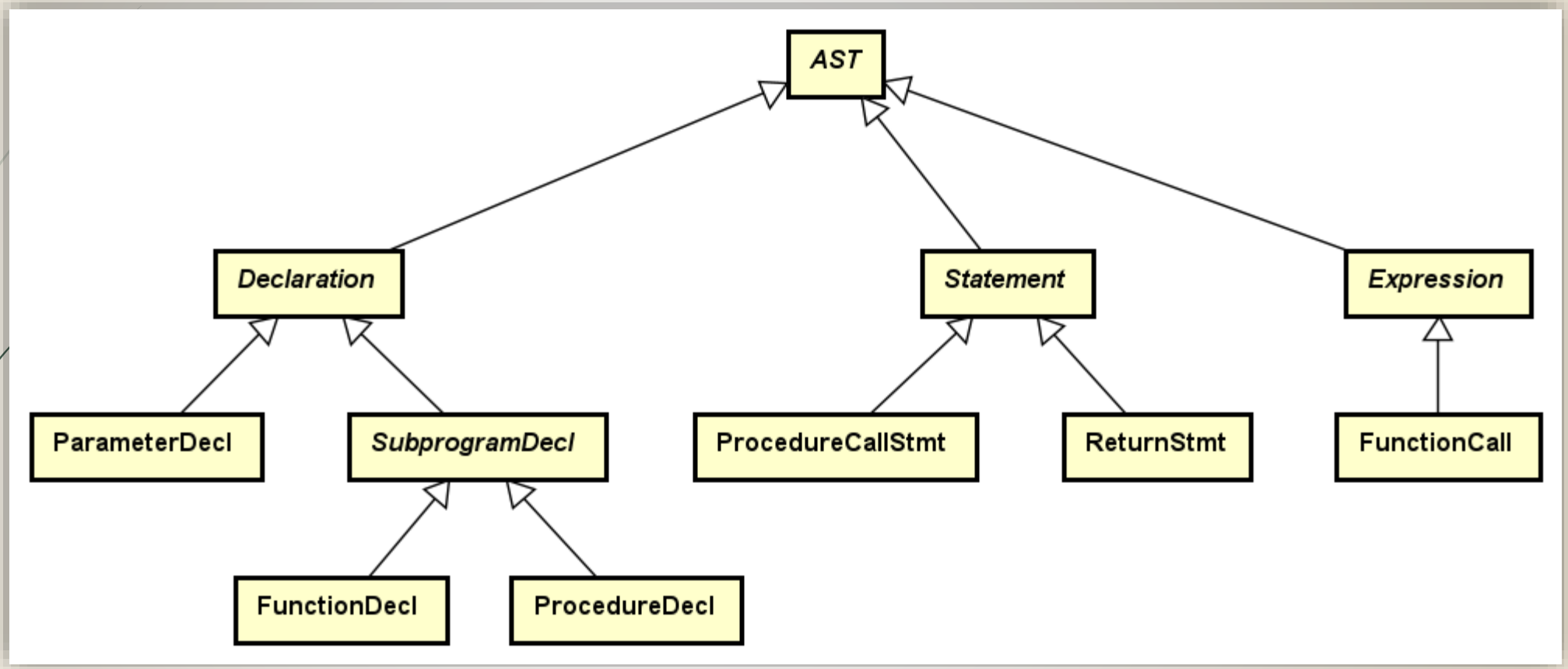
Regras Gramaticais Relacionadas aos Subprogramas

```
  subprogramDecls = ( subprogramDecl )* .

   subprogramDecl = procedureDecl | functionDecl .

    procedureDecl = "procedure" procId ( formalParameters )? "is"
                    initialDeclPart statementPart procId ";" .

     functionDecl = "function" funcId ( formalParameters )?
                    "return" typeName "is" initialDeclPart statementPart funcId ";" .

 formalParameters = "(" parameterDecl ( "," parameterDecl )* ")" .

    parameterDecl = ( "var" )? paramId ":" typeName .

procedureCallStmt = procId ( actualParameters )? ";" .

 actualParameters = "(" expressions ")" .

       returnStmt = "return" ( expression )? ";" .

     functionCall = funcId ( actualParameters )? .
```

Métodos de Análise Relevantes
(baseados nas regras gramaticais)

- `List<SubprogramDecl> parseSubprogramDecls()`
- `SubprogramDecl parseSubprogramDecl()`
- `ProcedureDecl parseProcedureDecl()`
- `FunctionDecl parseFunctionDecl()`
- `List<ParameterDecl> parseFormalParameters()`
- `ParameterDecl parseParameterDecl()`
- `ProcedureCallStmt parseProcedureCallStmt()`
- `List<Expression> parseActualParameters()`
- `ReturnStmt parseReturnStmt()`
- `FunctionCall parseFunctionCall()`

Classes Relevantes da AST



Exemplo de Procedimentos
Parâmetros

```
var x : Integer;

procedure inc( var n : Integer ) is
begin
   n := n + 1;
end inc;

begin
   x := 5;
   inc( x );
   writeln x;
end.
```

Qual valor é impresso? Se **var** for removido da declaração do parâmetro, qual valor será impresso?

Exemplo de Procedimentos
Escopo

```
var x : Integer;
var y : Integer;

procedure P1 is
   var x : Integer;
   var n : Integer;
begin
   x := 5;    // qual x?
   n := y;    // qual y?
end P1;

begin
   x := 8;    // qual x?
end.
```

As variáveis e constantes podem ser declaradas em nível de programa (global) ou em nível de subprograma (local), introduzindo o conceito de escopo.

O Nível de Escopo de uma Declaração de Variável

- Durante a geração de código, quando uma variável ou um valor nomeado são referenciados na parte de instruções de um programa ou subprograma, precisaremos ser capazes de determinar onde a variável foi declarada;
- A classe `IdTable` contém o método `getScopeLevel()` que retorna o nível de escopo de para o escopo atual:
  - `PROGRAM` para objetos declarados no escopo mais externo (escopo de programa);
  - `SUBPROGRAM` para objetos declarados dentro de um subprograma;
- Quando uma variável é **declarada**, a declaração é inicializada com o nível atual:

```
varDecl = new VarDecl( identifiers,
                       varType,
                       idTable.getScopeLevel() );
```

Exemplo de Níveis de Escopo

```
var x : Integer;      // o nível de escopo da declaração é PROGRAM
var y : Integer;      // o nível de escopo da declaração é PROGRAM

procedure P1 is       // o nível de escopo da declaração é PROGRAM
   var x : Integer;   // o nível de escopo da declaração é SUBPROGRAM
   var b : Integer;   // o nível de escopo da declaração é SUBPROGRAM
begin
   ... x ...         // x foi declarado no escopo SUBPROGRAM
   ... b ...         // b foi declarado no escopo SUBPROGRAM
   ... y ...         // y foi declarado no escopo PROGRAM
end P1;

begin
    ... x  ...       //  x foi declarado no escopo PROGRAM
    ... y  ...       //  y foi declarado no escopo PROGRAM
    ... P1 ...       // P1 foi declarado no escopo PROGRAM
end.
```

Classe `IdTable`

- A classe `IdTable` tem a habilidade de abrir novos escopos e de buscar por declarações, tanto no escopo atual quanto em escopos mais externos;
- A classe `IdTable` é implementada usando uma pilha de mapas em que são associadas as strings dos identificadores (seus nomes) às suas declarações;
  - Note que como não permitidos o aninhamento de subprogramas, nossa pilha sempre terá no máximo dois níveis;
- Quando um novo escopo é aberto, um novo mapa é empilhado na pilha. Quando um escopo é fechado, o mapa do topo é desempilhado da pilha;
- Dentro de um subprograma, a busca por declarações envolve buscar dentro do nível atual (o mapa no topo da pilha que contém todos os identificadores declarados no escopo `SUBPROGRAM`) e então dentro do escopo mais externo (o mapa abaixo do topo, contendo todos os identificadores declarados no escopo `PROGRAM`).

Métodos Importantes da Classe `IdTable`

```
/**
 * Retorna o nível de escopo atual.
 */
public ScopeLevel getScopeLevel()

/**
 * Abre um novo escopo para identificadores.
 */
public void openScope()

/**
 * Fecha o escopo mais interno.
 */
public void closeScope()
```

`ScopeLevel` é uma enumeração com apenas dois valores: `PROGRAM` e `SUBPROGRAM`.

Métodos Importantes da Classe `IdTable`

```
/**
 * Insere uma declaração no nível de escopo atual.
 * @throws ParserException se o token associado à declaração
 * já tiver sido definido dentro do escopo atual.
 */
public void add( Declaration decl ) throws ParserException

/**
 * Retorna a Declaration associada ao texto do token do identificador.
 * Returna null se o identificador não for encontrado.
 * Esse método busca em escopos mais externos caso necessário.
 */
public Declaration get( Token idToken )
```

Regras de Restrição para Subprogramas

- Instrução **`return`**:
  - **Regra de Tipo:** se a instrução retorna o valor de uma função, então o tipo da expressão que será retornada deve ser do mesmo tipo do retorno da função;
  - **Regra Variada:** se a instrução `return` retorna um valor, então ela deve estar aninhada à declaração de uma função;
  - **Regra Variada:** se a instrução `return` está aninhada a uma função, então ela deve retornar um valor;
  - **Regra Variada:** a instrução `return` deve estar aninhada a um subprograma, o que é tratado pelo *parser* usando `SubprogramContext`.

Regras de Restrição para Subprogramas

- Declaração de Função:
  - **Regra Variada:** não pode haver nenhuma `var` nos parâmetros;
  - **Regra Variada:** é necessário que haja pelo menos uma instrução de retorno;
- Chamada de Subprograma, tanto para procedimentos quanto para funções:
  - **Regra de Tipo:** a quantidade de argumentos (*actual parameters*) precisa ser a mesma da quantidade de parâmetros formais e cada par deve ter o mesmo tipo;
- Chamada de Procedimento:
  - **Regra Variada:** se um parâmetro formal é um parâmetro `var`, então o argumento deve ser um valor nomeado, não uma expressão arbitrária.

Organização em Tempo de Execução para Subprogramas

- O entendimento da organização em tempo de execução de subprogramas envolve quatro conceitos principais:
  - Registros de ativação (*activation records/stack frames*);
  - Endereçamento de variáveis;
  - Passagem de parâmetros e retorno de valores de funções;
  - Instruções da CVM para subprogramas.

Instruções da CVM para Subprogramas

Opcodes usados em Subprogramas

PROC: *procedure/function*
LDLADDR: *load local address*
LDGADDR: *load global address*
CALL: *call subprogram*
RET: *return from a subprogram*

Note que na CVM não possui instruções separadas para procedimentos e funções.

Subprogramas Ativos

- Quando um programa está em execução, um subprograma é denominado **ativo** se ele foi invocado e ainda não retornou;
- Quando um subprograma é invocado, precisamos alocar espaço na pilha para seus parâmetros e variáveis locais. Adicionalmente, se o subprograma é uma função, precisamos alocar espaço na pilha para o valor de retorno;
- Quando um subprograma retorna, o espaço alocado na pilha é liberado.

Um subprograma ativo é aquele para o qual este espaço (registro de ativação) está atualmente na pilha.

Registro de Ativação
Activation Record/Stack Frame/Frame

- Um registro de ativação é composto por uma estrutura de tempo de execução para cada subprograma ativo. Um novo registro de ativação é criado toda vez que um subprograma é chamado;
- Consiste em até cinco partes:
  - **A parte do valor de retorno** (somente para funções);
  - **A parte dos parâmetros**, que pode ser vazia caso eles não existam;
  - **A parte do contexto** (sempre duas palavras):
    - Valores salvos para os registradores PC e BP;
  - **A parte das variáveis locais**, que pode ser vazia caso elas não existam;
  - **A parte temporária**:
    - Armazena operandos e resultados durante a execução das instruções;
    - Está sempre vazia no início e no fim de cada instrução do subprograma.

A Parte do Valor de Retorno de um Registro de Ativação

- A chamada de uma função precisa, primeiramente, alocar espaço na pilha para o valor de retorno. O número de bytes alocados é o número de bytes do tipo de retorno da função;
- O método `emit()` na classe `FunctionCall` contém o seguinte código:

```
// aloca espaço na pilha para o valor de retorno
emit( "ALLOC " + funcDecl.getType().getSize() );
```

A Parte dos Parâmetros de um Registro de Ativação

- Para cada **parâmetro de valor**, uma chamada de subprograma precisa emitir código para deixar o valor do argumento no topo da pilha;
- Para cada **parâmetro variável (var)**, uma chamada de procedimento precisa emitir código para deixar o endereço do argumento no topo da pilha. O argumento precisa ser um valor nomeado, não uma expressão.

A Parte Contextual de um Registro de Ativação

- **Link Dinâmico (*Dynamic Link*):** endereço base (BP) do registro de ativação para o subprograma invocado;
- **Endereço de retorno:** endereço da próxima instrução que segue à chamada do subprograma.

Os valores de BP e PC relativos ao subprograma invocado são salvos (empilhados) na pilha pela instrução CALL da CVM e então são restaurados pela instrução RET da CVM.

A Parte das Variáveis Locais de um Registro de Ativação

- Se existem variáveis locais declaradas em um subprograma, então há a necessidade de se alocar espaço na pilha de execução para essas variáveis;
- A instrução `PROC` (*procedure*) da CVM possui um parâmetro inteiro usado para definir o comprimento das variáveis do subprograma;
- **Exemplo:**

```
procedure P2 is
   var m, n : Integer;
   var b : Boolean;
begin
   ...
end P2;
```

- Este procedimento precisará alocar nove bytes para as variáveis locais: 4 para cada inteiro e 1 para o booleano. A instrução `PROC 9` será emitida para alocar o espaço necessário na pilha de execução;
- Uma instrução na forma `PROC 0` é emitida sempre que um subprograma não possuir nenhuma variável local.

A Parte Temporária de um Registro de Ativação

- A parte temporária de um registro de ativação é análoga ao uso da pilha de execução para manter valores temporários;
- Assim que as instruções de máquina de um subprograma são executadas a parte temporária cresce e diminui;
- A parte temporária de um registro de ativação estará vazia no início e no fim da execução de cada instrução da CPRL do subprograma.

A Parte Temporária de um Registro de Ativação - Exemplo

- Assumindo que:
  - O registrador `BP` tem o valor `200`;
  - A variável local `x` do tipo inteiro tem endereço relativo `8`;
  - A variável local `y` do tipo inteiro tem valor `6` e endereço relativo `12`;
- A instrução de atribuição da CPRL:

```
x := y + 1;
```

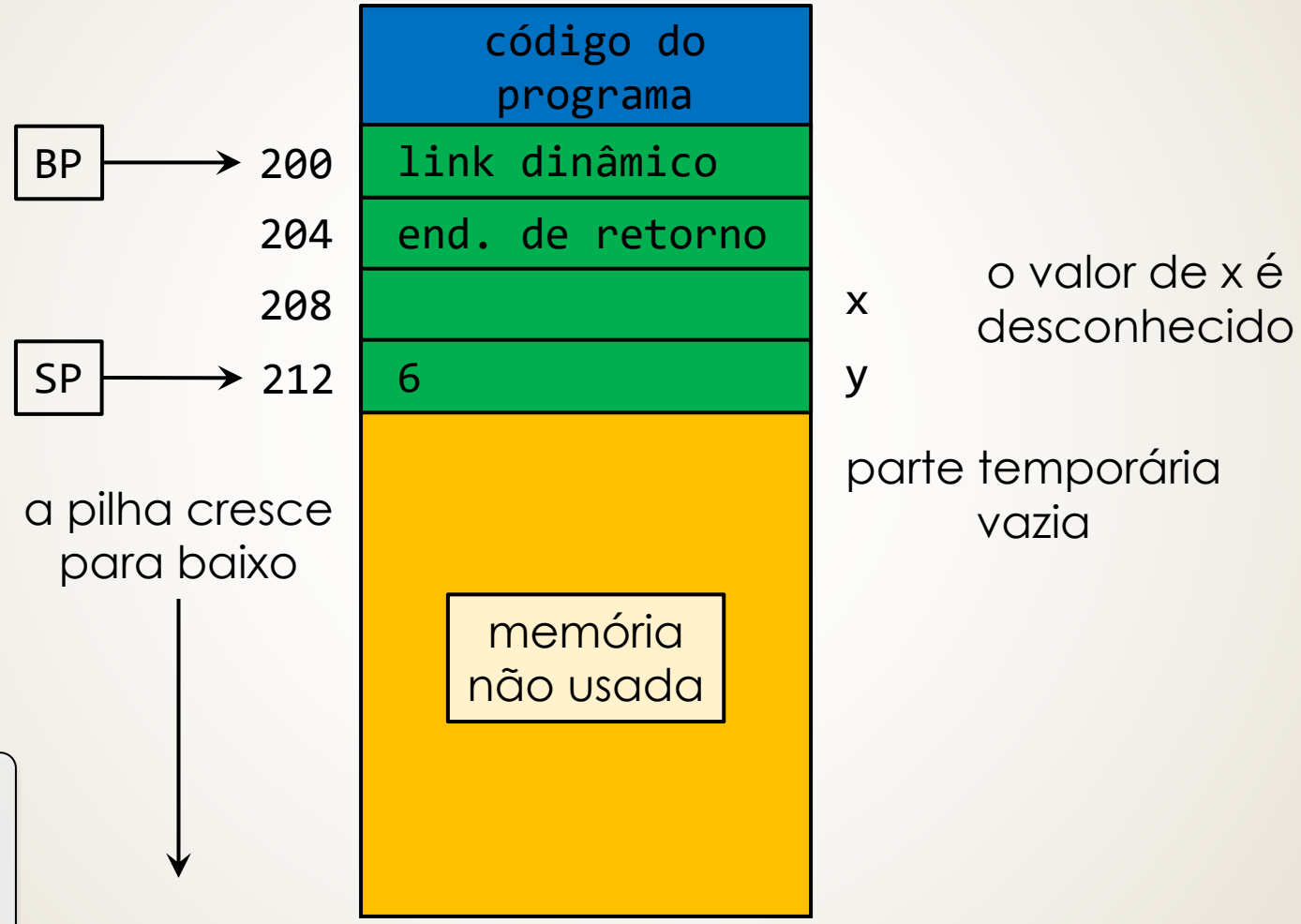
Compilará para as seguintes instruções da CVM:

```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT 1
ADD
STOREW
```

Será otimizada para `LDCINT1`

A Parte Temporária de um Registro de Ativação - Exemplo

A parte temporária da pilha está vazia no início de uma instrução da CPRL.



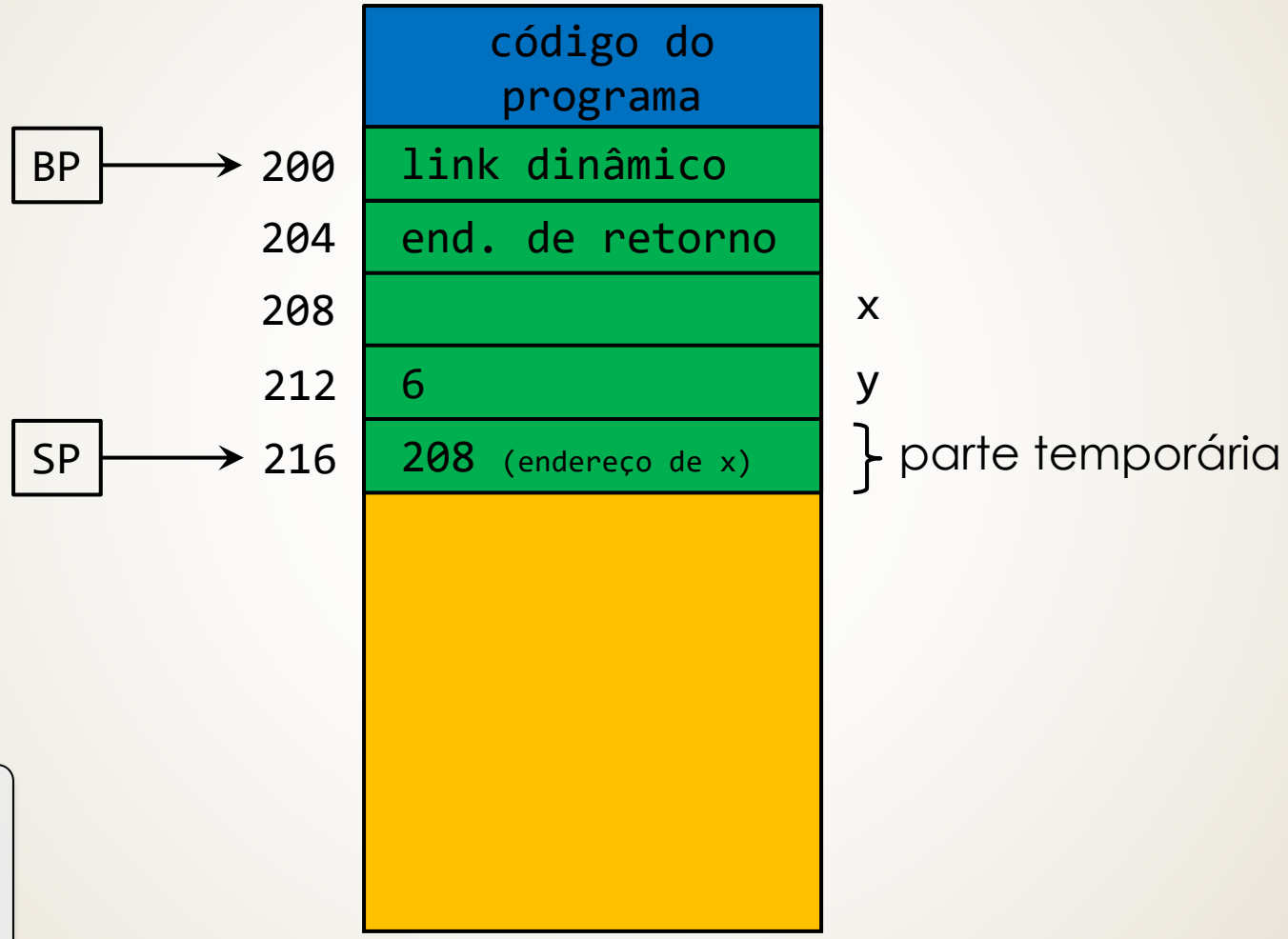

```
x := y + 1;
```

```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT1
ADD
STOREW
```

```
        Load/Store Opcodes
LDCINT1: load constant integer 1
LDLADDR: load local address
  LOADW: load word
 STOREW: store word

        Arithmetic Opcodes
ADD: add
```

A Parte Temporária de um Registro de Ativação - Exemplo

Após a execução de
`LDLADDR 8`




```
x := y + 1;
```

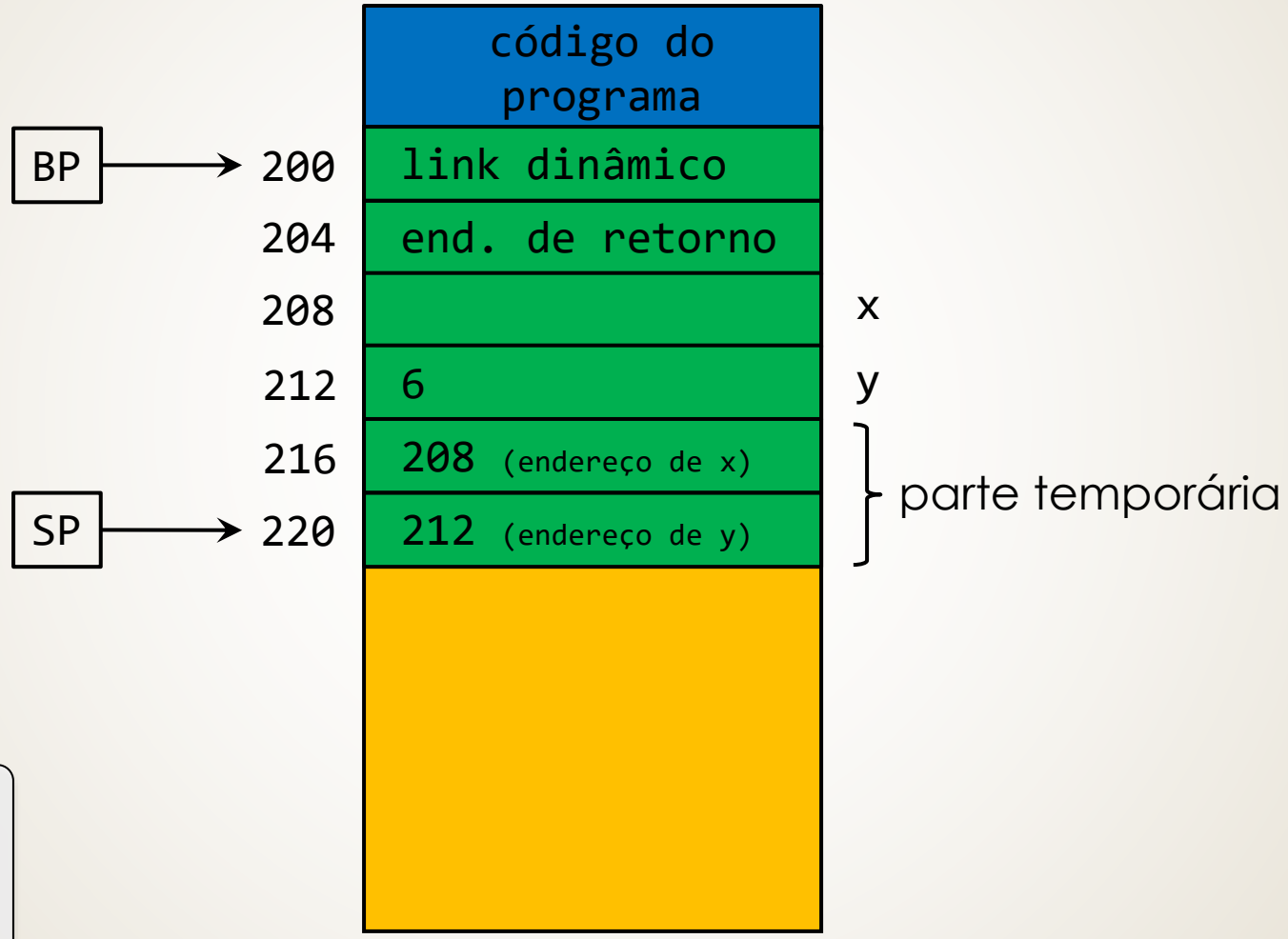
```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT1
ADD
STOREW
```

***Load/Store* Opcodes**
LDCINT1: *load constant integer 1*
LDLADDR: *load local address*
LOADW: *load word*
STOREW: *store word*

***Arithmetic* Opcodes**
ADD: *add*

A Parte Temporária de um Registro de Ativação - Exemplo

Após a execução de
LDLADDR 12




```
x := y + 1;
```

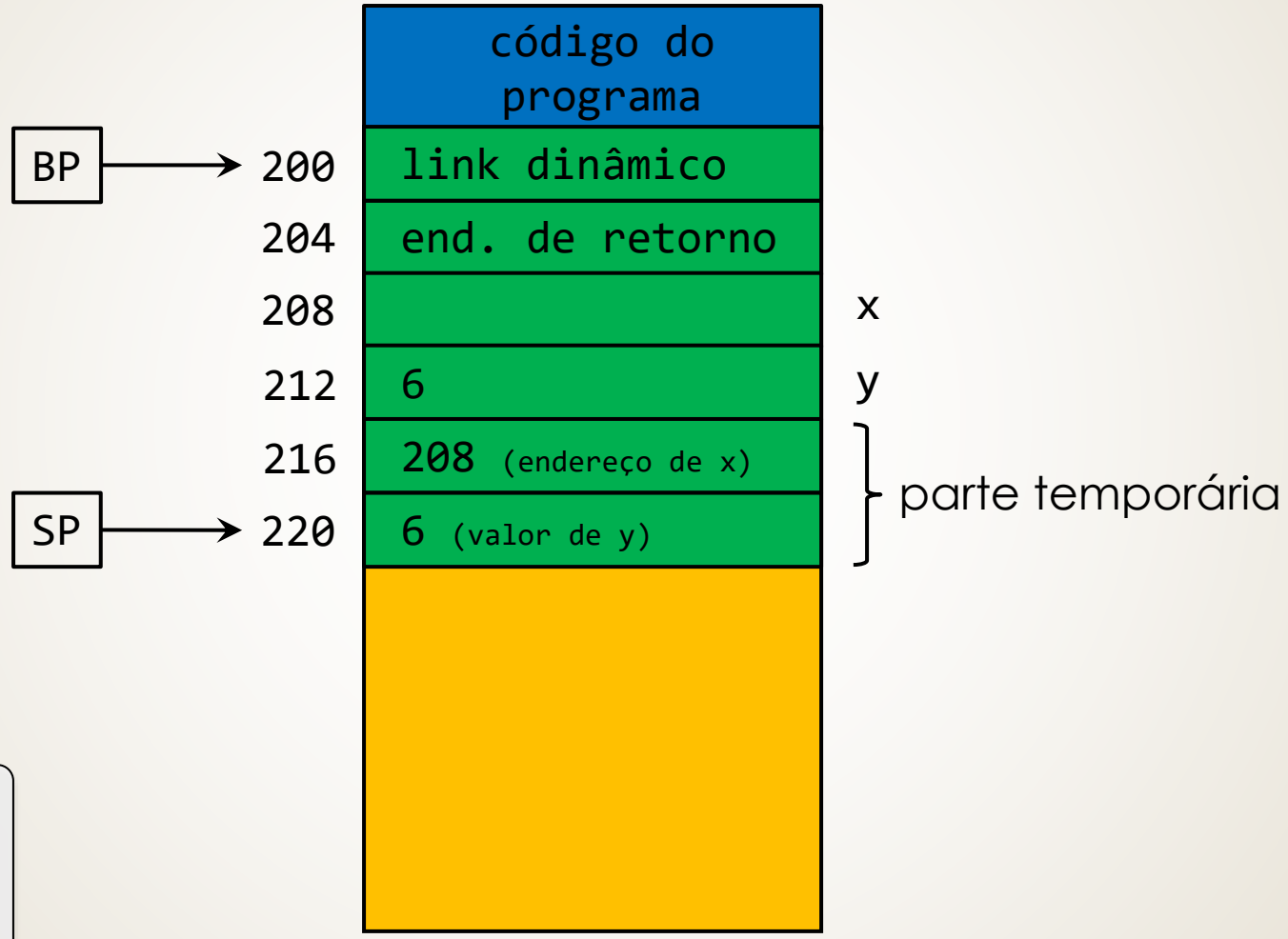
```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT1
ADD
STOREW
```

***Load/Store* Opcodes**
LDCINT1: *load constant integer 1*
LDLADDR: *load local address*
LOADW: *load word*
STOREW: *store word*

***Arithmetic* Opcodes**
ADD: *add*

A Parte Temporária de um Registro de Ativação - Exemplo

Após a execução de
LOADW




```
x := 6 + 1;
```

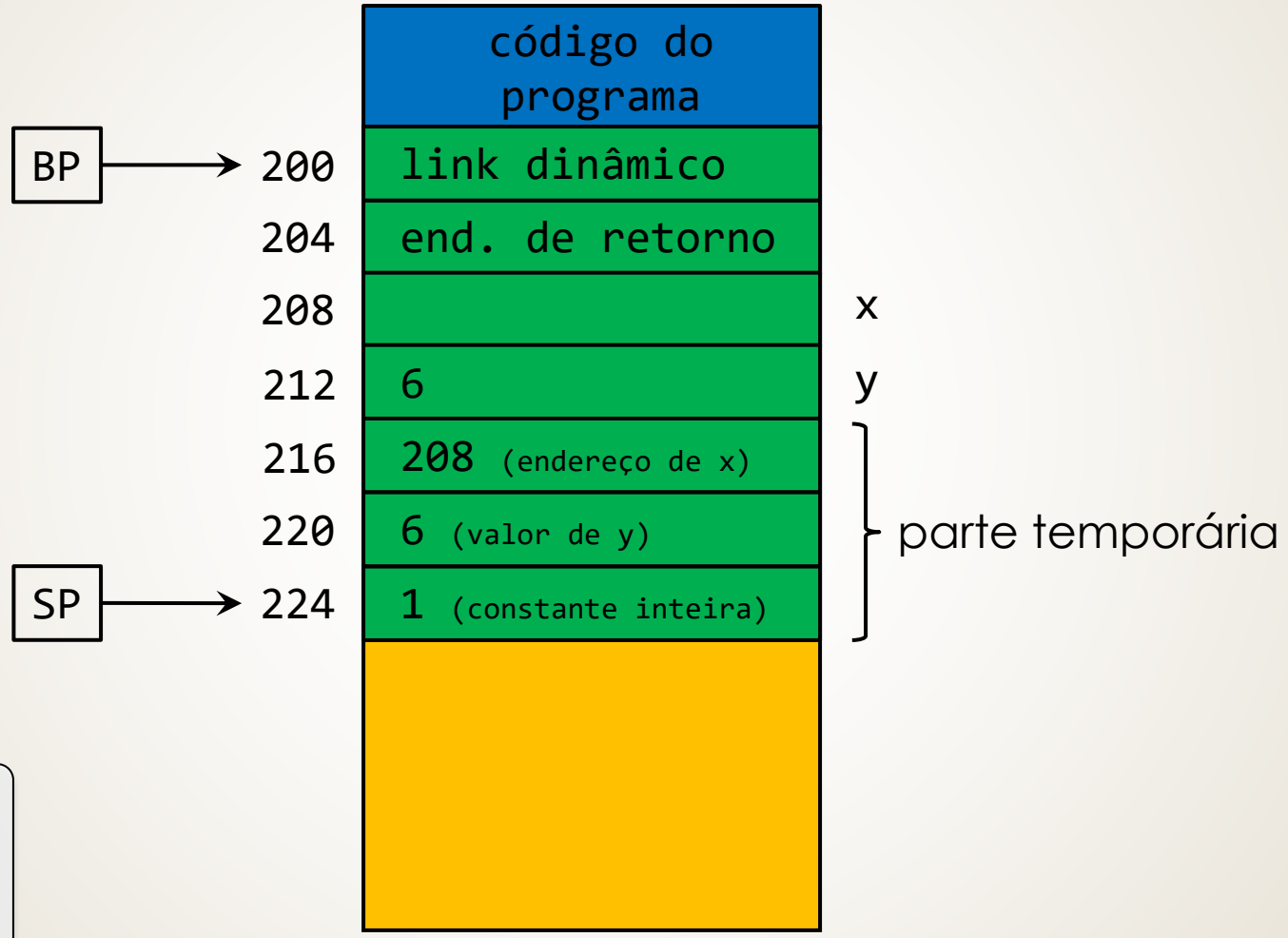
```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT1
ADD
STOREW
```

***Load/Store* Opcodes**
LDCINT1: *load constant integer 1*
LDLADDR: *load local address*
LOADW: *load word*
STOREW: *store word*

***Arithmetic* Opcodes**
ADD: *add*

A Parte Temporária de um Registro de Ativação - Exemplo

Após a execução de
LDCINT1




```
x := 6 + 1;
```

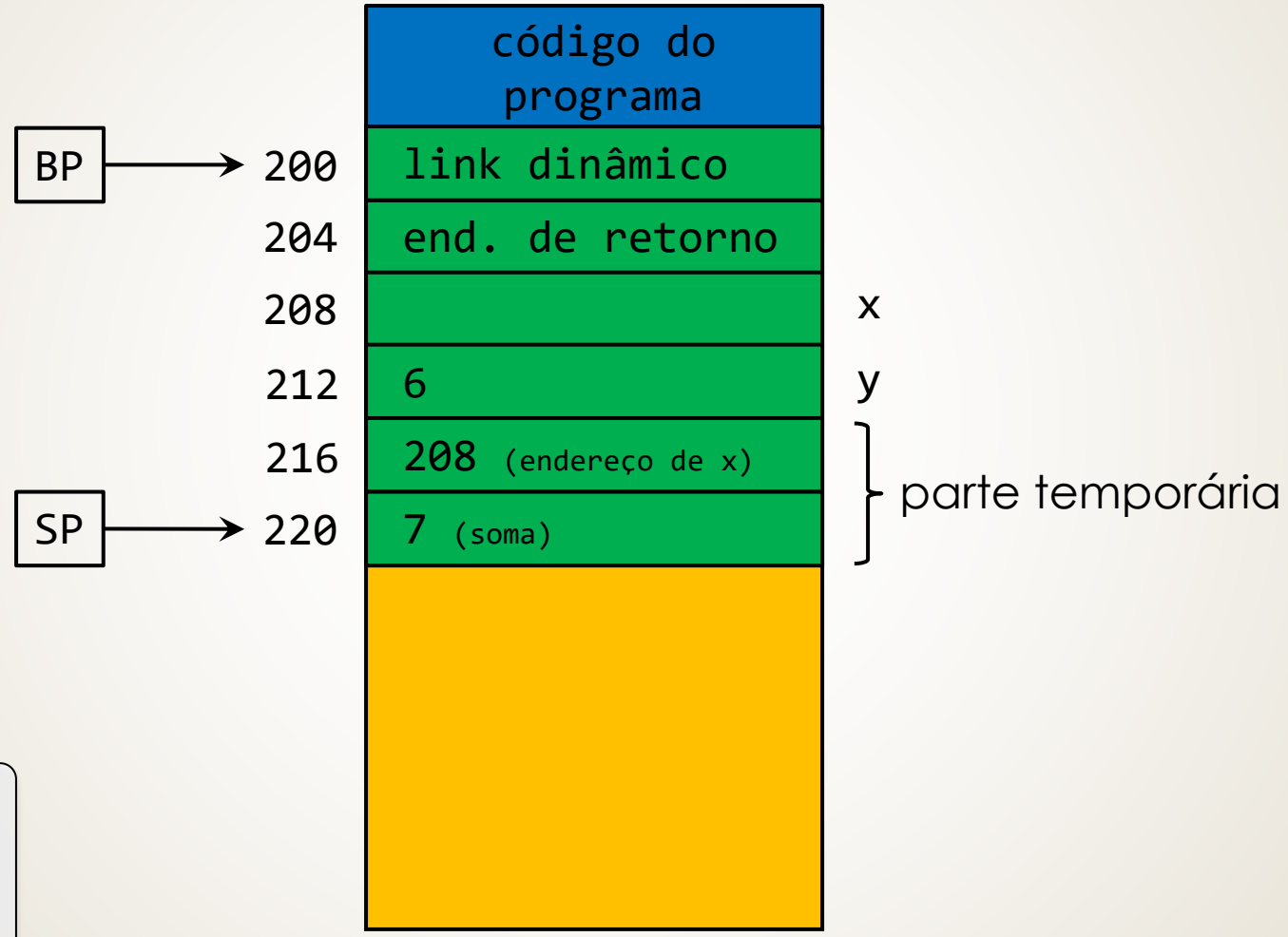
```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT1
ADD
STOREW
```

Load/Store Opcodes
LDCINT1: *load constant integer 1*
LDLADDR: *load local address*
LOADW: *load word*
STOREW: *store word*

Arithmetic Opcodes
ADD: *add*

A Parte Temporária de um Registro de Ativação - Exemplo

Após a execução de ADD



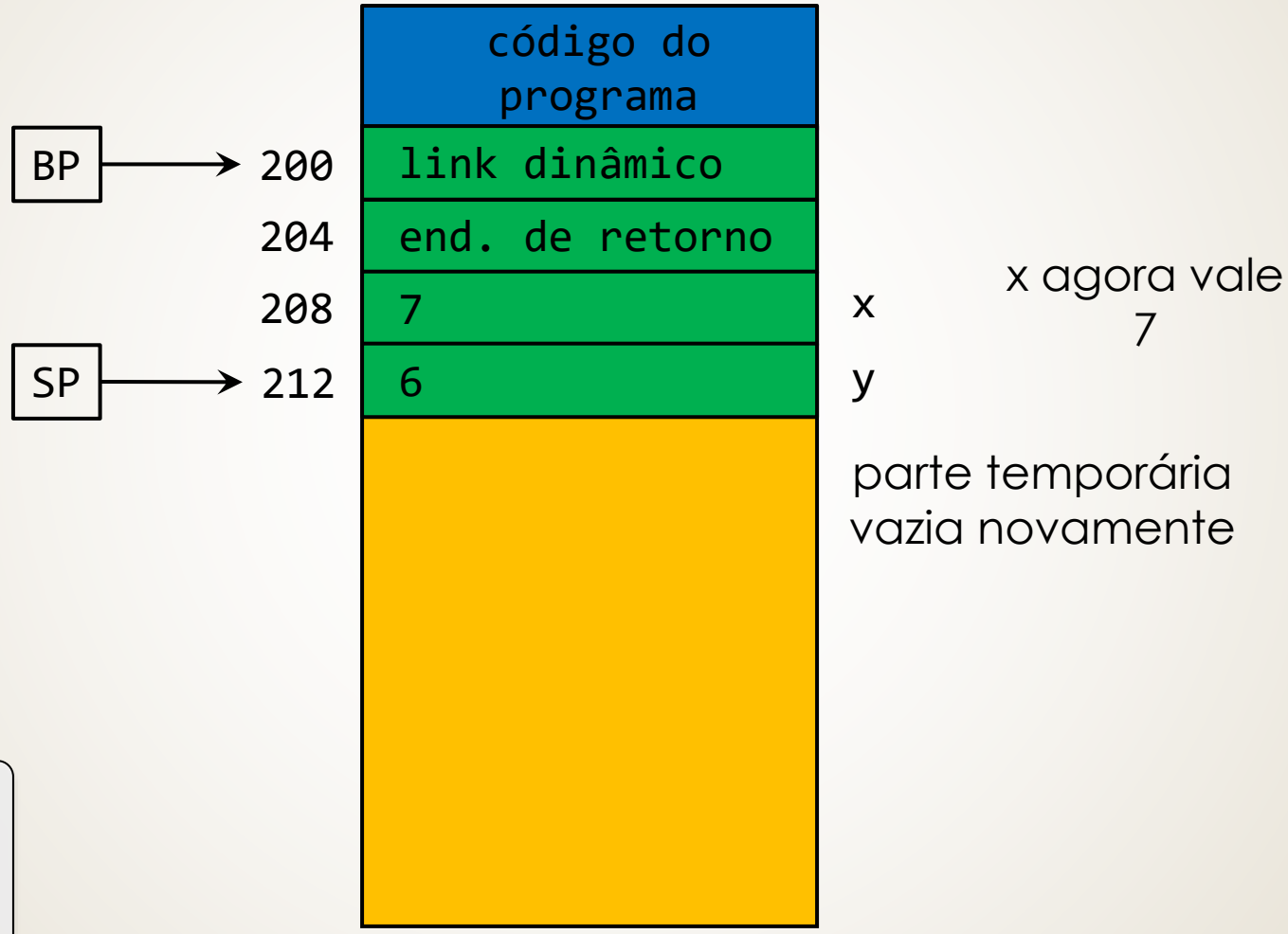

```
x := 7;
```

```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT1
ADD
STOREW
```

```
      Load/Store Opcodes
LDCINT1: load constant integer 1
LDLADDR: load local address
  LOADW: load word
 STOREW: store word

      Arithmetic Opcodes
ADD: add
```

A Parte Temporária de um Registro de Ativação - Exemplo

Após a execução de
STOREW




```
x := 7;
```

```
LDLADDR 8
LDLADDR 12
LOADW
LDCINT1
ADD
STOREW
```

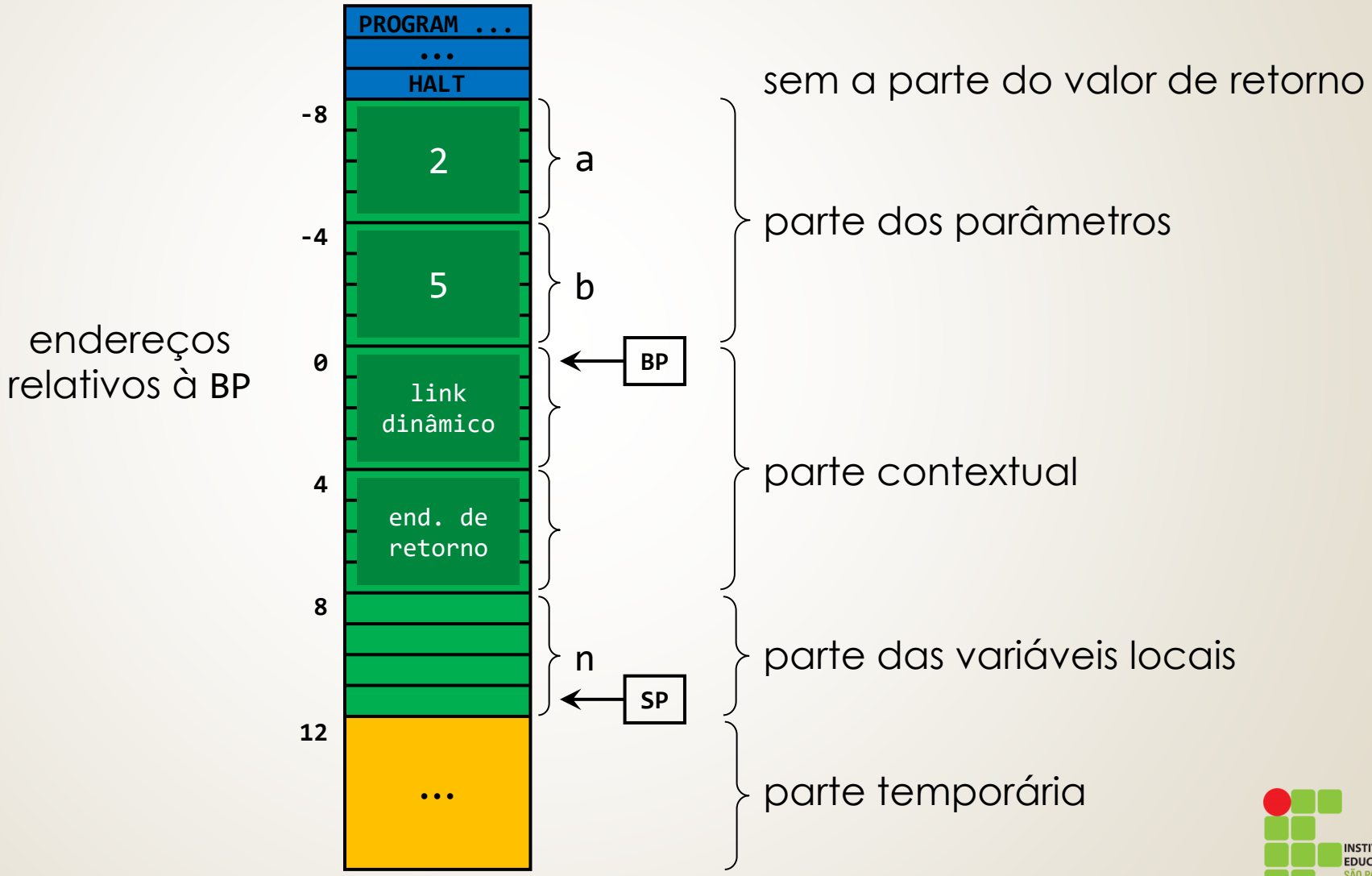
```
      Load/Store Opcodes
LDCINT1: load constant integer 1
LDLADDR: load local address
  LOADW: load word
 STOREW: store word

      Arithmetic Opcodes
ADD: add
```

Exemplo de Subprograma com Parâmetros

```
var x : Integer;

procedure P3( a : Integer, b : Integer ) is
   var n : Integer;
begin
   ...
end P3;

begin
   ...
   P3( 2, 5 );
   ...
end.
```

Exemplo de Subprograma com Parâmetros

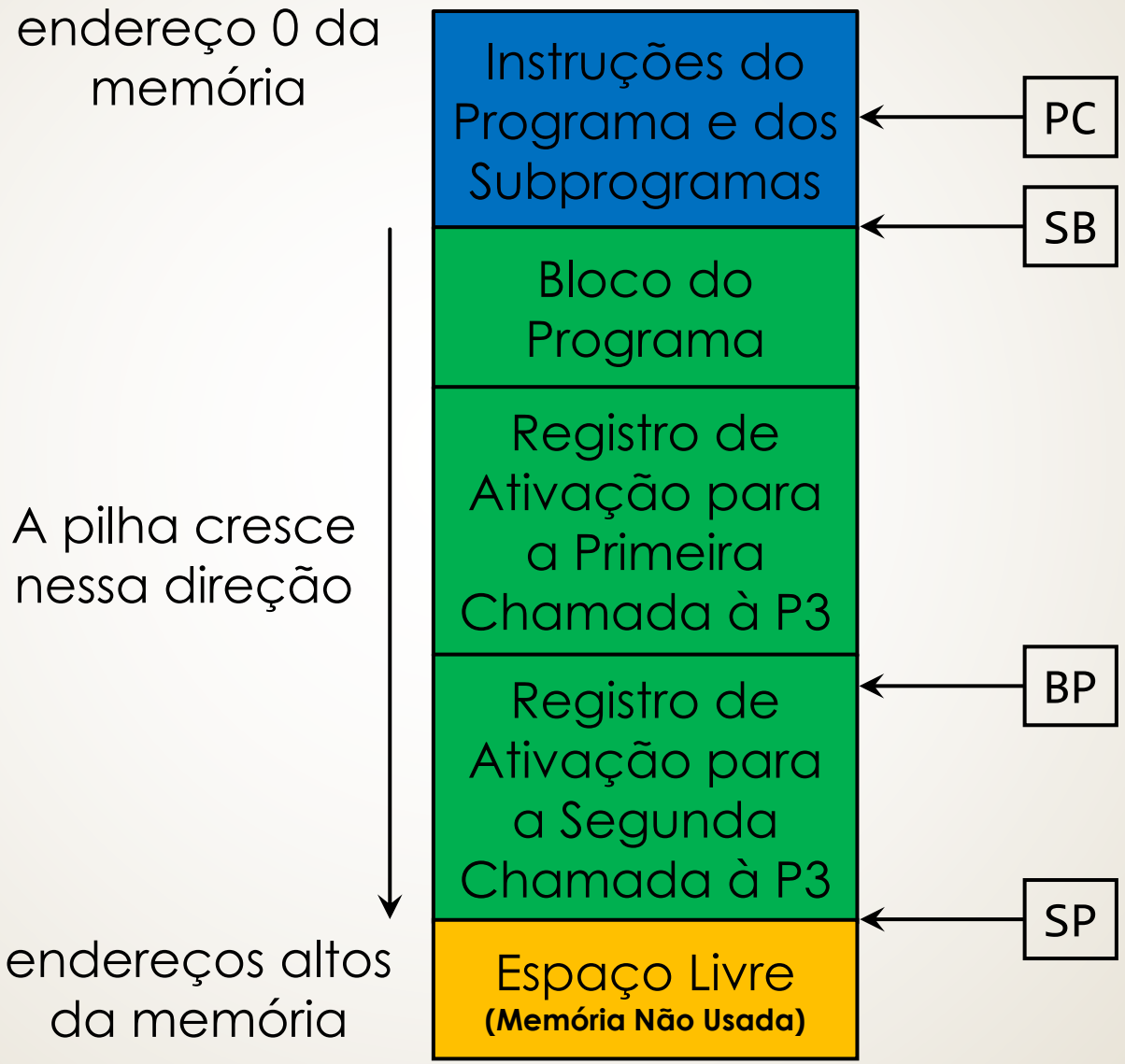
Registro de Ativação para o Procedimento P3



Suporte à Recursão

- Visto que um novo registro de ativação é criado toda vez que um subprograma é invocado, a CPRL suporta chamadas recursivas;
- Para ilustrar, suponha que um programa invoque o procedimento P3 e então P3 faz uma chamada recursiva à si mesmo. Cada chamada a P3 tem seu próprio registro de ativação, o que significa que cada chamada tem sua própria cópia de parâmetros, declarações de variáveis locais etc.

Suporte à Recursão



Carregando um Programa

- O código objeto é carregado no início da memória iniciando no endereço 0;
- O registrador PC é inicializado com 0, o endereço da primeira instrução;
- Os registradores SB e BP são inicializados com o endereço que segue a última instrução, ou seja, o primeiro byte livre na memória;
- O registrador SP é inicializado com $BP - 1$, visto que a pilha de execução está vazia.

Discussão Sobre Parâmetros

- As funções podem ter somente parâmetros de valor, mas os procedimentos podem ter tanto parâmetros variáveis (**var**) quanto de valor;
- O código que gerencia a passagem de valor para esses dois tipos de parâmetros em uma chamada de procedimento é de certa forma análogo a como tratamos uma instrução de atribuição na forma "`x := y`", onde geramos código diferente para os lados esquerdo e direito;
  - **Lado esquerdo:** gerar código para deixar o **endereço** na pilha;
  - **Lado direito:** gerar código para deixar o **valor** na pilha.

Discussão Sobre Parâmetros

- A analogia para os parâmetros:
  - **Parâmetros variáveis (var):** gerar código similar à forma que se lida com o lado esquerdo de uma instrução de atribuição;
  - **Parâmetros de valor:** gerar código similar à forma que se lida com o lado direito de uma instrução de atribuição;
- Ao se analisar o código dos argumentos (*actual parameters*), por padrão sempre invocamos `parseExpression()`:
  - Gera código que deixa o valor na expressão na pilha;
  - Correto para um parâmetro de valor, mas não para um parâmetro variável;
- Note que o código para a classe `Variable` contém um construtor que possui um único parâmetro do tipo `NamedValue` que é usado para construir um objeto do tipo `Variable`.

Convertendo um `NamedValue` em uma `Variable`

- Quando temos uma expressão de `NamedValue` que corresponde à um parâmetro variável, precisamos convertê-la em uma `Variable`;
- Uma possível abordagem: no método `checkConstraints()` da classe `ProcedureCall`, quando se itera e se compara a lista de parâmetros formais e argumentos:
  - Se o parâmetro formal é um parâmetro variável e o argumento não é um `NamedValue`, gerar uma mensagem de erro (não se pode passar uma expressão arbitrária à um parâmetro variável);
  - Se o parâmetro formal é um parâmetro variável e o argumento é um `NamedValue`, converter o `NamedValue` em uma `Variable`.

Convertendo um `NamedValue` em uma `Variable`

Código Dentro do Método `checkConstraints()` de `ProcedureCallStmt`

```
for ( int i = 0; i < actualParams.size(); i++ ) {

    Expression expr = actualParams.get( i );
    ParameterDecl param = formalParams.get( i );

    ...  // verifica se os tipos combinam

    // verifica se os valores nomeados estão sendo
    // passados para os parâmetros variáveis
    if ( param.isVarParam() ) {
        if ( expr instanceof NamedValue ) {
            // troca o valor nomeado por uma variável
            expr = new Variable( (NamedValue) expr );
            actualParams.set( i, expr );
        } else {
            throw error( expr.getPosition(),
                         "Expression for a var parameter must be a variable." );
        }
    }
}
```

Invocando um Subprograma

- Quando um subprograma é invocado:
  - Para uma função é alocado espaço na pilha para o valor de retorno;
  - Os argumentos são empilhados na pilha:
    - Valores de expressões para parâmetros de valor;
    - Endereços para parâmetros variáveis;
  - A instrução `CALL` empilha a parte contextual na pilha;
  - A instrução `PROC` do subprograma aloca espaço na pilha para as variáveis locais do subprograma.

Instrução PROC versus Instrução ALLOC

- Para a CVM, as instruções PROC e ALLOC são equivalentes e podem ser usadas intercaladamente;
- Ambas as instruções movem SP para alocar espaço na pilha, ou seja, para o valor de retorno de uma função ou uma variável local de um subprograma.

Instrução de Retorno

- A instrução de retorno (RET) da CVM indica o número de bytes usado pelos parâmetros dos subprogramas, permitindo que eles possam ser removidos da pilha;
- **Exemplo:**

```
RET 8
```

Retornando de um Subprograma

- Quando uma instrução de retorno é executada:
  - BP é configurado com o valor do link dinâmico:
    - Restaura BP ao registro de ativação do chamador;
  - PC é configurado com o endereço de retorno:
    - Restaura PC às instruções do chamador;
  - SP é configurado de forma a restaurar a pilha ao seu estado antes da instrução da invocação ser executada:
    - Para procedimentos, SP é configurado com o endereço de memória antes o registro de ativação;
    - Para funções, SP é configurado com o endereço de memória do último byte do valor de retorno. O valor de retorno é mantido na pilha.

Referenciando Variáveis Locais e Parâmetros

- A instrução `LDLADDR` é usada para referenciar os parâmetros e as variáveis locais de um subprograma:
  - Computa o valor absoluto do endereço de uma variável local usando seu endereço relativo à `BP` e empilha esse valor computado na pilha;
  - Seu uso em subprogramas é similar ao uso de `LDGADDR` para variáveis de um programa, com a exceção de que o endereço relativo da primeira variável local é 8 ao invés de 0, dado que existem 8 bytes na parte contextual do registro de ativação;
  - Endereços relativos podem ser negativos ao se carregar endereços de parâmetros.

Referenciando Variáveis Globais

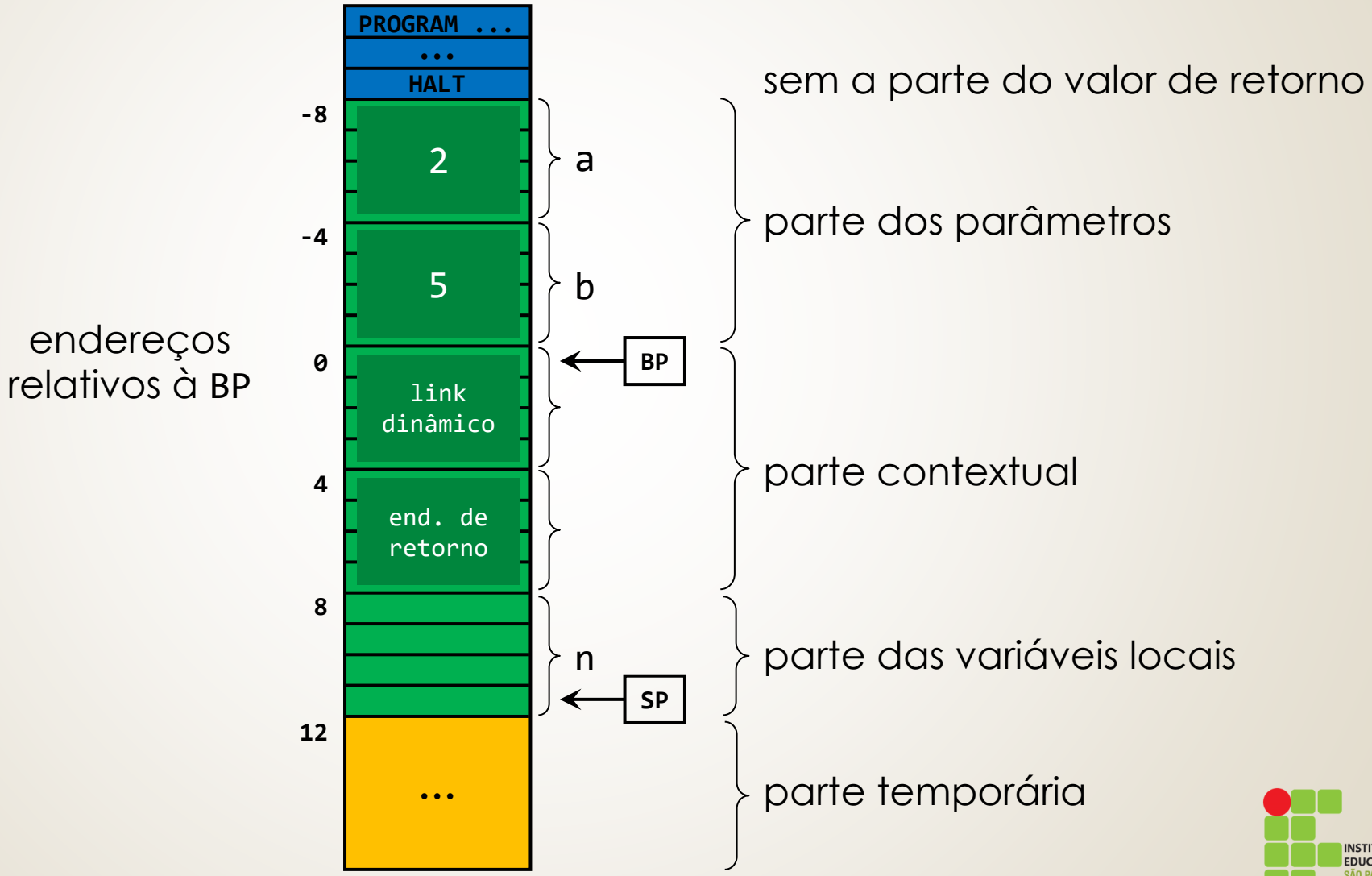
- A instrução **LDGADDR** é usada dentro de um subprograma para referenciar variáveis globais, ou seja, variáveis declaradas em nível de programa:
  - Computa o valor absoluto do endereço de uma variável global usando seu endereço relativo à **SB** e empilha esse valor computado na pilha;
  - Note que **LDGADDR** é usada em subprogramas da mesma forma que em programas.

Exemplo de Subprograma com Parâmetros

```
var x : Integer;

procedure P3( a : Integer, b : Integer ) is
   var n : Integer;
begin
   ...
end P3;

begin
   ...
   P3( 2, 5 );
   ...
end.
```

Exemplo de Subprograma com Parâmetros

Registro de Ativação para o Procedimento P3



Referenciando Variáveis e Parâmetros para o Procedimento P3

`LDLADDR -8`	Carrega o endereço do parâmetro a na pilha de execução.
`LDLADDR -4`	Carrega o endereço do parâmetro b na pilha de execução.
`LDLADDR 8`	Carrega o endereço da variável local n na pilha de execução.
`LDGADDR 0`	Carrega o endereço da variável global x na pilha de execução.

Parâmetros Variáveis (`var`)

- Para parâmetros variáveis (`var`), o endereço do argumento é passado, ou seja, o valor contido no parâmetro formal é o endereço do argumento;
- Usaremos essas duas instruções para carregar (empilhar) o endereço do argumento na pilha.

Exemplo de Subprograma com Parâmetros Variáveis

```
var x : Integer;

procedure P4( var a : Integer, b : Integer ) is
   var n : Integer;
begin
   ...
end P4;

begin
   x := 5;
   P4( x, 6 );
end.
```

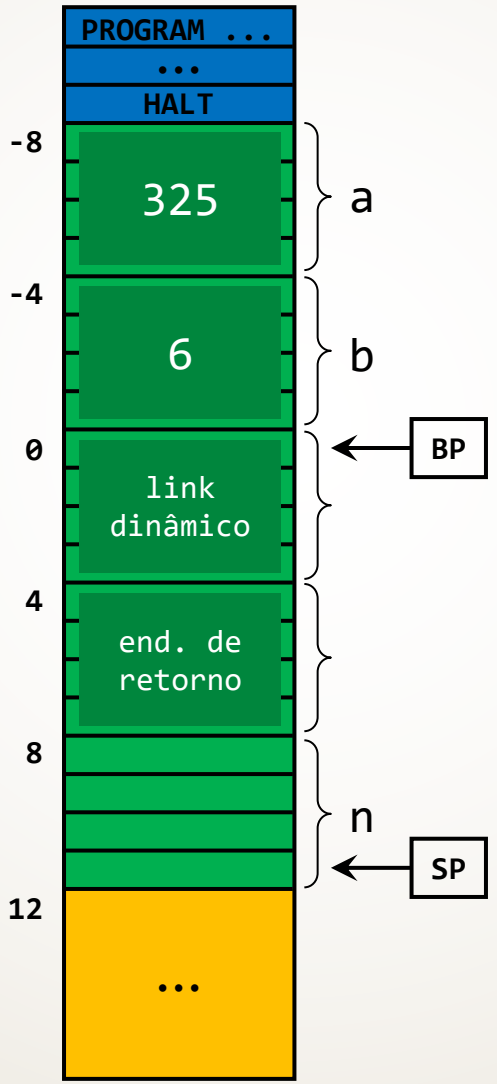
Note que o parâmetro a é um parâmetro variável.

Registro de Ativação para o Procedimento P4

Após a Chamada à P4(x, 6)

o endereço absoluto de x é 325

endereços relativos à BP



sem a parte do valor de retorno

parte dos parâmetros

parte contextual

parte das variáveis locais

parte temporária

Referenciando Variáveis e Parâmetros para o Procedimento P4

```
LDLADDR -8
LOADW
```

Carrega o endereço do argumento x na pilha de execução.

```
LDLADDR -4
```

Carrega o endereço do parâmetro b na pilha de execução.

```
LDLADDR 8
```

Carrega o endereço da variável local n na pilha de execução.

```
LDGADDR 0
```

Carrega o endereço da variável global x na pilha de execução.

O Método `emit()` para a Classe `Variable`

Carrega o Endereço da Variável na Pilha

```
@Override
public void emit() throws CodeGenException, IOException {

    if ( decl instanceof ParameterDecl &&
         ( ( ParameterDecl ) decl ).isVarParam() ) {

        // o endereço do argumento é o valor do parâmetro variável (var)
        emit( "LDLADDR " + decl.getRelAddr() );
        emit( "LOADW" );

    } else if ( decl.getScopeLevel() == ScopeLevel.PROGRAM ) {
        emit( "LDGADDR " + decl.getRelAddr() );
    } else {
        emit( "LDLADDR " + decl.getRelAddr() );
    }

}
```

O código desse método ainda será estendido quando formos tratar da implementação dos arrays da CPRL.

Bibliografia

MOORE JR., J. I. **Introduction to Compiler Design: an Object Oriented Approach Using Java**. 2. ed. [s.l.]:SoftMoore Consulting, 2020. 284 p.

AHO, A. V.; LAM, M. S.; SETHI, R. ULLMAN, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson, 2008. 634 p.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. 2. ed. Rio de Janeiro: Campus Elsevier, 2014. 656 p.

JOSÉ NETO, J. **Introdução à Compilação**. São Paulo: Elsevier, 2016. 307 p.

SANTOS, P. R.; LANGOLOIS, T. **Compiladores: da teoria à prática**. Rio de Janeiro: LTC, 2018. 341 p.