Geração de Código para uma Máquina Hipotética a Pilha

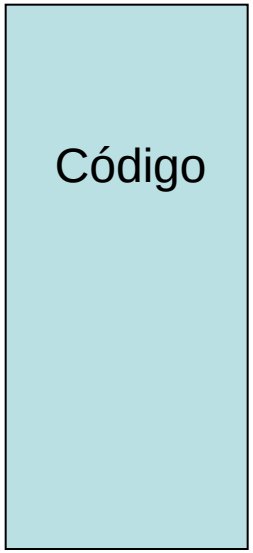
Detalhes da geração de código

Usando a técnica *ad hoc, amarrada aos procedimentos sintáticos, igual à análise semântica*

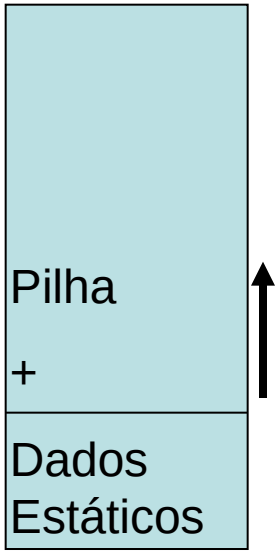
(Kowaltowski, capítulo 10)

MEPA

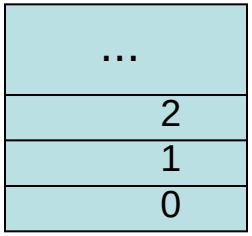
- É composta de 3 Regiões
- Não tem HEAP, pois FRANKIE não permite variáveis dinâmicas



Região de programa: P(i)



Região da Pilha de Dados: M(s)



Display: D(b)

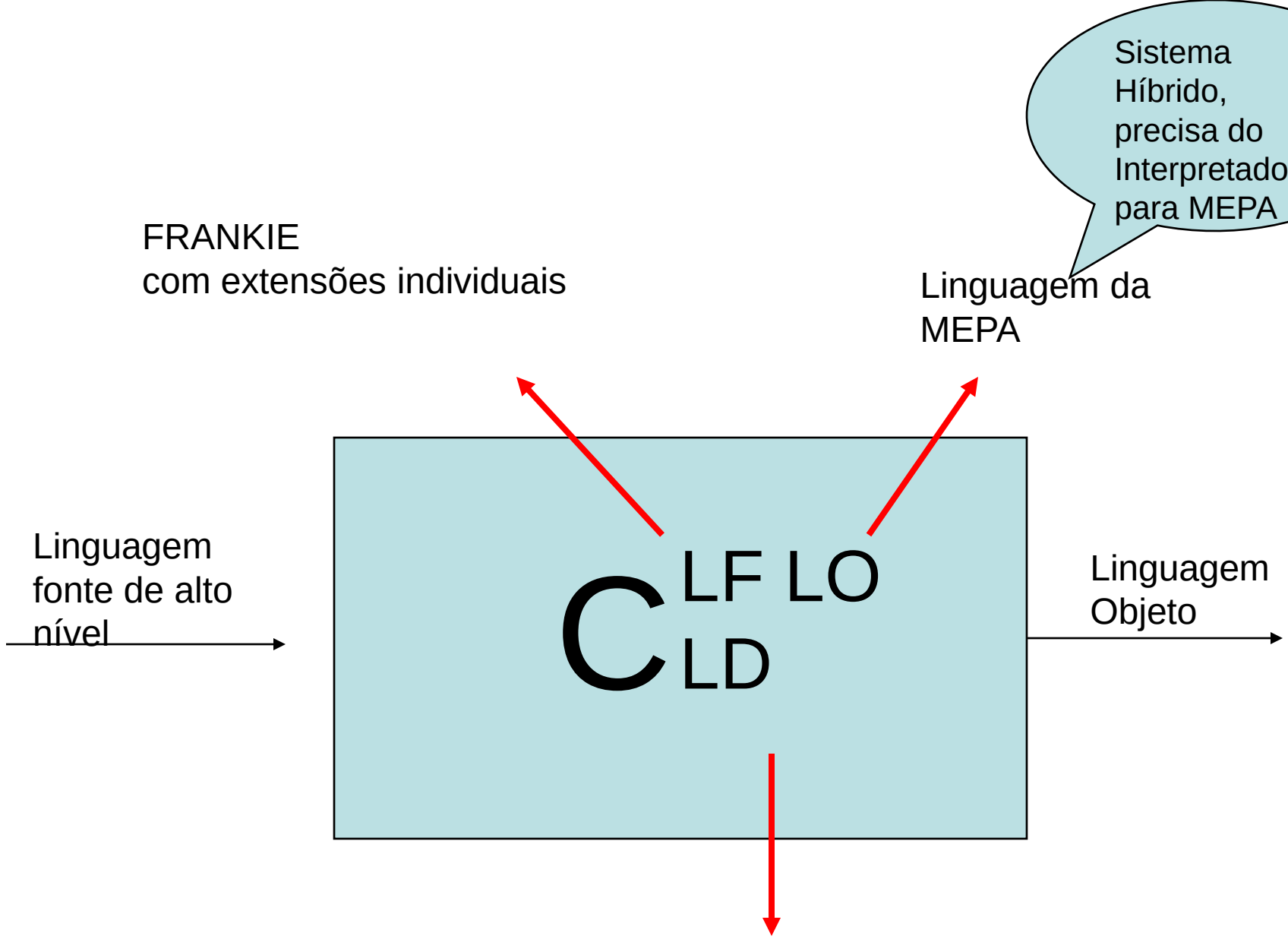
(Vetor dos Registradores de Base que apontam para M)

Geração de Código

- Uma vez que o programa da MEPA está carregado na região P, e os registradores têm seus valores iniciais, o funcionamento da máquina é muito simples:
  - As instruções indicadas pelo registrador i são executadas até que seja encontrada a instrução de parada, ou ocorra algum erro.
  - A execução de cada instrução aumenta de 1 o valor de i, exceto as instruções que envolvem desvios.
- Em nível de projeto, o vetor P será construído pelo compilador que o gera como saída. Esta saída passa a ser a entrada de um programa interpretador deste código.

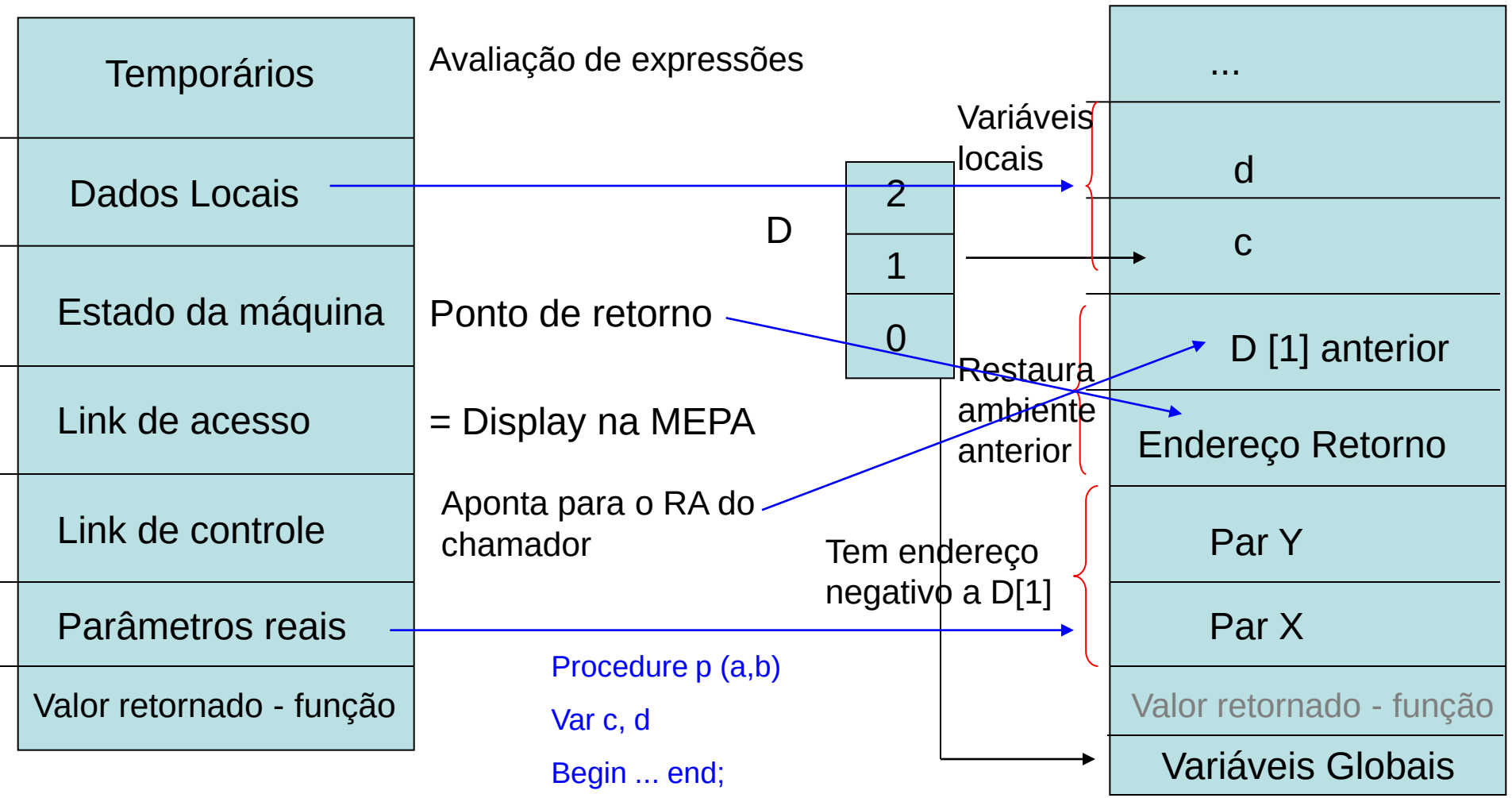
Porque falei em vetor P e não arquivo para P???

- As áreas M e D e os registradores i, s, b, estão fora do escopo do compilador e, portanto, são definidos e manipulados no programa interpretador, como mostra a especificação inicial para nosso projeto:


Sistema Híbrido, precisa do Interpretador para MEPA
FRANKIE
com extensões individuais
Linguagem da MEPA
Linguagem fonte de alto nível
C
LF LO
LD
Linguagem Objeto
JAVA com JavaCC

Registro de ativação

- Gerencia as info necessárias para uma única execução de um procedimento



Geração de Código para a MEPA: implementação Cap 10 de Kowaltowski, p 165 – 172

- MEPA terá uma memória composta de 3 áreas:
  - **A área de Código**, que conterá as instruções geradas pelo compilador.
    - Esta área será simulada como um array - P - onde cada registro é uma instrução ou pseudo-código;
    - ou seja, a Geração de Código consiste no preenchimento deste array que será "interpretado" posteriormente.

- **A área de Dados**, que na realidade é uma pilha que conterá os valores manipulados pelas instruções da MEPA.
  - Esta área só existirá realmente durante a interpretação do Código Gerado.
  - Toda instrução agirá sobre o topo desta pilha - M - ou sobre o topo e seu antecessor, no caso de uma operação binária.

- **A área da pilha dos registradores de base D.**
  - Cada registrador, quando ativo, aponta para um índice de M que marca o início de um novo escopo.

Além das 3 áreas, MEPA possui registradores especiais:

- o registrador de programa i aponta para próxima instrução a ser executada, portanto P(i).
- o registrador s indicará o topo da pilha M, cujo valor será portanto M(s).
- o registrador b indicará o topo da pilha D cujo valor será D(b).

- Como a área M só manipula dados e Frankie básica só manipula inteiros e booleanos, então:
- var

  M: array [0 .. tpilha] of integer;

  s: -1 .. tpilha;

  - O campo endereço da Tabela de Símbolos - TS - deverá ser preenchido com: end_rel (endereço relativo à base na pilha M) e prim_instr (endereço da 1ª instrução no array P, no caso do identificador ser do tipo procedimento ou função).

Instrução: rótulo/nada, código, seguido de 0,1 ou 2 argumentos

- Devemos criar a rotina

  GERA(rótulo, código, par1, par2)

  – para gerar as instruções da MEPA que estão no Apêndice 3 (dado em sala) e dizem respeito à parte A (instruções comuns à versão básica e estendida) e parte B (instruções da parte básica)

  – SEM tratamento de rótulos, passagem de procedimentos e funções como parâmetros.

- e gravá-las no vetor P.

TS implementada estaticamente como uma “pilha”

```
Type categoria = (constante, tipo, variavel, procedimento,funcao, parametro);
        classet = (valor, referencia, procedimento, funcao);
        dim = record
                   inf, sup: integer
                  end;
        item = record
                     ident: string[tam_max];
                     nivel: integer;
                     case categ: categoria of
                       constante: (case tipoc: integer of
                                          1: (valori: integer);
                                          2: (valorc: char);
                                          3: (valorr: real);
                                          4: (valors: string);
                                          5: (valorb: boolean););
                       tipo: (nbytes: integer; dimensao: dim; tipo_elem: integer);
                       procedimento: (npar1: integer; end1; integer);
                       funcao: (npar2: integer; end2: integer; tipo_f: integer);
                       parametro: (classe: classet; end3: integer; tipo_p: integer);
                       variavel: (end4: integer; tipo_v: integer)
                  end;
        TS: record
                 pilha: array [1..max] of item;
                 topo: integer
               end;
```

Processamento de Expressões para Frankie básica

```
procedure termo (var t: string);
var t1, t2: string;
begin
  Fator (t);
  Enquanto simbolo in [cod_*,cod_div, cod_and] faça
  begin
    s1:= simbolo;
    simbolo := analex(s);
    Fator(t1);

    Caso s1 seja
        cod_* : t2 := {´inteiro´; GERA(branco,”MULT”)};
        cod_div: t2 := { ´inteiro´; GERA(branco,”DIVI”)};
        cond_and : t2 := ´booleano´;GERA(branco,”CONJ”)};
    end;
    Se (t <> t1) ou (t <> t2) então erro(´incompatibilidade de tipos´)
  end
end;
```

Operação após os 2 fatores

Expressão, Expressão Simples são similares a termo

- E tratam dos relacionais (**=** | **<>** | **<** | **<=** | **>=** | **>) e sinais com +** | **-** | **or, respectivamente**
- Fator já é bem diferenciado, pois vai cuidar das instruções de CRCT (constante numérica e também as booleanas), CRVL e CRVI (variável simples e parâmetros passados por valor e referência) e da negação

```
procedure fator (var t: string);
Inicio Caso simbolo seja
Número: {t:= ´inteiro´; GERA(branco,”CRCT”,conversão(s)); simbolo := analex(s)};
Identificador:
  {Busca(Tab: TS; id: string; ref: Pont_entrada; declarado: boolean);
  Se declarado = false então erro; Obtem_atributos(ref: Pont_entrada; AT: atributos);
  Caso AT.categ seja
variavel: {t:= AT.tipo_v;GERA(branco, “CRVL”,AT.nivel, AT.end4); simbolo := analex(s);}
parametro: {t:= AT.tipo_p;
       caso AT.classe seja
             valor: GERA(branco,”CRVL”,AT.nivel,AT.end3);
             referencia: GERA(branco,”CRVI”,AT.nivel,AT.end3);
        end; simbolo := analex(s);}
constante: {t:= ‘boolean’;
        caso s seja
             “true”:GERA(branco,”CRCT”,1); “false”:GERA(branco,”CRCT”,0);
        end; simbolo := analex(s);}
função: “tratamento de funções”;
 Else erro; end;
Cod_abre_par: {simbolo := analex(s); expressao(t); se simbolo <> Cod_fecha_par then
     erro; simbolo := analex(s);}
Cod_neg: {simbolo := analex(s); fator(t); se t <> ‘booleano’ então erro;
GERA(branco,”NEGA”)};
Else erro; end;
Fim;
```

Rotina Proximo_Rotulo(L), coloca na variável L o próximo rótulo da forma Li, com i inteiro, positivo

```
Procedure comando_condicional;
Inicio
simbolo := analex(s);{ já foi analisado o “IF”}
Proximo_Rotulo(L1);
Expressão(t); se t <> ‘booleano’ então erro;
GERA(branco,”DSVF”,L1);
Se simbolo <> cod_then então erro;
simbolo := analex(s); comando;
Caso simbolo seja
    cod_else: {Proximo_Rotulo(L2);
    GERA(branco,”DSVS”,L2); GERA(L1,”NADA”);
    simbolo := analex(s); comando; GERA(L2,”NADA”)} ;
    outros: GERA(L1,”NADA”)
Fim
Fim
```

```
        ... } tradução de E
        DSVF L1
        ...} tradução de C1
        DSVS L2
L1      NADA
        ... } tradução de C2
L2      NADA
```

```
        ... } tradução de E
        DSVF L1
        ... } tradução de C
L1      NADA
```

Declaração de Procedimento

- Deve cuidar das instruções:

ENPR nível e

RTPR nível, nro parâmetros

e guardar o endereço do procedimento (obtido da rotina **Proximo_Rotulo(L))** na TS, pois este endereço é usado como rótulo da instrução ENPR e CHPR:

L ENPR nível

CHPR L

A info sobre o nro de parâmetros vem da TS, assim como o nível, que é referente ao corpo do proc e não ao nome do proc

Como distinguir Bloco do PP dos blocos de procedimentos e funções?

- Quando há subrotinas, o controle deve ser deslocado para o pp ser executado primeiro
- Mas como distinguir entre os blocos desde que pp e subrotinas usam as mesmas rotinas sintáticas?

1. <programa> ::=

    program <identificador> ;

    <bloco>.

2. <bloco> ::=

    [<parte de definições de constantes>]

    [<parte de definições de tipos>]

    [<parte de declarações de variáveis>]

    [<parte de declarações de sub-rotinas>]

    <comando composto>

17. <declaração de procedimento> ::=

    procedure <identificador> [<parâmetros formais>] ;

    <bloco>

Bloco

- Responsável por gerar a instrução

  DMEM nro de variáveis ao encontrar um end; (procedimento ou função) ou end. (programa principal)

  – Declaração de variáveis deve passar este número para BLOCO

- Como Bloco embute declaração de procs e funcs,

  ao ser declarado o **primeiro** PROC **ou** a **primeira** FUNCTION uma instrução de **DSVS L1** deve ser gerada, sendo L1 o endereço do programa principal. Também, uma instrução **L1 NADA**, antecedendo a primeira instrução do pp, deve ser gerada.

  Um flag pode ajudar.

Regras de Declaração de Varíáveis

<parte de declarações de variáveis> ::=

var <declaração de variáveis>

{**;** <declaração de variáveis>}**;**

<declaração de variáveis>::=

<lista de identificadores 1> **:** <tipo>

<lista de identificadores 1> ::=

<identificador> {**,** <identificador> }

Endereços das variáveis na TS e a alocação de espaço (AMEM)

```
Procedure parte_declaração_variáveis;
Inicio
simbolo := analex(s); { já foi analisado o “VAR”}
ender:= 0;
Repita
    n:= 0; término := falso;
    repita
    se s <> cod_ID então erro;
    se Declarado(TS, s, nivel_corr) então erro;
    insere(s,variavel,nivel_corr,tipo_indef,ender);
    ender := ender + 1; n := n + 1; simbolo := analex(s);
    caso s seja
            cod_virg: nada;
            cod_dois_pontos: termino := verdadeiro
            outros: erro
    fim; simbolo := analex(s);
Até termino;
GERA(branco, “AMEM”,n);  .... MAIS PARTES AQUI ....
Até s <> cod_ID
Fim;
```