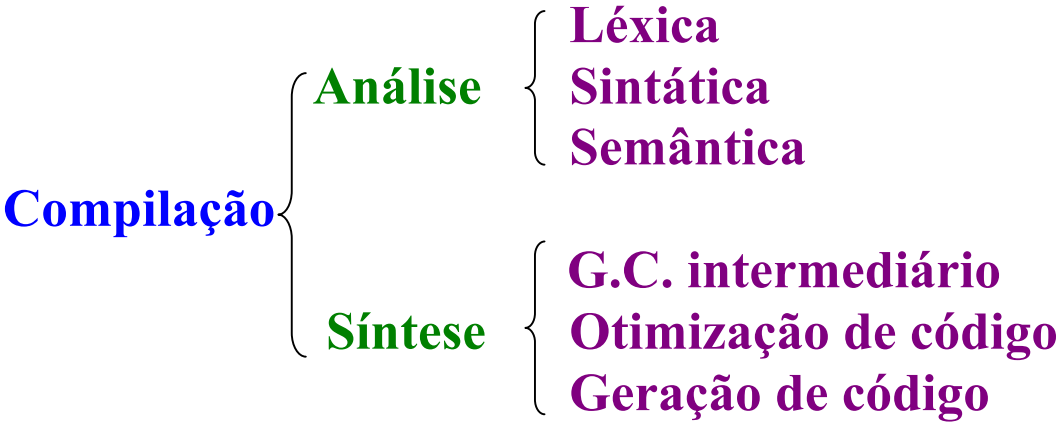
CAP. VII – GERAÇÃO DE CÓDIGO

VII . 1 - INTRODUÇÃO



- **Síntese**

  - **Tradução do programa fonte (léxica, sintática e semanticamente correto) para um programa objeto equivalente.**

  - **Estabelecimento de um significado (uma semântica) para o programa fonte, em termos de um código executável (diretamente ou via interpretação).**

- **Esquema de tradução dirigida pela sintaxe**
  - **Geração de cód. Interm. ou executável.**
  - **Uso de ações de geração de código**
    - **Similares às ações semânticas**
    - **Inseridas na G.L.C.**
    - **Ativadas pelo Parser ou A. Semântico**
    - **pode ser integrado ao semântico**
  - **Generalizando:**
    - **Ações semânticas**
      - **Ações de verificação**
      - **Ações de geração de código**
  - **Na prática ...**
    - **Unificação de ações de Verificação semântica e de Geração de Código**

CÓD. INTERMEDIÁRIO X CÓD. BAIXO NÍVEL

MÁQUINA HIPOTÉTICA X MÁQUINA REAL

VII.2 - CÓDIGO INTERMEDIÁRIO

Representação intermediária entre o programa fonte (L. alto nível) e o programa objeto (L. baixo nível).

- **Vantagens**
  - **Geração menos complexa**
    - **Abstração de detalhes da máquina real**
    - **Repertório de instruções definido em função das construções da linguagem fonte**
  - **Facilidades para geração de código executável em um passo subseqüente**
    - **Representação uniforme**
  - **Facilita otimização**
    - **Redução do tempo de execução e/ou do tamanho do código gerado**

- Possibilita interpretação
- Aumento da portabilidade
  - Diferentes interpretadores ou geradores de código para diferentes máquinas
  - Base necessária para construção de Just-in-time Compilers
  - Exemplos: máquina P (Pascal)
    JVM (Java)
- Facilita extensibilidade
  - Via introdução de novas construções
  - Via sofisticação do ambiente operacional

- Desvantagens
  - Acréscimo do tempo de compilação
    - Passo extra para geração de código objeto a partir do código intermediário.
  - Aumento do projeto total
    - Necessidade de definição de uma Linguagem intermediária.
    - Necessidade de definição de uma máquina hipotética
    - Necessidade de um interpretador para validação do código intermediário
  - Quando o C.I. é o código alvo ...
    - Tempo de execução (via interpretação) maior se comparado com cód. compilado.

• Formas de código intermediário

1. Triplas e Quádruplas

- Instruções
  - Operador
  - Operando 1, operando 2, [resultado]
- Operadores
  - Aritméticos, lógicos, armaz., desvio, carga, ...
  - Exemplos:

Triplas : w * x + (y +z)

(1) * , w , x
(2) + , y , z
(3) + , (1), (2)

Quádruplas: (A + B) * (C + D) – E

+ , A , B, T_1
+ , C , D, T_2
* , T_1 , T_2, T_3
- , T_3 , E, T_4

OBS.:

1. Quádruplas facilitam a otimização de código
2. Algoritmos de otimização clássicos (AHO e ULLMAN) são todos baseados em quádruplas.
3. Gerenciamento de temporárias é problemático

2. - Máquinas de PILHA

(de Acumulador, ou de um Operando)

- O código é similar a um assembler simplificado
- Usa registradores apenas para funções especiais
- Usa uma PILHA para armazenar valores de Variáveis (globais e locais) e Resultado das operações realizadas.
- Formato das Instruções
  - Operador (Código da operação)
  - Operando – o qual pode ser:
    - referência a uma variável
    - valor constante
    - endereço de uma instrução
- Exemplo: D := A + B * C

```
1 - LOAD  A     4 - MULT  -
2 - LOAD  B     5 - SOMA  -
3 - LOAD  C     6 - ARMZ  D
```

Usando Endereço Relativo

Variável	Nível	Deslocamento
A	1	0
B	1	1
C	1	2
D	1	3

```
LOAD  1,0
LOAD  1,1
LOAD  1,2
MULT  -,-
SOMA  -,-
ARMZ  1,3
```

3. Outras formas de CI

- Notação Polonesa, Árvores Sintáticas Abstratas

VII.3 - Máquinas Hipotéticas (virtuais, abstratas)

- **Destinam-se a produção de compiladores e interpretadores**
  - **+ Portáveis**
  - **+ Adaptáveis**

- **Composição**
  - **Arquitetura**
    - **Área de código**
    - **Área de dados (pilha)**
    - **Registradores**
      - **Uso geral**
      - **Uso específico**
  - **Repertório de instruções**
    - **Compõem a linguagem de máquina da "MÁQUINA VIRTUAL"**
    - **Normalmente é o próprio CI**
    - **Diretamente relacionado às construções da linguagem fonte**
    - **Formato das instruções**
      - **Dependente da arquitetura**
      - **Pode ou não ser uniforme**

        Ex: OP, operando

        OP, operando1, operando2

        OP, operando1, operando2, operando3

- **Interpretador**
  - **Simula o "Hardware" na execução do código da "máquina"**

    - **Exemplos**
      - **Máquina P (Pascal)**
        - **Pilha, CI → triplas**
      - **JVM (Java)**
        - **Pilha, CI → não uniforme**

VII.4 - UMA MÁQUINA HIPOTÉTICA DIDÁTICA

OBJETIVOS:

- **Visão completa do processo de compilação.**
- **Noção do processo de geração de código.**
- **Permitir a interpretação de programas exemplos**
  - **Validar ações semânticas**
    - **Verificação**
    - **Geração de código**

Definição da Arquitetura

(* Baseada na máquina P, simplificada *)

- Área de instruções
  - Contém as instruções a serem executadas
- Área de dados
  - Alocação de dados manipulados pelas inst.
  - Estrutura de pilha
    - Cada célula (≡) 1 palavra (inteira)
    - Contem:
      - valores de constantes
      - valores assumidos por var.
      - ponteiro para estruturas
      - resultados intermediários
  - Registradores de uso específico
    - PC → apontador de instruções
      - Aponta para a próxima instrução a ser executada (área de instruções)
    - Topo → Aponta para o topo da pilha usada como área de dados
    - Base → Aponta para o endereço (pos. na pilha) inicial de um segmento de dados
      → Usado no cálculo de endereços (endereço = base + desloc.)

- **Definição do repertório de instruções**
  - **Definição do código intermediário**
  - **Forma geral das instruções**

OPERADOR	OPERANDO

OPERADOR - código da instrução (mnemônico)
OPERANDO - subdividido em PARTE1 e PARTE2

- **O significado depende da instrução - Exemplos:**
  - **1 – Instruções que referenciam endereços:**
    - **PARTE1 – Nível**
    - **PARTE2 – Deslocamento**
  - **2 – Instruções aritméticas**
    - **PARTE1 e PARTE2 são nulos!**
    - **(Operam sobre topo/sub-topo da pilha)**
  - **3 – Instruções de desvio**
    - **PARTE1 – Nulo**
    - **PARTE2 – Endereço de uma Instrução**

- **Grupos de instruções**
  - **Aritméticas, lógicas e relacionais**
  - **Carga/armazenamento**
  - **Alocação de espaço para variáveis**
  - **Fluxo**
    - **Desvios**
    - **Chamada / retorno de procedimento**
  - **Específicas** { **Leitura, impressão** / **Inicio e Fim de execução** / **Nada (nop)**

→ Instruções de carga e armazenamento

- **CRVL l , a** (* carrega valor de variável *)
  onde: l – nível; a – deslocamento

  Topo: = * + 1
  Pilha [topo] : = Pilha [base (l) + a]

- **CRCT __ , K** (* carrega constante *)
  onde K é o valor da constante

  Topo : = topo + 1
  Pilha [topo] : = K

- **ARMZ l , a** (* armazena conteúdo do topo da pilha no endereço (l + a) da pilha *)
  pilha [base (l) + a] : = pilha [topo]
  topo : = topo – 1

→ Instruções aritméticas

- **SOMA __ , __** (* operação de adição *)

  pilha [topo – 1] : = pilha [topo - 1] + pilha [topo]
  topo : = topo – 1

- **SUB __ , __** (* operação de subtração *)
- **MULT __ , __** (* operação de multiplicação *)
- **MUN __ , __** (* Menos UNário – muda sinal *)

→ Instruções lógicas

- **CONJ __ , __ (* operação "and" ≡ "E" *)**
  se pilha [topo – 1] = 1 e pilha [topo] = 1
  então pilha [topo – 1] : = 1 (true) "verdadeiro"
  senão pilha [topo – 1] : = ø (false) "falso";
  topo : = topo – 1

- **DISJ __ , __ (* Operação "OR" ≡ "ou" *)**
  Se pilha [topo – 1] = 1 ou [pilha topo] = 1
  ... idem CONJ ...

- **NEGA __ , __ (* Operação "NOT" ≡ "Não" *)**
  Pilha [topo] : = 1 – pilha [topo]

→ Instruções relacionais

- **CMIG __ , __ (* Compara igual "=" *)**
  Se pilha [topo – 1] = pilha [topo]
  Então pilha [topo – 1] : = 1 (true)
  Senão pilha [topo – 1] : = ø (false)
  Topo: = topo – 1
- **CMDF __ , __ (* Compara diferente "<>" *)**
- **CMMA __ , __ (* Compara maior ">" *)**
- **CMME __ , __ (* Compara menor "<" *)**
- **CMEI __ , __ (* Compara menor igual "<=" *)**
- **CMAI __ , __ (* Compara maior igual ">=" *)**

→ Instruções de desvio

- **DSVS __ , a (* desvia sempre para a instrução "a" *)**
  PC := a

- **DSVF __ , a (*se falso, desvia para "a" *)**
  se pilha [topo] = 0 (* falso *)
  entao PC := a;
  topo := topo – 1

- **CALL l , a (* chamada de procedimento *)**
- **RETU __ , __ (* retorno de procedimento *)**

→ Alocação de espaço

- **AMEM __ , a (* aloca "a" posições de memória *)**
  topo := topo + a

- **DMEM __ , a (* Desaloca "a" posições de memória *)**
  topo := topo - a

→ Entada / Saída

(* operam com valores e endereços do topo da pilha *)

- **LEIA __ , __ (* lê valor numérico *)**
- **IMPR __ , __ (* imprime valor numérico *)**
- **IMPRLIT __ , __ (* imprime literal *)**

→ Instruções auxiliares

- **INICIO __ , __ (* define inicio da execução *)**
- **NADA __ , __ (* nada faz ! *)**
- **FIM __ , __ (* define término da execução *)**

REPERTÓRIO DE INSTRUÇÕES DA MV

```
INICIO   __, __   (* Inicio da interpretação *)
AMEM      __, a   (* Aloca "a" posições de memória *)
DMEM      __, a   (* Desaloca "a" posições de memória *)
CRVL      l , a   (* Carrega Variável *)
CVET1     l , a   (* Carrega Variável indexada uni-dim. *)
CVET2     l , a   (* Carrega Variável indexada bi-dim. *)
CRVLIND   l , a   (* Carrega Valor indiretamente*)
CREN      l , a   (* Carrega Endereço *)
CRCT      __, k   (* Carrega constante *)
ARMZ      l , a   (* Armazena em uma var. *)
AVET1     l , a   (* Armazena em uma var. indexada uni-dim.*)
AVET2     l , a   (* Armazena em uma var. indexada bi-dim.*)
ARMZIND   l , a   (* Armazena de forma indireta *)
SOMA     __, __   (* Adição *)    SUB   __, __   (* Subtração *)
MULT     __, __   (* Multiplicação *)
DIVI     __, __   (* Divisão de inteiros*)
DUP      __, __   (* Duplica o valor do topo da pilha*)
MUN      __, __   (* Menos Unário *)
CONJ     __, __   (* And *) ∴ "E"  DISJ __, __   (* Or *) ∴ "Ou"
NEGA     __, __   (* Not *) ∴ "Não"
CMIG     __, __   (* = *)     CMDF     __, __   (* <> *)
CMMA     __,__    (* > *)     CMME     __, __   (* < *)
CMEI     __,__    (* <= *)    CMAI     __, __   (* >= *)
DSVS      __,  a  (* Desvia sempre *)
DSVF      __,  a  (* Desvia se falso *)
CALL      l ,  a  (* Chama procedimento *)
RETU     __, __   (* retorna de procedimento *)
LEIA     __, __   (* Lê valor *)
IMPR     __, __   (* Imprime valor numérico*)
IMPRLIT  __, __   (* Imprime literal *)
NADA     __, __   (* Nada faz *)
FIM      __, __   (* Finaliza execução *)
```

VII.5 - Geração de Código Intermediário

1. Alocação de espaço para as variáveis

```
var A, B, C: inteiro;                    AMEM - , 3
var X, Y: vetor[1 .. 10] de inteiro;     AMEM - , 20
```

2. Comando de Leitura

```
Leia ( X ) :      LEIA   - , -
                  ARMZ  X (* end. Relativo de X *)
```

3. Comando escreva

```
Escreva ( ' total = ', tot )
        CRCT    -, ind-tab-lit
        IMPRLIT  - , -
        CRVL    tot   (* end. Relativo de tot *)
        IMPR    - , -
```

4. Comando de atribuição

- **Forma geral:** VAR := EXPR

```
____
____   } CÓDIGO p/ EXPR
____
ARMZ   VAR
```

- **Exemplo de um programa:**

```
Programa ex1;
var A, B, C: inteiro;
{
     leia (A, B);
     C := (A + B) * ( A – B )
     escreva ("resultado = ", C);
}.
```

5. Comando enquanto-faca

- **Forma geral:** **enquanto < E > faca < C >**

```
L1:   NADA  - , -
      ____
      ____      } <E>
      ____
      DSVF      - , L2
      ____
      ____      } <C>
      ____
      DSVS      - , L1
L2:   NADA      - , -
```

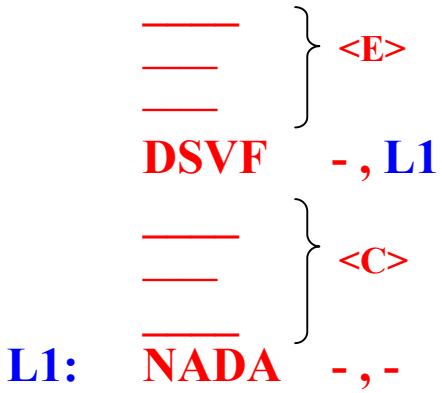
- **Exemplo:**

```
...
I:= 1;
enquanto I < N faça
{
      escreva (I, I*I);
      I:= I + 1;
}
...
```

6. Comando se-entao / se-entao-senao

6.1 – se-então

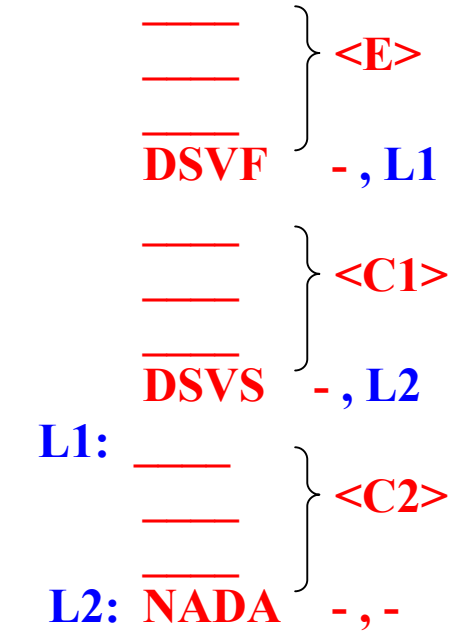
- **Forma geral : se <E> entao <C>**



- **Exemplo : se A > B então MAIOR := A**

6.2 - se-entao-senao

- **Forma geral : se <E> entao <C1> senão <C2>**



- **Exemplo : se A > B**
  então MAIOR := A
  senão MAIOR := B

7. Estruturas de controle aninhadas

- **Enquanto-faca**
  - **Forma Geral: enquanto < E > faca < C >**
  - **Pilha de controle ∴ PENQ [TPE]**
  - **Ações de geração de código**

    <C> ::= enquanto #w1 <E> #w2 faca <C> #w3

    #w1 – Gera instrução NADA __, __
    Guarda endereço da instr. NADA __,__
    em PENQ - TPE:= * +1
    PENQ [TPE]:= PC

    #w2 – Gera instrução DSVF __, ?
    Guarda endereço de DSVF em PENQ

    #w3 – Completa DSVF do topo de PENQ com
    PC + 1 (próxima instrução)
    Decrementa TPE → TPE:= * - 1
    gera DSVS __, PENQ [TPE]
    Decrementa TPE → TPE:= * - 1

  - **Exemplo:**

```
...
I:= 1;
enquanto I < N faca
{
      K:= 1;
      enquanto K < M faca
      {
         escreva (I * K);  K:= K + 1;
      };
      I:= I + 1;
};
```

- **Se-entao-senao**
  - **Forma Geral: se < E > entao < C1 > senao < C2>**
  - **Pilha de controle ∴ PSE [TPSE]**
  - **Ações de geração de código**

    <C> ::= se <E> #Y1 entao <C> <else-parte> #Y3

    <else-parte> ::= #Y2 senao <C> | ε

    #Y1 - gera DSVF __, ?
    guarda endereço na PSE

    #Y2 - completa DSVF do topo de PSE
    com PC + 1; decrementa TPSE
    gera DSVS __, ?
    guarda endereço na PSE

    #Y3 - completa instrução do topo de PSEF
    → PSE [TPSE]:= PC
    decrementa TPSE

  - **Exemplo: se A > B**
    entao se A > C
    entao escreva (A)
    senão escreva (C)
    senão se B > C
    entao escreva (B)
    senao escreva (C);

8. Constantes com Tipo ≠ Pilha

CRCT __, K ⟶ CRCT 1, Ind.Tab.Literais
CRCT 2, Ind.Tab.Reais

9. Variáveis Simples com Tipo ≠ Pilha

- **Carga / Armazenamento**

  CRVL l , a - CRVLX l , i ← i: Ind.Tab.Tipo_X

  ARMZ l , a ARMZX l , i

  nível deslocamento tipo nível

- **Operações (Aritméticas, Lógicas, E/S)**

  CMIG __, __ { CMIGX __, __ ∴ x = tipo var
  CMIG X , __

  SOMA __, __ { SOMAX __, __
  SOMA X, __

  LEIA __, __ { LEIAX __, __
  LEIA X, __

  IMPR __, __ { IMPR 1 , __ (ou IMPLIT __,__)
  IMPR 2 , __ (ou IMPREAL __,__)