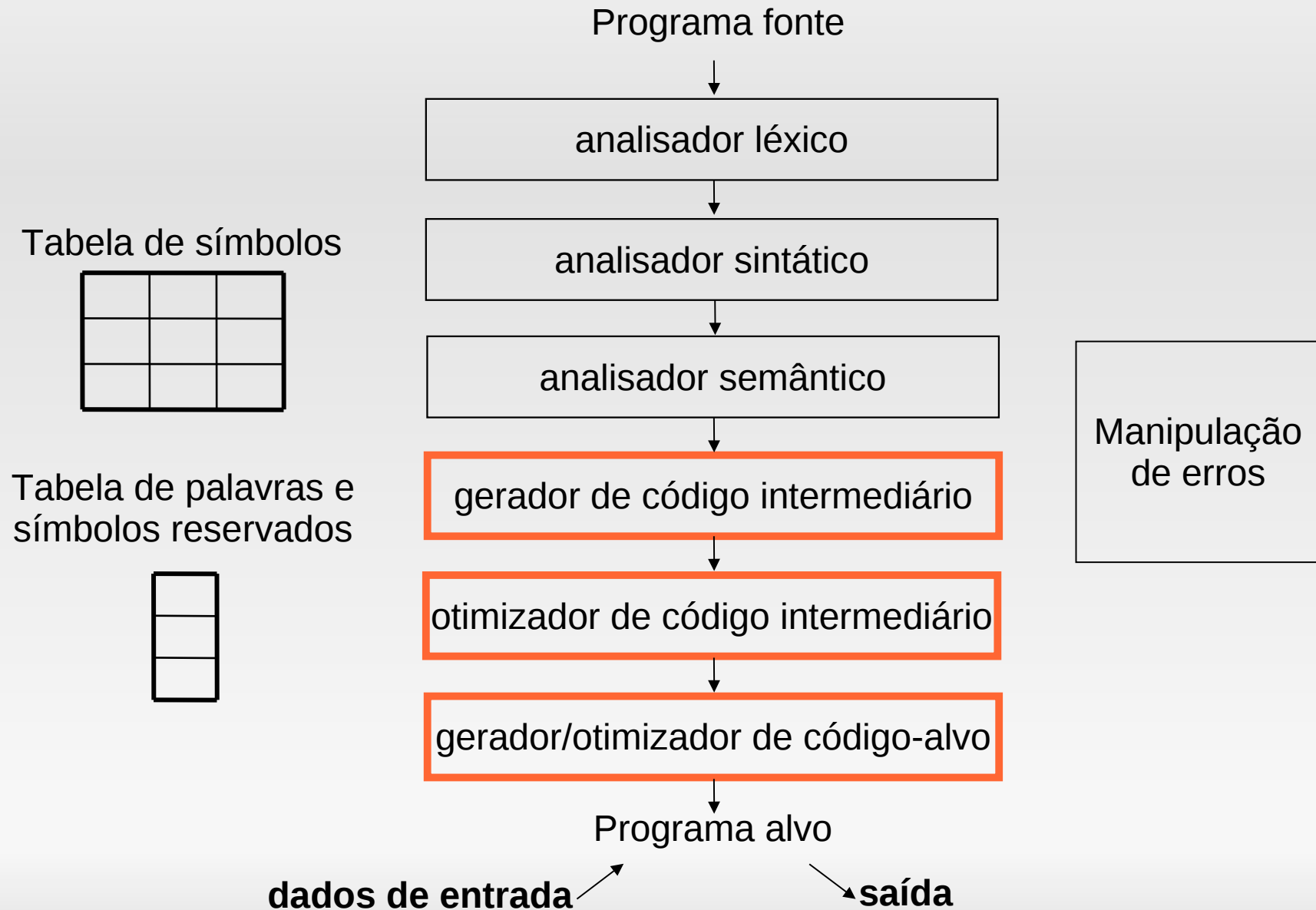


# Construção de Compiladores

Geração de Código  
Otimização de Código

Profa. Helena Caseli  
[helenacaseli@dc.ufscar.br](mailto:helenacaseli@dc.ufscar.br)

# Processo de Tradução



# Geração de Código

- O que é?
  - Etapa na qual o programa-fonte (estático) é transformado em código-alvo (executável)
  - Etapa mais complexa de um compilador, pois depende de
    - Características da linguagem-fonte
    - Informações detalhadas da arquitetura-alvo
    - Estrutura do ambiente de execução
    - Sistema operacional da máquina-alvo
- ➔ Envolve também, normalmente, tentativas de otimizar ou melhorar a velocidade e/ou tamanho do código-alvo

# Geração de Código Intermediário

- O que é?
  - Etapa na qual é gerada uma interpretação intermediária explícita para o programa fonte
- Código intermediário X Código alvo
  - O código intermediário não especifica detalhes
    - Quais registradores serão usados, quais endereços de memória serão referenciados etc.
  - Geração de código em mais de um passo
    - Vantagens
      - Otimização de código intermediário
      - Simplificação da implementação do compilador
      - Tradução de código intermediário para diversas máquinas
    - Desvantagem
      - Compilador requer um passo a mais

# Geração de Código Intermediário

- Entrada
  - Uma representação intermediária do programa-fonte
    - Árvore sintática abstrata
  - Tabela de Símbolos
- Saída – Código intermediário
  - Representação intermediária "mais próxima" ao código-alvo
  - Diferentes formas de acordo com
    - Maior ou menor nível de abstração
    - Uso ou não de informação detalhada da máquina alvo
    - Incorporação ou não de informações na Tabela de Símbolos
  - ➔ Linearização da árvore sintática

# Geração de Código Intermediário

- Duas formas populares
  - Código de três endereços
  - P-código

# Geração de Código Intermediário

- Código de três endereços
  - Instrução básica
    - Avaliação de expressões aritméticas

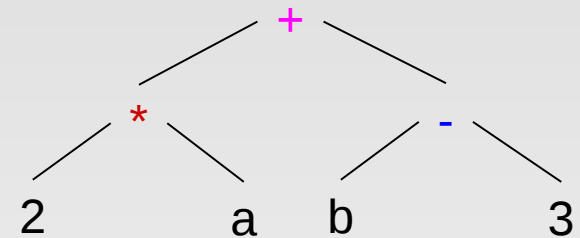
$$x = y \text{ op } z$$

- *op* é um operador aritmético (+ ou -, por exemplo) ou algum outro operador que possa operar sobre os valores de *y* e *z*
- O nome advém dessa forma de instrução na qual ocorre, no máximo, a manipulação de 3 endereços na memória: *x*, *y* e *z*
  - Atenção: o uso de *x* é diferente do uso de *y* e *z*
- São necessárias outras formas para expressar as características da linguagem e esse é um dos motivos de não ser padronizado

# Geração de Código Intermediário

- Código de três endereços
  - Exemplo

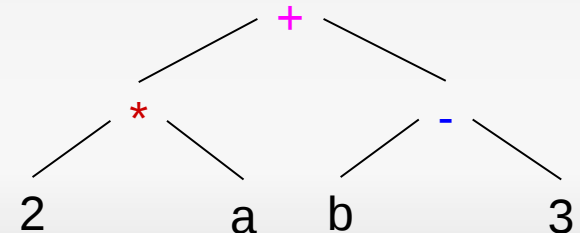
$2*a+(b-3) \longrightarrow$   
 $t1 = 2 * a$   
 $t2 = b - 3$   
 $t3 = t1 + t2$



- O código de três endereços foi obtido a partir da árvore sintática
  - Por meio da linearização da esquerda para a direita, ou seja, percurso em pós-ordem

- Outra possibilidade

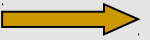
$2*a+(b-3) \longrightarrow$   
 $t1 = b - 3$   
 $t2 = 2 * a$   
 $t3 = t2 + t1$

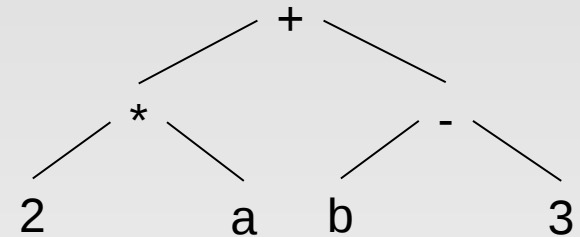




# Geração de Código Intermediário

- Código de três endereços
  - Exemplo

$2*a+(b-3)$    $t1 = 2 * a$   
 $t2 = b - 3$   
 $t3 = t1 + t2$

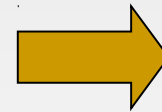


- Temporários
  - Os nomes (t1, t2 e t3) devem ser diferentes dos nomes existentes no código-fonte
  - Correspondem aos nós interiores da árvore sintática e representam seus valores computados
  - O temporário final (t3) representa o valor da raiz
  - Podem ser mantidos na memória ou em registradores

# Geração de Código Intermediário

- Código de três endereços
  - Exemplo – programa que computa o fatorial (em uma linguagem fictícia) e um possível código de três endereços

```
{ Programa exemplo
  -- computa o fatorial
}
read x; { inteiro de entrada }
if 0 < x then { não computa se x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { fatorial de x como saída }
end
```



```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

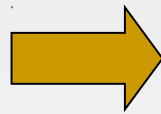
# Geração de Código Intermediário

- Código de três endereços
  - Normalmente não é implementado em forma de texto
  - Implementação com estruturas de dados
    - Cada instrução = uma estrutura de registros com vários campos (uma quádrupla)
      - Um para a operação
      - Três para os endereços
        - Para instruções com menos de 3 endereços, há campos vazios
        - Ponteiros para os nomes na Tabela de Símbolos podem ser usados
    - Sequência de instruções = uma matriz ou lista ligada

# Geração de Código Intermediário

- Código de três endereços
  - Exemplo – **quádruplas** para o código de três endereços do programa que computa o fatorial

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

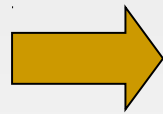


```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

# Geração de Código Intermediário

- Código de três endereços
  - Exemplo – **triplas** para o código de três endereços do programa que computa o fatorial

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```



```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

```
(0) (rd,x,_)
(1) (gt,x,0)
(2) (if_f,(1),(11))
(3) (asn,1,fact)

(4) (mul,fact,x)
(5) (asn,(4),fact)
(6) (sub,x,1)
(7) (asn,(6),x)
(8) (eq,x,0)
(9) (if_f,(8),(4))
(10) (wri,fact,_)

(11) (halt,_,_)
```

# Geração de Código Intermediário

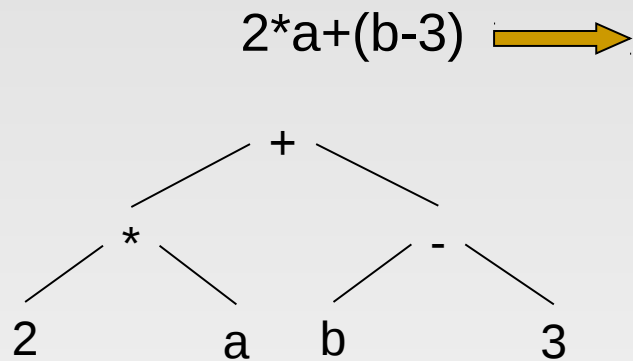
- P-código
  - Surgiu como um código de montagem-alvo padrão produzido pelos compiladores Pascal
  - Foi projetado como **código de uma máquina hipotética** baseada em pilhas (P-máquina) com interpretador para diversas máquinas reais
    - A ideia era facilitar a **portabilidade** – apenas o interpretador da P-máquina deveria ser reescrito para uma nova plataforma
  - Se mostrou útil também como **código intermediário**
    - Diversas extensões e modificações têm sido usadas em diversos compiladores de código nativo, a maioria para linguagens derivadas de Pascal

# Geração de Código Intermediário

- P-código
  - Características
    - Projetado para ser executado diretamente, então contém uma descrição implícita de um ambiente de execução
      - Informações específicas da P-máquina (tamanho de dados, formação da memória etc.)
    - Representação e implementação similares ao código de três endereços
    - Neste curso usaremos uma versão abstrata simplificada da P-máquina
      - Memória de código
      - Memória de dados (não especificada para variáveis com nomes)
      - Pilha de dados temporários
      - Registradores para manter a pilha e dar suporte à execução

# Geração de Código Intermediário

- P-código
  - Exemplo



ldc 2	; carrega constante 2
lod a	; carrega valor da variável a
mpi	; multiplicação de inteiros
lod b	; carrega valor da variável b
ldc 3	; carrega constante 3
sbi	; subtração de inteiros
adi	; adição de inteiros



# Geração de Código Intermediário

- P-código
  - Exemplo

$x := y + 1$



```
lđa x      ; carrega endereço de x
lod y      ; carrega valor de y
lđc 1      ; carrega constante 1
adi        ; adição
sto        ; armazena topo no endereço
           ; abaixo do topo & retira os dois
```

# Geração de Código Intermediário

- P-código
  - Exemplo - programa que computa o fatorial (em uma linguagem fictícia) e seu P-código

```
{ Programa exemplo
  -- computa o fatorial
}
read x; { inteiro de entrada }
if 0 < x then { não computa se x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { fatorial de x como saída }
end
```

```
lda x      ; carrega endereço de x
rdi        ; lê um inteiro, armazena no
           ; endereço no topo da pilha (& o retira)
lod x      ; carrega o valor de x
ldc 0      ; carrega a constante 0
grt        ; retira da pilha e compara os dois valores do topo
           ; coloca na pilha o resultado booleano
fjp L1     ; retira o valor booleano, salta para L1 se falso
lda fact   ; carrega endereço de fact
ldc 1      ; carrega constante 1
sto        ; retira dois valores, armazena primeiro
           ; em endereço representado pelo segundo
lab L2     ; definição do rótulo L2
lda fact   ; carrega endereço de fact
lod fact   ; carrega valor de fact
lod x      ; carrega valor de x
mpi        ; multiplica
sto        ; armazena topo em endereço do segundo & retira
lda x      ; carrega endereço de x
lod x      ; carrega valor de x
ldc 1      ; carrega constante 1
sbi        ; subtrai
sto        ; armazena (como no caso anterior)
lod x      ; carrega valor de x
ldc 0      ; carrega constante de 0
equ        ; teste de igualdade
fjp L2     ; salta para L2 se falso
lod fact   ; carrega valor de fact
wri        ; escreve topo da pilha & retira
lab L1     ; definição do rótulo L1
stp
```

# Geração de Código Intermediário

- Código de três endereços X P-código
  - Código de três endereços
    - É mais compacto (menos instruções)
    - É auto-suficiente
      - Não precisa de uma pilha para representar o processamento
- P-código
  - Mais próximo do código de máquina
  - Com a pilha implícita
    - As instruções exigem menos endereços (nenhum ou apenas um), pois os endereços omitidos estão na pilha
    - O compilador não precisa atribuir nomes aos temporários, pois estes estão na pilha

# Geração de Código Intermediário

- Código de três endereços X P-código
  - Cálculo de endereços
    - Código de três endereços – Notação em C

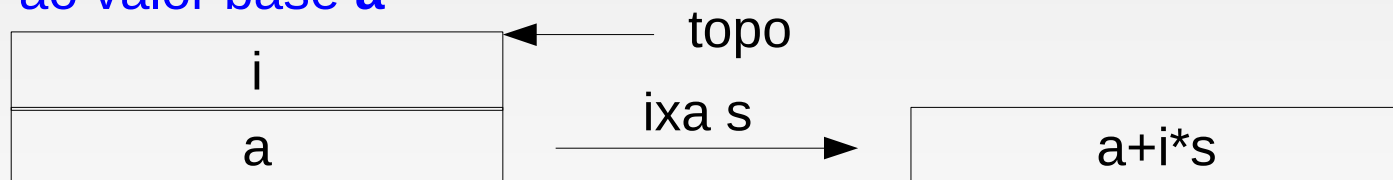
$t1 = \&x + 10$       Armazena o valor constante 2 no  
 $*t1 = 2$               endereço da variável  $x + 10$  bytes

- P-código – 2 novas instruções

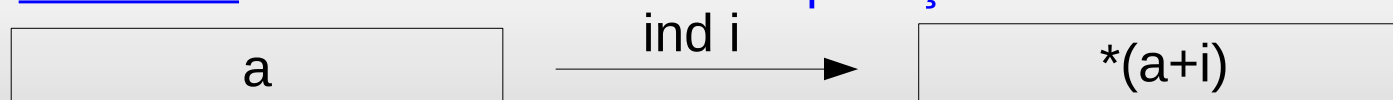
- **ixa** (endereço indexado) – substitui os valores na pilha pelo endereço resultante da aplicação do deslocamento  $i$  na escala  $s$  ao valor base  $a$

lda x  
ldc 10  
ixa 1  
ldc 2  
sto

$x + 10 * 1$



- **ind** (carga indireta) – substitui o endereço na pilha pelo conteúdo no local resultante da aplicação do deslocamento



# Geração de Código Intermediário

- Código de três endereços X P-código
  - Referências de matrizes

$a[i+1] = a[j*2] + 3;$

supondo  $i, j$  inteiros e  $a$  uma matriz de inteiros para a qual  $a[\text{indice}]$  é

$\text{endereço\_base}(a) + (\text{indice} - \text{lim\_inferior}(a)) * \text{tam\_elemento}(a)$

# Geração de Código Intermediário

- Código de três endereços X P-código
  - Referências de matrizes

$a[i+1] = a[j*2] + 3;$

- Código de três endereços

```
t1 = i+1
t2 = t1*elem_size(a)
t3 = &a+t2
t4 = j*2
t5 = t4*elem_size(a)
t6 = &a+t5
t7 = *t6
t8 = t7+3
*t3 = t8
```

- P-código

```
lda a          ixa elem_size(a)
lod i          ind 0
ldc 1          ldc 3
adi            adi
ixa elem_size(a) sto
lda a
lod j
ldc 2
mpi
```

`elem_size(a)`: tamanho do elemento na matriz `a` na máquina alvo (pode ser dado pela tabela de símbolos)

# Geração de Código Intermediário

- Código de três endereços X P-código
  - Declarações de controle e expressões lógicas

if ( E ) S1 else S2

- Código de três endereços

```
<código para t1 = avaliação de E>  
if_false t1 goto L1  
<código para S1>  
goto L2  
label L1  
<código para S2>  
label L2
```

if\_false: testa se t1 é falso  
goto: salto incondicional

- P-código

```
<código para avaliar E>  
fjp L1  
<código para S1>  
ujp L2  
lab L1  
<código para S2>  
lab L2
```

fjp: salta se valor no topo da  
pilha é falso  
ujp: salto incondicional

# Geração de Código Intermediário

- Código de três endereços X P-código
  - Declarações de controle e expressões lógicas

while ( E ) S

- Código de três endereços

```
label L1  
<código para t1 = avaliação de E>  
if_false t1 goto L2  
<código para S>  
goto L1  
label L2
```

- P-código

```
lab L1  
<código para avaliar E>  
fjp L2  
<código para S>  
ujp L1  
lab L2
```



# Geração de Código Intermediário

- Exercício
  - Dada a gramática que representa um subconjunto de expressões em C

$\text{exp} \rightarrow \text{id} = \text{exp} \mid \text{aexp}$   
 $\text{aexp} \rightarrow \text{aexp} + \text{fator} \mid \text{fator}$   
 $\text{fator} \rightarrow (\text{exp}) \mid \text{num} \mid \text{id}$

- Dê o P-código para a expressão  $(x=x+3)+4$

lda x  
lod x  
ldc 3  
adi  
stn  
ldc 4  
adi

DICA: use stn  
- **stn**, assim como **sto**,  
armazena o valor no topo da  
pilha no endereço tb na pilha  
- diferentemente, esse valor  
fica no topo da pilha enquanto  
o endereço é descartado

# Geração de Código Intermediário

- Exercício
  - Dada a gramática que representa um subconjunto de expressões em C

$\text{exp} \rightarrow \text{id} = \text{exp} \mid \text{aexp}$   
 $\text{aexp} \rightarrow \text{aexp} + \text{fator} \mid \text{fator}$   
 $\text{fator} \rightarrow (\text{exp}) \mid \text{num} \mid \text{id}$

- Dê o código de três endereços para a expressão  $(x=x+3)+4$

$t1 = x + 3$   
 $x = t1$   
 $t2 = t1 + 4$

# Geração de Código

- Técnicas para geração de código (intermediário e alvo)
  - Gramática de atributos
    - O código gerado é visto como um **atributo sintetizado** do tipo cadeia de caracteres
    - O código é gerado diretamente durante a análise sintática ou por um percurso em pós-ordem da árvore sintática
  - Procedimentos/funções de geração
    - Baseados na árvore sintática e gramática de atributos
    - *Ad hoc*

# Geração de Código

- Técnicas para geração de código (intermediário e alvo)
  - Gramática de atributos
    - Exemplo – gramática que representa um pequeno subconjunto das expressões em C com cálculo de *pcod*

$exp \rightarrow id = exp \mid aexp$   
 $aexp \rightarrow aexp + fator \mid fator$   
 $fator \rightarrow (exp) \mid num \mid id$

|| = concatena e não pula linha

strval = valor da cadeia

++ = concatena e pula linha

- **stn**, assim como **sto**, armazena o valor no topo da pilha no endereço  
 - diferentemente, esse valor fica no topo da pilha enquanto o endereço é descartado

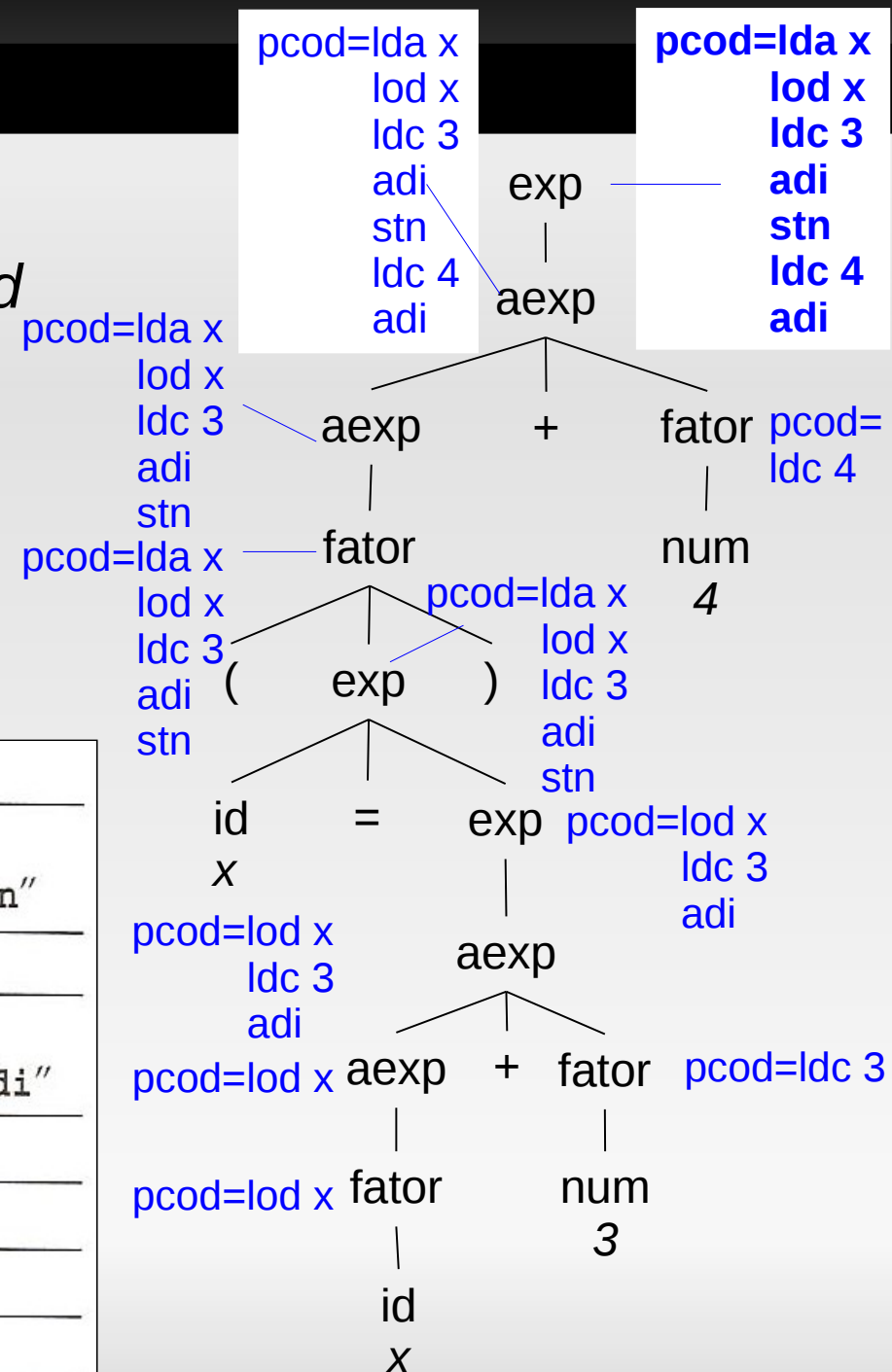
Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow id = exp_2$	$exp_1.pcod = "lda"    id.strval$ $++ exp_2.pcod ++ "stn"$
$exp \rightarrow aexp$	$exp.pcod = aexp.pcod$
$aexp_1 \rightarrow aexp_2 + fator$	$aexp_1.pcod = aexp_2.pcod$ $++ fator.pcod ++ "adi"$
$aexp \rightarrow fator$	$aexp.pcod = fator.pcod$
$fator \rightarrow ( exp )$	$fator.pcod = exp.pcod$
$fator \rightarrow num$	$fator.pcod = "ldc"    num.strval$
$fator \rightarrow id$	$fator.pcod = "lod"    id.strval$

# Geração de Código

- Cálculo do atributo sintetizado *pcod*

$exp \rightarrow id = exp \mid aexp$   
 $aexp \rightarrow aexp + fator \mid fator$   
 $fator \rightarrow (exp) \mid num \mid id$

Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow id = exp_2$	$exp_1.pcod = "lda" \parallel id.strval$ $++ exp_2.pcod ++ "stn"$
$exp \rightarrow aexp$	$exp.pcod = aexp.pcod$
$aexp_1 \rightarrow aexp_2 + fator$	$aexp_1.pcod = aexp_2.pcod$ $++ fator.pcod ++ "adi"$
$aexp \rightarrow fator$	$aexp.pcod = fator.pcod$
$fator \rightarrow ( exp )$	$fator.pcod = exp.pcod$
$fator \rightarrow num$	$fator.pcod = "ldc" \parallel num.strval$
$fator \rightarrow id$	$fator.pcod = "lod" \parallel id.strval$



# Geração de Código

- Técnicas para geração de código (intermediário e alvo)
  - Gramática de atributos
    - Exemplo – gramática que representa um pequeno subconjunto das expressões em C com cálculo de *tacod*

$exp \rightarrow id = exp \mid aexp$   
 $aexp \rightarrow aexp + fator \mid fator$   
 $fator \rightarrow (exp) \mid num \mid id$

Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow id = exp_2$	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode ++$ $id.strval \parallel "=" \parallel exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2 + fator$	$aexp_1.name = newtemp()$ $aexp_1.tacode =$ $aexp_2.tacode ++ fator.tacode$ $++ aexp_1.name \parallel "=" \parallel aexp_2.name$ $\parallel "+" \parallel fator.name$
$aexp \rightarrow fator$	$aexp.name = fator.name$ $aexp.tacode = fator.tacode$
$fator \rightarrow ( exp )$	$fator.name = exp.name$ $fator.tacode = exp.tacode$
$fator \rightarrow num$	$fator.name = num.strval$ $fator.tacode = ""$
$fator \rightarrow id$	$fator.name = id.strval$ $fator.tacode = ""$

*newtemp* = gera novo nome temporário, que é guardado no atributo "name"

"" = cadeia vazia



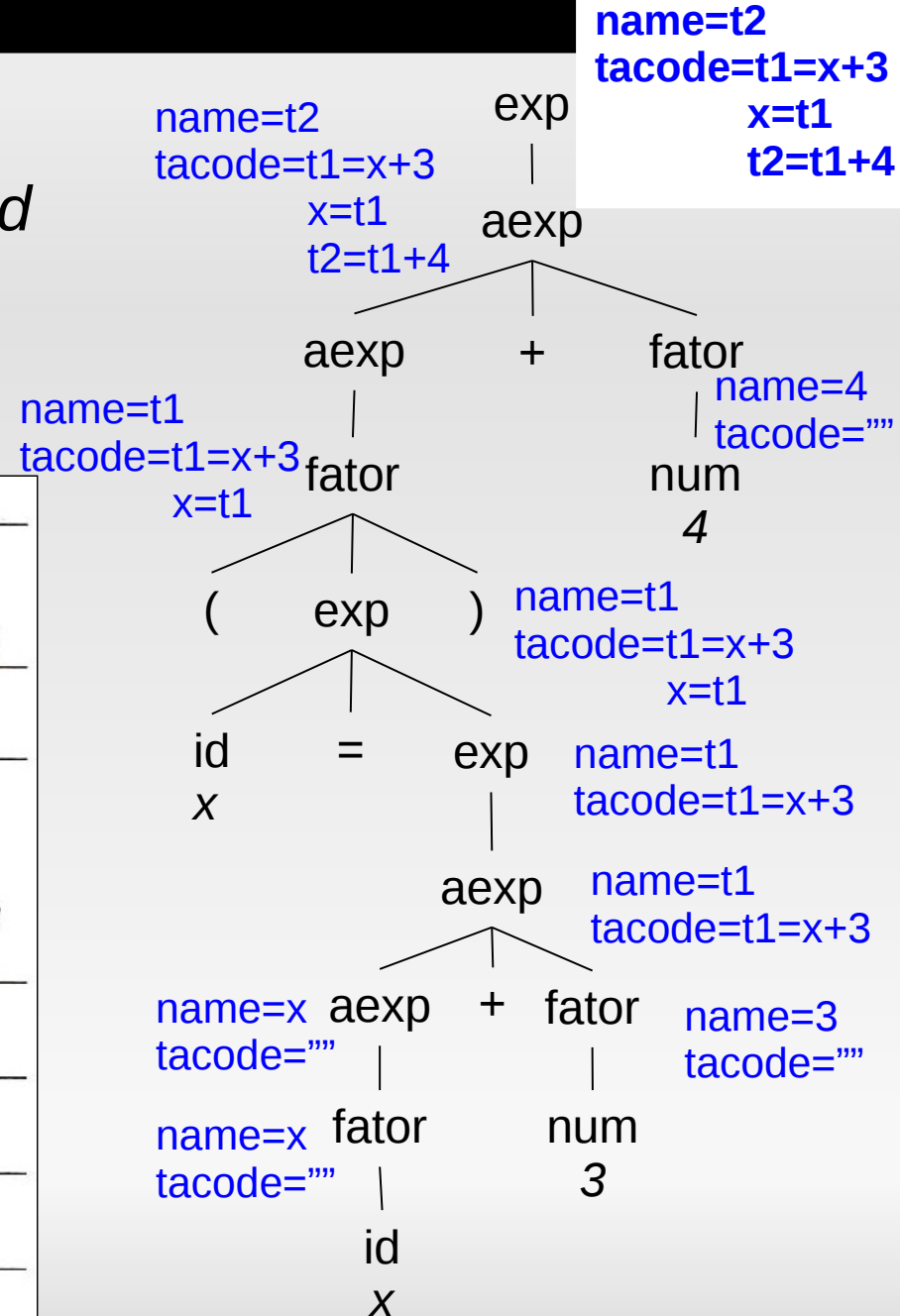
# Geração de Código

- Cálculo do atributo sintetizado *tacod*

$$\text{exp} \rightarrow \text{id} = \text{exp} \mid \text{aexp}$$
$$\text{aexp} \rightarrow \text{aexp} + \text{fator} \mid \text{fator}$$

fator  $\rightarrow$  (exp) | num | id

Regra Gramatical	Regras Semânticas
$exp_1 \rightarrow id = exp_2$	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode ++$ $id.strval    "="    exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2 + fator$	$aexp_1.name = newtemp()$ $aexp_1.tacode =$ $aexp_2.tacode ++ fator.tacode$ $++ aexp_1.name    "="    aexp_2.name$ $   "+"    fator.name$
$aexp \rightarrow fator$	$aexp.name = fator.name$ $aexp.tacode = fator.tacode$
$fator \rightarrow ( exp )$	$fator.name = exp.name$ $fator.tacode = exp.tacode$
$fator \rightarrow num$	$fator.name = num.strval$ $fator.tacode = ""$
$fator \rightarrow id$	$fator.name = id.strval$ $fator.tacode = ""$



# Geração de Código

- Técnicas para geração de código (intermediário e alvo)
  - Gramática de atributos
    - Vantagens
      - Apresenta com clareza as relações entre as sequências de código de diferentes partes da árvore
      - Permite comparar diferentes métodos de geração de código
    - Desvantagem
      - Não é prática como técnica para geração efetiva de código
        - Há desperdício de memória na concatenação de cadeias
        - Não é possível gerar e, em seguida, gravar pequenos trechos de código à medida que vão sendo gerados
        - É possível que a gramática fique bem complicada (por exemplo, com atributos herdados)



# Geração de Código

- Técnicas para geração de código (intermediário e alvo)
  - Procedimentos/funções de geração
    - Baseados na gramática de atributos e na árvore sintática
      - Busca em pós-ordem da árvore sintática modificada ou
      - Ações equivalentes durante a análise sintática se a árvore não for gerada explicitamente
  - *Ad hoc*
    - Amarrado aos procedimentos sintáticos

# Geração de Código

- Técnicas para geração de código (intermediário e alvo)
  - Procedimentos/funções de geração
    - Exemplo – gerando P-código para exemplo anterior com base na gramática de atributos e árvore sintática

```
procedure genCod(T: nó-árvore);  
begin
```

```
  if T não é nulo then
```

```
    if ('+') then
```

```
      genCode(t->leftchild);  
      genCode(t->rightchild);  
      write("adi");
```

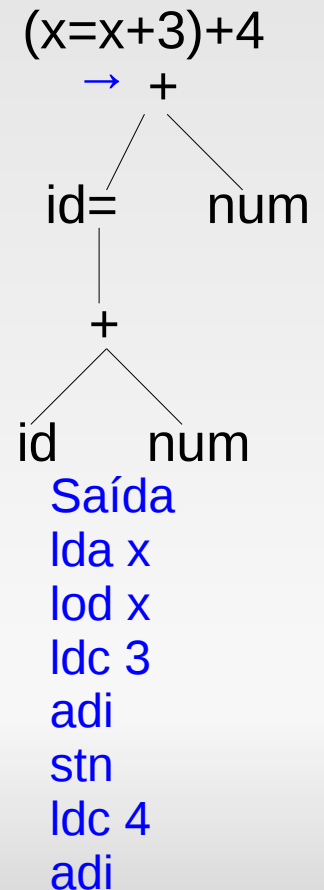
```
    else if ('id=') then
```

```
      write("lda"+id.strval);  
      genCode(t->leftchild);  
      write("stn");
```

```
    else if ('num') then write("ldc"+num.strval);
```

```
    else if ('id') then write("lod"+id.strval);
```

```
  end;
```



# Geração de Código

- Técnicas para geração de código (intermediário e alvo)
  - Procedimentos/funções de geração
    - Ad hoc
      - Geração de código amarrada aos procedimentos sintáticos

função fator(Seg): **string**;

Início

declare **cod: string**;

se (simbolo='(') então início

obtem\_simbolo(cadeia,simbolo);

**cod=exp(Seg+{''})**;

se (simbolo=')')

então obtem\_simbolo(cadeia,simbolo);

senão ERRO(Seg);

fim

senão se (simbolo='num') então início

**cod="ldc "+cadeia**;

obtem\_simbolo(cadeia,simbolo);

fim

senão se (simbolo='id') então início

**cod="lod "+cadeia**;

obtem\_simbolo(cadeia,simbolo);

fim

senão ERRO(Seg);

**retorne cod**;

fim

fator → (exp)  
| num  
| id

# Geração de Código

- Geração de código-alvo a partir de código intermediário
  - Duas técnicas padrão
    - Expansão de macros
      - Encara cada linha de código como uma macro e a substitui por uma porção de código correspondente do código-alvo
        - Pode gerar código ineficiente ou redundante
    - Simulação estática
      - Requer uma simulação direta dos efeitos do código intermediário e a geração de código-alvo correspondente a esses efeitos
        - Pode variar de muito simples a muito sofisticada

# Geração de Código

- Geração de código-alvo a partir de código intermediário
  - Expansão de macros
    - Exemplo – suponha que temos código intermediário de três endereços e queremos gerar código-alvo P-código
      - A expressão  $a=b+c$  pode ser substituída pela sequência

lda a  
lod b (ou ldc b, se b é constante)  
lod c (ou ldc c, se c é constante)  
adi  
sto

- A expressão  $a=b$  pode ser substituída pela sequência

lda a  
lod b (ou ldc b, se b é constante)  
sto

# Geração de Código

- Geração de código-alvo a partir de código intermediário
  - Expansão de macros
    - Exemplo – suponha que temos código intermediário de três endereços e queremos gerar código-alvo P-código
      - Aplicando a expansão de macro com base na substituição de expressões

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

```
lda t1
lod x
ldc 3
adi
sto
lda x
lod t1
sto
lda t2
lod t1
ldc 4
adi
sto
```

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

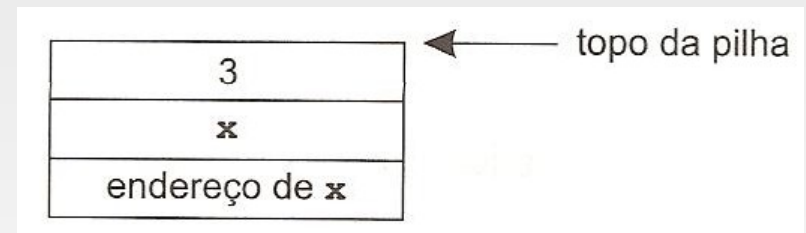
Para eliminar os temporários adicionais é necessário um esquema mais sofisticado

# Geração de Código

- Geração de código-alvo a partir de código intermediário
  - Simulação estática
    - Exemplo – suponha que temos código intermediário P-código e queremos gerar código-alvo de três endereços
      - Precisamos gerar os temporários e, por isso, é necessário que se simule a pilha e seu funcionamento

```
lda x  
lod x  
ldc 3  
adi  
stn  
ldc 4  
adi
```

Após processar as  
3 primeiras instruções



# Geração de Código

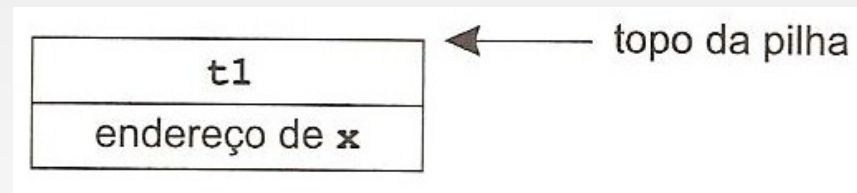
- Geração de código-alvo a partir de código intermediário
  - Simulação estática
    - Exemplo – suponha que temos código intermediário P-código e queremos gerar código-alvo de três endereços
    - Precisamos gerar os temporários e, por isso, é necessário que se simule a pilha e seu funcionamento

```
lda x
lod x
ldc 3
adi }
stn
ldc 4
adi
```

Processando adi



Código de três endereços:  
 $t1 = x + 3$





# Geração de Código

- Geração de código-alvo a partir de código intermediário
  - Simulação estática
    - Exemplo – suponha que temos código intermediário P-código e queremos gerar código-alvo de três endereços
    - Precisamos gerar os temporários e, por isso, é necessário que se simule a pilha e seu funcionamento

```
lda x
lod x
ldc 3
adi
stn }
ldc 4
adi
```

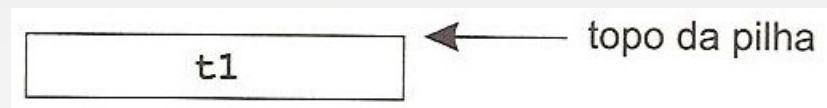
Processando stn



Código de três endereços:

$t1 = x + 3$

$x = t1$



# Geração de Código

- Geração de código-alvo a partir de código intermediário
  - Simulação estática
    - Exemplo – suponha que temos código intermediário P-código e queremos gerar código-alvo de três endereços
    - Precisamos gerar os temporários e, por isso, é necessário que se simule a pilha e seu funcionamento

```
lda x
lod x
ldc 3
adi
stn
ldc 4 }
adi
```

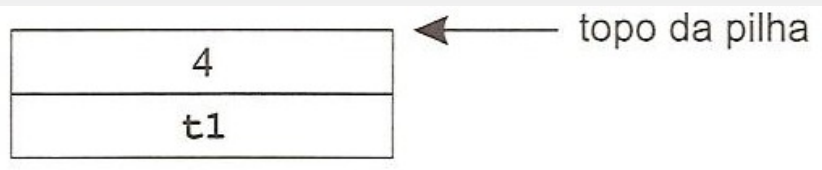
Processando ldc 4



Código de três endereços:

$t1 = x + 3$

$x = t1$



# Geração de Código

- Geração de código-alvo a partir de código intermediário
  - Simulação estática
    - Exemplo – suponha que temos código intermediário P-código e queremos gerar código-alvo de três endereços
    - Precisamos gerar os temporários e, por isso, é necessário que se simule a pilha e seu funcionamento

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi }
```

Processando adi

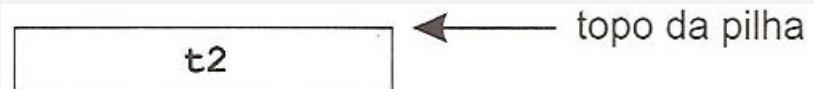


Código de três endereços:

$t1 = x + 3$

$x = t1$

$t2 = t1 + 4$



# Otimização de Código

- O que é?
  - Etapa na qual tenta-se melhorar o código de tal forma que resulte em um código equivalente porém mais compacto ou mais rápido em tempo de execução
    - Pontos a serem melhorados
      - Velocidade
      - Tamanho
      - Memória utilizada para temporários
  - Nome enganoso – pois é raro gerar código "ótimo"
    - A geração de código ótimo é um problema indecidível
  - Na prática
    - Heurísticas são usadas para tentar otimizar o código ao máximo
  - Custo-benefício
    - A otimização torna o compilador mais lento encarecendo-o, por isso é importante analisar a necessidade de código otimizado

# Otimização de Código

- Onde se aplica?
  - Código fonte da linguagem
  - Código intermediário gerado pelo compilador
  - Código em linguagem de montagem
  - Programa em forma de árvore sintática abstrata
- Como é feita a otimização?
  - Normalmente, o processo de otimização se desenvolve em duas fases:
    - Otimização de código intermediário
    - Otimização de código objeto

# Otimização de Código

- Principais fontes de otimização de código
  - Alocação de registradores
    - Boa alocação de registradores para melhorar a qualidade de código
      - Quanto maior o número de registradores e melhor seu uso, maior a velocidade do código gerado
  - Operações desnecessárias
    - Evitar a geração de código para operações redundantes ou desnecessárias como

```
#define DEBUG 0
```

```
...
```

```
if (DEBUG) { ... }
```

→ Não é preciso gerar código-alvo para o código entre chaves, que é inatingível

original

```
if debug = 1 goto L1  
goto L2  
L1: imprimir informações  
L2:
```

→ Salto desnecessário

otimizado

```
if debug ≠ 1 goto L2  
imprimir informações  
L2:
```

# Otimização de Código

- Principais fontes de otimização de código
  - Operações caras
    - Redução de força
      - Expressões caras são substituídas por mais baratas (uma potência  $x^3$  pode ser implementada como multiplicação  $x*x*x$ )
    - Empacotamento e propagação de constantes
      - Reconhecimento e troca de expressões constantes pelo valor calculado (por exemplo, troca-se  $2+5$  por  $7$ )
    - Procedimentos
      - Alinhamento de procedimentos – substitui a ativação do procedimento pelo código do corpo do procedimento (com a substituição apropriada de parâmetros por argumentos)
      - Identificação e remoção de recursão de cauda – quando a última operação de um procedimento é ativar a si mesmo
  - Uso de dialetos de máquina
    - Instruções mais baratas oferecidas por máquinas específicas

# Otimização de Código

- Principais fontes de otimização de código
  - Previsão de comportamento de programa
    - Conhecimento do comportamento do programa para otimizar saltos, laços e procedimentos ativados mais frequentemente
      - A maioria dos programas gasta 80-90% do seu tempo de execução em 10-20% de seu código



# Otimização de Código

- Níveis de otimização
  - Otimização em pequena escala (peephole)
    - Aplicada a pequenas sequências de instruções
  - Otimização local
    - Aplicada a segmentos de código de linha reta – a sequência maximal de código de linha reta é chamada “bloco básico”
      - Relativamente fácil de efetuar
  - Otimização global
    - Estende-se para além dos blocos básicos, mas é confinada a um procedimento individual
      - Exige análise de fluxo de dados
  - Otimização interprocedimento
    - Estende-se para além dos limites dos procedimentos, podendo atingir o programa todo
      - A mais complexa, exigindo diversos tipos de informações e rastreamentos do programa
- As técnicas de otimização podem ser combinadas e aplicadas recursivamente na otimização de código intermediário ou objeto

# Otimização de Código

- Otimização peephole

- Tenta melhorar o desempenho do programa alvo substituindo pequenas sequências de instruções (peepholes) por sequências mais curtas ou mais rápidas

- Eliminação de instruções redundantes

(1) MOV R0, a

(1) MOV R0, a

(2) MOV a, R0

- Simplificação algébrica

$x = y + 0$

$x = y$

$x = y * 1$

$x = y$

- Redução de força

$x^2$

$x * x$

$x + 1$  (add x, 1)

inc

# Otimização de Código

- Otimização local

- Bloco Básico

- Uma sequência de enunciados consecutivos na qual o controle entra no início e o deixa no fim, sem uma parada ou possibilidade de ramificação, exceto ao final
    - Exemplo – dado o código de três endereços para fatorial

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Inícios possíveis de um novo bloco básico:

- Primeira instrução
- Cada rótulo que é alvo de um salto
- Cada instrução após um salto

# Otimização de Código

- Otimização local

- Bloco Básico

- Uma sequência de enunciados consecutivos na qual o controle entra no início e o deixa no fim, sem uma parada ou possibilidade de ramificação, exceto ao final
    - Exemplo – dado o código de três endereços para fatorial

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

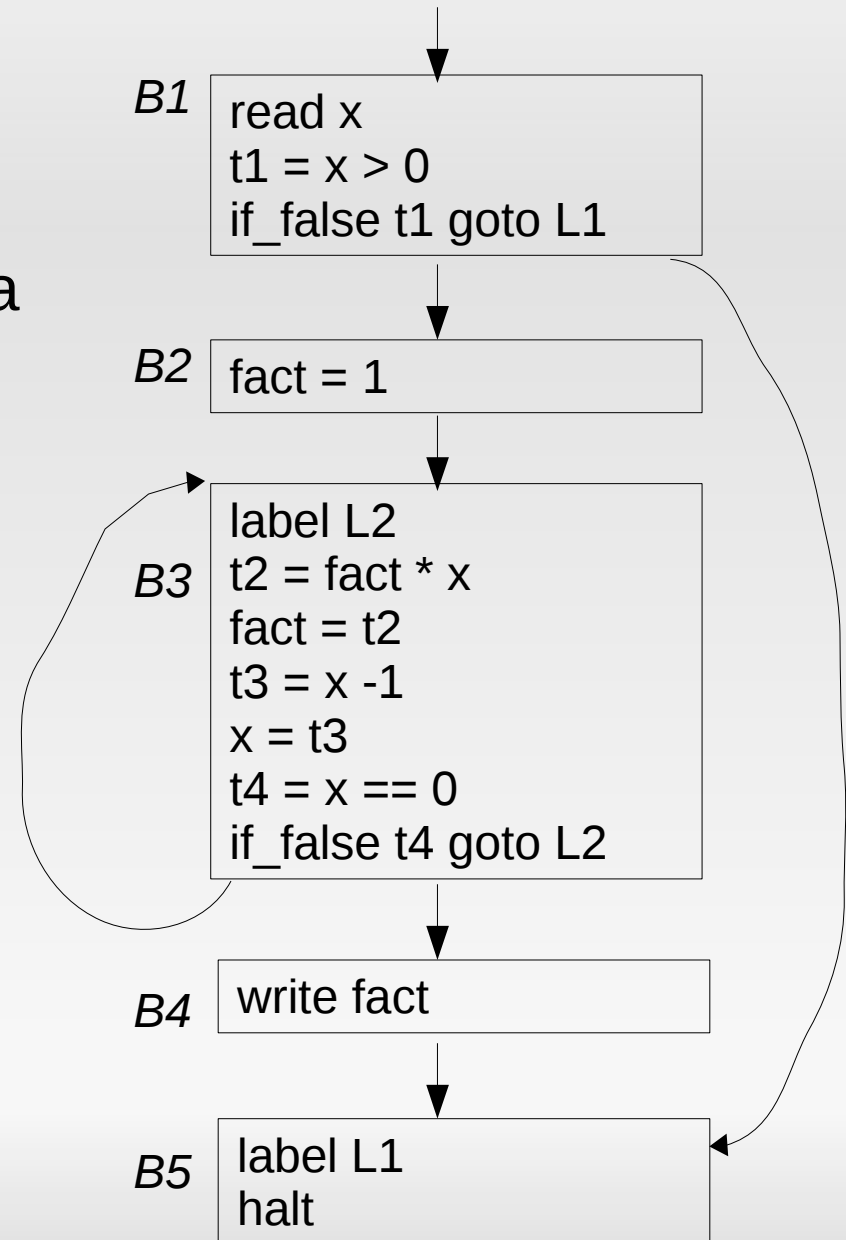
# Otimização de Código

- Otimização global
  - Grafo de fluxo de execução
    - O fluxo de execução de um programa pode ser visualizado criando-se um GDA a partir de seus blocos básicos
      - Cada vértice do grafo é um bloco básico
      - Uma aresta de um bloco B1 para um bloco B2 existe se B2 puder ser executado imediatamente após B1
    - ➔ Pode ser construído com uma única passada pelo código
    - ➔ Principal estrutura de dados requerida para a análise de fluxo de dados

# Otimização de Código

- Otimização global
  - Grafo de fluxo de execução
    - Exemplo – Grafo de fluxo para programa fatorial em código de três endereços

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```



# Otimização de Código

- Otimização local
  - É possível criar um GDA para cada bloco básico
    - Nós folha = valores usados no bloco provenientes de outro ponto
    - Nós interiores = operações sobre os valores folha e outros interiores
    - A atribuição é representada pela junção de um nome ao nó que representa o valor atribuído
    - Exemplo – B3 anterior

GDA correspondente

label L2

t2 = fact \* x

fact = t2

t3 = x - 1

x = t3

t4 = x == 0

if\_false t4 goto L2

Código otimizado equivalente

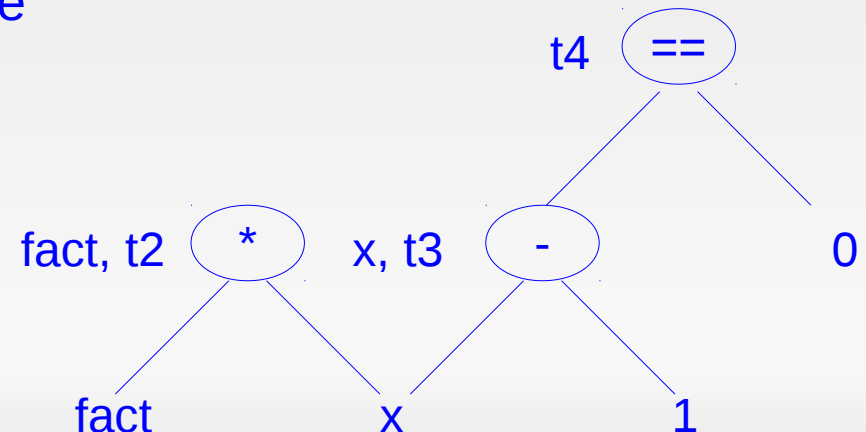
label L2

fact = fact \* x

x = x - 1

t4 = x == 0

if\_false t4 goto L2



# Otimização de Código

- Algoritmo para transformar Bloco Básico em GDA

O algoritmo supõe que cada instrução do bloco básico segue um dos três formatos:

- (i)  $x = y \text{ op } z$
- (ii)  $x = \text{op } y$
- (iii)  $x = y$

Execute os passos (1) e (2) para cada instrução do Bloco Básico

- (1) Se o nó  $y$  ainda não existe no grafo, crie uma folha para  $y$ . Tratando-se do caso (i) faça o mesmo para  $z$ .
- (2) No caso (i), verifique se existe um nó  $\text{op}$  com filhos  $y$  e  $z$  (nesta ordem). Caso exista, chame-o também de  $x$ ; senão, crie um nó  $\text{op}$  com nome  $x$  e dois arcos dirigidos do nó  $\text{op}$  para  $y$  e para  $z$ .  
No caso (ii), verifique se existe um nó  $\text{op}$  com filho  $y$ . Se não existir, crie tal nó em um arco direcionado desse nó para  $y$ . Chame de  $x$  o nó criado ou encontrado.  
No caso (iii), chame também de  $x$  o nó  $y$ .



# Otimização de Código

- Algoritmo para transformar Bloco Básico em GDA

Exemplo: Dado o Bloco Básico a seguir, gere o GDA correspondente

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$

(i)  $x = y \text{ op } z$

Execute os passos (1) e (2) para cada instrução do Bloco Básico

(1) Se o nó  $y$  ainda não existe no grafo, crie uma folha para  $y$ . Tratando-se do caso (i) faça o mesmo para  $z$ .

(2) No caso (i), verifique se existe um nó  $op$  com filhos  $y$  e  $z$  (nesta ordem). Caso exista, chame-o também de  $x$ ; senão, crie um nó  $op$  com nome  $x$  e dois arcos dirigidos do nó  $op$  para  $y$  e para  $z$ .

No caso (ii), verifique se existe um nó  $op$  com filho  $y$ . Se não existir, crie tal nó em um arco direcionado desse nó para  $y$ . Chame de  $x$  o nó criado ou encontrado.

No caso (iii), chame também de  $x$  o nó  $y$ .

# Otimização de Código

- Algoritmo para transformar Bloco Básico em GDA

Exemplo:

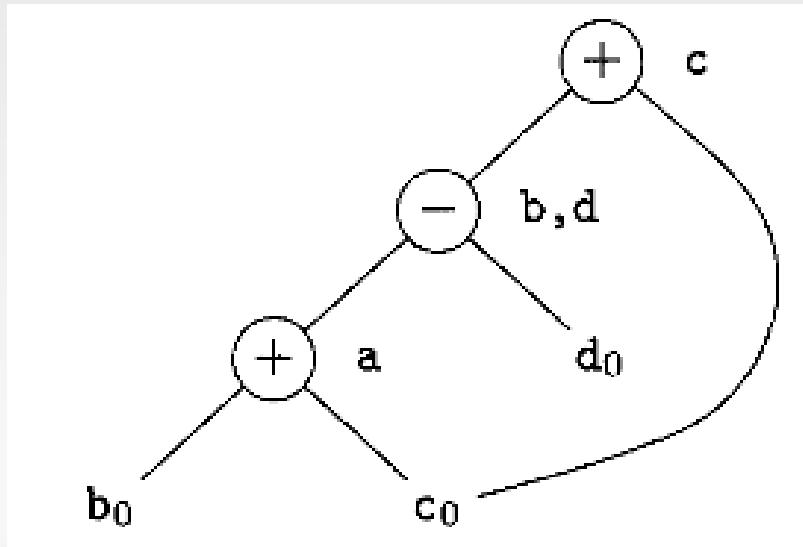
$$a = b + c$$

$$b = a - d$$

$$c = b + c$$

$$d = a - d$$

GDA correspondente:



# Otimização de Código

- Otimização local
  - Eliminação de subexpressões comuns
    - Reescrita de código para eliminação de trechos que realizam a mesma computação
  - Em relação ao GDA
    - As subexpressões comuns podem ser detectadas notando que ao adicionar um novo nó M ao GDA já existe um nó N com os mesmos filhos, na mesma ordem, e com o mesmo operador
    - Nesse caso, N calcula o mesmo valor que M e pode ser usado no seu lugar

# Otimização de Código

- Otimização local
  - Eliminação de subexpressões comuns

Exemplo:

$a = b + c$

$b = a - d$

$c = b + c$

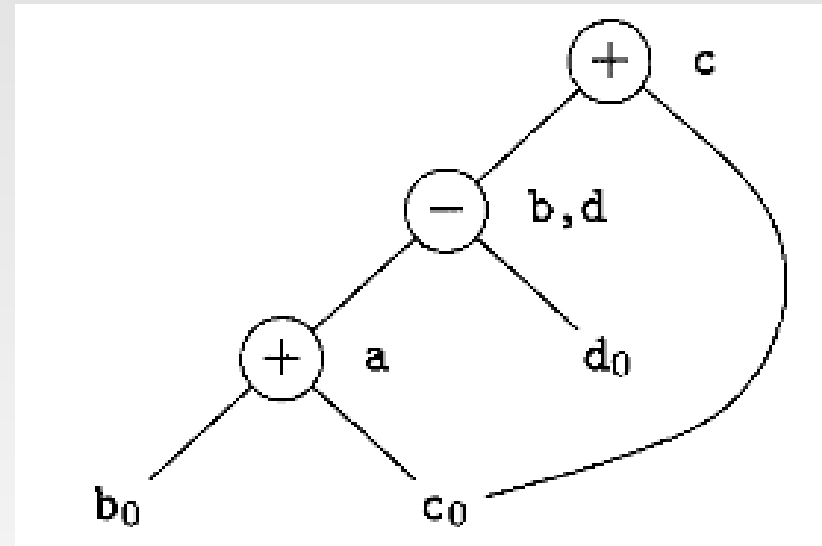
$d = a - d$

Poderia ser otimizado para:

$a = b + c$

$d = a - d$

$c = d + c$



- ➔ Contudo, se por exemplo  $b$  for usado após esse bloco básico então é necessário guardar seu valor

# Otimização de Código

- Otimização global
  - Eliminação de subexpressões comuns envolvendo vários blocos básicos
    - É necessário empregar algoritmos de análise de fluxo de execução para descobrir quais são as subexpressões comuns do programa

```
a = 4 * i;  
if (i > 10) {  
    i++;  
    b = 4 * i;  
}  
else  
    c = 4*i;
```

```
a = 4 * i;  
if (i > 10) {  
    i++;  
    b = 4 * i;  
}  
else  
    c = a;
```

# Otimização de Código

- Otimização global
  - Eliminação de código morto (inatingível)
    - Código morto (inatingível) = código que nunca será executado, independente do fluxo de execução do programa

```
int f (int n) {  
    int i = 0;  
    while ( i < n) {  
        if (g == h) {  
            break;  
            g = 1;           // morto  
        }  
        i++;  
        g--;  
    }  
    return g;  
    g++;           // morto  
}
```

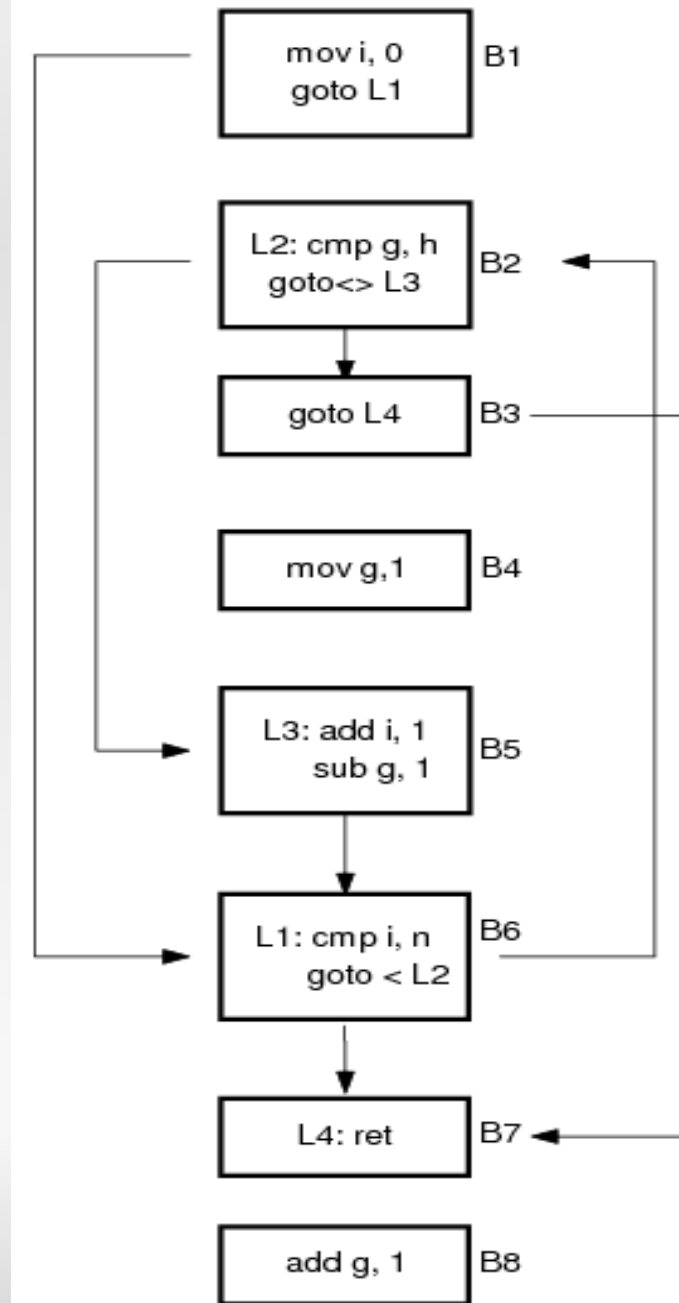
```
int f (int n) {  
    int i = 0;  
    while ( i < n) {  
        if (g == h) {  
            break;  
        }  
        i++;  
        g--;  
    }  
    return g;  
}
```

# Otimização de Código

- Otimização global
  - Eliminação de código morto
    - Código morto pode ser identificado por meio do GDA
    - Exemplo:
      - Traduzindo para assembly e criando o GDA correspondente

```
int f (int n) {  
    int i = 0;  
    while ( i < n) {  
        if (g == h) {  
            break;  
            g = 1;  
        }  
        i++;  
        g--;  
    }  
    return g;  
    g++;  
}
```

B4 e B8 são  
blocos mortos



# Otimização de Código

- Otimização global
  - Otimização de laço
    - Movimentação de código (Code Motion)
      - Expressões para as quais os valores permanecem os mesmos independente do número de vezes que o laço é executado, devem estar fora do laço

```
i = 0
while (i <= n - 2) do
begin
    write(i);
    i = i + 1;
end
```

```
i = 0
t = n-2
while (i <= t) do
begin
    write(i);
    i = i + 1;
end
```



# Otimização de Código

- Otimização com Variáveis
  - Alocação de registradores para variáveis
    - Instruções envolvendo apenas operadores em registradores são mais rápidas do que as que envolvem operadores na memória
    - Exemplo: colocar as variáveis mais usadas (empregadas em laços internos, por exemplo) em registradores

```
for (i=0; i < n; i++)  
  for (j=0; j < n; j++)  
    for (k=0; k < n; k++)  
      s[i][j][k] = 0;
```

as variáveis mais usadas são  
 $k > j > i$

# Otimização de Código

- Otimização com Variáveis
  - Reuso de registradores
    - Se duas variáveis locais a uma subrotina nunca estão vivas ao mesmo tempo, elas podem ocupar a mesma posição de memória ou registrador

```
void f() {  
    int i, j;  
    for (i=0; i < 10; i++)  
        count << i << endl;  
    for (j=10; j < 0; j--)  
        count << j << endl;  
}
```

```
void f() {  
    int i, j;  
    for (i=0; i < 10; i++)  
        count << i << endl;  
    for (i=10; i < 0; i--)  
        count << i << endl;  
}
```

→ i e j podem ser a mesma variável

# Otimização de Código

- Otimização de Procedimentos
  - Passagem de parâmetros/valor de retorno por registradores
    - O compilador pode adotar a passagem de parâmetros e o armazenamento de valores de retorno usando alguns registradores específicos
    - Essa opção evita a passagem pela pilha, que é mais lenta

# Otimização de Código

- Otimização de Procedimentos
  - Expansão em linha de procedimentos
    - Procedimentos pequenos podem ser expandidos no lugar onde são chamados evitando-se, assim, a execução de tarefas como:
      - a) passagem de parâmetros
      - b) empilhamento do endereço de retorno
      - c) salto para o procedimento
      - d) salvamento e inicialização de registrador para variáveis locais
      - e) alocação das variáveis locais
      - => execução do corpo do procedimento
      - f) destruição das variáveis locais
      - g) salto para o endereço de retorno

# Otimização de Código

- Otimização de Procedimentos
  - Recursão em cauda
    - Substituição de uma chamada recursiva ao final da execução do procedimento por um desvio incondicional para o início do procedimento

```
void P (int a)    {  
    if (a > 2)  
        P(a-1);  
    else if (a == 2) cout << "0" << endl;  
        cout << "0" << endl;  
    else  
        P(10);  
}
```

```
void P (int a) {  
L:  
    if (a > 2) {  
        a = a - 1;  
        goto L;  
    }  
    else if (a == 2) cout << "0" << endl;  
    else {  
        a = 10;  
        goto L;  
    }  
}
```

Esse exemplo só pode ser otimizado porque não há instrução fora do if-then-else

# Processo de Tradução

