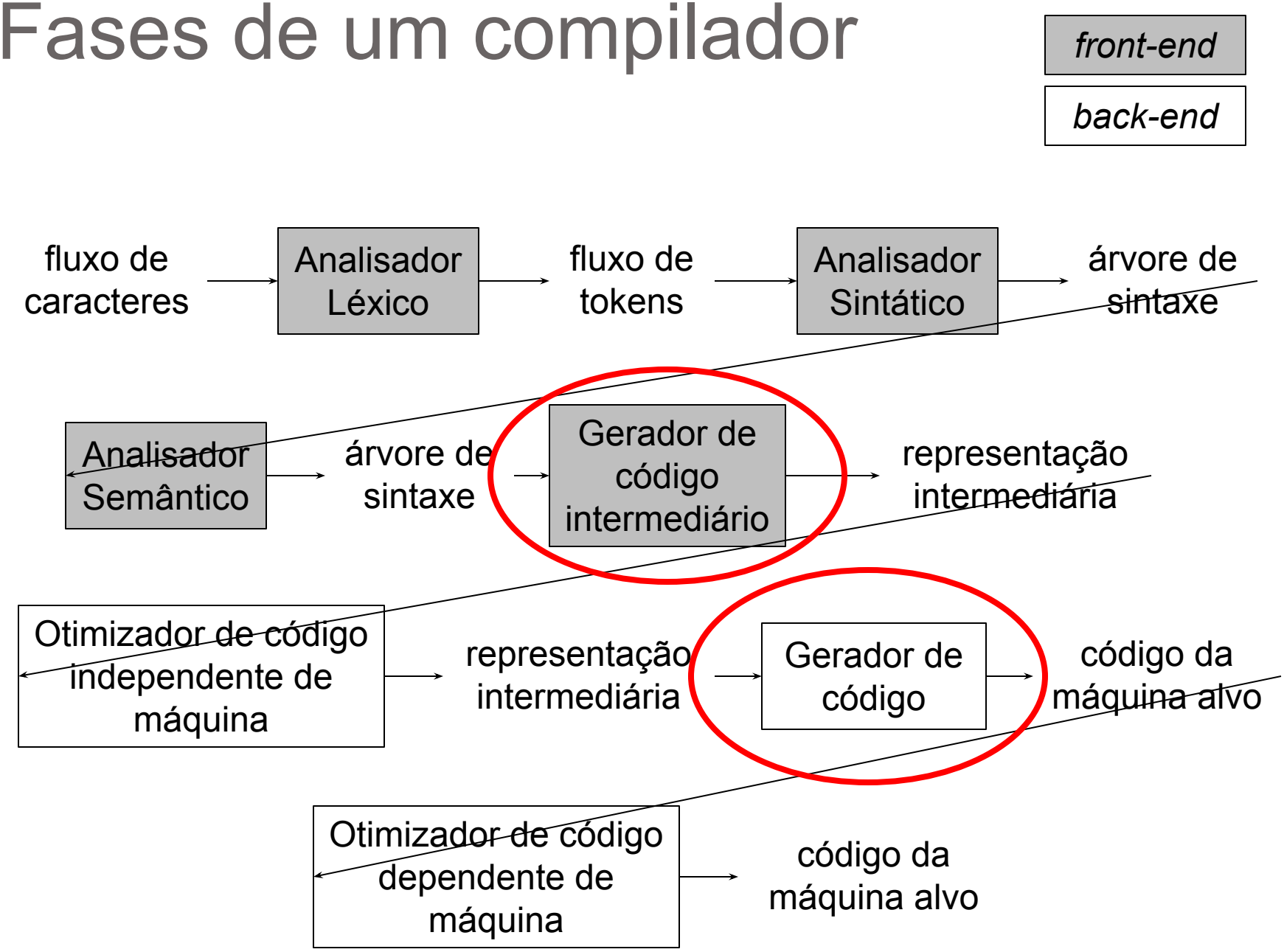
Construção de compiladores

Profs. Mário César San Felice (e Helena Caseli,
Murilo Naldi, Daniel Lucrédio)
Departamento de Computação - UFSCar
1º semestre / 2018
Tópico 8 - Geração de Código e Otimização

Fases de um compilador



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 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 intermediário

- 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
  - 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 da máquina alvo
    - Adição (ou não) de dados da Tabela de Símbolos
  - Linearização da árvore sintática

Diferentes níveis de representação intermediária

- Ao longo do processo de compilação
  - São geradas (explicitamente ou implicitamente) diferentes representações intermediárias

Ex: árvores de análise sintática
- mostram a estrutura hierárquica natural
- boa para tarefas como a checagem estática de tipos

Ex: código 3-endereços
- boa para tarefas dependentes de máquina:
- geração de código
- alocação de registradores
- seleção de instruções
- otimizações

Diferentes níveis de representação intermediária

- Exemplo:
  - Laço

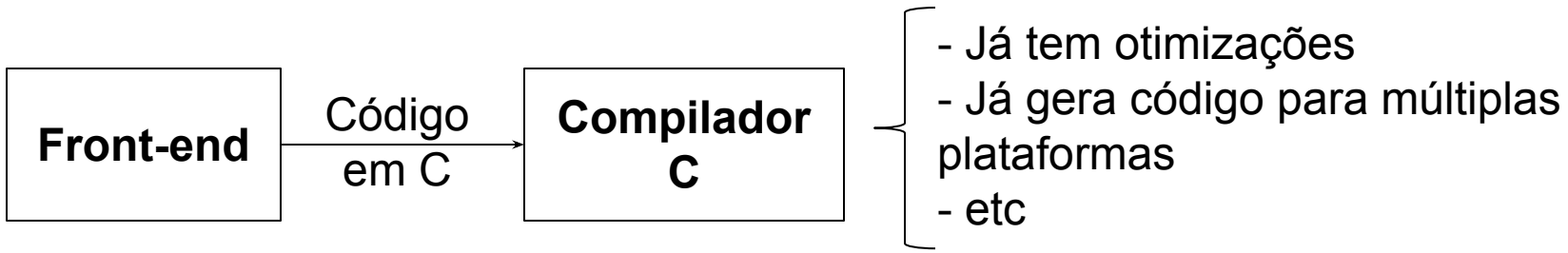
    - A árvore armazena os elementos do laço (condição, incremento, etc)

    - O código 3-endereços contém rótulos, instruções de salto e outras, parecidas com código de máquina

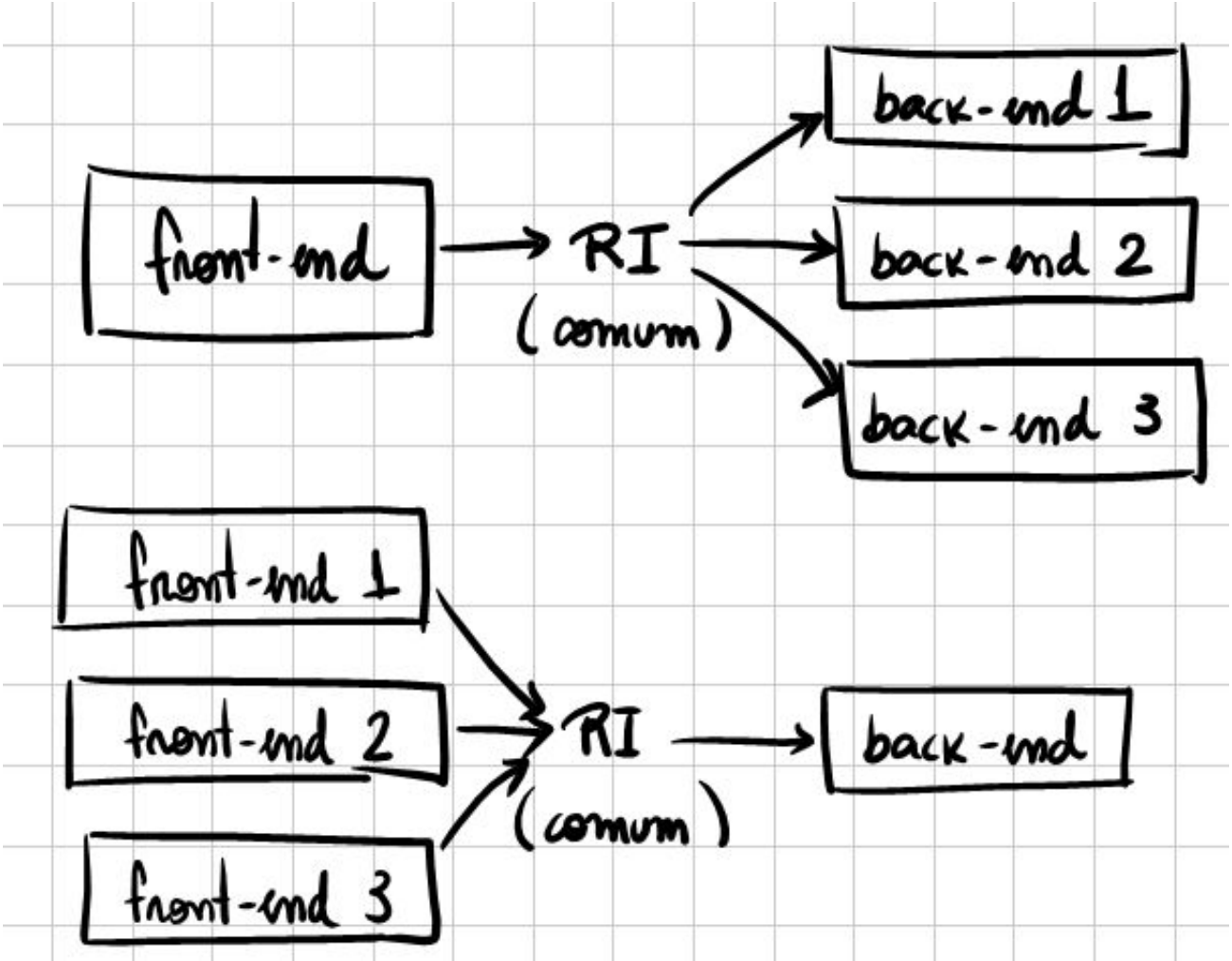
Código intermediário

- Pode ser código mesmo
  - Ou estruturas de dados

- Exemplo: C pode ser usada como R.I.
  - O compilador C é o back-end
    - Reutilização de back-end
    - Evita a necessidade de se construir um novo
  - O compilador C++ original era assim
    - Mas depois evoluiu, permitindo um processo mais otimizado



Código intermediário



Código intermediário

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

Código de 3-endereços

- Instrução básica
  - Avaliação de expressões aritméticas
    - **x = y op z**
  - op é um operador aritmético (+ ou -, por exemplo)

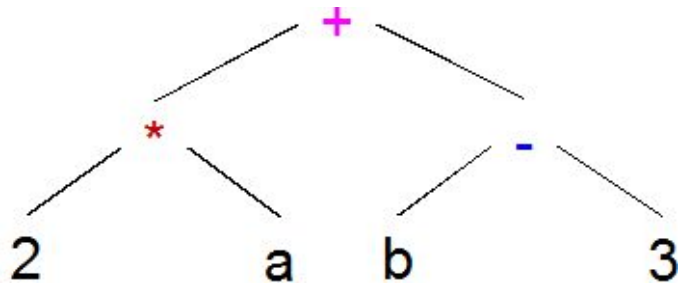
- O nome advém dessa forma de instrução na qual ocorre a manipulação de **até** 3 endereços na memória: x, y e z

- Nem sempre as instruções seguem esse formato
  - Ex: **a = - b**

Código de 3-endereços

- Exemplo

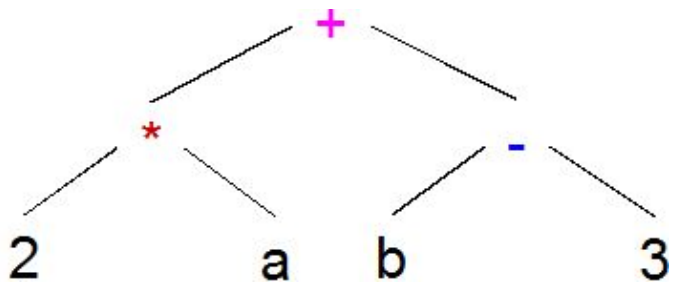
2*a+(b-3) → t1 = 2 * a
t2 = b - 3
t3 = t1 + t2



  - Código de três endereços obtido da árvore sintática
    - Por meio da linearização da esquerda para a direita
    - Percurso em pós-ordem

- Outra possibilidade

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

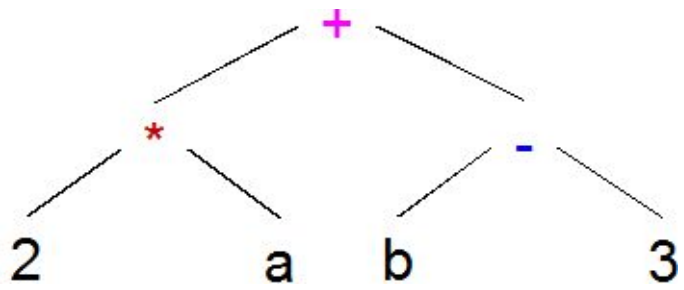


Código de 3-endereços

- Exemplo

2*a+(b-3) ⟶

t1 = 2 * a
t2 = b - 3
t3 = t1 + t2

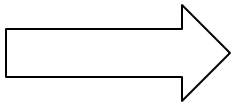


  - Temporários
    - Os nomes (t1, t2 e t3) devem ser diferentes dos nomes existentes no código-fonte
    - Correspondem aos nós interiores da árvore sintática e representam seus valores computados
    - O temporário final (t3) representa o valor da raiz
    - Podem ser mantidos na memória ou em registradores

Código de 3-endereços

- Ex: 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
```

Código de 3-endereços

- Normalmente não é implementado em forma de texto

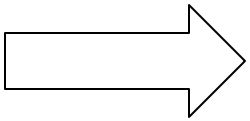
- Implementação com estruturas de dados
  - Cada instrução = uma estrutura de registros com vários campos (uma quádrupla)
    - Um para a operação
    - Três para os endereços
  - Para instruções com menos de 3 endereços, há campos vazios
  - Ponteiros para os nomes na Tabela de Símbolos podem ser usados

- Sequência de instruções = um vetor ou lista ligada

Código de 3-endereços

- Exemplo: quádruplas

```
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,_,_,_)
```

Código de 3-endereços

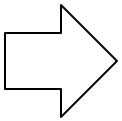
- Quádruplas X Triplas
  - O quarto elemento das quádruplas (quando presente) é sempre o endereço de uma variável temporária
  - Portanto, é possível eliminá-lo
    - E deixar a implementação mais eficiente

- Conceito de triplas
  - Cada instrução tem um índice
  - Esse índice pode ser referenciado em outras instruções

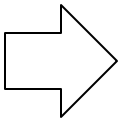
Código de 3-endereços

- Exemplo: triplas

```
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,_,_)
```

Código de 3-endereços

- Quádruplas X Triplas
  - Vantagens do uso de Triplas
    - Economia em cada entrada já que agora 1 endereço não é mais necessário
    - Diminuição no número de instruções pois não há mais a necessidade de rótulos

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

P-código

- 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 compiladores de código nativo
  - A maioria para linguagens derivadas de Pascal

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

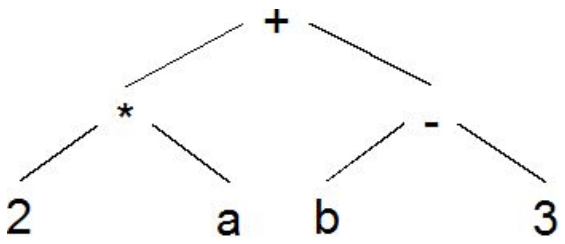
P-código

- Iremos considerar 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

P-código

- Exemplo

2*a+(b-3)




```
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
```

P-código

- Exemplo

x:=y+1

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

P-código

- Exemplo

x:=y+1

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

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
```

Código de 3-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

- Veremos agora alguns exemplos de como traduzir o programa-fonte
  - Já analisado sintaticamente e semanticamente

- Para código intermediário
  - Pronto para otimização / geração de código-alvo

- Veremos apenas algumas situações
  - Para outras, consulte as referências da disciplina
  - A ideia geral é a mesma
    - Mas os detalhes para cada situação são muitos

Geração de código

- Veremos:
  - Cálculo de endereços
  - Referências de matrizes
  - Declarações de controle
  - Expressões lógicas

- Não veremos
  - Estrutura de registros
  - Referências de ponteiros
  - Geração de rótulos
  - Ajuste retroativo (backpatching)
  - Chamadas de procedimentos e funções

Cálculo de endereços

- Para a geração de código final, não é possível usar nomes de variáveis
  - É necessário substituir por endereços de memória
    - Endereços absolutos de memória
    - Registradores
  - Os endereços podem ser armazenados na tabela de símbolos

Cálculo de endereços

- Mesmo na geração do código intermediário
  - É necessário calcular os endereços

- Por exemplo, referências para
  - Matrizes
  - Ponteiros
  - Etc...

Cálculo de endereços

- Veremos agora como usar
  - Código de 3-endereços e P-código
  - Para calcular endereços

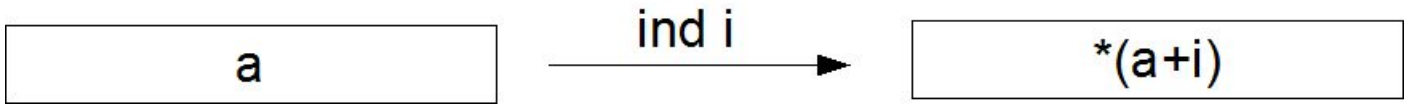
- Código de 3-endereços : Notação em C

```
t1 = &x+10
*t1 = 2
```

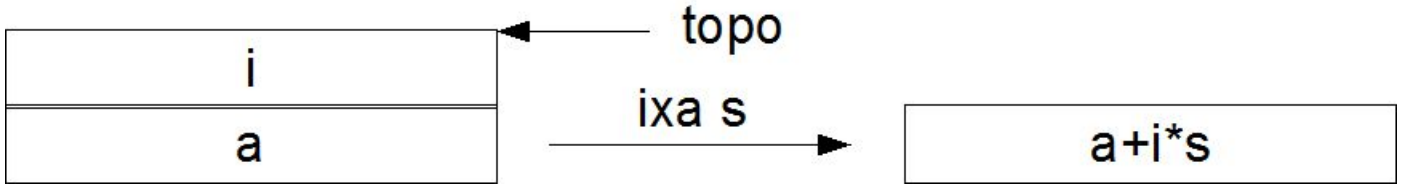
- Armazena o valor da constante 2 na posição de memória referente ao endereço da variável x + 10 bytes

Cálculo de endereços

- P-código : 2 novas instruções
  - **ind** (carga indireta) – substitui o endereço na pilha pelo **conteúdo** no local resultante da aplicação do deslocamento



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



Cálculo de endereços

- Ex: código 3-endereços

```
t1 = &x+10
*t1 = 2
```

- P-código equivalente

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

&x+10*1

Referências de matrizes

- Ex:

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

- supondo i, j inteiros e a uma matriz de inteiros para a qual a[indice] é

endereço_base(a) + (indice – limite_inferior(a)) * tamanho_elemento(a)

- Por exemplo, em C o endereço a[i+1] é
  - a + (i+1-0) * sizeof(int)

Referências de matrizes

- Ex: `a[i+1] = a[j*2] + 3;`

- Código de 3-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
```

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

Referências de matrizes

- Ex: `a[i+1] = a[j*2] + 3;`

- 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
```

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

Declarações de controle

- Ex:

```
if ( E ) S1 else S2
```

- Código de 3-endereços

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

Obs:
if_false = testa se t1 é falso
goto = salto incondicional

Declarações de controle

- Ex:

```
if ( E ) S1 else S2
```

- P-código

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

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

Declarações de controle

- Ex:

```
while ( E ) S
```

- Código de 3-endereços

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

Obs:
if_false = testa se t1 é falso
goto = salto incondicional

Declarações de controle

- Ex:

```
while ( E ) S
```

- P-código

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

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

Outras considerações

- Alguns rótulos exigem duas passadas
  - Sempre que houver declarações de controle
  - Os saltos para um rótulo podem ser gerados antes da definição do próprio rótulo
    - Ou seja, não se sabe o endereço real do rótulo para o qual se está saltando
  - Uma solução é gerar nomes lógicos primeiro
    - Depois, em uma segunda passada, trocar os nomes por endereços reais

Outras considerações

- Existe uma técnica que evita as duas passadas
  - Ajuste retroativo (backpatching)
  - Uso de um repositório temporário que armazena os rótulos que ainda serão gerados
    - Que é atualizado quando necessário

Técnicas para geração de código

- Procedimentos/funções de geração
  - Baseados na árvore sintática
  - Busca em pós-ordem

ou

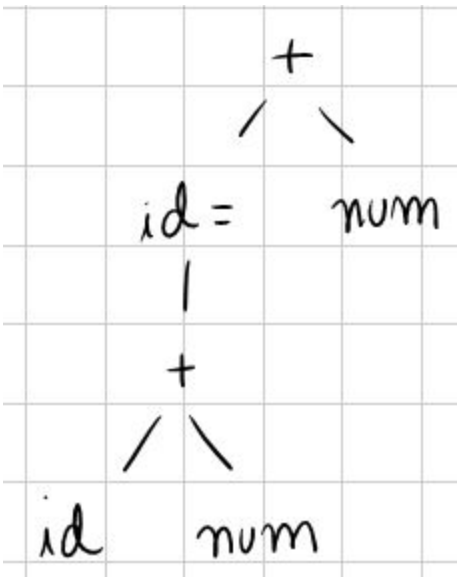
  - Ações equivalentes durante a análise sintática se a árvore não for gerada explicitamente

- Ad hoc
  - Amarrado aos procedimentos sintáticos

Procedimentos/função de geração

- Exemplo – gerando P-código com base na árvore sintática

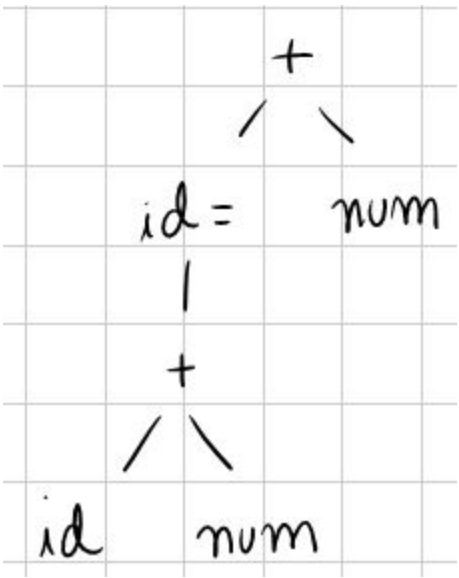
```
procedure genCode(T: nó-árvore);
begin
    if T não é nulo then
            if ('+') then
                genCode(t->leftchild);
                genCode(t->rightchild);
                write("adi");
            else if ('id=') then
                write("lda "+id.strval);
                genCode(t->leftchild);
                write("stn");
            else if ('num') then write("ldc
  "+num.strval);
            else if ('id') then write("lod "+id.strval);
end;
```



Procedimentos/função de geração

- Exercício – gere P-código para a expressão (x=x+3)+4

```
procedure genCode(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;
```



Procedimentos/função de geração

- Código gerado:

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

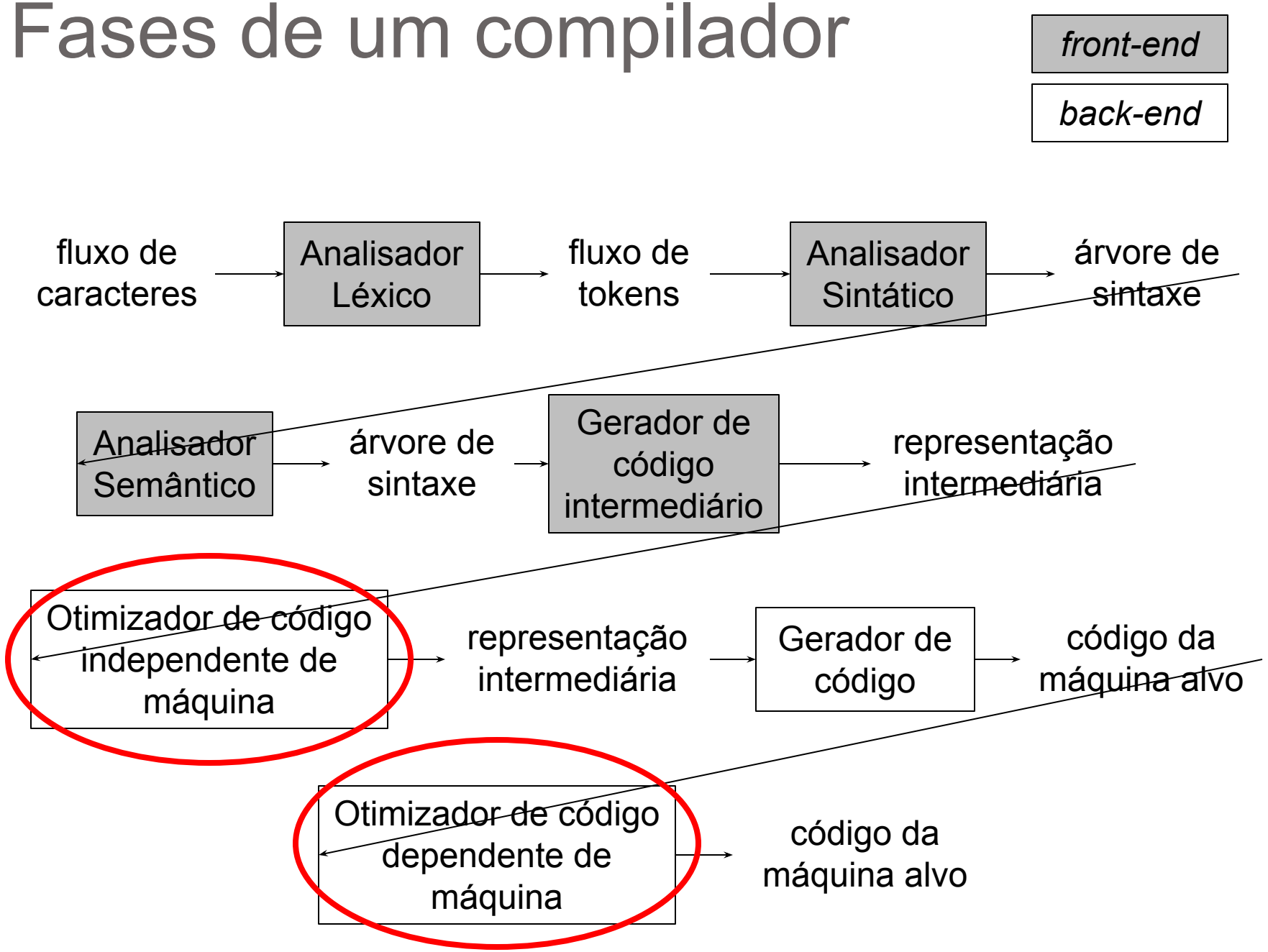
Ad hoc

- Geração de código amarrada aos procedimentos sintáticos

```
função fator(Seg): string;
inicio
   declare cod: string;
      se (simbolo='(') então inicio
         obtem_simbolo(cadeia,simbolo);
         cod='('+exp(Seg)+')';
         se (simbolo=')')
            então obtem_simbolo(cadeia,simbolo);
            senão ERRO(Seg);
         fim
      senão se (simbolo='num') então inicio
         cod="ldc "+cadeia;
         obtem_simbolo(cadeia,simbolo);
         fim
      senão se (simbolo='id') então inicio
         cod="lod "+cadeia;
         obtem_simbolo(cadeia,simbolo);
         fim
      senão ERRO(Seg);
   retorne cod;
fim
```

Gramática:
fator → (exp)
| num
| id

Fases de um compilador



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

- Pontos a serem melhorados
  - Velocidade
  - Tamanho
  - Memória utilizada para temporários

Otimização de código

- Nome enganoso
  - Pois é raro gerar código ”ótimo”
  - Encontrar o código ótimo é um problema indecidível
- Na prática
  - Heurísticas são usadas para melhorar o código
- Custo-benefício
  - A otimização torna o compilador mais lento
  - Por isso é importante analisar se código otimizado é necessário

Otimização de código

- Onde pode ser aplicado?
  - 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

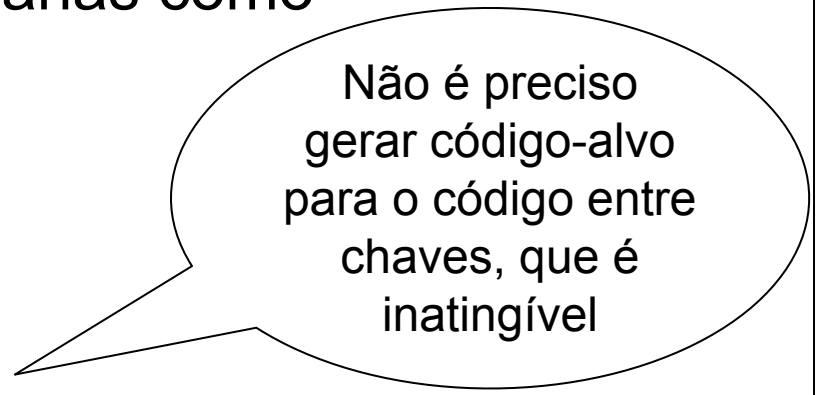
Principais fontes de otimização

- 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

Principais fontes de otimização

- Operações desnecessárias
  - Evitar a geração de código para operações redundantes ou desnecessárias como
    - Código inatingível

```
#define DEBUG 0
...
if (DEBUG) { ... }
```



Principais fontes de otimização

- Operações desnecessárias
  - Evitar a geração de código para operações redundantes ou desnecessárias como
    - Saltos desnecessários

original

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

otimizado

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

Principais fontes de otimização

- 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)

Principais fontes de otimização

- 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 chamada recursiva de um procedimento é a última operação realizada

Principais fontes de otimização

- Uso de dialetos de máquina
  - Instruções mais baratas oferecidas por máquinas específicas

- 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

Níveis de otimização de código

- 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

Níveis de otimização de código

- 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 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
```

Otimização peephole

- 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 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
  - 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 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
  - 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
```

Primeira instrução inicia um bloco básico

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
  - 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
```

Primeira instrução inicia um bloco básico

Cada rótulo alvo de um salto inicia um bloco básico

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
  - 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
```

Primeira instrução inicia um bloco básico

Cada instrução após um salto inicia um bloco básico

Cada rótulo alvo de um salto inicia um bloco básico

Cada instrução após um salto inicia um bloco básico

Otimização local

- Bloco Básico
  - Dentro dele algumas propriedades podem ser assumidas
    - Por exemplo, uma variável carregada em um registrador irá permanecer lá até o fim do bloco básico
      - A não ser que explicitamente removida

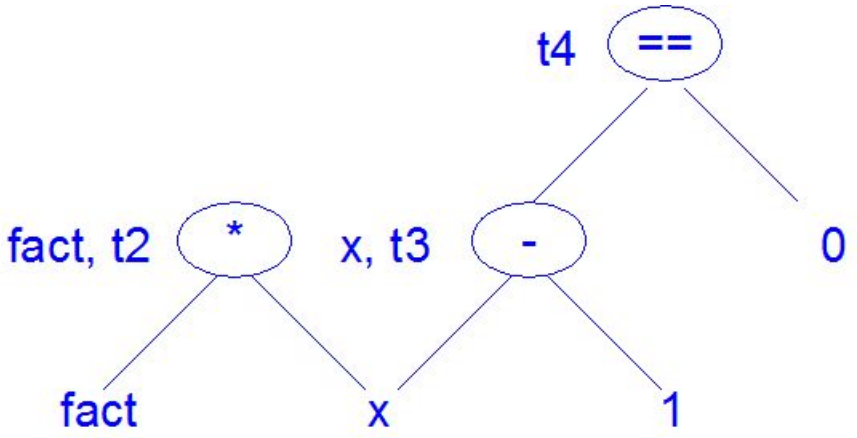
```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x -1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Otimização local

- É possível criar um GAD para cada bloco básico
  - Nós origem = valores provenientes de outro ponto
  - Demais nós = operações sobre outros valores
  - A atribuição é representada pela junção de um nome ao nó que representa o valor atribuído
  - Exemplo – B3 anterior

```
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
```

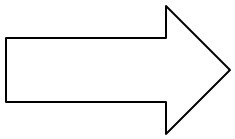
GAD correspondente



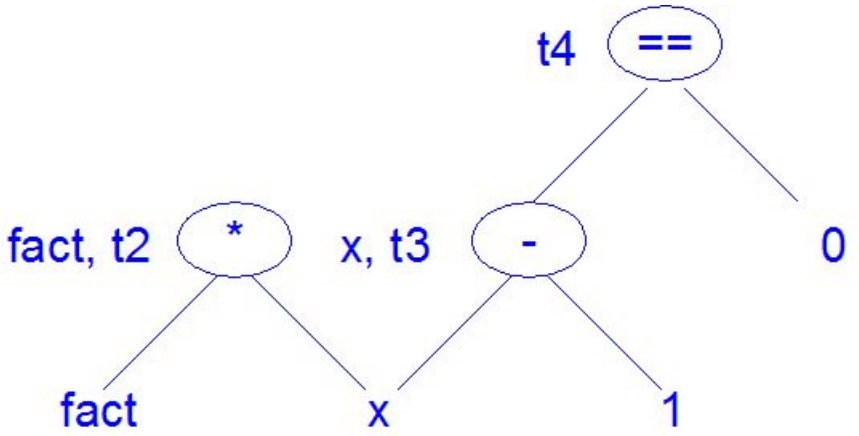
Otimização local

- Exemplo – B3 anterior sendo otimizado conforme o GAD local

```
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
```



```
label L2
fact = fact * x
x = x - 1
t4 = x == 0
if_false t4 goto L2
```



Otimização local

- Algoritmo para transformar Bloco Básico em GAD
  - O algoritmo supõe que cada instrução do bloco básico segue um dos três formatos:
    - (i) x = y op z
    - (ii) x = 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 um nó origem para y. Tratando-se do caso (i) faça o mesmo para z.

Otimização local

- (2) Analise o tipo da instrução

  - No caso (i) [x = y op z], 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 dos nós y e z para op.

  - No caso (ii) [x = op y], verifique se existe um nó op com filho y. Se não existir, crie tal nó e um arco direcionado de y para esse nó. Chame de x o nó destino.

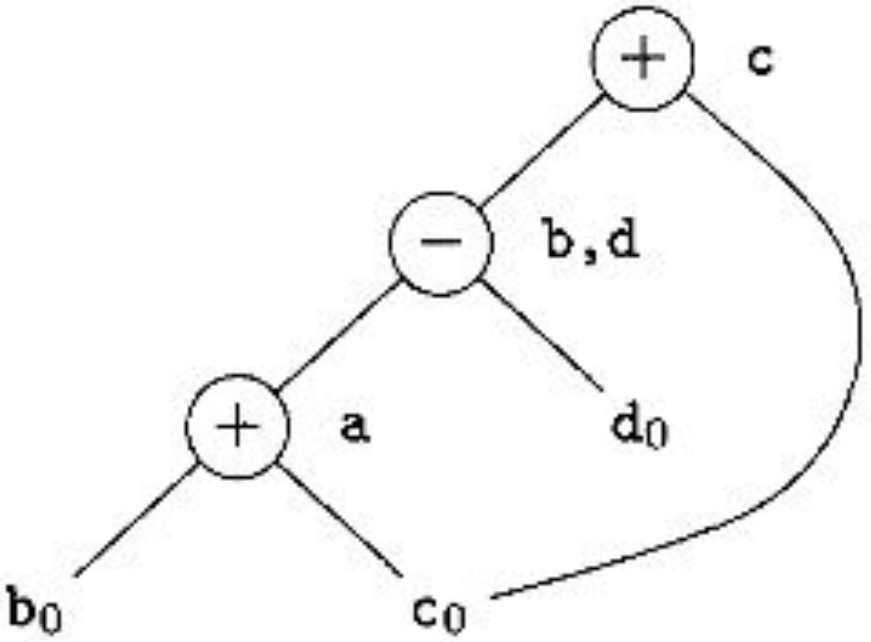
  - No caso (iii) [x = y], chame também de x o nó y.

Otimização local

- Algoritmo para transformar Bloco Básico em GAD
  - Exemplo:

```
a = b + c
b = a - d
c = b + c
d = a - d
```

  - GAD correspondente:



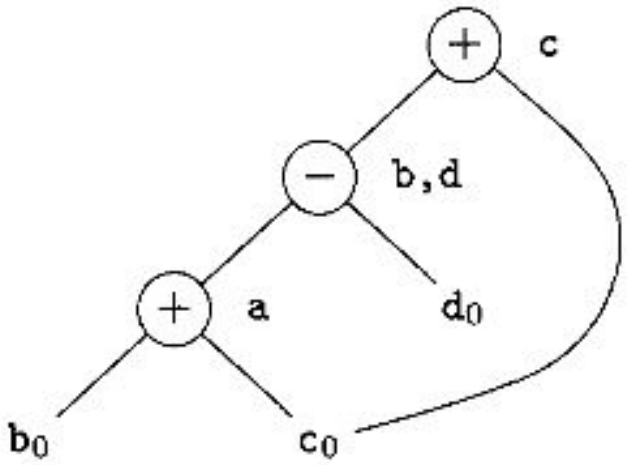
Otimização local

- Eliminação de subexpressões comuns
  - Reescrita de código para eliminação de trechos que realizam a mesma computação
  - Em relação ao GAD
    - As subexpressões comuns podem ser detectadas notando que ao adicionar um novo nó M ao GAD 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 local

- 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
```

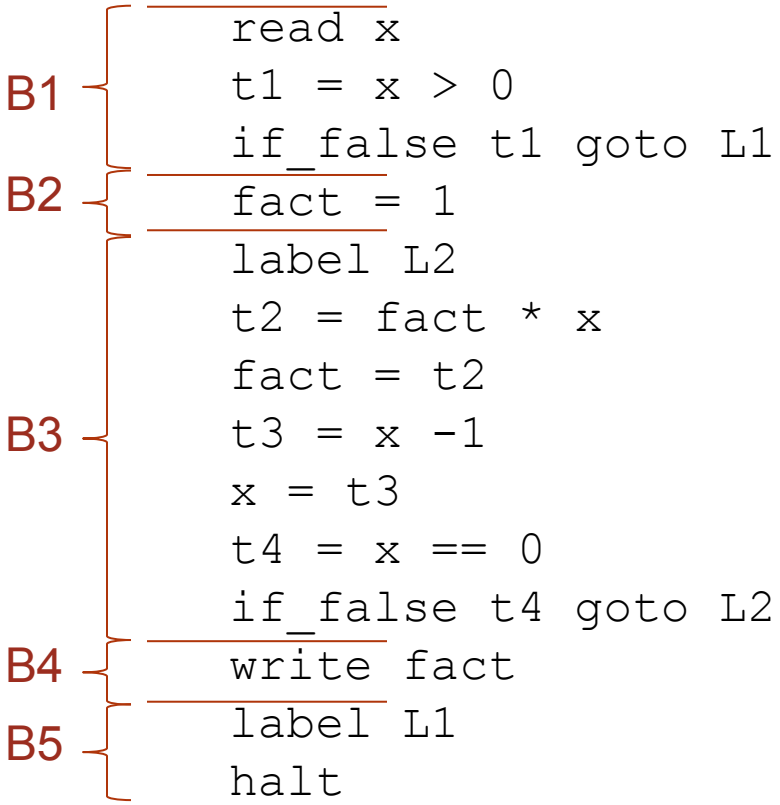
- Contudo, se b for usado após esse bloco básico então é necessário guardar seu valor

Otimização global

- O fluxo de execução de um programa pode ser visualizado criando-se um GAD 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 global

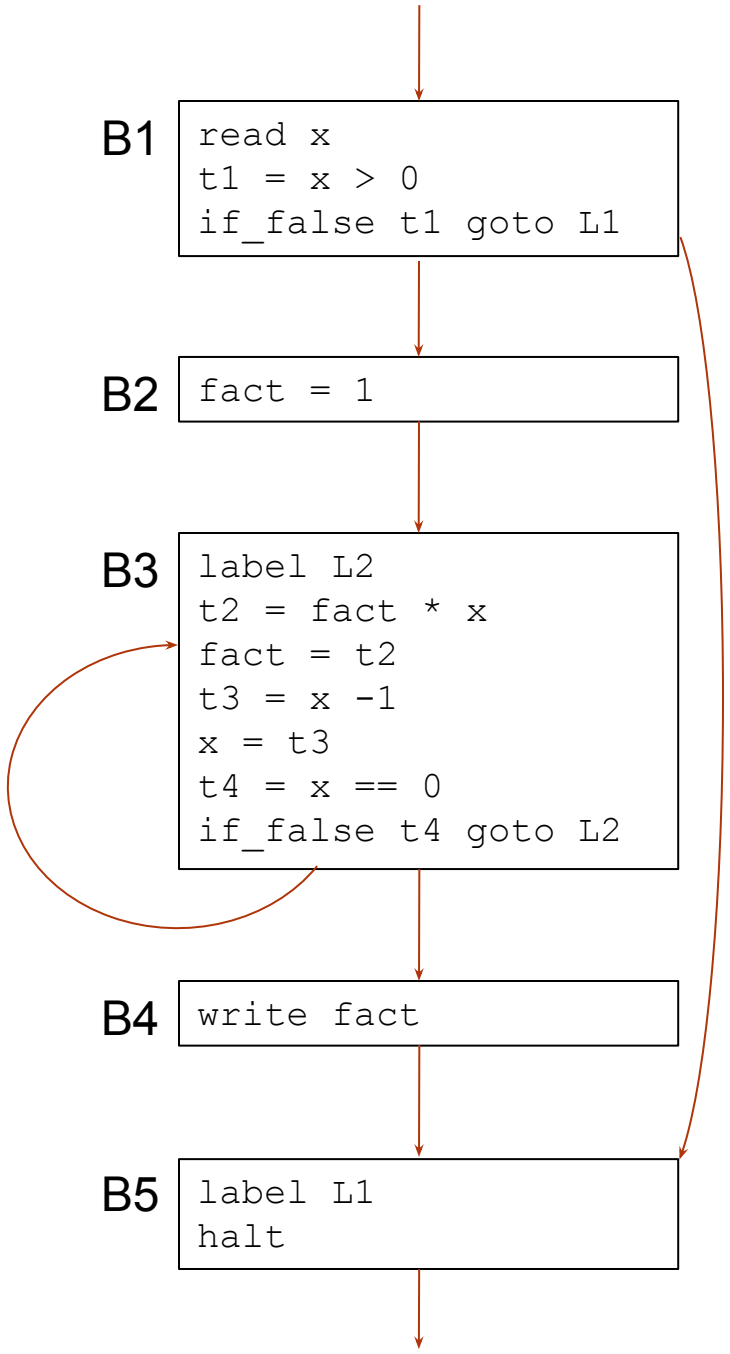
- Grafo de fluxo de execução
- Exemplo: programa fatorial em código 3-endereços



Otimização global

- Grafo de fluxo de execução
- Exemplo: programa fatorial em código 3-endereços

```
B1  read x
    t1 = x > 0
    if_false t1 goto L1
B2  fact = 1
    label L2
    t2 = fact * x
    fact = t2
B3  t3 = x -1
    x = t3
    t4 = x == 0
    if_false t4 goto L2
B4  write fact
B5  label L1
    halt
```



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;
```

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 global

- Eliminação de 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
}
```

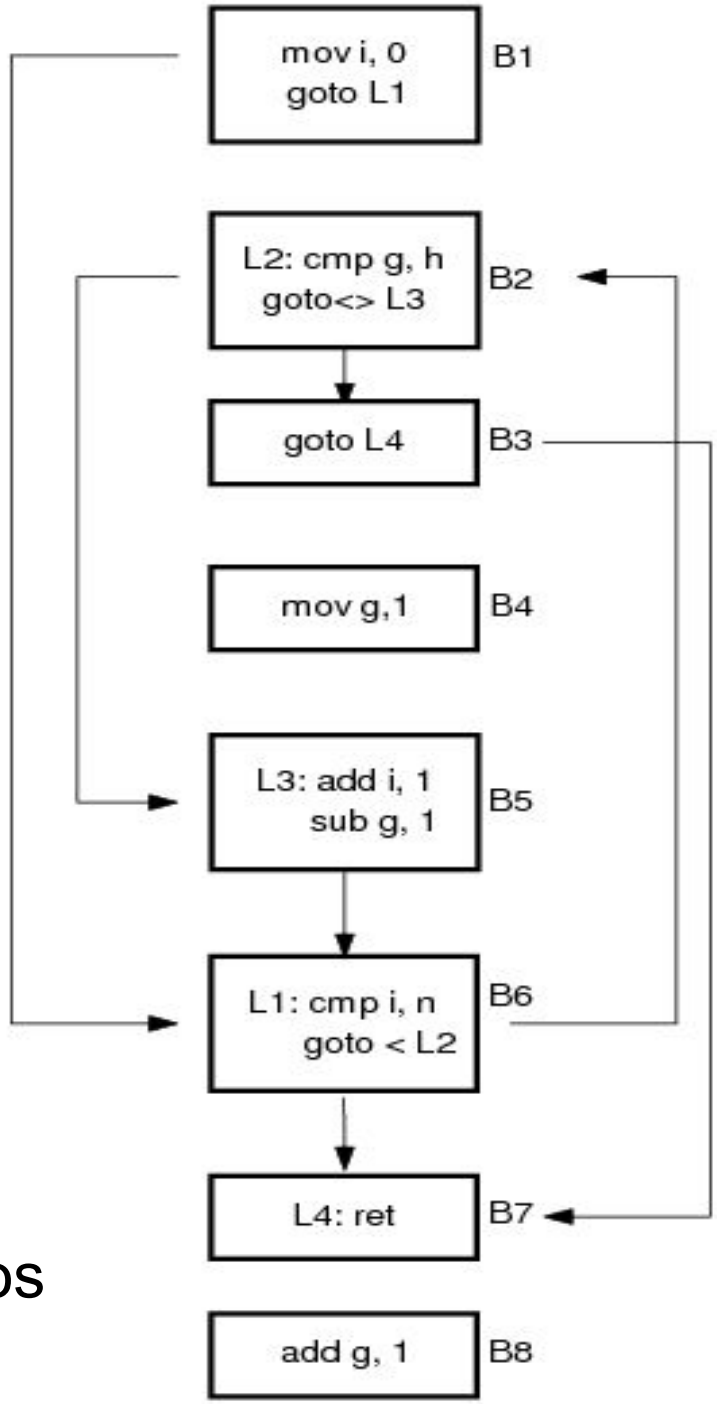
```
int f (int n) {
    int i = 0;
    while ( i < n) {
        if (g == h) {
            break;

        }
            i++;
            g--;
    }
    return g;

}
```

Otimização global

- Eliminação de código morto
- Código morto pode ser identificado por meio do GAD
- Ex.: Traduzindo para assembly e criando o GAD 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 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
```

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 global

- 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;
```

Otimização global

- 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++)          // as variáveis
    for (j=0; j < n; j++)      // mais usadas são
        for (k=0; k < n; k++)  // k > j > i
            s[i][j][k] = 0;
```

Otimização global

- 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++)
      cout << i << endl;
   for (j=10; j < 0; j--)
      cout << j << endl;
}
```

Otimização global

- 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++)
      cout << i << endl;
   for (j=10; j < 0; j--)
      cout << j << endl;
}
```

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

i e j podem ser a mesma variável

Otimização interprocedimentos

- 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 interprocedimentos

- 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 interprocedimentos

- 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;
   else
      P(10);
}
```

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

Otimização interprocedimentos

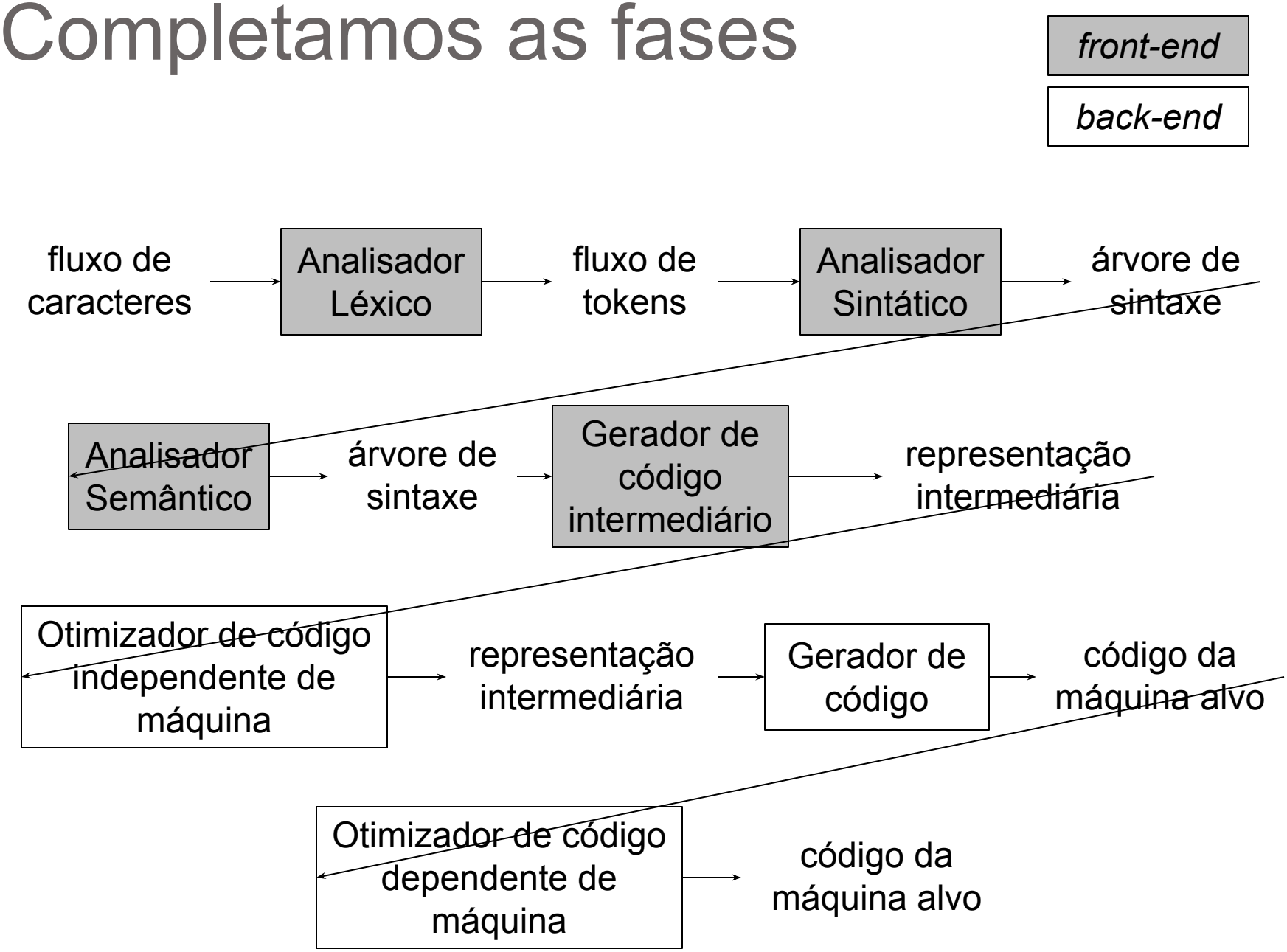
- Recursão em cauda
  - Substituição de uma chamada recursiva ao final da execução do procedimento por um desvio incondicional para o início do procedimento

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

Completamos as fases



Demonstração

- Agora veremos um processo completo
  - Geração de código

- Usaremos um ambiente de execução simples
  - Baseado em P-código
  - Totalmente estático
  - Sem procedimentos

- Durante a demonstração, tente imaginar o que seria necessário para
  - Procedimentos
  - Recursividade

Chegando ao fim do curso

E agora? O que estudar?

- Construção de compiladores 2
  - Muitos dos aspectos vistos aqui na teoria serão praticados

- Na vida profissional
  - Hoje em dia não é necessário construir compiladores
  - A maioria só vai usar um compilador
  - Mas muitas soluções que são implementadas "na mão" poderiam usar linguagens

E agora? O que estudar?

- Na vida profissional
  - Mas muitas soluções que são implementadas "na mão" poderiam usar linguagens
    - Seriam muito mais elegantes!!
    - Sugestão: se você algum dia se deparar com uma situação onde acha que "cabe" uma linguagem
      - Não tenha medo!
      - Existem ferramentas que facilitam sua vida
      - É mais simples do que parece

E agora? O que estudar?

- Na vida profissional
  - Caso se torne um profissional que trabalha com linguagens
  - Os livros utilizados (Dragão e Louden) contém muitos detalhes interessantes não vistos
    - Destrinchá-los e conhecê-los a fundo é essencial para o projetista de linguagens
  - O livro oficial do ANTLR também é bastante instrutivo, além de prático

E agora? O que estudar?

- Na vida acadêmica
  - Muitas teses de mestrado/doutorado precisam de um compilador

- Pode ser que você precise revisitar o assunto

O que lembrar?

- Fases de um compilador:
  - O que são e para que servem
- Análise léxica:
  - Expressões regulares
- Análise sintática:
  - Gramáticas livres de contexto
  - As diferenças entre LL e LR

O que lembrar?

- Análise sintática:
  - As transformações mais comuns
    - Eliminação de ambiguidade
    - Fatoração à esquerda
    - Eliminação de recursão à esquerda

- Análise semântica:
  - Esquemas de Tradução Dirigida por Sintaxe (TDS)

O que lembrar?

- Geração e otimização de código:
  - Guarde as ideias principais
    - Talvez você não as utilize em um compilador
    - Mas pode precisar em outros projetos onde é necessário melhorar desempenho/consumo

- Prática
  - ANTLR é bastante útil e merece ser acompanhado

Fim