

# Conteúdo Programático e Cronograma

---

## 1º Semestre

~~Organização e estrutura de compiladores~~

~~Análise Léxica~~

~~Análise Sintática~~

~~Ferramentas de geração automática de compiladores~~

## 2º Semestre

~~Análise Semântica~~

Geração de código

Análise de fluxo de dados

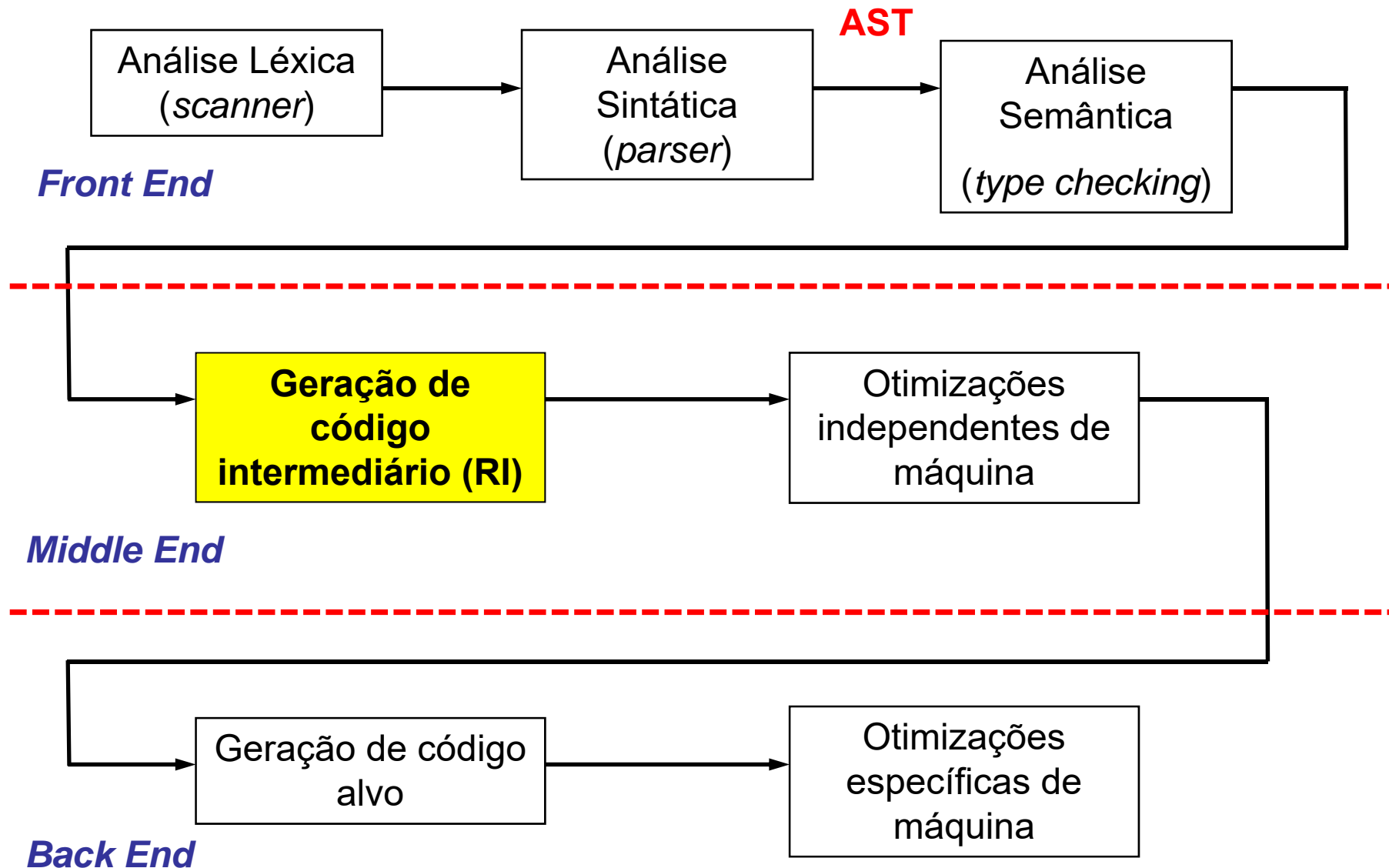
Otimização de código

Alocação de registradores

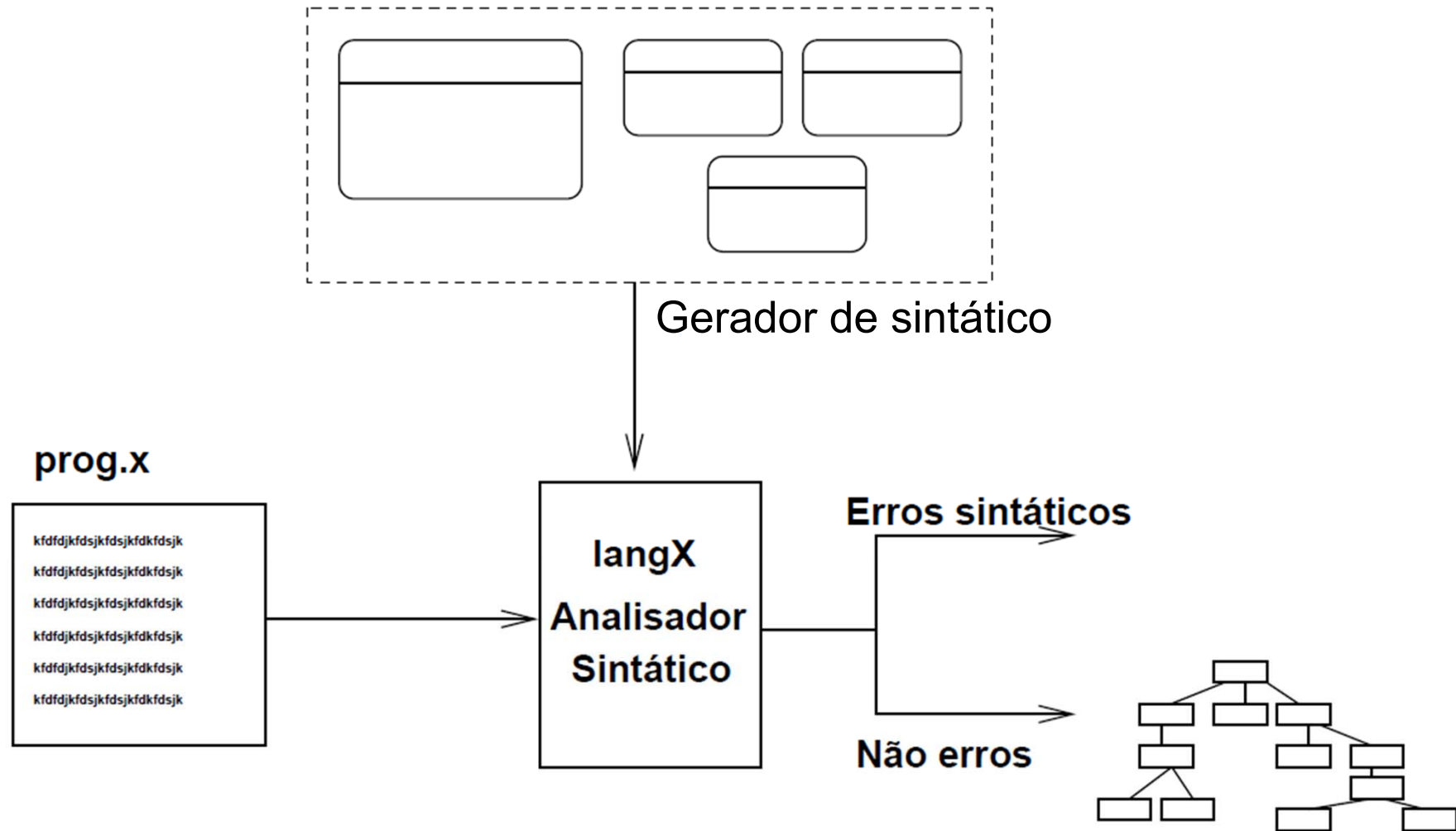
---

# **Representação Intermediária**

# Fluxo do Compilador

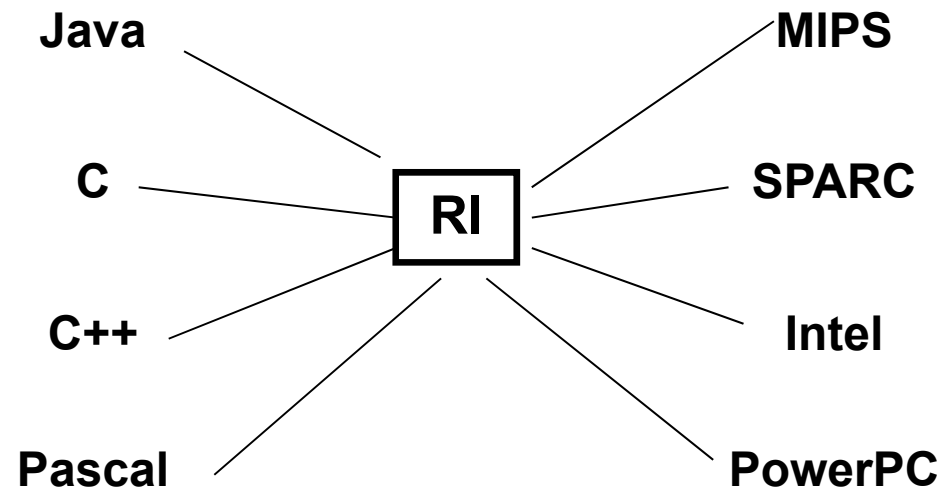


# Representação Intermediária (RI)



# Representação Intermediária (RI)

---



Queremos compiladores para **N** linguagens, direcionados para **M** máquinas diferentes.

RI nos possibilita elaborar **N** front-ends e **M** back-ends, ao invés de **N.M** compiladores.

## Escolha de uma RI

---

- Reuso: adequação à linguagem e à arquitetura alvo, custos.
- Projeto: nível, estrutura, expressividade
- “O projeto de uma representação intermediária é uma arte e não uma ciência.” - Steven S. Muchnick
- Pode-se adotar mais de uma RI

# Tipos de RI

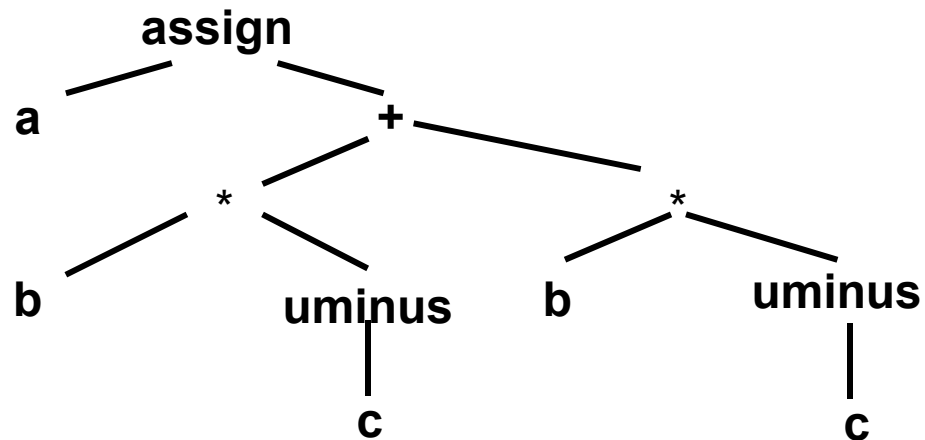
---

- Representação gráfica:
  - Árvores Sintáticas ou DAGs
  - Manipulação pode ser feita através de gramáticas
  - pode ser tanto de alto-nível como nível médio
- Representação Linear
  - RI's se assemelham a um pseudo-código para alguma máquina abstrata
  - Java: Bytecode (interpretado ou traduzido)
  - Código de três endereços

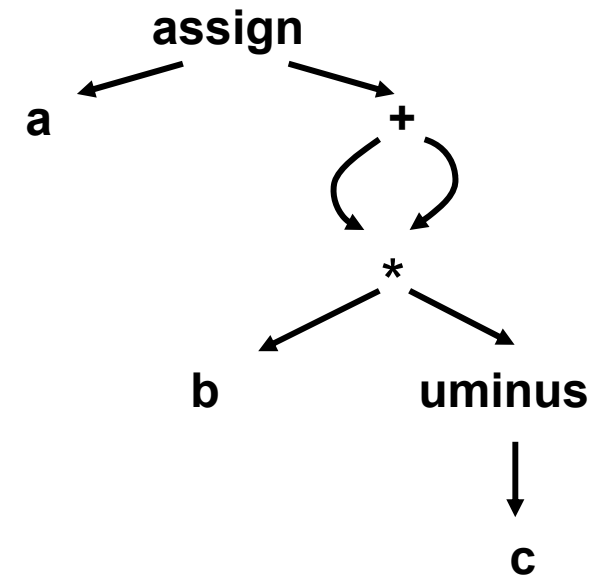
# Representação em Árvores vs DAG

---

**$a := b * -c + b * -c$**



**Árvore**



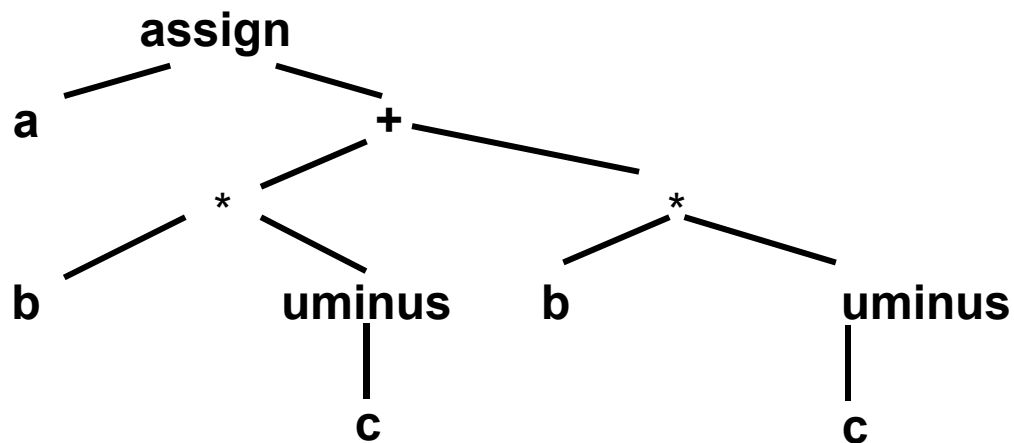
**DAG**



## Código de três endereços

---

- Forma geral:  **$x := y \text{ op } z$**
- Representação linearizada de uma árvore sintática, ou DAG



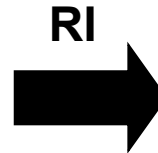
**$a := b * -c + b * -c$**

```
t1 := - c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

## Exemplo: Produto Interno

---

```
{  
  ...  
  prod = 0;  
  i = 1;  
  while (i <= 20) {  
    prod = prod + a[i] * b[i];  
    i = i + 1;  
  }  
  ...  
}
```



```
(1) prod := 0  
(2) i := 1  
(3) t1 := 4 * i  
(4) t2 := a [ t1]  
(5) t3 := 4 * i  
(6) t4 := b [t3]  
(7) t5 := t2 * t4  
(8) t6 := prod + t5  
(9) prod := t6  
(10) t7 := i + 1  
(11) i := t7  
(12) if i <= 20 goto (3)
```

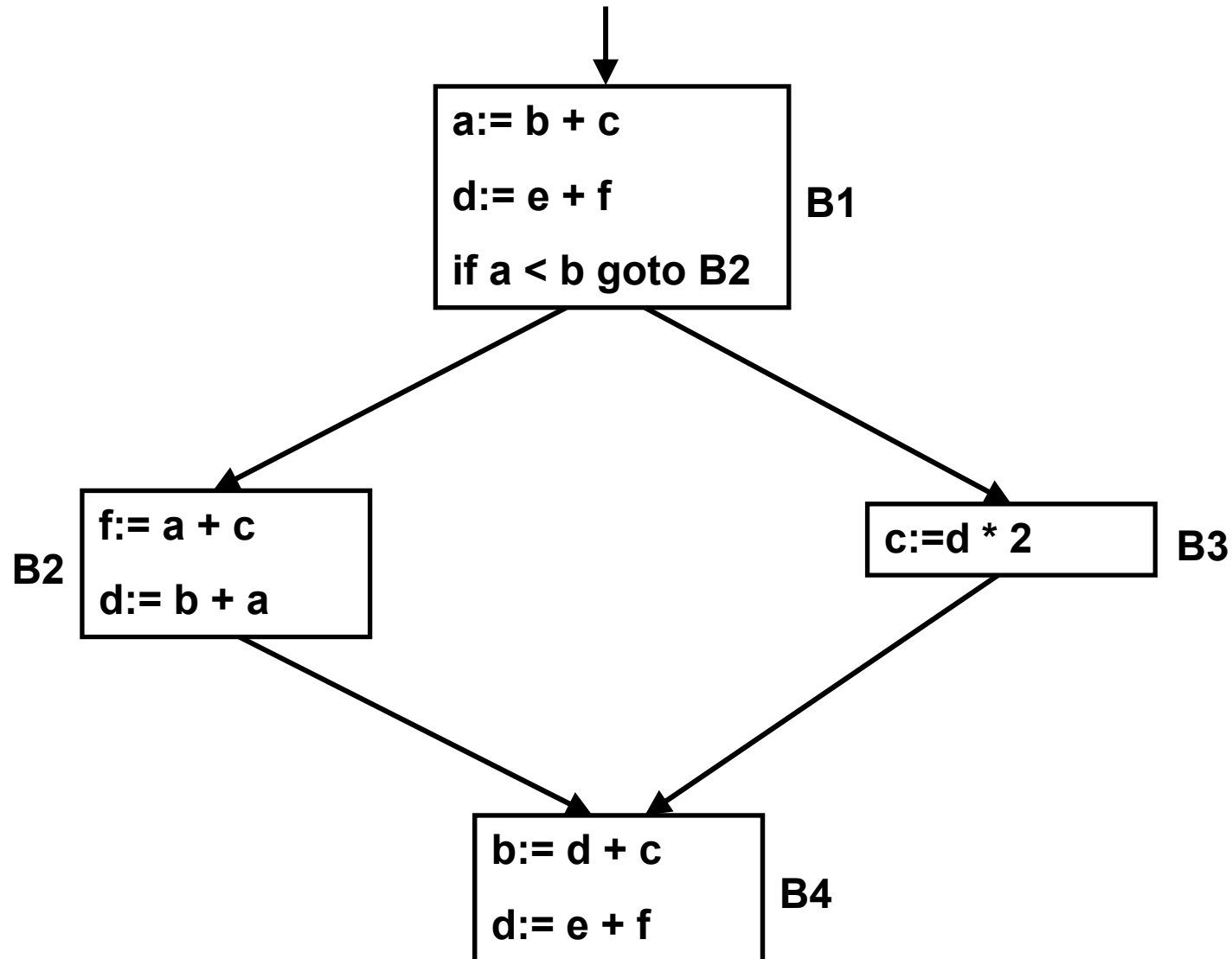
## Representação Híbrida

---

- Combina-se elementos tanto das RI's gráficas (estrutura) como das lineares.
  - Usar RI linear para blocos de código seqüencial e uma representação gráfica (CFG: *control flow graph*) para representar o fluxo de controle entre esses blocos

## Exemplo: CFG com código de 3 endereços

---



## Caso Real - GCC

---

- Várias linguagens: pascal, fortran, C, C++
- Várias arquiteturas alvo: MIPS, SPARC, Intel, Motorola, PowerPC, etc
- Utiliza mais de uma representação intermediária
  - árvore sintática: construída por ações na gramática
  - RTL: tradução de trechos da árvore

## Caso Real - GCC

---

`d := (a+b)*c`

```
(insn 8 6 10 (set (reg:SI 2)
                  (mem:SI (symbol_ref:SI ("a")))))
(insn 10 8 12 (set (reg:SI 3)
                  (mem:SI (symbol_ref:SI ("b")))))
(insn 12 10 14 (set (reg:SI 2)
                  (plus:SI (reg:SI 2) (reg:SI 3))))
(insn 14 12 15 (set (reg:SI 3)
                  (mem:SI (symbol_ref:SI ("c")))))
(insn 15 14 17 (set (reg:SI 2)
                  (mult:SI (reg:SI 2) (reg:SI 3))))
(insn 17 15 19 (set (mem:SI (symbol_ref:SI ("d")))
                  (reg:SI 2)))
```

## Tradução para RI

---

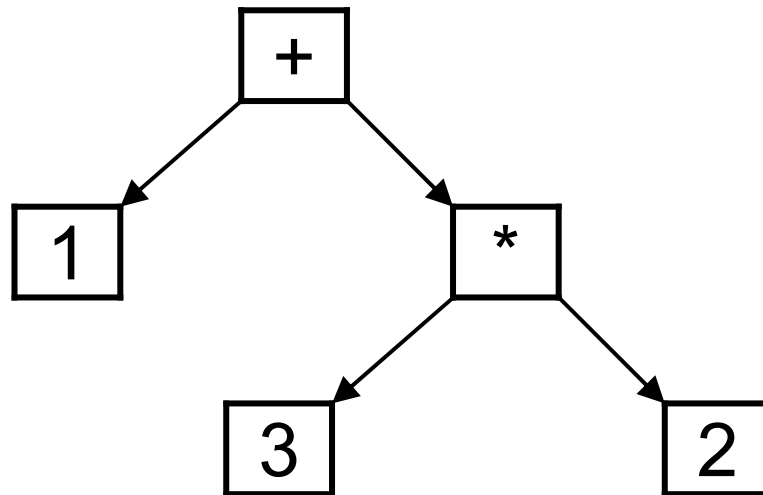
- A RI deveria ter componentes descrevendo operações simples
  - MOVE, um simples acesso, um JUMP, etc
- A idéia é quebrar pedaços complicados da AST em um conjunto de operações de máquina
- Cada operação ainda pode gerar mais de uma instrução *assembly* no final

# Tradução de AST para código de 3 endereços

---

Expressão Matemática:  $1+3*2$

Como seria o código de 3 endereços para a AST?





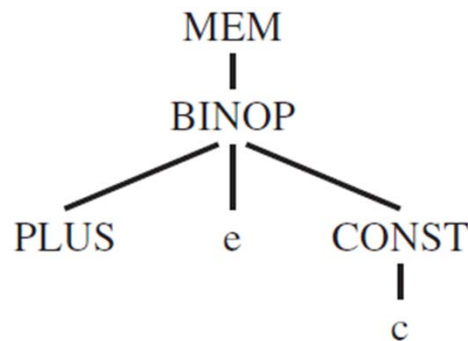
---

# Seleção de Instruções

# Introdução

---

- A árvore da RI expressa uma operação “simples” em cada nó, como por exemplo:
  - Acesso à memória
  - Operador Binário
  - Salto condicional
- Instruções da máquina podem realizar uma ou mais dessas operações
- Que instrução seria essa?



- Encontrar o conjunto de instruções de máquina que implementa uma dada árvore da RI é o objetivo da **Seleção de Instruções**

# Padrões de Árvores

---

- Expressam as instruções da máquina
- Seleção de instruções:
  - Cubra a árvore da RI com um número de padrões existentes para a máquina alvo, segundo alguma métrica.
- Exemplo:
  - Máquina Jouette
    - r0 contém sempre zero

# Padrão Jouette

Name	Effect	Trees
—	$r_i$	TEMP
ADD	$r_i \leftarrow r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$
MUL	$r_i \leftarrow r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$
SUB	$r_i \leftarrow r_j - r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$
DIV	$r_i \leftarrow r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$
ADDI	$r_i \leftarrow r_j + c$	$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
SUBI	$r_i \leftarrow r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \end{array}$
LOAD	$r_i \leftarrow M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\   \quad   \quad   \quad   \\ + \quad + \quad \text{CONST} \quad   \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
STORE	$M[r_j + c] \leftarrow r_i$	$\begin{array}{c} \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\   \quad   \quad   \quad   \\ + \quad + \quad \text{CONST} \quad   \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$
MOVEM	$M[r_j] \leftarrow M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\   \quad   \end{array}$

# Padrão Jouette

---

- Primeira linha não gera instrução
  - TEMP é implementado como registrador
- Duas últimas instruções não geram resultado em registrador
  - Alterações na memória
- Uma instrução pode ter mais de um padrão associado
- Objetivo é cobrir a árvore toda, sem sobreposição entre padrões

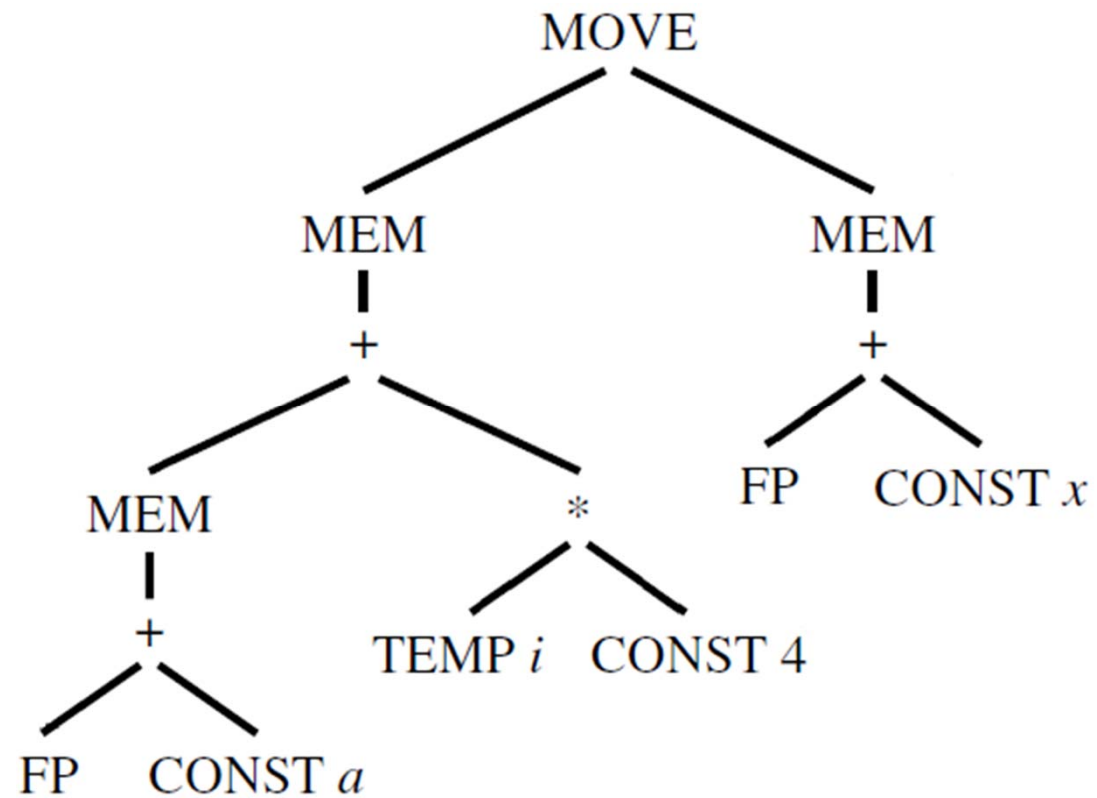
# Seleção de Instruções: Maximal Munch

---

- O maior padrão é aquele com maior número de nós
- Se dois padrões do mesmo tamanho encaixam, a escolha é arbitrária
- Facilmente implementado através de funções recursivas
  - Ordene as cláusulas com a prioridade de tamanho dos padrões
  - Se para cada tipo de nó da árvore existir um padrão de cobertura de um nó, nunca pode ficar travado.
- Bastante simples
  - Inicie na raiz
  - Encontre o maior padrão que possa ser encaixado nesse nó
    - Cubra o raiz e provavelmente outros nós
  - Repita o processo para cada sub-árvore a ser coberta
- A cada padrão selecionado, uma instrução é gerada
- Ordem inversa da execução! A raiz é a última a ser executada

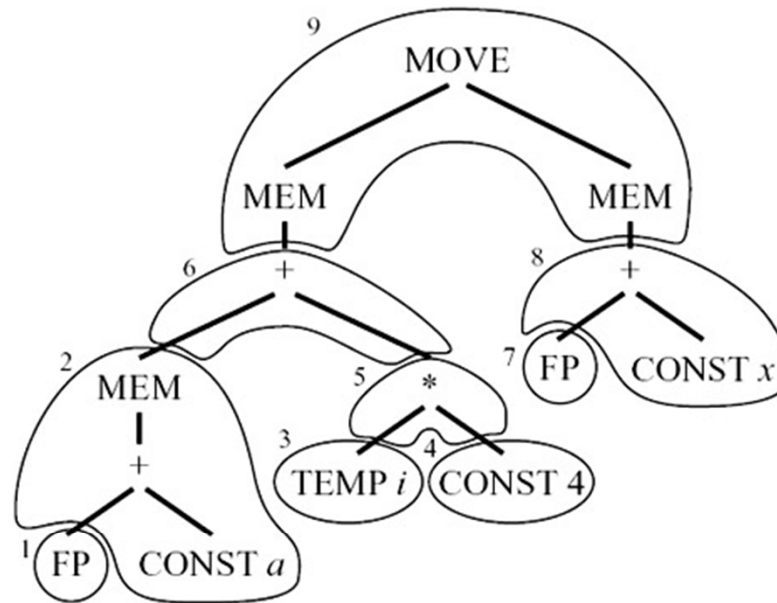
## Seleção de Instruções: Maximal Munch

---



# Seleção de Instruções: Maximal Munch

---



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$



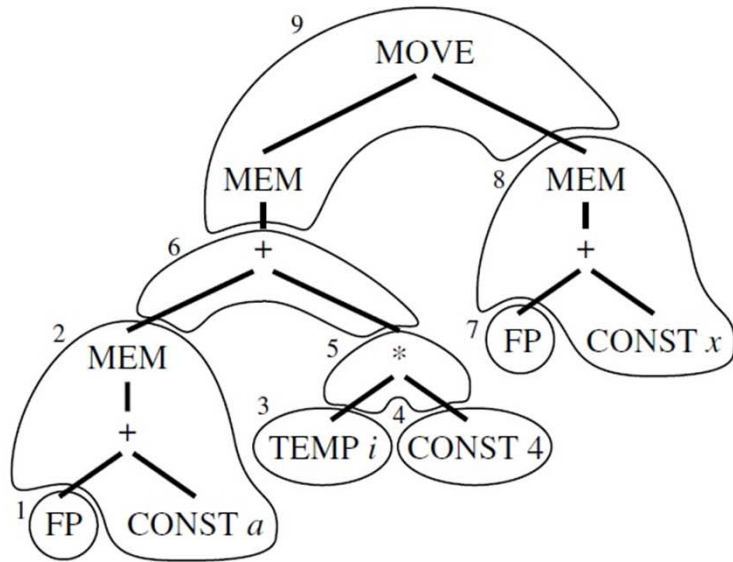
## Seleção de Instruções: Minimal Munch

---

- ADDI  $r1 \leftarrow r0 + a$
- ADD  $r1 \leftarrow \mathbf{fp} + r1$
- LOAD  $r1 \leftarrow M[r1 + 0]$
- ADDI  $r2 \leftarrow r0 + 4$
- MUL  $r2 \leftarrow r1 \times r2$
- ADD  $r1 \leftarrow r1 + r2$
- ADDI  $r2 \leftarrow r0 + x$
- ADD  $r2 \leftarrow \mathbf{fp} + r2$
- LOAD  $r2 \leftarrow M[r2 + 0]$
- STORE  $M[r1 + 0] \leftarrow r2$

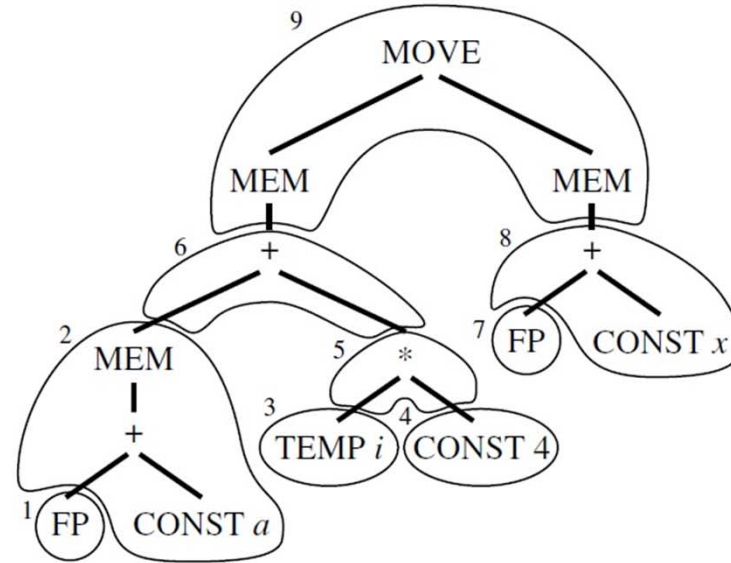
Cobertura da árvore utilizando  
Minimal Munch

# Seleção de Instruções: Outras Possibilidades



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

(a)



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

(b)

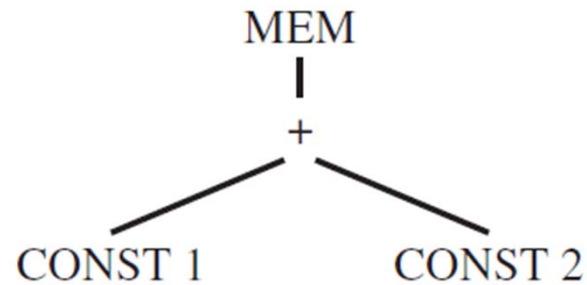
## Seleção de Instruções: Programação Dinâmica

---

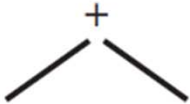
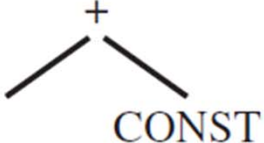
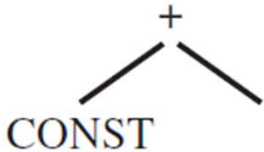
- Encontra uma cobertura ótima (optimum)
- PD monta uma solução ótima baseada em soluções ótimas de sub-problemas
- O algoritmo atribui um custo a cada nó da árvore
  - A soma do custo de todas as instruções da melhor cobertura da sub-árvore com raiz no respectivo nó
  - Para um dado nó  $n$ 
    - Encontra o melhor custo para suas sub-árvores
    - Analisa os padrões que podem cobrir  $n$
    - Algoritmo *Bottom-up*

## Seleção de Instruções: Programação Dinâmica

---

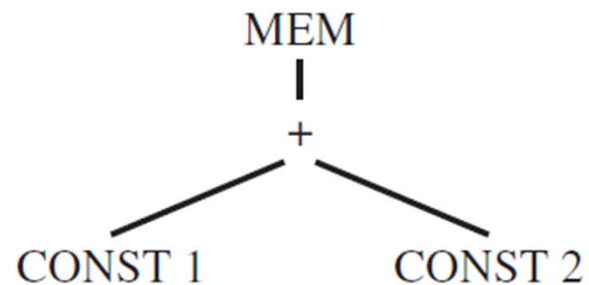


**CONST possui custo 1**

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	ADD	1	1+1	3
	ADDI	1	1	2
	ADDI	1	1	2

## Seleção de Instruções: Programação Dinâmica

---



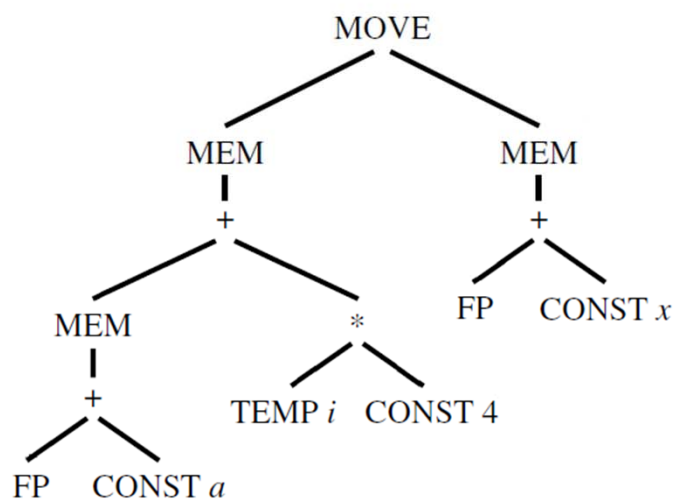
Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	LOAD	1	2	3
	LOAD	1	1	2
	LOAD	1	1	2

## Seleção de Instruções: Programação Dinâmica

---

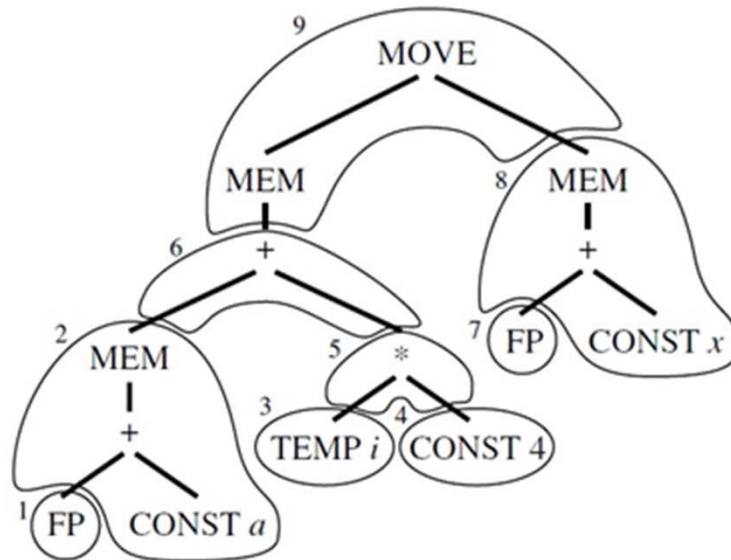
- Após computar o custo da raiz, emitir as instruções
- Emissão de  $(n)$ 
  - Para cada folha  $f$  **do padrão** selecionado para  $n$ , execute  $\text{emissão}(f)$
  - Emita a instrução do padrão de  $n$

# Seleção de Instruções: Programação Dinâmica



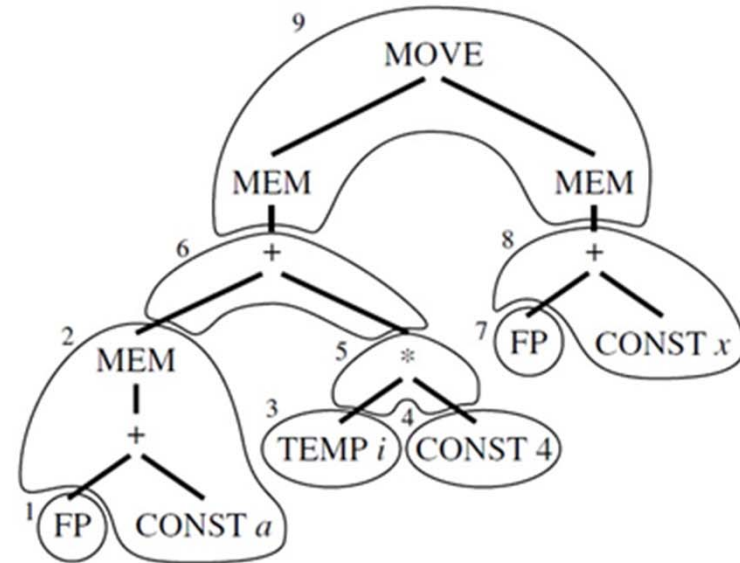
Name	Effect	Trees	Tile Cost
—	$r_i$	TEMP	
ADD	$r_i \leftarrow r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$	1
MUL	$r_i \leftarrow r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$	
SUB	$r_i \leftarrow r_j - r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$	1
DIV	$r_i \leftarrow r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$	
ADDI	$r_i \leftarrow r_j + c$	$\begin{array}{c} + \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$	1
SUBI	$r_i \leftarrow r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \end{array}$	1
LOAD	$r_i \leftarrow M[r_j + c]$	$\begin{array}{c} \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\   \quad   \quad   \quad   \\ + \quad + \quad \text{CONST} \quad   \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$	1
STORE	$M[r_j + c] \leftarrow r_i$	$\begin{array}{c} \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \quad \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \quad \text{MEM} \quad \text{MEM} \\   \quad   \quad   \quad   \\ + \quad + \quad \text{CONST} \quad   \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{CONST} \quad \text{CONST} \quad \text{CONST} \quad \text{CONST} \end{array}$	2
MOVEM	$M[r_j] \leftarrow M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\   \quad   \end{array}$	3

# Seleção de Instruções



2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	LOAD	$r_2 \leftarrow M[\mathbf{fp} + x]$
9	STORE	$M[r_1 + 0] \leftarrow r_2$

Programação Dinâmica

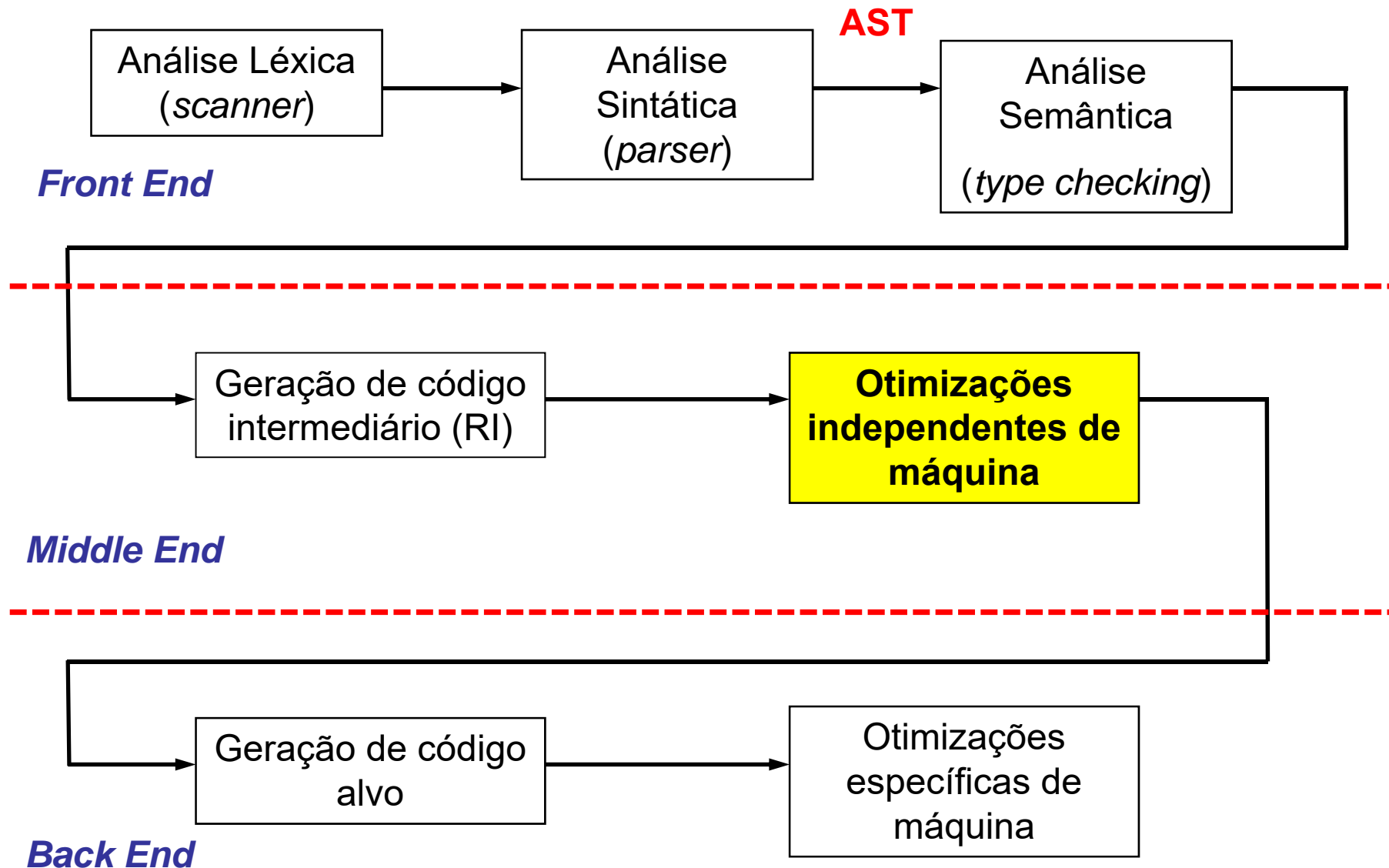


2	LOAD	$r_1 \leftarrow M[\mathbf{fp} + a]$
4	ADDI	$r_2 \leftarrow r_0 + 4$
5	MUL	$r_2 \leftarrow r_i \times r_2$
6	ADD	$r_1 \leftarrow r_1 + r_2$
8	ADDI	$r_2 \leftarrow \mathbf{fp} + x$
9	MOVEM	$M[r_1] \leftarrow M[r_2]$

Maximal Munch



# Fluxo do Compilador



---

# Conceitos de Otimização de Código

# Introdução

---

- **Melhorar o algoritmo é tarefa do programador**
- **O compilador pode ser útil para:**
  - Aplicar transformações que tornam o código gerado mais eficiente
  - Deixa o programador livre para escrever um código limpo

# Quick Sort

---

```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if ( n <= m ) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while(1) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if ( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

**Fig. 10.2.** C code for quicksort.

# Quick Sort

---

(1) $i := m - 1$	(16) $t_7 := 4 * i$
(2) $j := n$	(17) $t_8 := 4 * j$
(3) $t_1 := 4 * n$	(18) $t_9 := a[t_8]$
(4) $v := a[t_1]$	(19) $a[t_7] := t_9$
(5) $i := i + 1$	(20) $t_{10} := 4 * j$
(6) $t_2 := 4 * i$	(21) $a[t_{10}] := x$
(7) $t_3 := a[t_2]$	(22) $\text{goto } (5)$
(8) $\text{if } t_3 < v \text{ goto } (5)$	(23) $t_{11} := 4 * i$
(9) $j := j - 1$	(24) $x := a[t_{11}]$
(10) $t_4 := 4 * j$	(25) $t_{12} := 4 * i$
(11) $t_5 := a[t_4]$	(26) $t_{13} := 4 * n$
(12) $\text{if } t_5 > v \text{ goto } (9)$	(27) $t_{14} := a[t_{13}]$
(13) $\text{if } i \geq j \text{ goto } (23)$	(28) $a[t_{12}] := t_{14}$
(14) $t_6 := 4 * i$	(29) $t_{15} := 4 * n$
(15) $x := a[t_6]$	(30) $a[t_{15}] := x$

**Fig. 10.4.** Three-address code for fragment in Fig. 10.2.

# Quick Sort

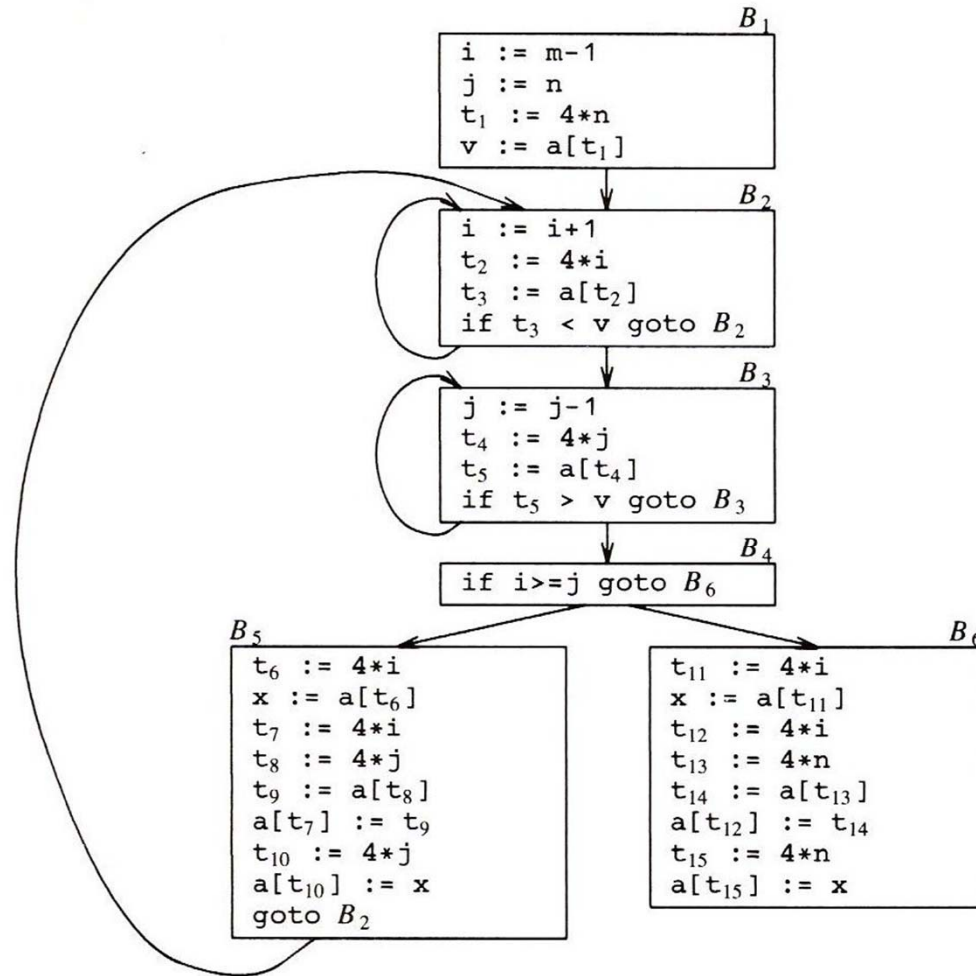


Fig. 10.5. Flow graph.

# Principais Fontes de Otimização

---

- **Transformações que preservam a funcionalidade**
  - Eliminação de Sub-Expressões comuns (CSE)
  - Propagação de Cópias
  - Eliminação de código morto
  - *Constant Folding*
- **Transformações Locais**
  - Dentro de um bloco básico
- **Transformações Globais**
  - Envolve mais de um bloco básico

## Local CSE

- $E$  é sub-expressão comum se

- $E$  foi previamente computada
- Os valores usados por  $E$  não sofreram alterações

$B_5$

```
t6 := 4*i  
x := a[t6]  
t7 := 4*i  
t8 := 4*j  
t9 := a[t8]  
a[t7] := t9  
t10 := 4*j  
a[t10] := x  
goto B2
```

(a) Before

$B_5$

```
t6 := 4*i  
x := a[t6]  
t8 := 4*j  
t9 := a[t8]  
a[t6] := t9  
a[t8] := x  
goto B2
```

(b) After

**Fig. 10.6.** Local common subexpression elimination.



# Global CSE

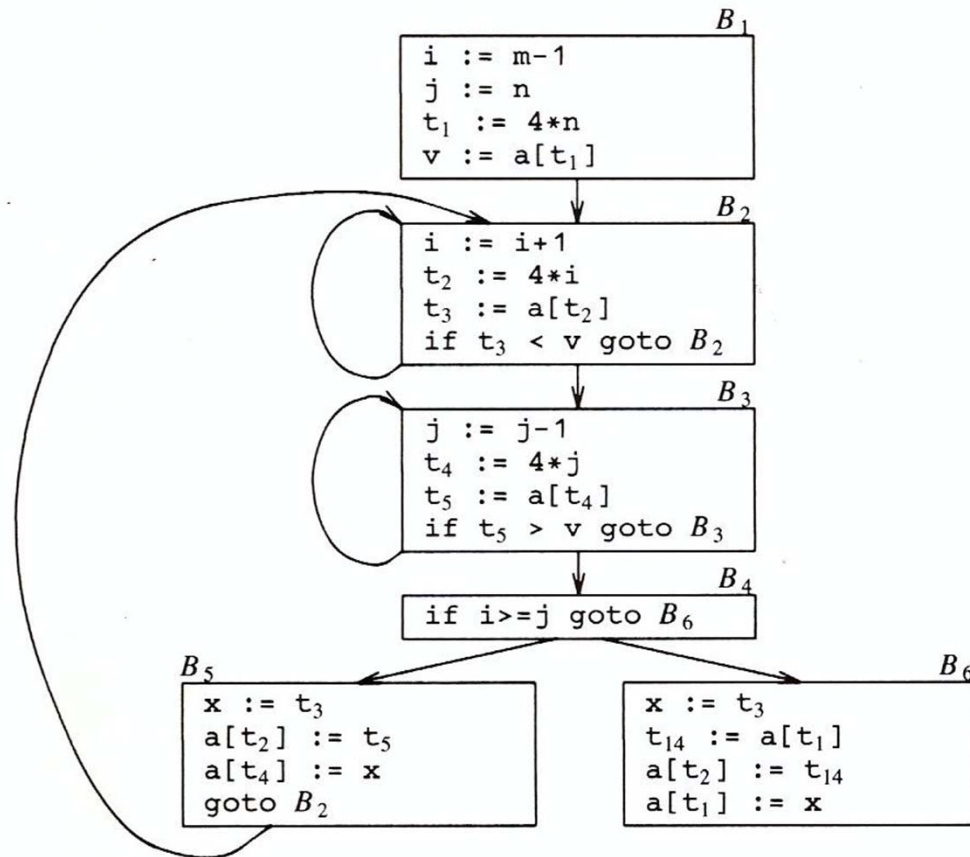


Fig. 10.7.  $B_5$  and  $B_6$  after common subexpression elimination.

# Copy Propagation

---

- Voltemos ao bloco B5
  - Dá para melhorá-lo ainda mais?

# Dead Code Elimination

---

- Código morto
  - Sentenças que computam valores que nunca são usados
- Pode ser inserido
  - Pelo programador
    - If( debug ) {...}
  - Por outras transformações
    - Copy propagation
      - Bloco B5 do exemplo anterior

---

# **Blocos Básicos e Grafos de Fluxo de Controle**

# Introdução

---

- Representação gráfica do código de 3 endereços é útil para entender os algoritmos de otimização
- **Nós:** computação
- **Arestas:** fluxo de controle
- Muito usado em coletas de informações sobre o programa

# Blocos Básicos

---

- Seqüência de instruções consecutivas
- Fluxo de Controle:
  - Entra no início
  - Sai pelo final
  - Não existem saltos para dentro ou do meio para fora da seqüência

**t1 = a \* a**

**t2 = a \* b**

**t3 = b \* 3**

**t4 = t2 - t3**

# Algoritmo para Quebra em BBs

---

- **Entrada:** seqüência de código de 3 endereços
- **Defina os líderes** (iniciam os BBs):
  - Primeira Sentença é um líder
  - Todo alvo de um **goto**, **condicional** ou **incondicional**, é um líder
  - Toda sentença que sucede imediatamente um **goto**, **condicional** ou **incondicional**, é um líder
- Os BBs são compostos pelos líderes e todas as instruções subsequentes até o próximo líder (exclusive)

# Quick Sort

---

(1) $i := m - 1$	(16) $t_7 := 4 * i$
(2) $j := n$	(17) $t_8 := 4 * j$
(3) $t_1 := 4 * n$	(18) $t_9 := a[t_8]$
(4) $v := a[t_1]$	(19) $a[t_7] := t_9$
(5) $i := i + 1$	(20) $t_{10} := 4 * j$
(6) $t_2 := 4 * i$	(21) $a[t_{10}] := x$
(7) $t_3 := a[t_2]$	(22) $\text{goto } (5)$
(8) $\text{if } t_3 < v \text{ goto } (5)$	(23) $t_{11} := 4 * i$
(9) $j := j - 1$	(24) $x := a[t_{11}]$
(10) $t_4 := 4 * j$	(25) $t_{12} := 4 * i$
(11) $t_5 := a[t_4]$	(26) $t_{13} := 4 * n$
(12) $\text{if } t_5 > v \text{ goto } (9)$	(27) $t_{14} := a[t_{13}]$
(13) $\text{if } i \geq j \text{ goto } (23)$	(28) $a[t_{12}] := t_{14}$
(14) $t_6 := 4 * i$	(29) $t_{15} := 4 * n$
(15) $x := a[t_6]$	(30) $a[t_{15}] := x$

**Fig. 10.4.** Three-address code for fragment in Fig. 10.2.



# Quick Sort

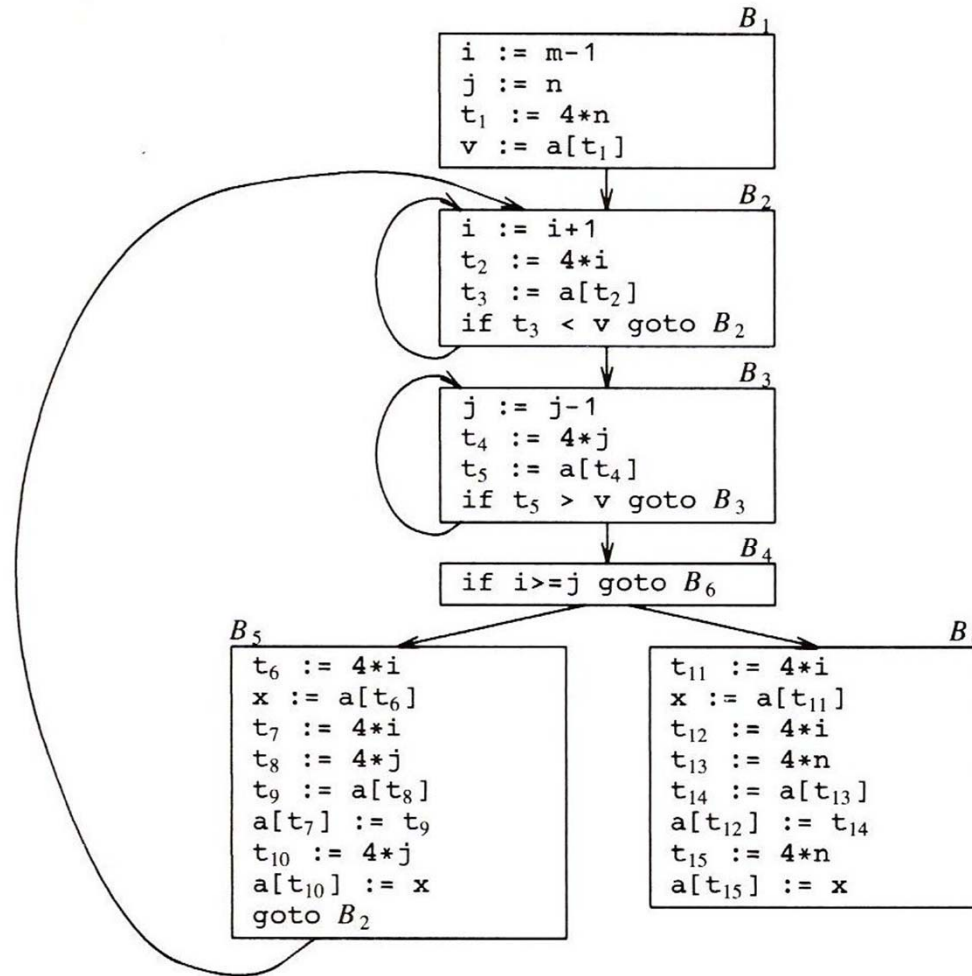


Fig. 10.5. Flow graph.

---

# **Análise de Fluxo de Dados**

# Introdução

---

- **Otimização**
  - Transformações para ganho de eficiência
  - Não podem alterar a saída do programa
- **Exemplos:**
  - ***Dead Code Elimination***: Apaga uma computação cujo resultado nunca será usado
  - ***Register Allocation***: Reaproveitamento de registradores
  - ***Common-subexpression Elimination***: Se uma expressão é computada mais de uma vez, elimine uma das computações
  - ***Constant Folding***: Se os operandos são constantes, calcule a expressão em tempo de compilação

# Introdução

---

- Otimizações são transformações feitas com base em informações coletadas do programa
- Coletar informações é trabalho da análise de fluxo de dados.
- Intraprocedural global optimization
  - Interna a um procedimento ou função
  - Engloba todos os blocos básicos

# Introdução

---

- **Idéia básica**
  - Atravessar o grafo de fluxo de controle do programa coletando informações sobre a execução
  - Conservativamente!
  - Modificar o programa para torná-lo mais eficiente em algum aspecto:
    - Desempenho
    - Tamanho
- **Análises são descritas através de equações de fluxo de dados:**

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

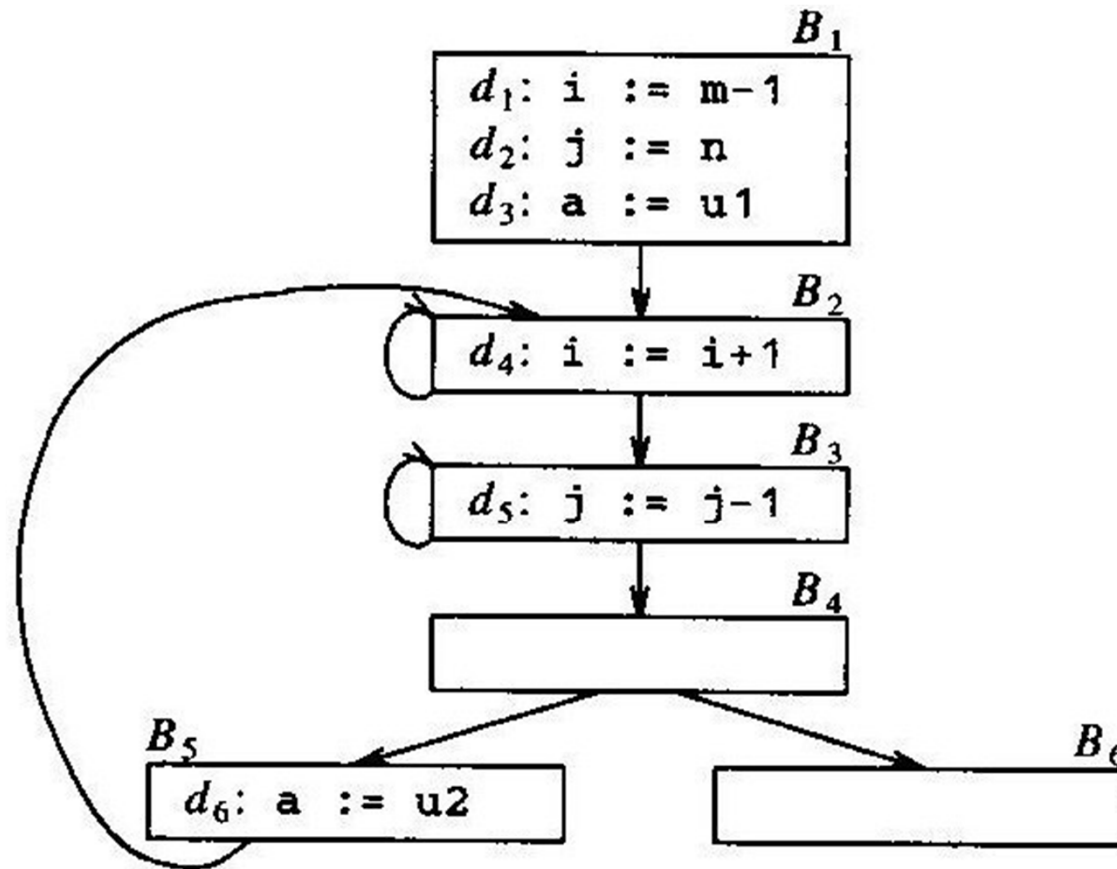
# Introdução

---

As equações podem mudar de acordo com a análise:

- As noções de *gen* e *kill* dependem da informação desejada
- Podem seguir o fluxo de controle ou não
  - Forward
  - Backward
- Chamadas de procedimentos, atribuição a ponteiros e a arrays não serão consideradas em um primeiro momento.

## Pontos e Caminhos



**Fig. 10.19.** A flow graph.

---

# **Análise de Fluxo de Dados: Reaching Definitions**



# Reaching Definitions

---

- **Definição não ambígua de uma variável  $t$ :**

$d: t := a \text{ op } b$

$d: t := M[a]$

- $d$  alcança um uso na sentença  $u$  se:
  - Se existe um caminho no CFG de  $d$  para  $u$
  - Esse caminho não contém outra definição não ambígua de  $t$
- **Definição ambígua**
  - Uma sentença que pode ou não atribuir um valor a  $t$ 
    - CALL
    - Atribuição a ponteiros

# Reaching Definitions

---

- Criam-se IDs para as definições:

**d1**:  $t \leftarrow x \text{ op } y$

- Gera a definição **d1**
  - Mata todas as outras definições da variável  $t$ , pois não alcançam o final dessa instrução.
- **defs(t)** ou **D<sub>t</sub>**: conjunto de todas as definições de  $t$

# Reaching Definitions

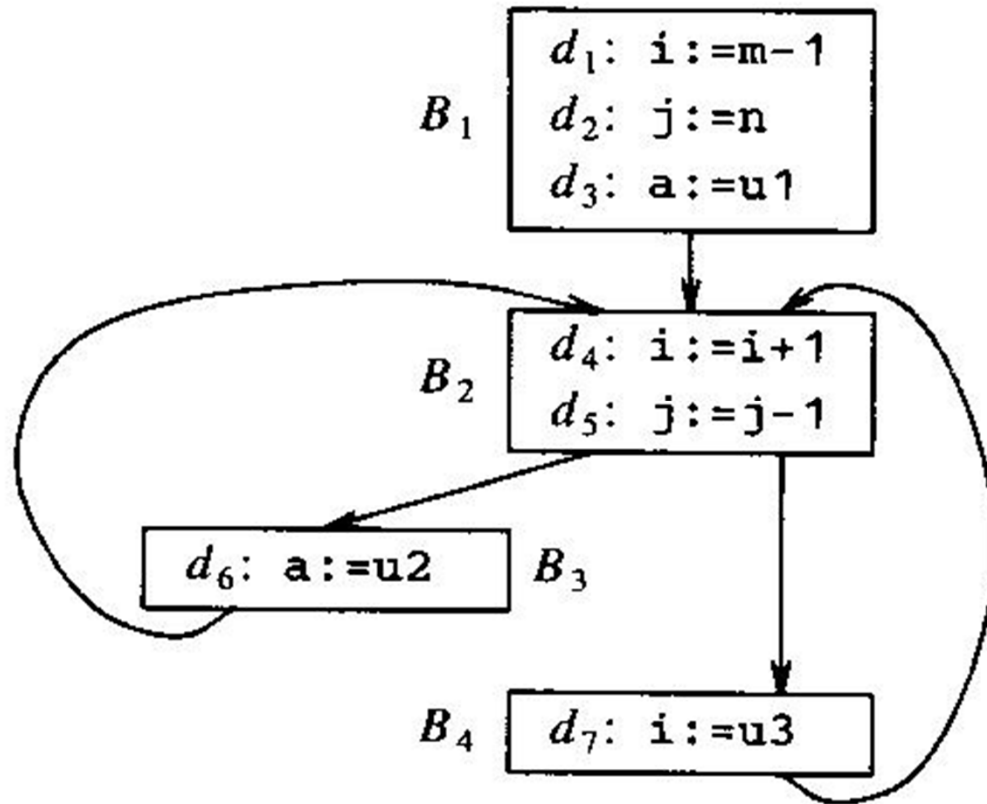
---

## Principal uso:

- Dada uma variável  $x$  em um certo ponto  $p$  do programa, pode-se inferir que o valor de  $x$  é limitado a um determinado grupo de possibilidades.

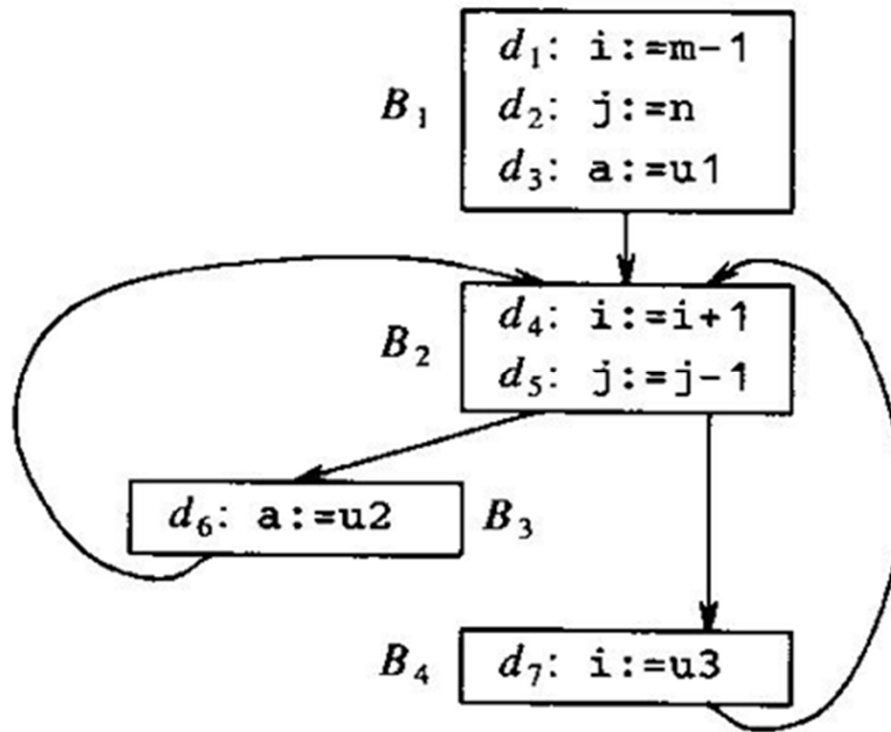
## Reaching Definitions: gen e kill

---



## Reaching Definitions: gen e kill

---



$$\begin{aligned} \text{gen}[B_1] &= \{d_1, d_2, d_3\} \\ \text{kill}[B_1] &= \{d_4, d_5, d_6, d_7\} \end{aligned}$$

$$\begin{aligned} \text{gen}[B_2] &= \{d_4, d_5\} \\ \text{kill}[B_2] &= \{d_1, d_2, d_7\} \end{aligned}$$

$$\begin{aligned} \text{gen}[B_3] &= \{d_6\} \\ \text{kill}[B_3] &= \{d_3\} \end{aligned}$$

$$\begin{aligned} \text{gen}[B_4] &= \{d_7\} \\ \text{kill}[B_4] &= \{d_1, d_4\} \end{aligned}$$

## Reaching Definitions: Equações de DFA

---

- Vendo B como uma sequência de uma ou mais sentenças
  - Pode-se definir
    - $in[B]$ ,  $out[B]$ ,  $gen[B]$ ,  $kill[B]$
  - Computando  $gen$  e  $kill$  para cada B como visto anteriormente
- Tem-se:

$$in[B] = \bigcup_{P \in Pred(B)} out[P]$$

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

## Reaching Definitions: Solução Iterativa

---

```
    /* initialize out on the assumption  $in[B] = \emptyset$  for all  $B$  */  
(1)  for each block  $B$  do  $out[B] := gen[B]$ ;  
(2)   $change := true$ ;          /* to get the while-loop going */  
(3)  while  $change$  do begin  
(4)       $change := false$ ;  
(5)      for each block  $B$  do begin  
(6)           $in[B] := \bigcup_{\substack{P \text{ a prede-} \\ \text{cessor of } B}} out[P]$ ;  
(7)           $oldout := out[B]$ ;  
(8)           $out[B] := gen[B] \cup (in[B] - kill[B])$ ;  
(9)          if  $out[B] \neq oldout$  then  $change := true$   
      end  
  end
```

**Fig. 10.26.** Algorithm to compute *in* and *out*.

# Reaching Definitions: Observações

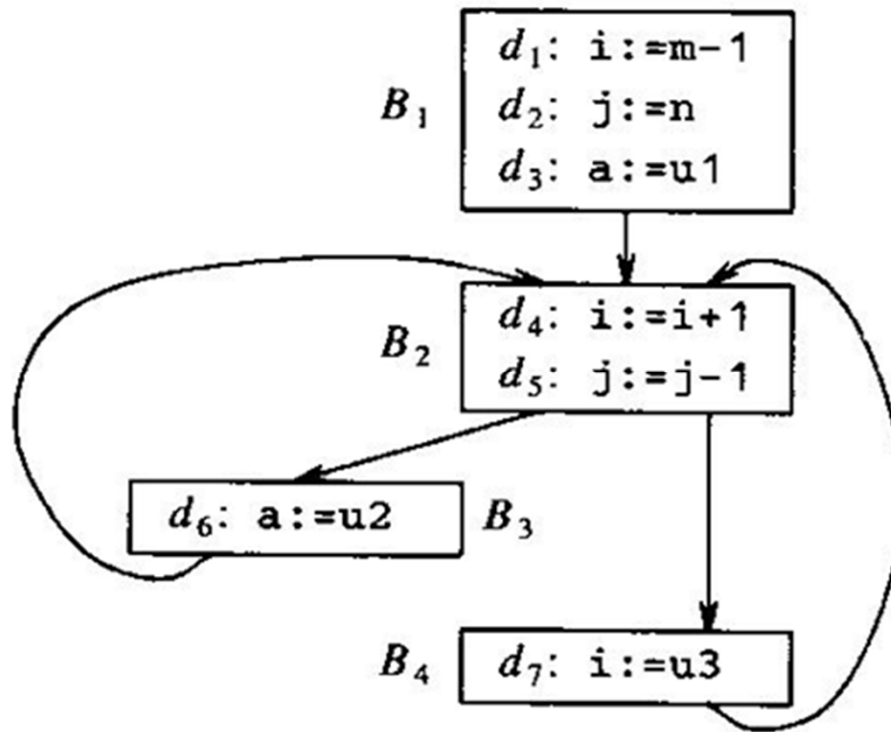
---

- O algoritmo propaga as definições
  - Até onde elas podem chegar sem serem mortas
  - “Simula” todos os caminhos de execução
- O algoritmo sempre termina:
  - $out[B]$  nunca diminui de tamanho
  - o número de definições é finito
  - se  $out[B]$  não muda,  $in[B]$  não muda no próximo passo
  - Limitante superior para número de iterações
    - Número de nós no CFG
    - Pode ser melhorado de acordo com a ordem de avaliação dos nós



# Reaching Definitions: Computar in/out

---



$$\begin{aligned} \text{gen}[B_1] &= \{d_1, d_2, d_3\} \\ \text{kill}[B_1] &= \{d_4, d_5, d_6, d_7\} \end{aligned}$$

$$\begin{aligned} \text{gen}[B_2] &= \{d_4, d_5\} \\ \text{kill}[B_2] &= \{d_1, d_2, d_7\} \end{aligned}$$

$$\begin{aligned} \text{gen}[B_3] &= \{d_6\} \\ \text{kill}[B_3] &= \{d_3\} \end{aligned}$$

$$\begin{aligned} \text{gen}[B_4] &= \{d_7\} \\ \text{kill}[B_4] &= \{d_1, d_4\} \end{aligned}$$