Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Material a seguir

- Parcialmente adaptado do material de aula dos professores
- Tim Teitelbaum (Cornell)
- Wes Weimer (University of Virginia)

Grafos

- Árvores fornecem uma representação natural da estrutura gramatical do código

- A estrutura no entanto é rígida demais para representar outras propriedades do programa

- Para modelar estes aspectos do comportamento dos programas, compiladores costumam usar grafos como representações intermediárias

Control-Flow Graph

- Representa o fluxo de controle do programa
- Nós correspondem a *blocos básicos* de código
- Arestas representam controle sendo transferido
- Representação gráfica das possibilidades do fluxo de execução do programa

Blocos Básicos

- Um bloco básico é uma sequência máxima de instruções *branch-free*
  - sem labels (exceto na primeira instrução)
  - sem desvios (exceto na última instrução)

Blocos Básicos

- Conceito geral: sequência de operações que sempre executam em conjunto
  - não pode desviar para o meio de um bloco básico (apenas no início)
  - não pode desviar no meio de um bloco básico (apenas no fim)
  - cada instrução em um bloco básico é executada após todas as instruções anteriores terem sido executadas

Exemplo

```
1.  L1:
2.      t:=2*x
3.      w:=t+x
4.      if w>0 goto L2
```

Não há como (3) ser executada
sem (2) ter sido executada antes.

Exemplo

```
1.  L1:
2.      t:=2*x
3.      w:=3*x
4.      if w>0 goto L2
```

Control-Flow Graph

- Grafo direcionado modelando o fluxo de controle entre blocos básicos de um programa
- Nós são blocos básicos de código
- Aresta liga um bloco A ao bloco B se a execução pode seguir da última instrução de A para a primeira instrução de B
  - a última instrução em A é um desvio pra B
  - B está sequencialmente após o bloco A

Control-Flow Graph

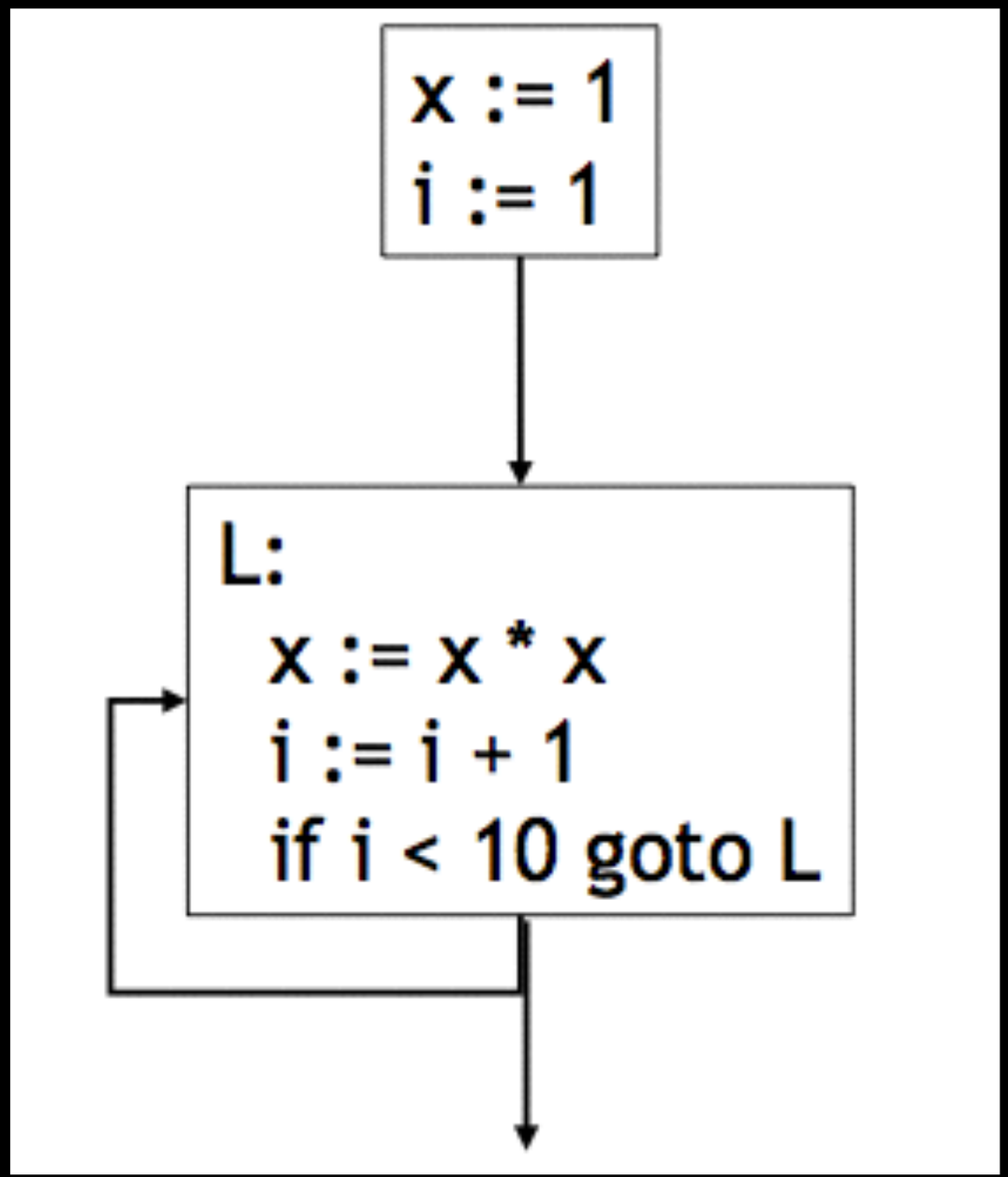
- O grafo fornece uma representação gráfica dos possíveis caminhos do programa em tempo de execução

- É diferente de uma AST em que as arestas denotam estruturas gramaticais

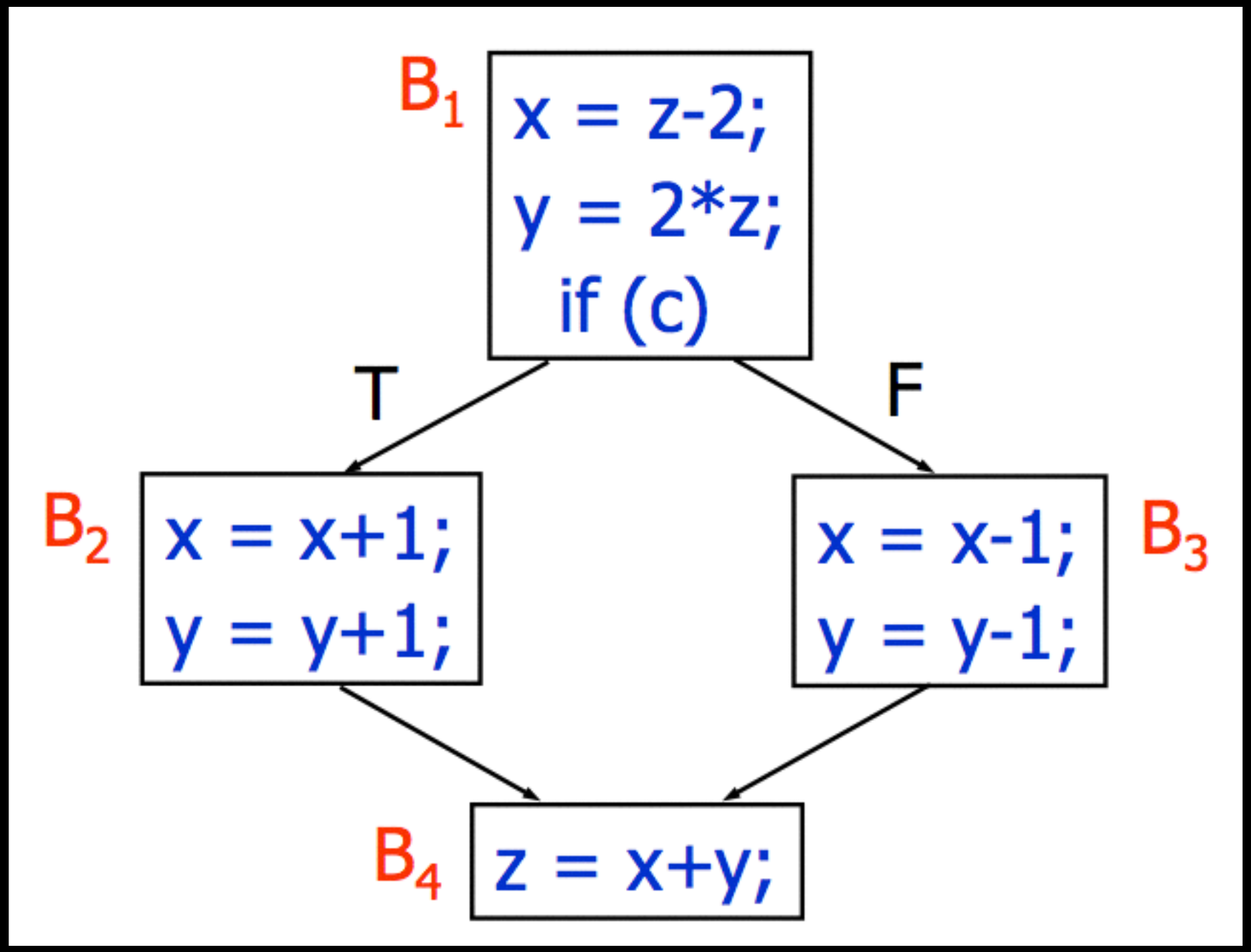
- Portanto, diferente de uma AST, é possível criar ciclos, ao modelarmos *loops*, por exemplo

Exemplo CFG

- O corpo de um método ou procedimento pode ser representado como um CFG
- Há um nó inicial (*start node*)
- Todos os nós de retorno são terminais

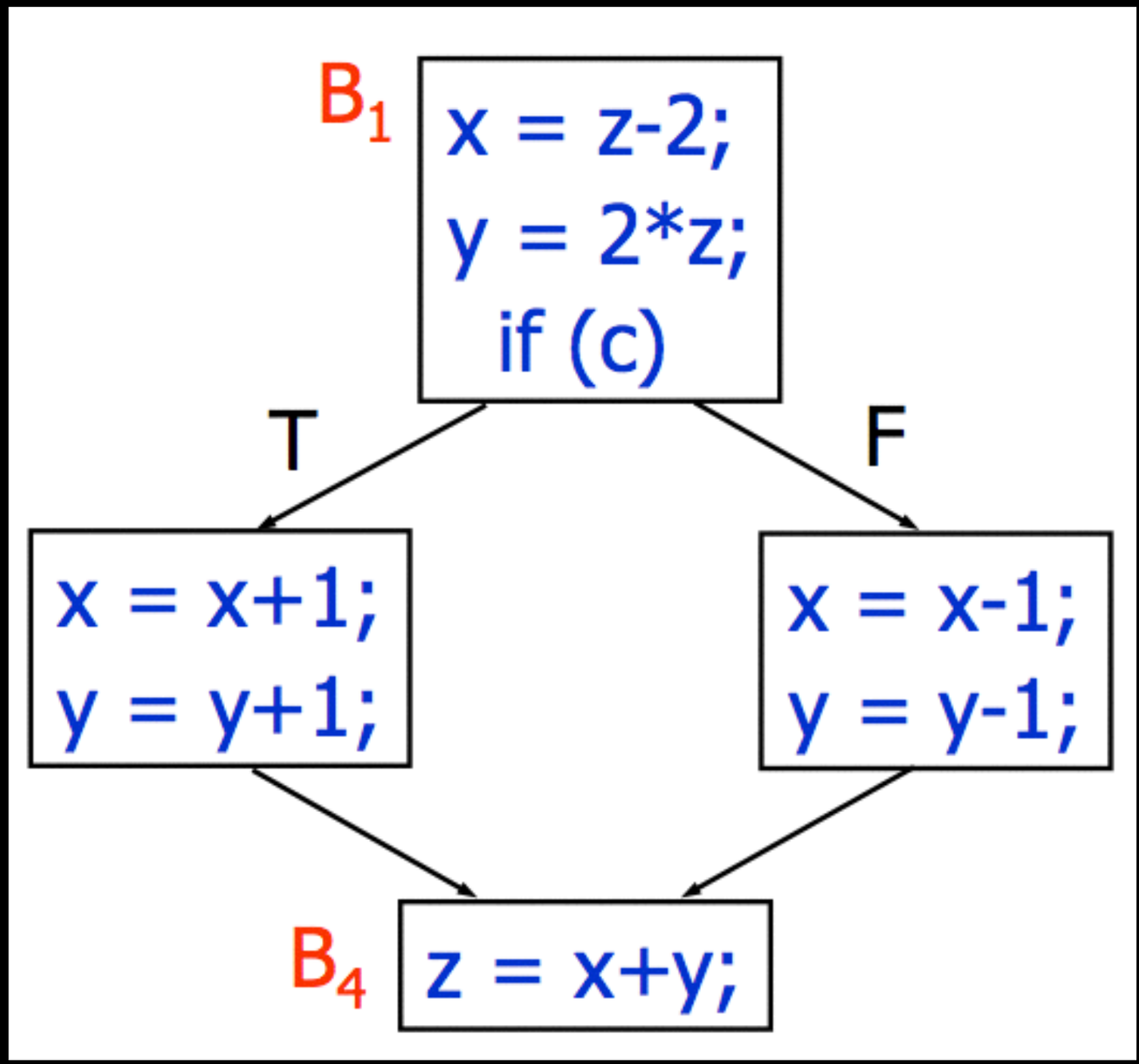


```
x = z-2 ;
y = 2*z;
if (c)  {
    x = x+1;
    y = y+1;
}
else {
    x = x-1;
    y = y-1;
}
z = x+y;
```



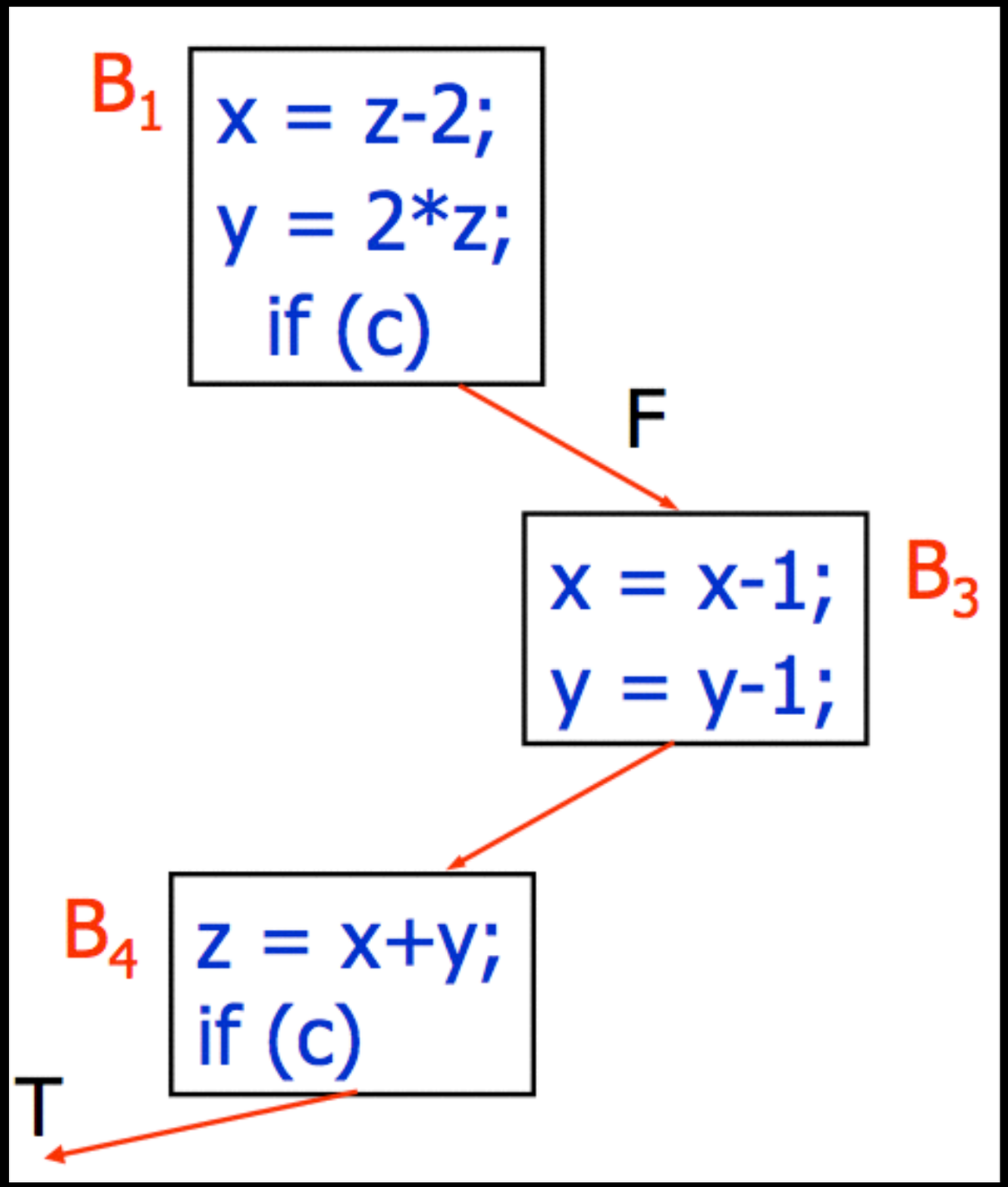
Fluxos

- CFG modela todos os possíveis fluxos do programa
- Uma execução possível é um caminho no grafo
- Se há múltiplos caminhos há múltiplas possíveis execuções do programa



Fluxos

- Um caminho no grafo é uma **possível** execução
- Podem existir caminhos impossíveis
- No exemplo ao lado, c não pode ser ao mesmo tempo *False* e *True*

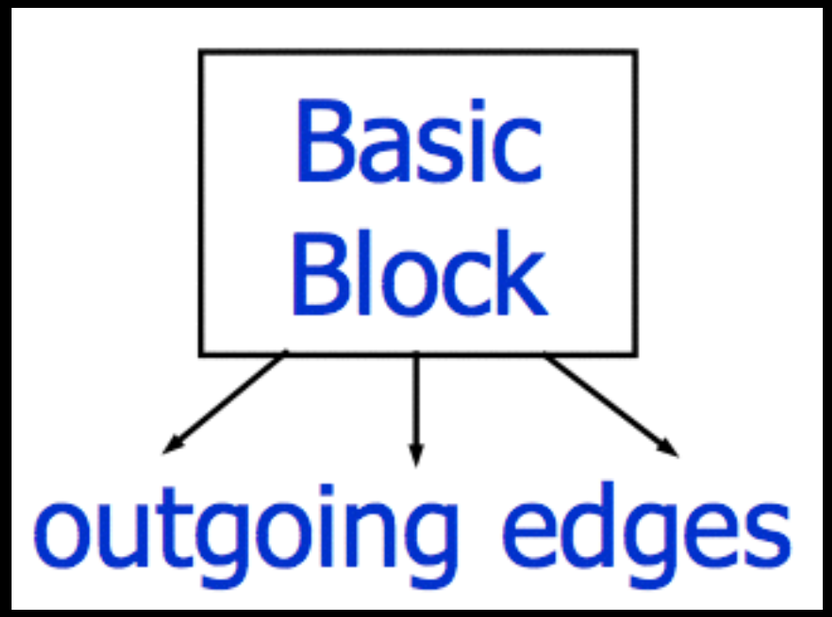


Outgoing

- Múltiplas arestas saindo de um nó indicam vários possíveis fluxos de controle do programa

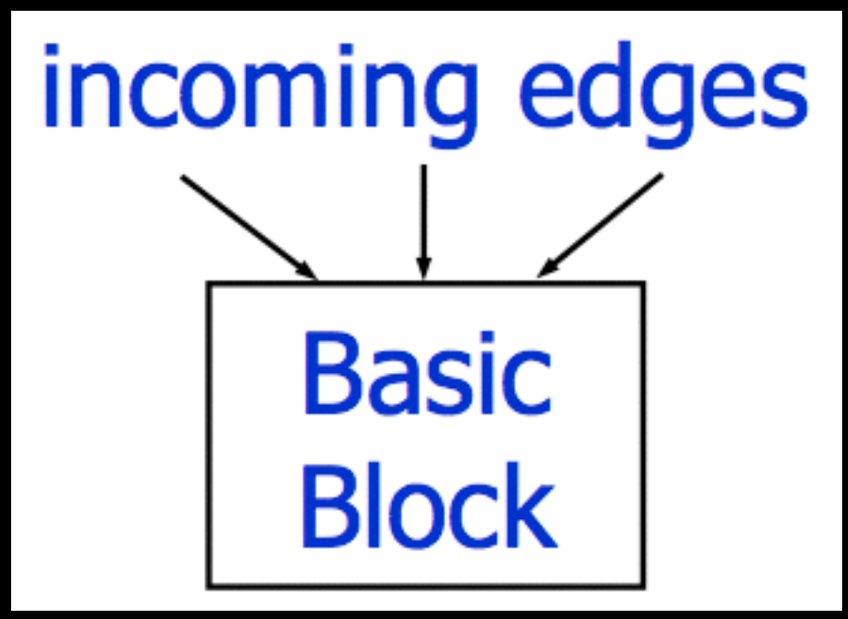
- Cada aresta representa uma possível execução do programa

- Próximo bloco a ser executado pode ser um dentre os sucessores



Incoming

- Fluxo de controle pode vir de qualquer um dos blocos predecessores
- Cada aresta representa uma possível execução do programa



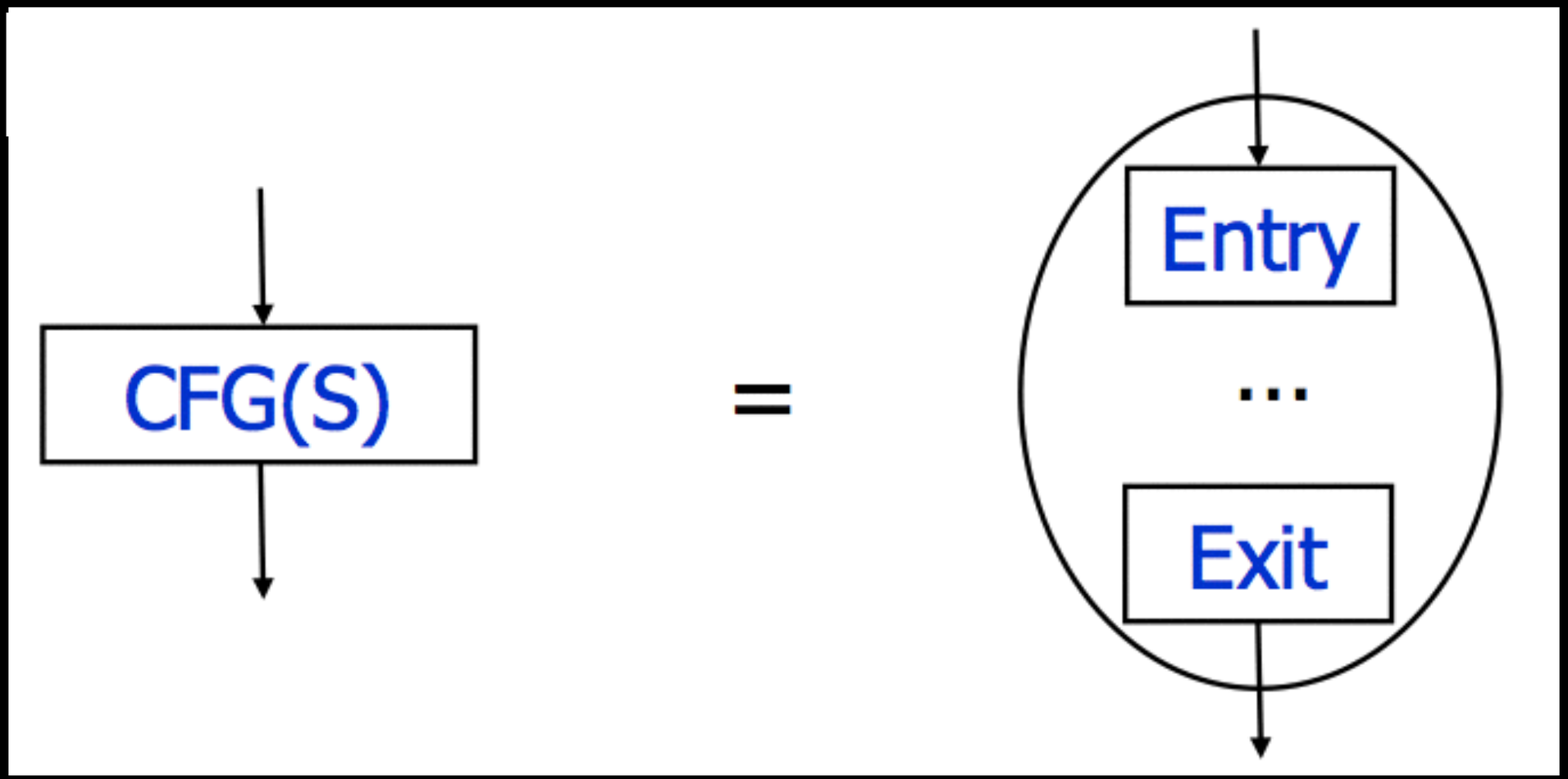
Construindo o CFG

- Podemos construir CFGs para representação intermediária de alto nível ou de baixo nível
- No caso de ASTs, a construção se dá traduzindo cada nó a um CFG e fazendo a composição
- No caso de representações mais baixo nível, como código de três endereços, por meio da análise de labels e instruções com desvios

CFG de um *Statement*

- CFG(S) = grafo de uma instrução alto nível S
- CFG(S) é um grafo de entrada e saída simples
  - um nó de entrada
  - um nó de saída
  - o CFG pode então ser definido recursivamente

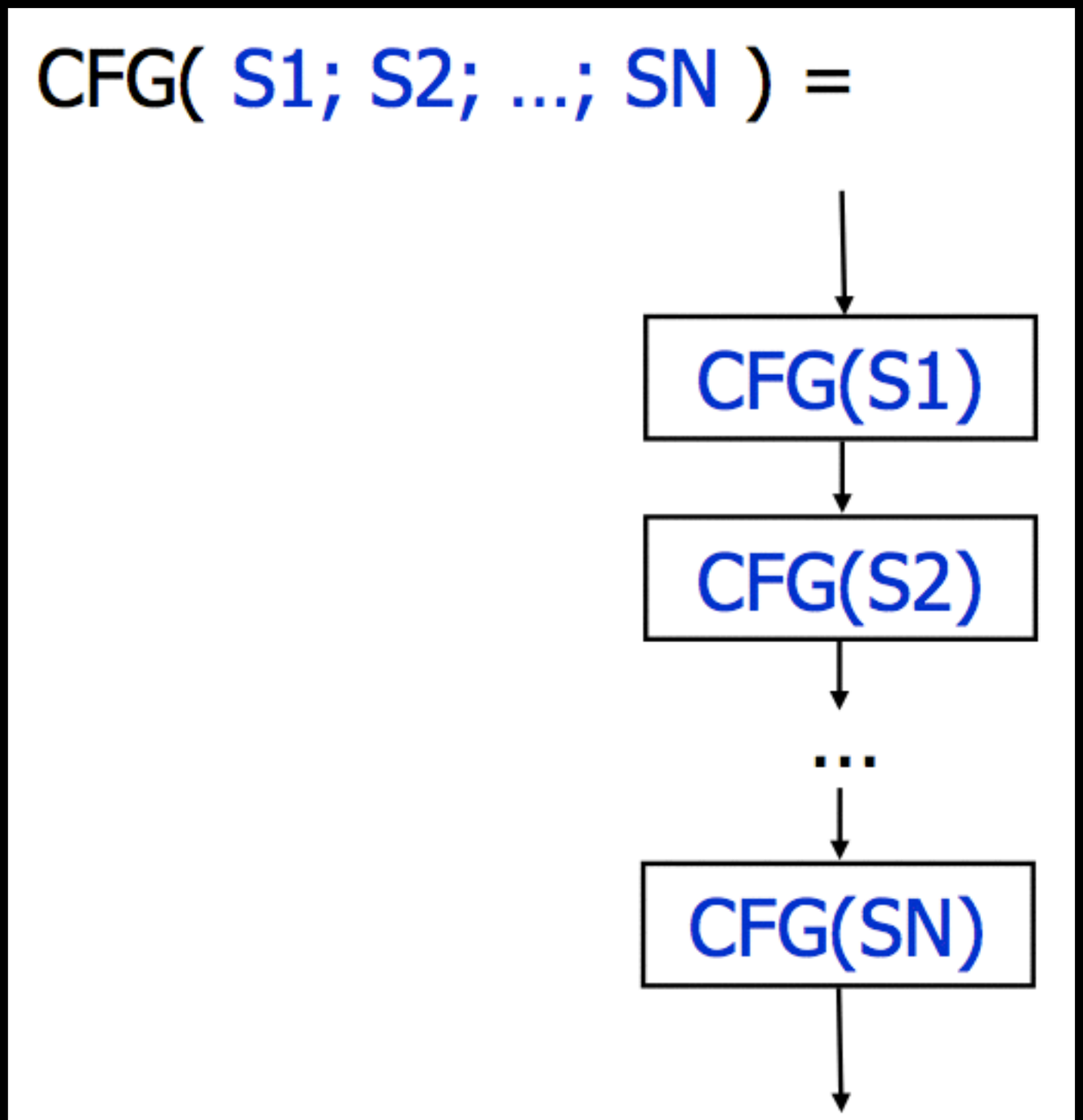
CFG de um *Statement*



Sequências

CFG(S1; S2; ...; SN) =

Sequências

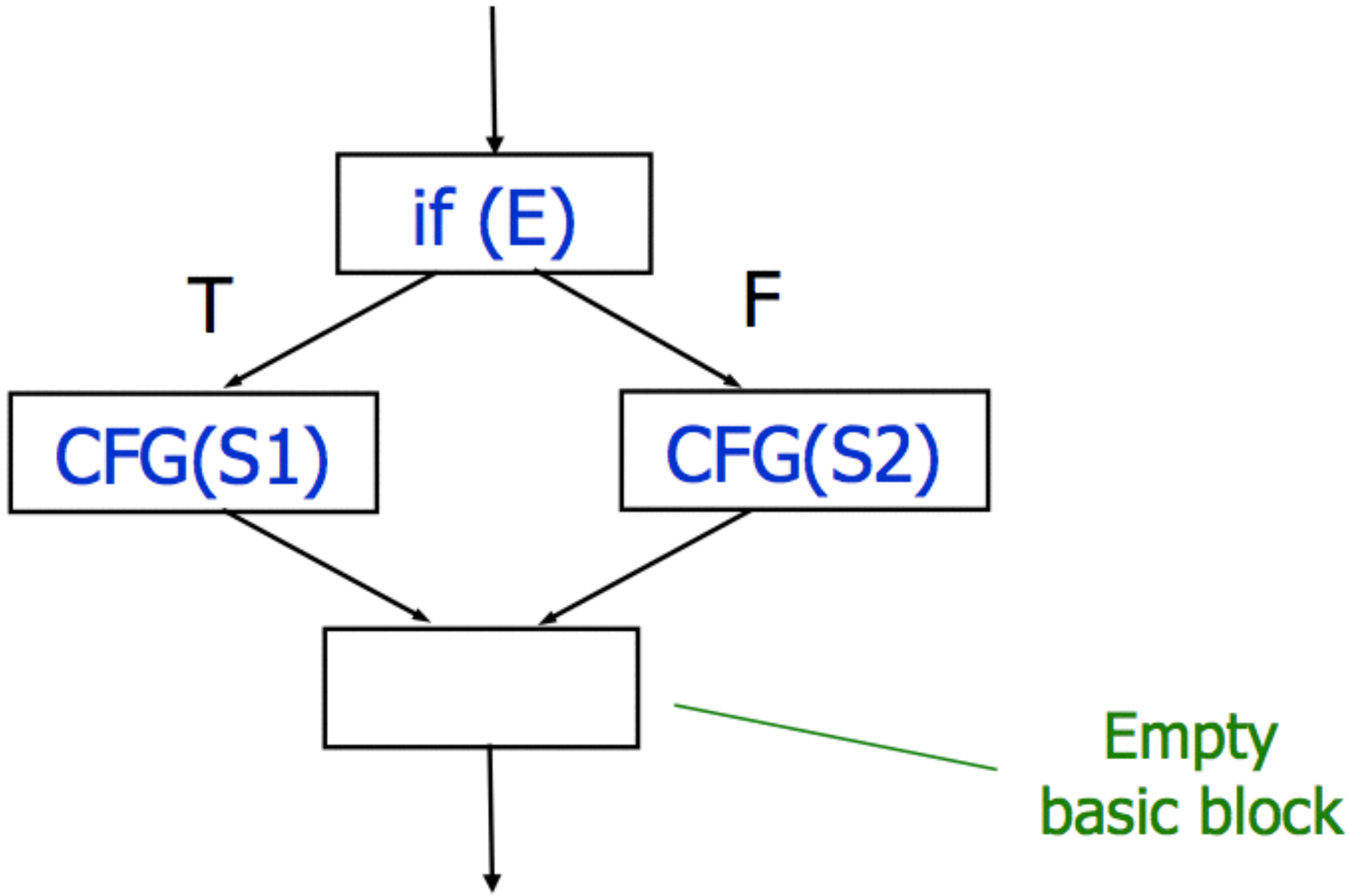


If-then-else

CFG (if (E) S1 else S2)

If-then-else

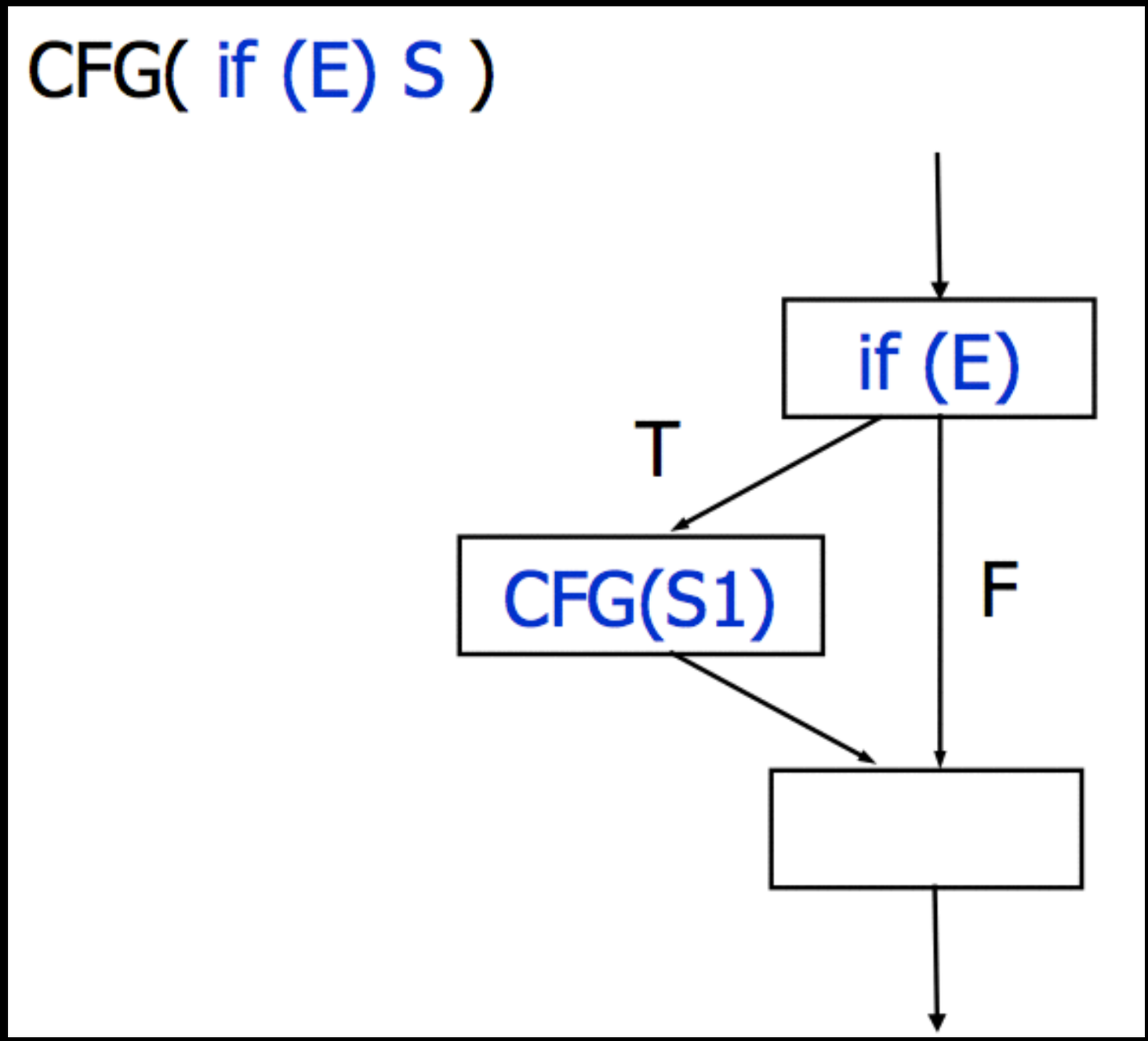
CFG (if (E) S1 else S2)



If-then

CFG(if (E) S)

If-then

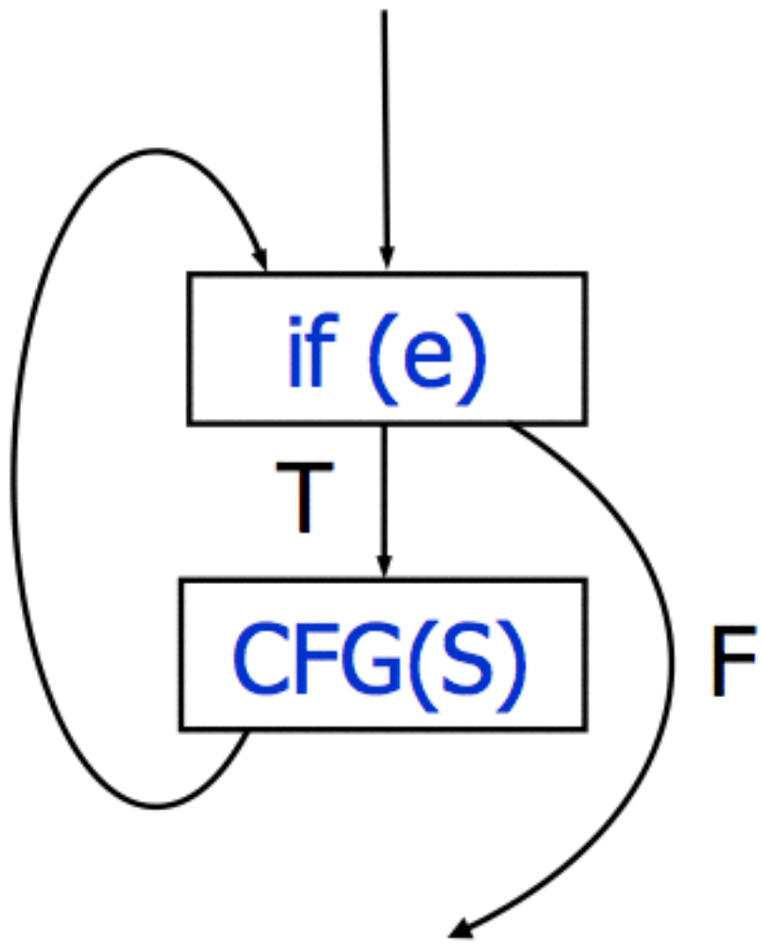


while

CFG for: while (e) S

while

CFG for: while (e) S



Construção recursiva

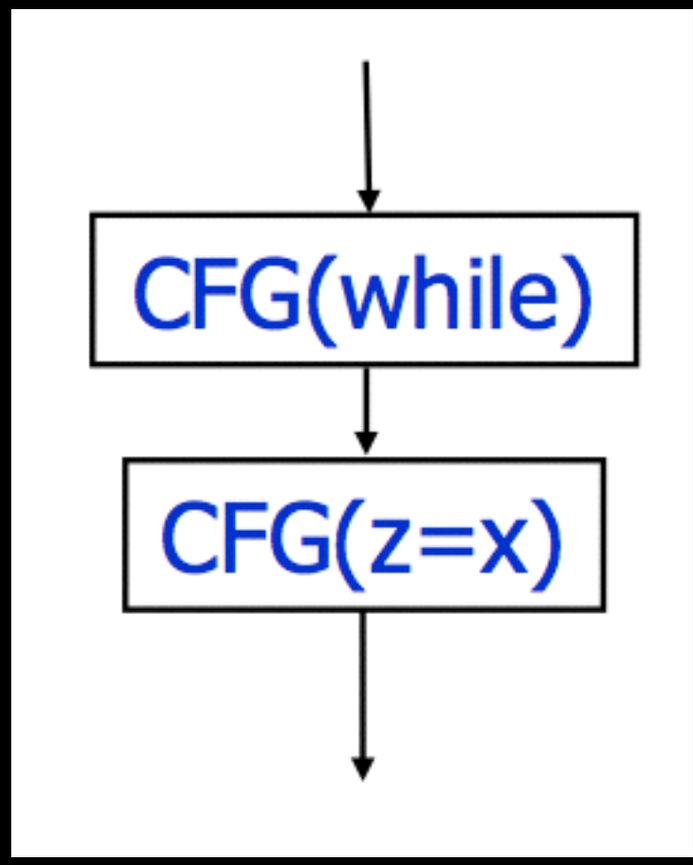
- Instruções aninhadas — construção recursiva dos CFGs ao visitar os nós da representação intermediária

Exemplo

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```

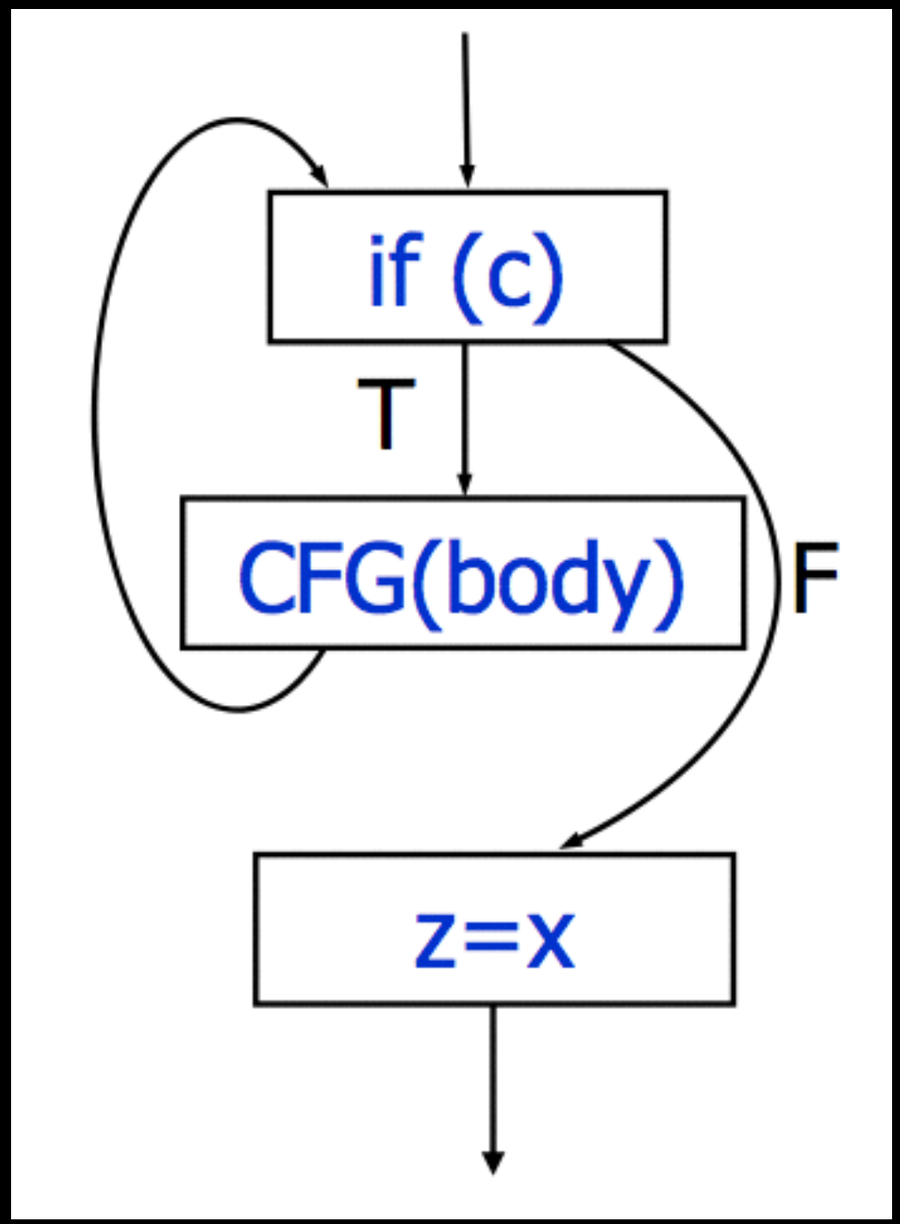
Exemplo

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



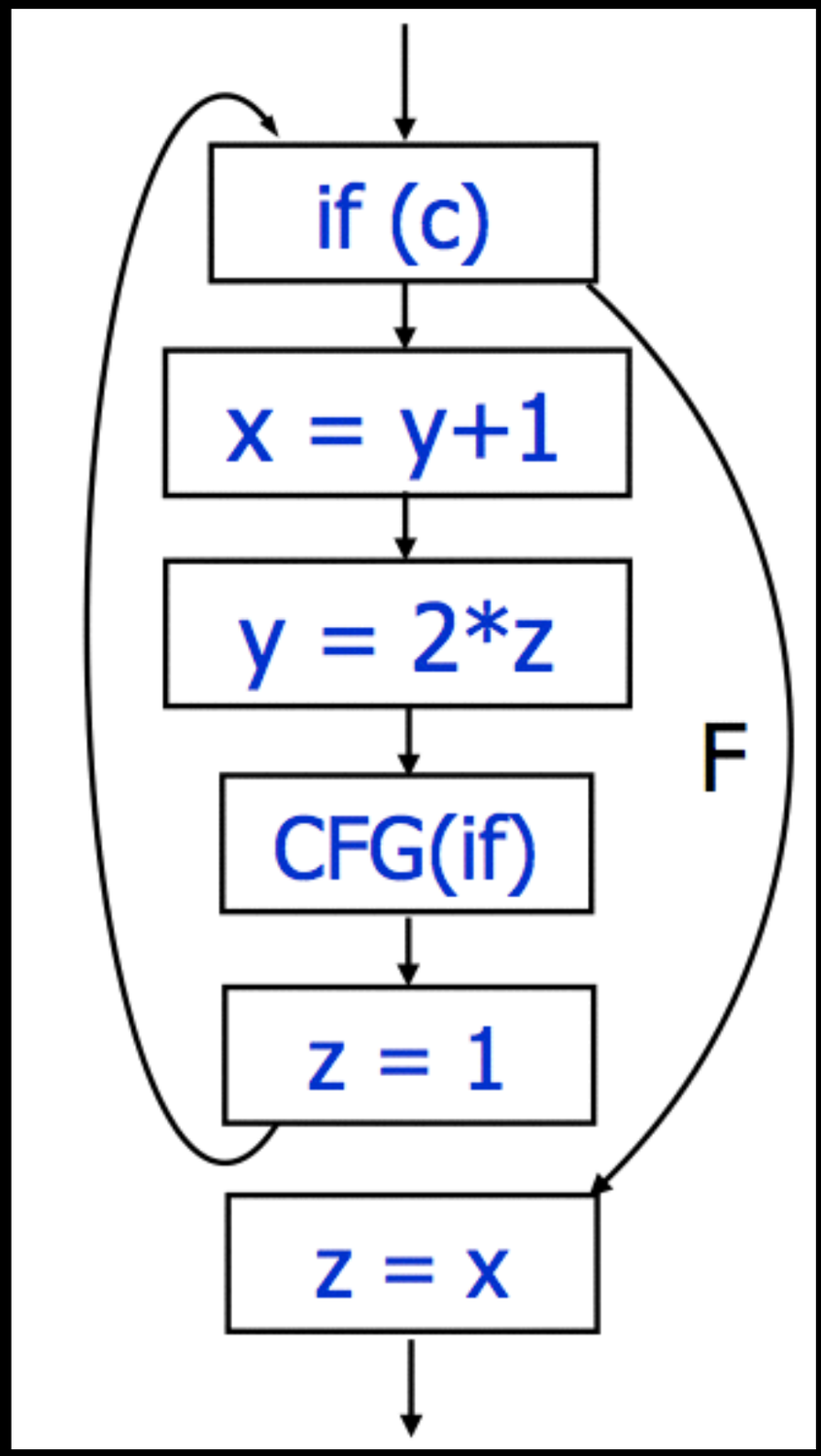
Exemplo

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



Exemplo

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



Construção recursiva

- Algoritmo simples
- CFG gerado
  - cada bloco tem apenas uma instrução
  - existem blocos básicos vazios
- No entanto, isto leva a muitos blocos
- Muitos blocos == ineficiência

CFGs grandes

- Blocos com poucas instruções significa um CFG muito grande
- Compiladores geralmente utilizam CFGs para realizar otimizações
- Muitos nós em um CFG significa que as otimizações serão caras em tempo e espaço

Usos de CFGs

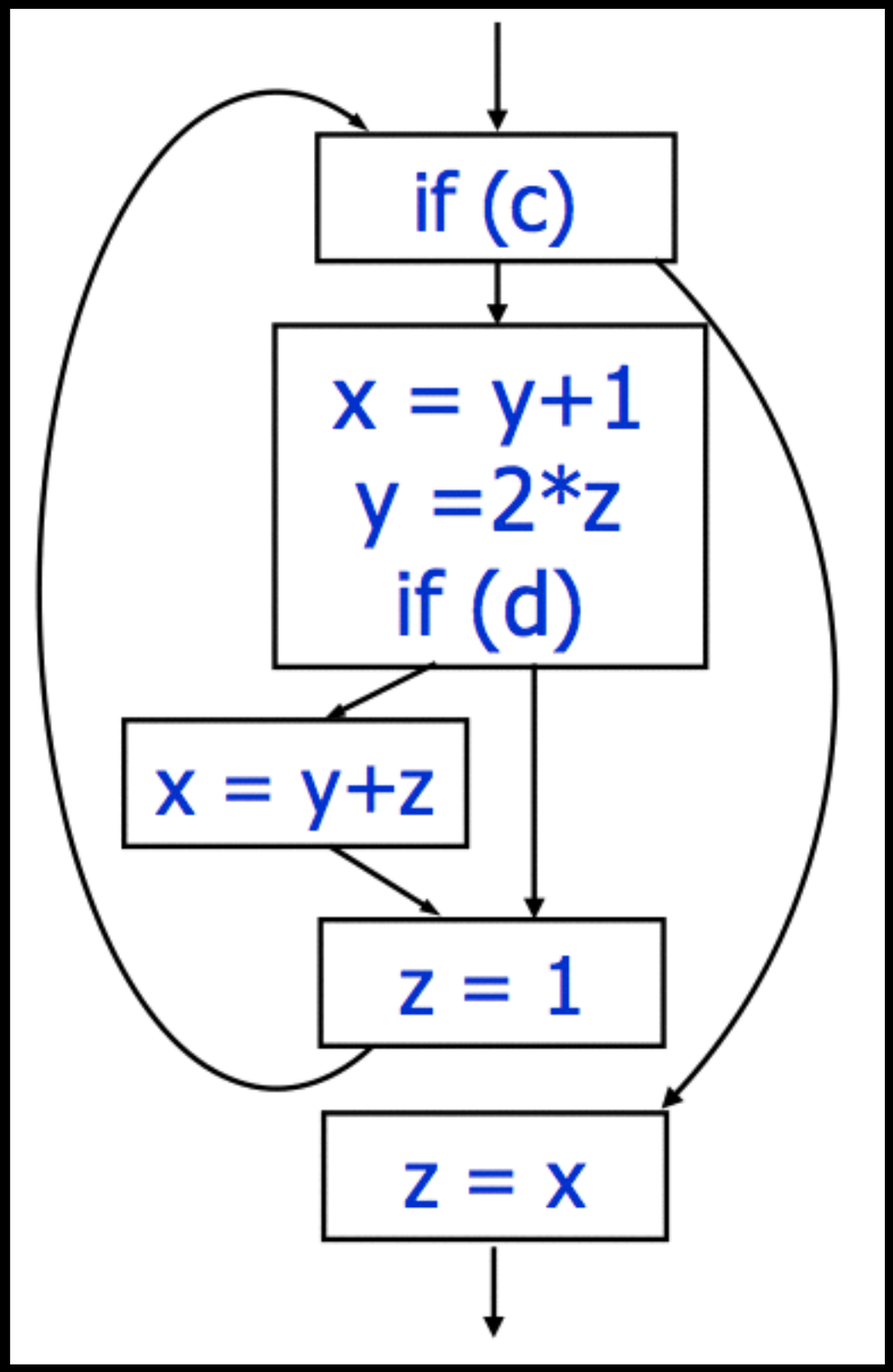
- Análise de programas, para realizar otimizações
- Instruction scheduling
- Alocação global de registradores

Construção eficiente

- Blocos básicos em um CFG
  - o mínimo de blocos possível, de maior tamanho possível
  - Não devem ocorrer pares de blocos (B_1, B_2) tal que
    - B_2 é sucessor de B_1
    - B_1 tem uma aresta outgoing
    - B_2 tem uma aresta incoming
- Não devem ocorrer blocos básicos vazios

Exemplo

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



CFGs para TAC

```
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3

z = 1;
jump L1
label L2
z = x;
```

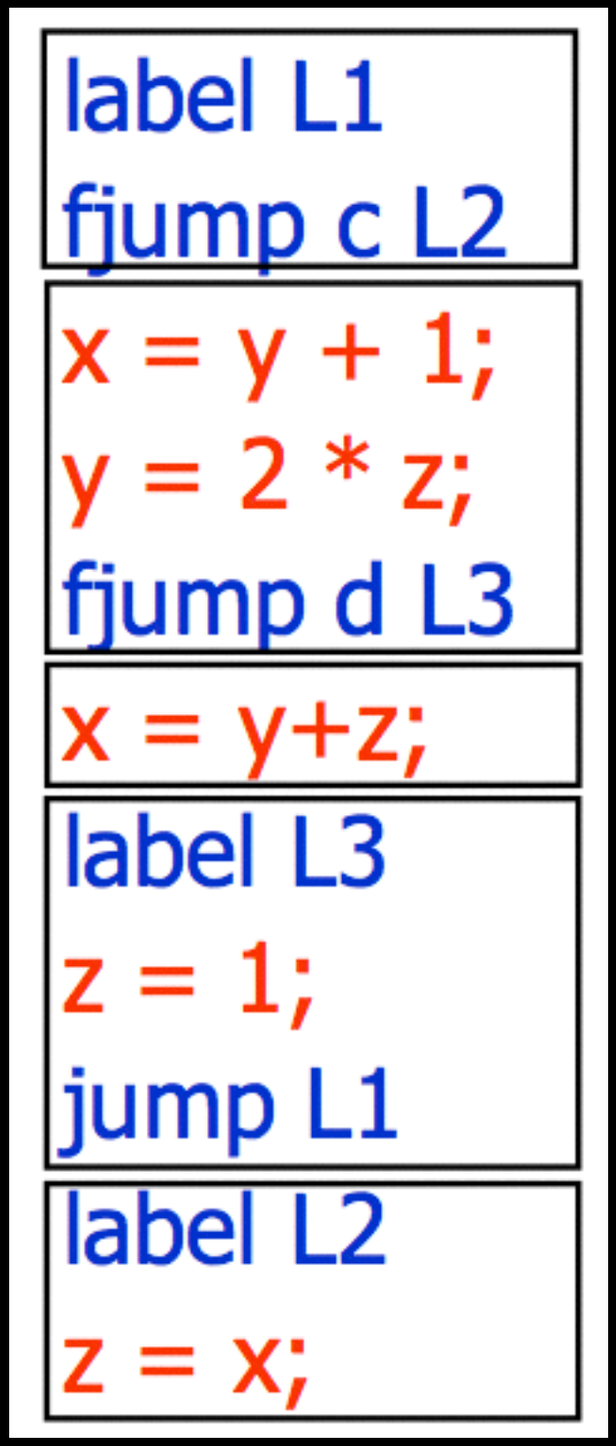
CFGs para TAC

- Identificar sequências de:
  - instruções sem desvio
  - instruções sem label
- Instruções sem desvio: controle não sai no meio do bloco básico
- Instruções sem label: controle não flui no meio do bloco básico

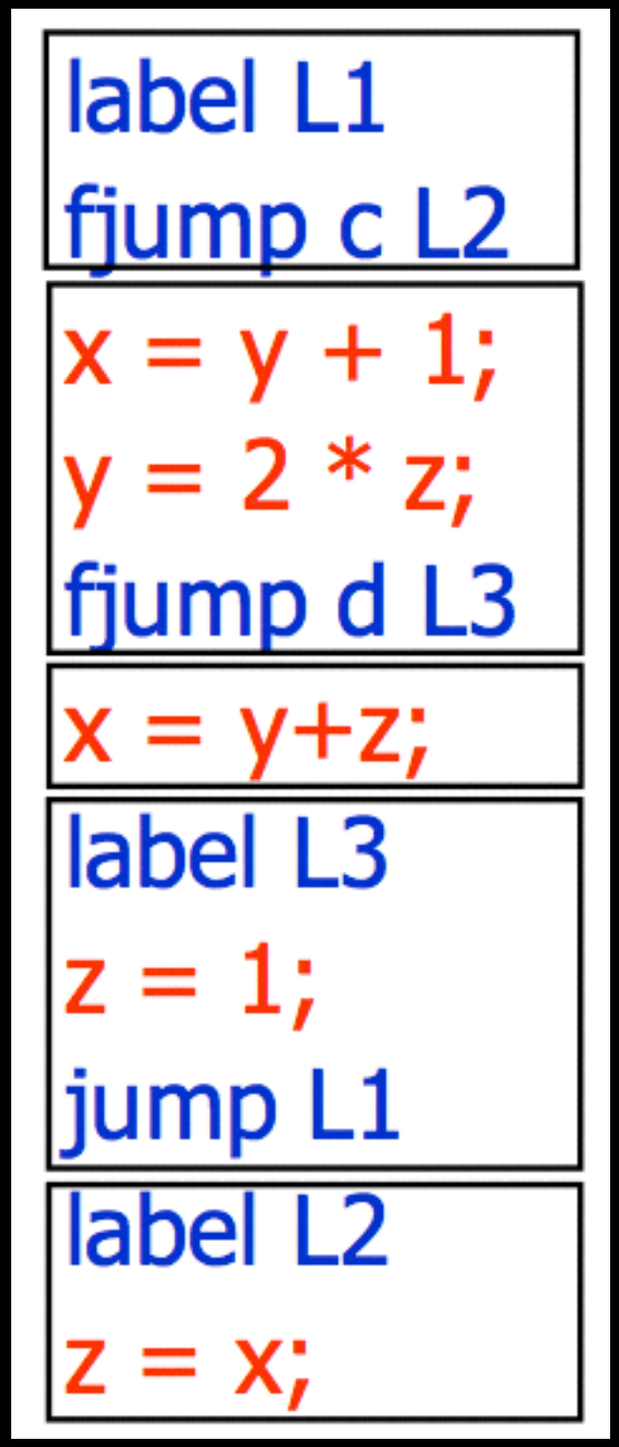
```
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3

z = 1;
jump L1
label L2
z = x;
```

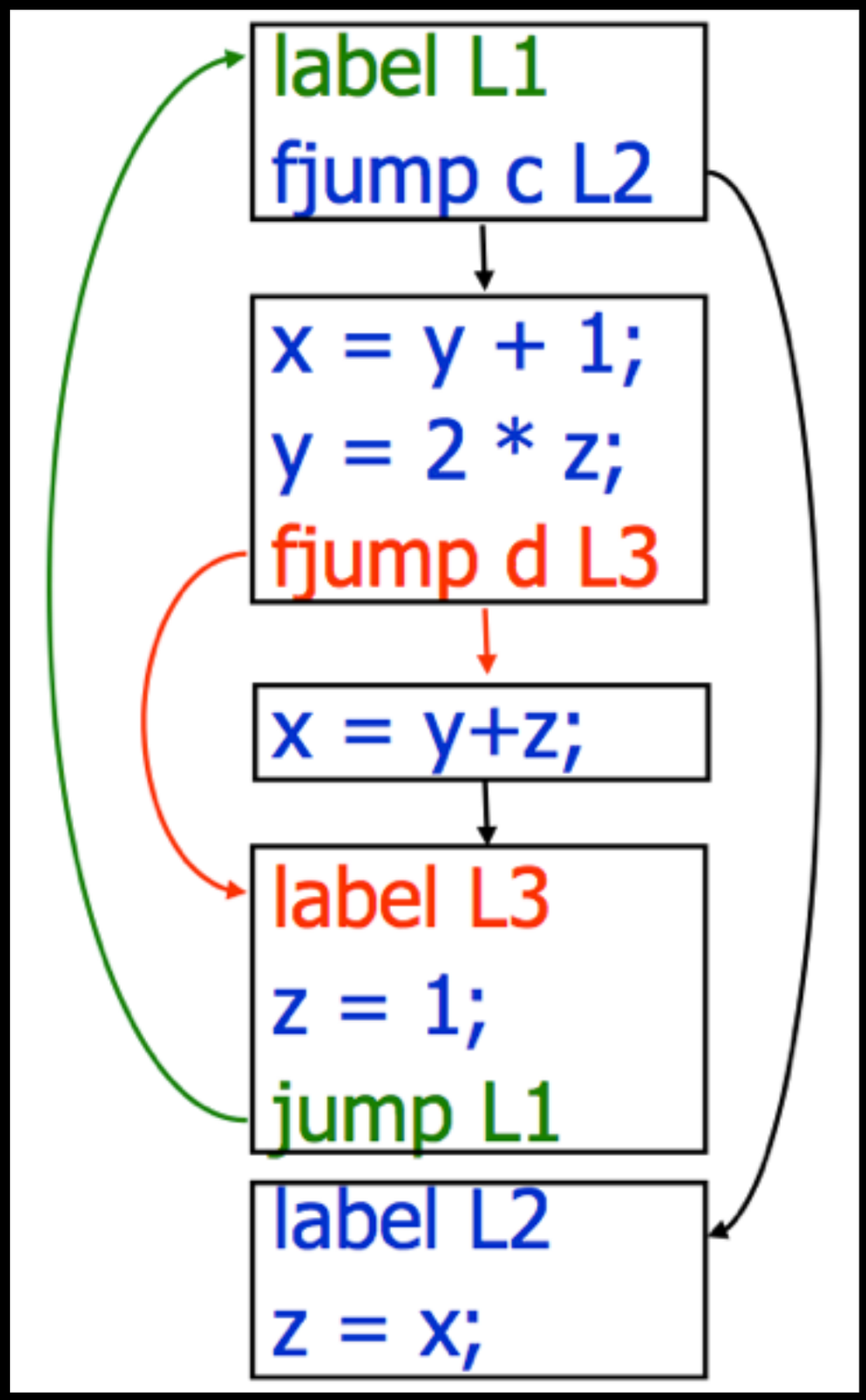
CFGs para TAC

- Blocos básicos iniciam em:
  - instruções com label
  - após instruções de desvio
- Blocos básicos terminam em:
  - instruções de desvio
  - antes de instruções com label

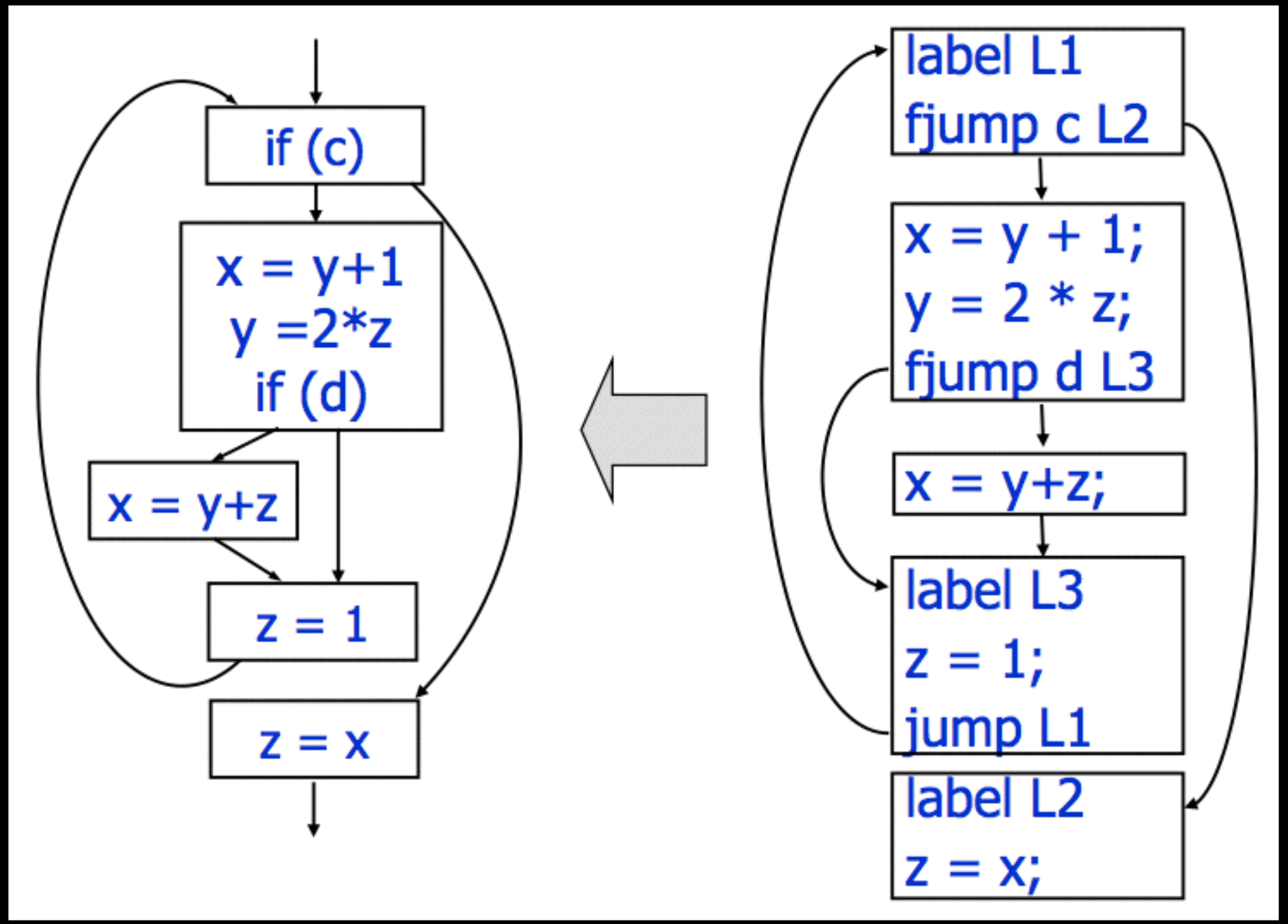


CFGs para TAC



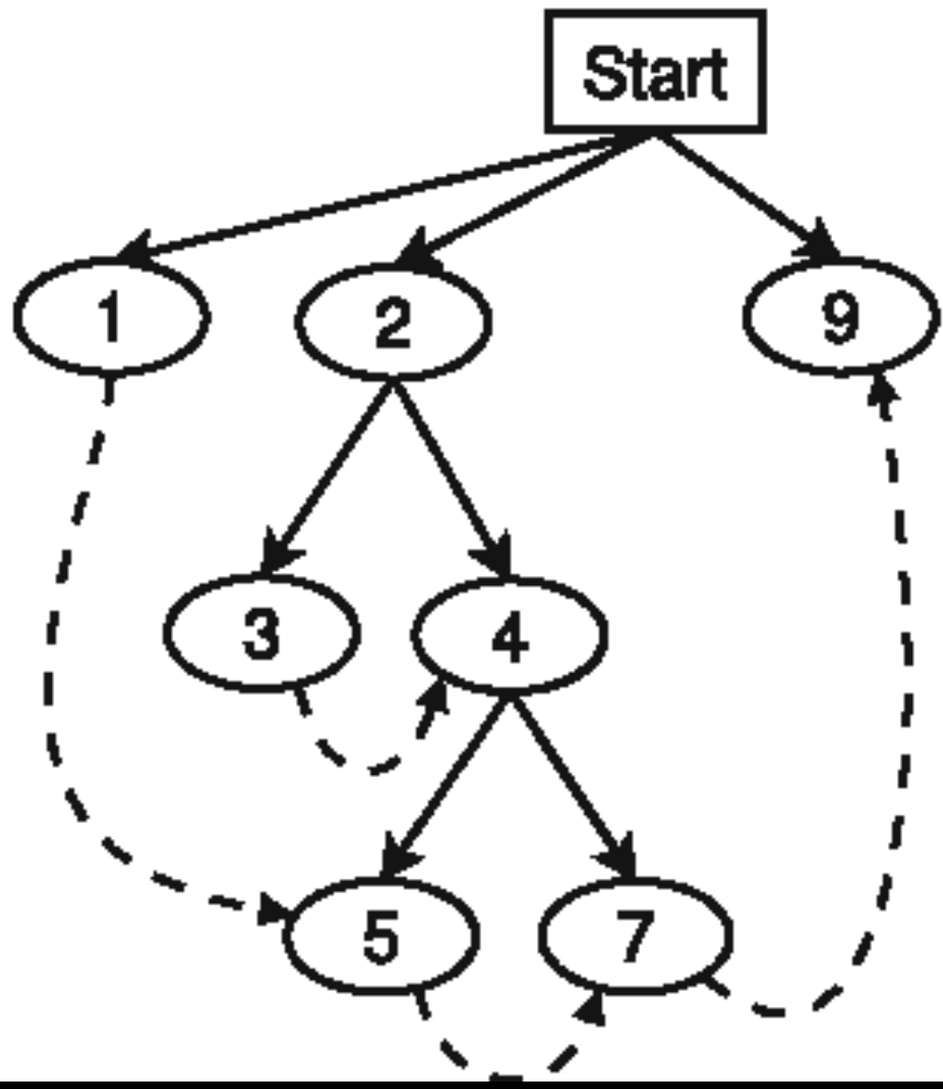


Relação

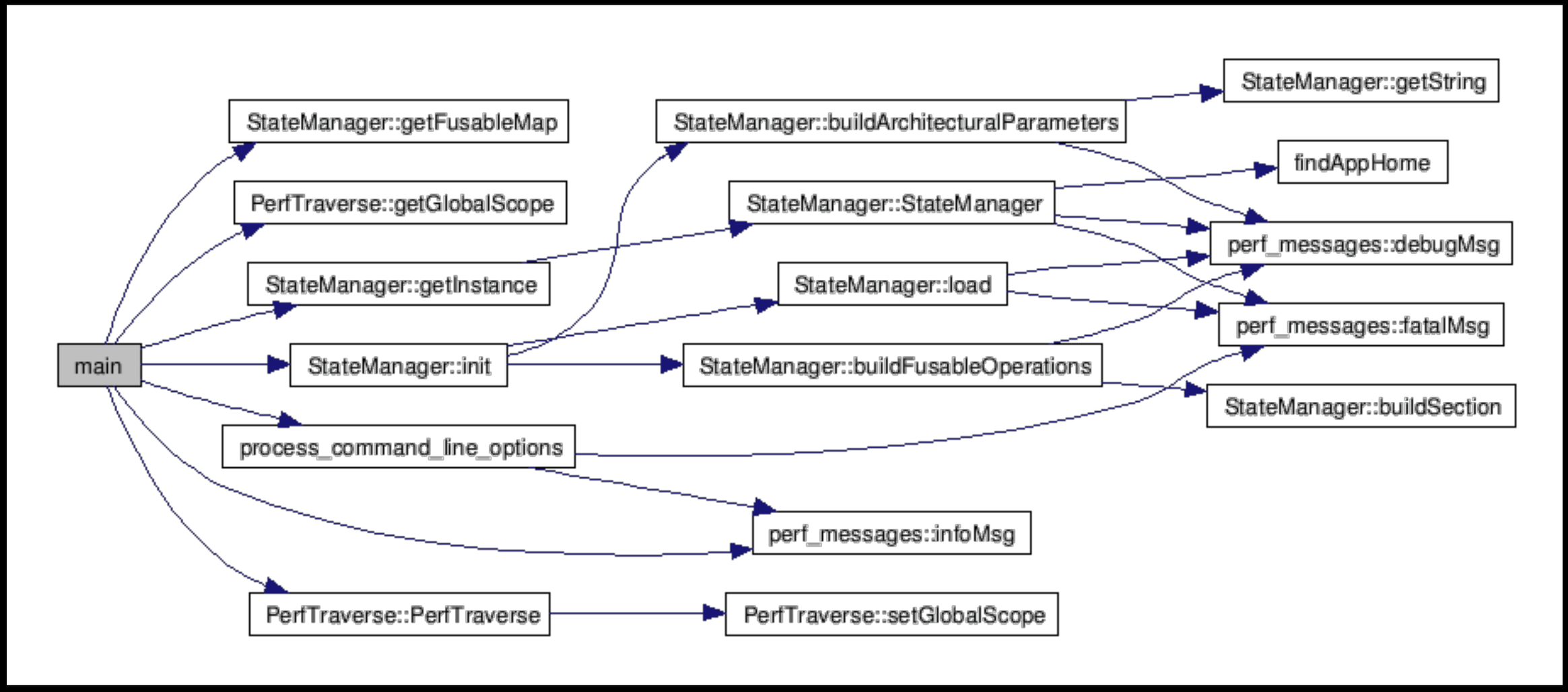

if (c)
x = y+1
y =2*z
if (d)
x = y+z
z = 1
z = x
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3
z = 1;
jump L1
label L2
z = x;

Dependence Graph

```
1 a = u();
2 while (f()) {
3   x = v();
4   if (x>0)
5     b = a;
6   else
7     c = b;
8 }
9 z = c;
```



Call graph


StateManager::getFusableMap
PerfTraverse::getGlobalScope
StateManager::getInstance
main
StateManager::init
process_command_line_options
PerfTraverse::PerfTraverse
StateManager::buildArchitecturalParameters
StateManager::StateManager
StateManager::load
StateManager::buildFusableOperations
perf_messages::infoMsg
PerfTraverse::setGlobalScope
StateManager::getString
findAppHome
perf_messages::debugMsg
perf_messages::fatalMsg
StateManager::buildSection