Compiladores (CC3001)
Aula 14: Análise de *Liveness* e Alocação de Registos

Mário Florido

DCC/FCUP

2024

[dcc] FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Esta aula

Formulação do Problema

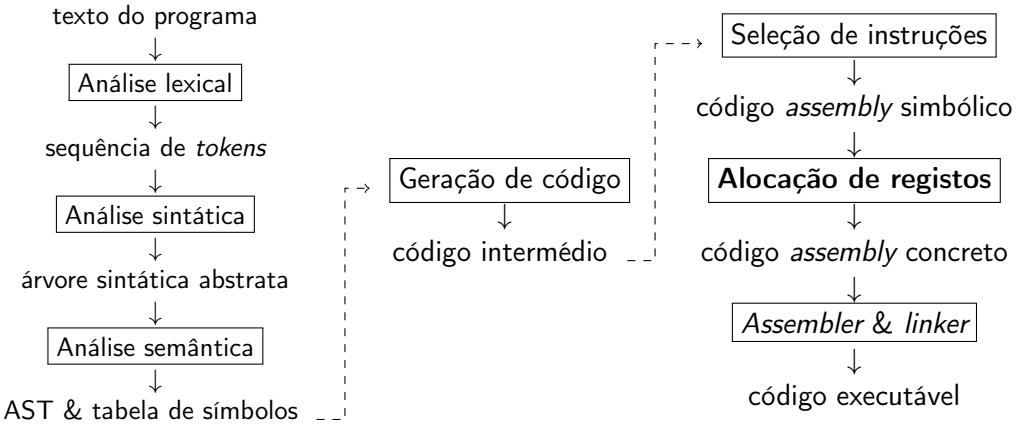
Análise de *Liveness*: Conceitos

Analise de *Liveness*: Equações e Aproximação

Alocação de Registos: Interferência e Algoritmo Básico

Alocação de Registos: Algoritmo com *Spilling*

Re-lembrar: fases dum compilador



Esta aula

Formulação do Problema

Análise de *Liveness*: Conceitos

Analise de *Liveness*: Equações e Aproximação

Alocação de Registos: Interferência e Algoritmo Básico

Alocação de Registos: Algoritmo com *Spilling*

O programa pode usar um número arbitrário de variáveis.

Os processadores têm um número limitado de registos:

X86	8 registos
ARM	16 registos
MIPS	31 registos

Solução:

- re-utilizar o mesmo registo para várias variáveis quando possível
- se não há registos suficientes: guardar variáveis em memória

Onde Fazer Alocação de Registos?

No código intermédio:

- \+ independente da arquitectura (parametrizado pelo número de registos disponível)
- − menor precisão: a tradução para código máquina pode remover/introduzir temporários

No código máquina simbólico:

- \+ alocação mais precisa (específica para um processador/arquitetura)
- − tem de ser re-implementada para cada arquitetura

Vamos ver alocação de registos ao nível de código intermédio.
As mesmas técnicas são aplicáveis a código máquina.

Âmbito da Alocação de Registos

- Ao nível de cada bloco de código sequencial:
  - \+ mais simples
  - – obriga a guardar variáveis em memória sempre que efetua saltos
- Ao nível de cada função/procedimento:
  - \+ mantém variáveis em registos durante saltos
  - – análise mais complexa
  - – obriga a guardar variáveis em memória antes de chamada de funções
- Módulo/programa completo:
  - \+ permite otimizar uso de registos entre funções
  - – análise bastante mais complexa

Vamos ver: alocação de registos ao nível da função/procedimento (mais comum).

Quando Partilhar Registos?

Exemplo: as variáveis a e c podem partilhar um registo?

```
a := 1
c := a + 1
a := c + 3
a := a + 2
```

Exemplo: as variáveis a e c podem partilhar um registo?

```
a := 1
c := a + 1
a := c + 3
a := a + 2
```

Intuição: Sim, porque as duas variáveis nunca são necessárias ao "mesmo tempo".

O programa seguinte calcula o mesmo resultado.

```
r := 1
r := r + 1
r := r + 3
r := r + 2
```

Esta aula

Formulação do Problema

Análise de *Liveness*: Conceitos

Analise de *Liveness*: Equações e Aproximação

Alocação de Registos: Interferência e Algoritmo Básico

Alocação de Registos: Algoritmo com *Spilling*

Intuição: duas variáveis podem partilhar um registo se não estão simultaneamente *vivas* em nenhum ponto do programa.

Liveness: uma variável está *viva* num ponto de programa se o seu valor nesse ponto será possivelmente usado no futuro.

Vamos representar o programa (corpo da função) como um grafo dirigido:

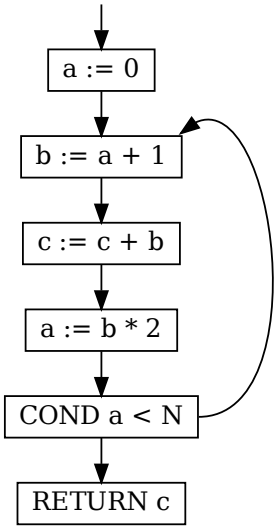
- cada instrução é um nó
- existe um arco $i \to j$ se a instrução j sucede à instrução i (i.e. j pode ser executada imediatamente depois de i).

Quase todas as instruções têm um único sucessor.

Exceção: saltos condicionais (dois sucessores) e a última instrução (nenhum sucessor).

Exemplo de Grafo de Controlo Fluxo

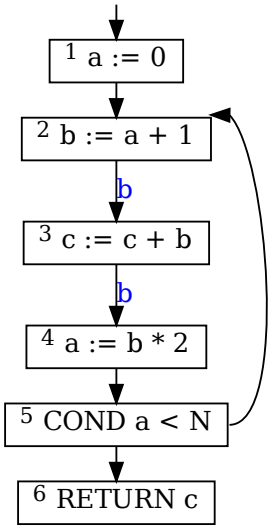
```
a := 0
LABEL L1
b := a + 1
c := c + b
a := b * 2
COND a<N L1 L2
LABEL L2
RETURN c
```



Podemos anotar os arcos do grafo com a *vida* de cada variável. Exemplo:

- a instrução 4 usa a variável b, logo b tem de estar *viva* em $3 \rightarrow 4$;
- a instrução 3 não modifica b, logo b tem de estar *viva* em $2 \rightarrow 3$;
- a instrução 2 atribui a b pelo que o valor b em $1 \rightarrow 2$ não é importante (b está *morta* em $1 \rightarrow 2$).

Logo: a variável b está *viva* em $2 \rightarrow 3 \rightarrow 4$.



1. Se o valor da variável x é usado na instrução i então x está viva à entrada de i
2. Se a instrução i atribui um valor à variável x (e a regra 1 não se aplica) então x está morta à entrada de i
3. Se uma variável está viva à saida de i (e a regra 2 não se aplica) então está viva à entrada de i
4. Uma variável está viva à saida de i *se e só se* está viva a entrada de alguma das instruções sucessoras de i

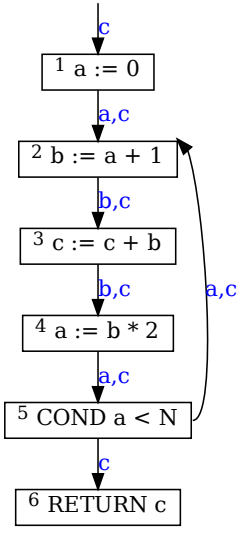
Podemos usar estas regras para *gerar* e *propagar* informação pelo grafo.

(Ver demonstração.)

Exemplo

```
1 a := 0
  LABEL L1
2 b := a + 1
3 c := c + b
4 a := b * 2
5 COND a<N L1 L2
  LABEL L2
6 RETURN c
```

- Em nenhum ponto as variáveis *a* e *b* estão simultaneamente vivas
- A variável *c* está viva à entrada: pode ser um *parâmetro da função* ou uma *variável local não inicializada* (erro)



As instruções são numeradas de 1 até n.

Para cada instrução i consideramos:

- $succ[i]$ O conjunto das instruções (números) que podem ser executados imediatamente após a instrução (número) i.
- $gen[i]$ O conjunto de variáveis cujos valores são usados na instrução i.
- $kill[i]$ O conjunto das variáveis que são escritas pela instrução i.
- $in[i]$ O conjunto das variáveis que estão vivas à entrada da instrução i.
- $out[i]$ O conjunto das variáveis que estão vivas à saida da instrução i.

Os conjuntos $succ[i]$, $gen[i]$ e $kill[i]$ são determinados pelo programa.

Objetivo: calcular $in[i]$ e $out[i]$ para i de 1 a n.

Conjunto dos Sucessores

- $succ[i] = \{i + 1\}$ excepto se a instrução i for `JUMP`, `COND`, `RETURN` ou a última instrução do programa.
- $succ[i] = \{j\}$ se a instrução i é `JUMP` l
  a instrução j é `LABEL` l.
- $succ[i] = \{j, k\}$ se a instrução i é `COND` c l_1 l_2
  as instruções j e k são `LABEL` l_1 e `LABEL` l_2.
- $succ[i] = \emptyset$ se a instrução i é `RETURN` ou a última instrução do programa (e não é `JUMP` ou `COND`).

Instrução i	$gen[i]$	$kill[i]$
`LABEL` l	$\emptyset$	$\emptyset$
$x := y$	$\{y\}$	$\{x\}$
$x := k$	$\emptyset$	$\{x\}$
$x := y$ binop z	$\{y, z\}$	$\{x\}$
$x := y$ binop k	$\{y\}$	$\{x\}$
$x := M[y]$	$\{y\}$	$\{x\}$
$M[x] :=$ y	$\{x, y\}$	$\emptyset$
`JUMP` l	$\emptyset$	$\emptyset$
`COND` x relop y l_t l_f	$\{x, y\}$	$\emptyset$
$x :=$ `CALL` $f(args)$	$args$	$\{x\}$
`RETURN` x	$\{x\}$	$\emptyset$

Exemplo Completo

```
1 a := 0
2 LABEL L1
3 b := a + 1
4 c := c + b
5 a := b * 2
6 COND a<N L1 L2
7 LABEL L2
8 RETURN c
```

i	$succ[i]$	$gen[i]$	$kill[i]$
1	$\{2\}$	$\emptyset$	$\{a\}$
2	$\{3\}$	$\emptyset$	$\emptyset$
3	$\{4\}$	$\{a\}$	$\{b\}$
4	$\{5\}$	$\{c, b\}$	$\{c\}$
5	$\{6\}$	$\{b\}$	$\{a\}$
6	$\{2, 7\}$	$\{a\}$	$\emptyset$
7	$\{8\}$	$\emptyset$	$\emptyset$
8	$\emptyset$	$\{c\}$	$\emptyset$

Esta aula

Vamos traduzir as regras para propagação de informação de *liveness* como equações relacionado os conjuntos *in*, *out* e *gen*, *kill* e *succ*.

Estas equações podem ser usadas para calcular automaticamente a informação de *liveness*.

Regra:

4. Uma variável está viva à saida de i *se e só se* está viva à entrada de alguma das instruções sucessoras de i.

Simbolicamente:

$$out[i] = \bigcup_{j \in succ[i]} in[j]$$

Regras:

1. Se o valor da variável x é usado na instrução i então x está viva à entrada de i
2. Se a instrução i atribui um valor à variável x (e a regra 1 não se aplica) então x está morta à entrada de i
3. Se uma variável está viva à saida de i (e a regra 2 não se aplica) então está viva à entrada de i

Simbolicamente:

$$in[i] = gen[i] \cup (out[i] \setminus kill[i])$$

$$in[i] = gen[i] \cup (out[i] \setminus kill[i]) \tag{1}$$

$$out[i] = \bigcup_{j \in succ[i]} in[j] \tag{2}$$

- Queremos determinar $in[i]$ e $out[i]$ (os restantes conjuntos são fixos pelo programa)
- As equações (1) e (2) são *recursivas*
- Podemos resolver de forma iterativa:
  1. inicializamos $in[i]$ e $out[i]$ com $\emptyset$ para todo i;
  2. usando (1) e (2) calculamos novos conjuntos $in[i]$, $out[i]$ para todo i;
  3. repetimos até estabilizar (i.e. nenhum dos conjuntos mudar).
- Este método converge independentemente da ordem das atualizações
- Mas converge mais depressa se atualizarmos por *ordem inversa* ($out[n], in[n], out[n-1], in[n-1], \ldots, out[1], in[1]$).

Exemplo: Conjuntos *Succ*, *Gen* e *Kill*

```
1 a := 0
2 LABEL L1
3 b := a + 1
4 c := c + b
5 a := b * 2
6 COND a<N L1 L2
7 LABEL L2
8 RETURN c
```

i	$succ[i]$	$gen[i]$	$kill[i]$
1	$\{2\}$	$\emptyset$	$\{a\}$
2	$\{3\}$	$\emptyset$	$\emptyset$
3	$\{4\}$	$\{a\}$	$\{b\}$
4	$\{5\}$	$\{c, b\}$	$\{c\}$
5	$\{6\}$	$\{b\}$	$\{a\}$
6	$\{2, 7\}$	$\{a\}$	$\emptyset$
7	$\{8\}$	$\emptyset$	$\emptyset$
8	$\emptyset$	$\{c\}$	$\emptyset$

Exemplo: Iterações

```
1 a := 0
2 LABEL L1
3 b := a + 1
4 c := c + b
5 a := b * 2
6 COND a<N L1 L2
7 LABEL L2
8 RETURN c
```

	Inicial		Iter 1		Iter 2		Iter 3	
i	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$
1			c	a, c	c	a, c	c	a, c
2			a, c	a, c	a, c	a, c	a, c	a, c
3			a, c	b, c	a, c	b, c	a, c	b, c
4			b, c	b, c	b, c	b, c	b, c	b, c
5			b, c	a, c	b, c	a, c	b, c	a, c
6			a, c	c	a, c	a, c	a, c	a, c
7			c	c	c	c	c	c
8			c		c		c	

Converge em 3 iterações.

Esta aula

Formulação do Problema

Análise de *Liveness*: Conceitos

Analise de *Liveness*: Equações e Aproximação

Alocação de Registos: Interferência e Algoritmo Básico

Alocação de Registos: Algoritmo com *Spilling*

A variável x interfere com a variável y se $x \neq y$ e existir uma instrução i tal que:

- $x \in kill[i]$,
- $y \in out[i]$,
- e a instrução i **não é** $x := y$.

Duas variáveis podem partilhar um registo se e só se não interferem uma com a outra.

x interfere com y **não é** exatamente equivalente a x e y estão vivas simultaneamente:

- mesmo que x, $y \in out[i]$ depois de $x := y$, podem partilhar um registo porque têm o mesmo valor;
- mesmo que $x \notin out[i]$ (i.e. x não está viva), se $x \in kill[i]$ não pode partilhar um registo com qualquer outra variável $y \in out[i]$.

Exemplo: *n*-ésimo Fibonnacci

```
fib(n):
1  a := 0
2  b := 1
3  z := 0
4  LABEL loop
5  COND n == z end body
6  LABEL body
7  t := a + b
8  a := b
9  b := t
10 n := n - 1
11 z := 0
12 JUMP loop
13 LABEL end
14 RETURN a
```

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2		a
2	3		b
3	4		z
4	5		
5	6, 13	n, z	
6	7		
7	8	a, b	t
8	9	b	a
9	10	t	b
10	11	n	n
11	12		z
12	4		
13	14		
14		a	

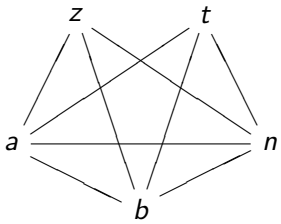
Iterações para Fibonnacci

i	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$
1			n	a, n	n	a, n	n	a, n
2			a, n	a, b, n	a, n	a, b, n	a, n	a, b, n
3			a, b, n	a, b, n, z	a, b, n	a, b, n, z	a, b, n	a, b, n, z
4			a, b, n, z	a, b, n, z	a, b, n, z	a, b, n, z	a, b, n, z	a, b, n, z
5			a, b, n, z	a, b, n	a, b, n, z	a, b, n	a, b, n, z	a, b, n
6			a, b, n	a, b, n	a, b, n	a, b, n	a, b, n	a, b, n
7			a, b, n	b, n, t	a, b, n	b, n, t	a, b, n	b, n, t
8			b, n, t	n, t	b, n, t	a, n, t	b, n, t	a, n, t
9			n, t	n	a, n, t	a, b, n	a, n, t	a, b, n
10			n		a, b, n	a, b, n	a, b, n	a, b, n
11					a, b, n	a, b, n, z	a, b, n	a, b, n, z
12					a, b, n, z	a, b, n, z	a, b, n, z	a, b, n, z
13			a	a	a	a	a	a
14			a		a		a	

Interferência no Exemplo Fibonacci

Podemos representar interferências como um grafo não-dirigido.

Instrução	Variável	Interfere com
1	a	n
2	b	n, a
3	z	n, a, b
7	t	b, n
8	a	t, n
9	b	n, a
10	n	a, b
11	z	n, a, b



Duas variáveis ligadas por um arco no grafo de interferências não podem partilhar um registo.

Queremos associar variáveis a números $1, 2, \ldots, K$ tal que:

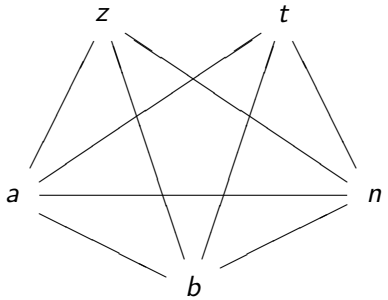
- duas variáveis ligadas por um arco têm números diferentes;
- $K \leq$ número de registos disponiveis.

Equivalente ao problema de K-coloração do grafo: colorir cada nó com uma de K cores tal que nós vizinhos têm cores distintas.

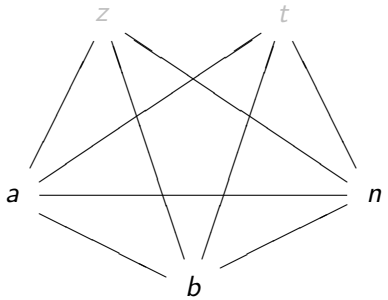
O problema de coloração de grafos é *NP-completo*: não se conhece uma solução eficiente (polinomial) no caso geral.

Uma heurística que funciona bem em muitos casos:

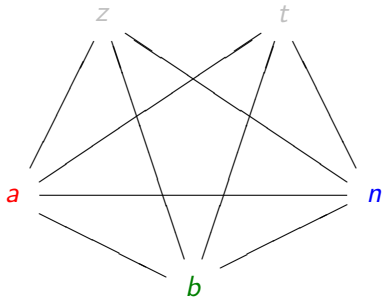
- começar por eliminar nós com menos de K vizinhos
  (colorirmos esses nós depois de escolher cores para os restantes);
- recursivamente resolver para os restantes.



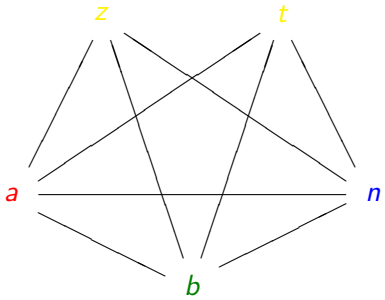
Os nós z e t têm apenas 3 vizinhos, portanto podemos deixá-los para o fim.



Podemos agora dar cores diferentes aos 3 nós sobrantes...



Agora damos a z e t cores distintas dos seus vizinhos.



Mas o que fazer se tivessemos apenas 3 cores (=registos)?

Esta aula

Formulação do Problema

Análise de *Liveness*: Conceitos

Analise de *Liveness*: Equações e Aproximação

Alocação de Registos: Interferência e Algoritmo Básico

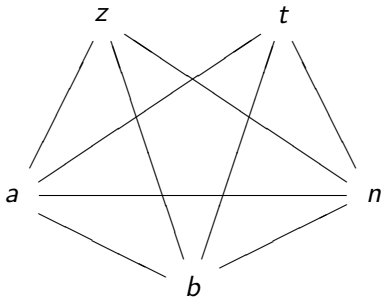
Alocação de Registos: Algoritmo com *Spilling*

Inicialização: Começar com uma pilha vazia.

Simplificação: 1) Se existe um nó com menos de K vizinhos:
(i) empilhar o nó juntamente com a lista de vizinhos;
(ii) remover o nó e os arcos incidentes.
2) Se não existe um nó com menos de K vizinhos: escolher um nó qualquer e fazer como acima.
3. Continuar até o grafo ficar vazio. Depois ir para *seleção*.

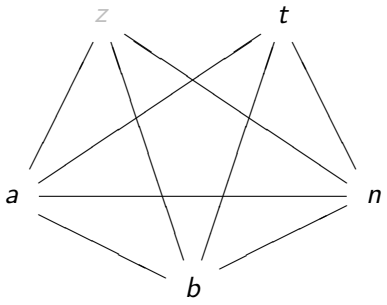
Seleção: 1) Tirar um nó e lista de vizinhos da pilha.
2) *Se possível, atribuir-lhe uma cor diferentes dos vizinhos.*
3) Se não for possível, marcar o nó para *spilling* (falha).
4) Repetir até a pilha estar vazia.

A qualidade depende da *(i) escolha dos nós na simplificação*, e da *(ii) escolha de cores na seleção*.



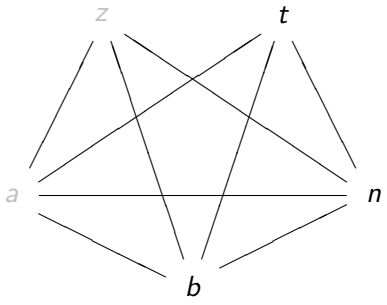
Nó	Vizinhos	Cor
z	a, b, n	

Nenhum nó tem < 3 vizinhos; escolhemos, por exemplo, z.



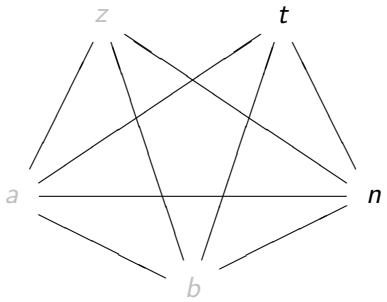
Nó	Vizinhos	Cor
a	b, n, t	
z	a, b, n	

Ainda nenhum nó tem < 3 vizinhos; escolhemos a.



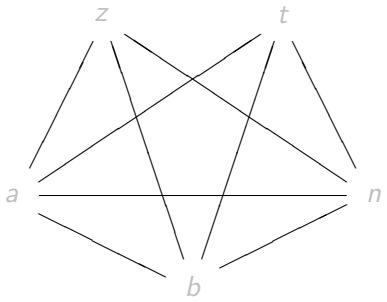
Nó	Vizinhos	Cor
b	t, n	
a	b, n, t	
z	a, b, n	

Escolhemos b porque tem 2 vizinhos.



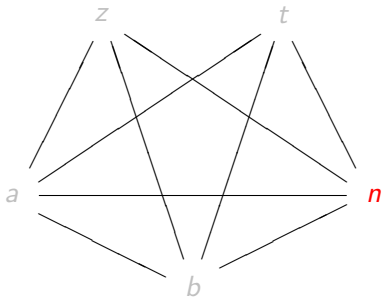
Nó	Vizinhos	Cor
n		
t	n	
b	t, n	
a	b, n, t	
z	a, b, n	

Por fim, escolhemos t e n.



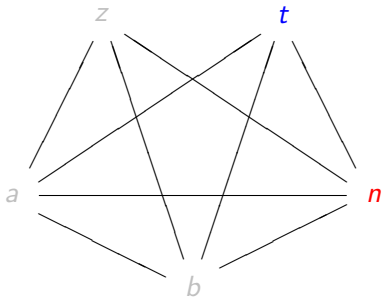
Nó	Vizinhos	Cor
n		1
t	n	
b	t, n	
a	b, n, t	
z	a, b, n	

n não tem vizinhos por isso escolhemos a cor 1.



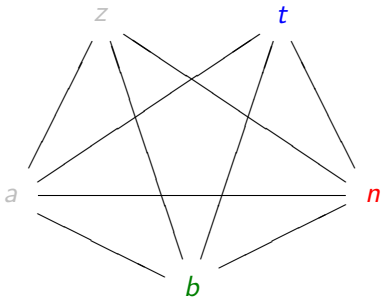
Nó	Vizinhos	Cor
n		1
t	n	2
b	t, n	
a	b, n, t	
z	a, b, n	

t só tem n como vizinho, por isso podemo colorir com 2.



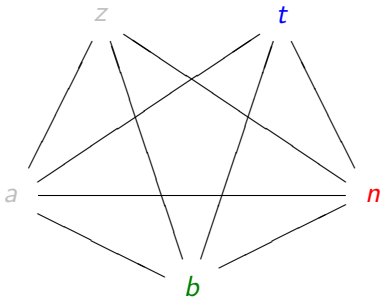
Nó	Vizinhos	Cor
n		1
t	n	2
b	t, n	3
a	b, n, t	
z	a, b, n	

b tem t e n como vizinhos, logo podemos colorir com 3.



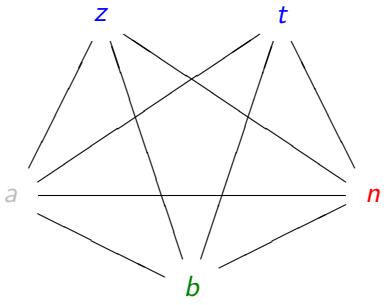
Nó	Vizinhos	Cor
n		1
t	n	2
b	t, n	3
a	b, n, t	*spill*
z	a, b, n	

a tem 3 vizinhos com cores diferentes, logo marcamos como *spill*.



Nó	Vizinhos	Cor
n		1
t	n	2
b	t, n	3
a	b, n, t	*spill*
z	a, b, n	2

z tem vizinhos com cores 1 e 3, logo podemos colorir com 2.



Nó	Vizinhos	Cor
n		1
t	n	2
b	t, n	3
a	b, n, t	*spill*
z	a, b, n	2

Terminamos, mas falta tratar o *spilling* da variável a.

Variáveis *spilled* vão residir na memória (excepto por periodos curtos).
Para cada variável x marcada *spill*:

1) Escolher um endereço na memória $addr_x$ para guardar o valor de x.
2) Em cada instrução i que usa x, localmente re-nomear x para x_i.
3) Antes de uma instrução i que lê x_i, inserir $x_i := M[addr_x]$.
4) Depois de uma instrução i que modifica x_i, inserir $M[addr_x] := x_i$.
5) Se x está viva no início da função/programa, inserir $M[addr_x] := x$ antes da primeira instrução.
6) Se x está viva no fim da função/programa, inserir $x := M[addr_x]$ depois da última instrução.

Finalmente: re-fazer análise de *liveness* e alocação de registos.

Exemplo: Fibonnaci após *Spilling*


```
fib(n):
1  a_1 := 0
   M[address_a] := a_1
2  b := 1
3  z := 0
4  LABEL loop
5  COND n == z end body
6  LABEL body
   a_7 := M[address_a]
7  t := a_7 + b
8  a_8 := b
   M[address_a] := a_8
9  b := t
10 n := n - 1
11 z := 0
12 JUMP loop
13 LABEL end
   a := M[address_a]
14 RETURN a
```

Conseguimos colorir o grafo com 3 cores após *spilling*: