Geração de Código Intermediário

Parte II – Traduzindo para IR

Traduzindo para Representação Intermediária

- Tradução para IR usando Esquemas de Tradução
  - Já vimos anteriormente como gerar uma AST

- Aqui analisaremos como gerar código de 3 endereços para:
  - Construção da tabela de símbolos
  - Comandos de atribuição/expressões
  - Verificação de tipos
  - Expressões Lógicas
  - Controle de Fluxo

Construção da tabela de símbolos

- Esquema para gerar as entradas na tabela de símbolos nas declarações de variáveis

- Exemplo para programas monolíticos

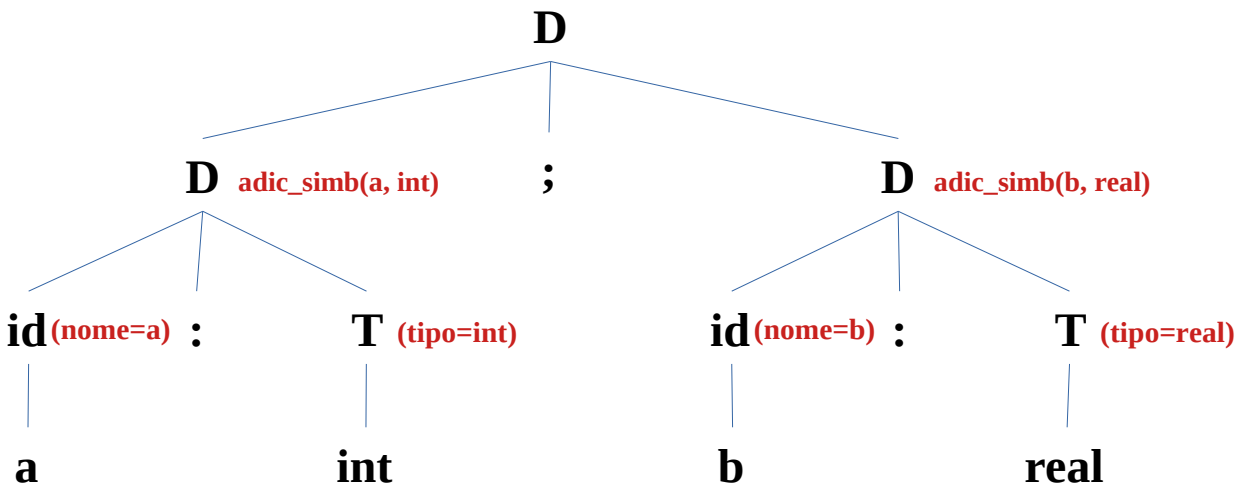
- $D \rightarrow D \; ; \; D$
- $D \rightarrow id : T$ {adic_simb(id.nome, T.tipo)}
- $T \rightarrow int$ {T.tipo=inteiro}
- $T \rightarrow real$ {T.tipo=float}
- $T \rightarrow array \; [num] \; of \; T_1$ {T.tipo=array(num.val, T_1.tipo)}

Construção da tabela de símbolos

- Exemplo:

  a:int; b:real

D → D ; D	
D → id : T	{adic_simb(id.nome, T.tipo)}
T → int	{T.tipo=inteiro}
T → real	{T.tipo=float}
T → array [num] of T_1	{T.tipo=array(num.val, T_1.tipo)}



Comandos de Atribuição/Expressões

- Geração de código para atribuições
- Funções:
  - *top.get*: retorna o índice da tabela de símbolos
  - *gen*: gera o código de 3 endereços

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\| \ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Comandos de Atribuição/Expressões

- Geração de código para atribuições
- Versão alternativa:
  - Gera e imprime diretamente o código

$$
\begin{array}{lll}
S \rightarrow \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\mid\ -\ E_1 & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);\ \} \\
\mid\ (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\
\mid\ \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \}
\end{array}
$$

Comandos de Atribuição/Expressões

- Exemplo:

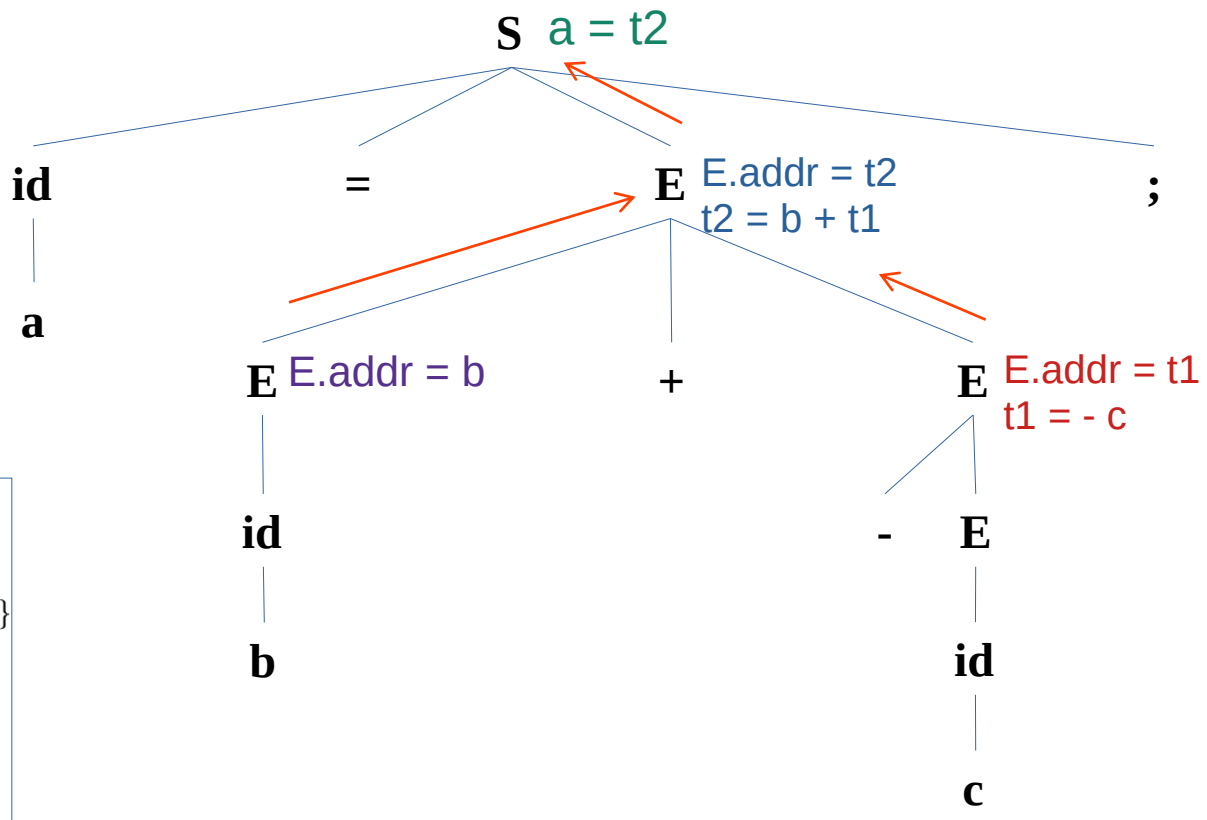
a = b + - c

t1 = minus c

t2 = b + t1

a = t2

S	$\rightarrow$	**id** = E ;	{ $gen($ $top.get($**id**$.lexeme)$ $'='$ $E.addr)$; }
E	$\rightarrow$	E_1 + E_2	{ $E.addr =$ **new** $Temp()$; $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$; }
	|	- E_1	{ $E.addr =$ **new** $Temp()$; $gen(E.addr\ '='\ '$**minus**$'\ E_1.addr)$; }
	|	(E_1)	{ $E.addr = E_1.addr$; }
	|	**id**	{ $E.addr = top.get($**id**$.lexeme)$; }



Comandos de Atribuição/Expressões

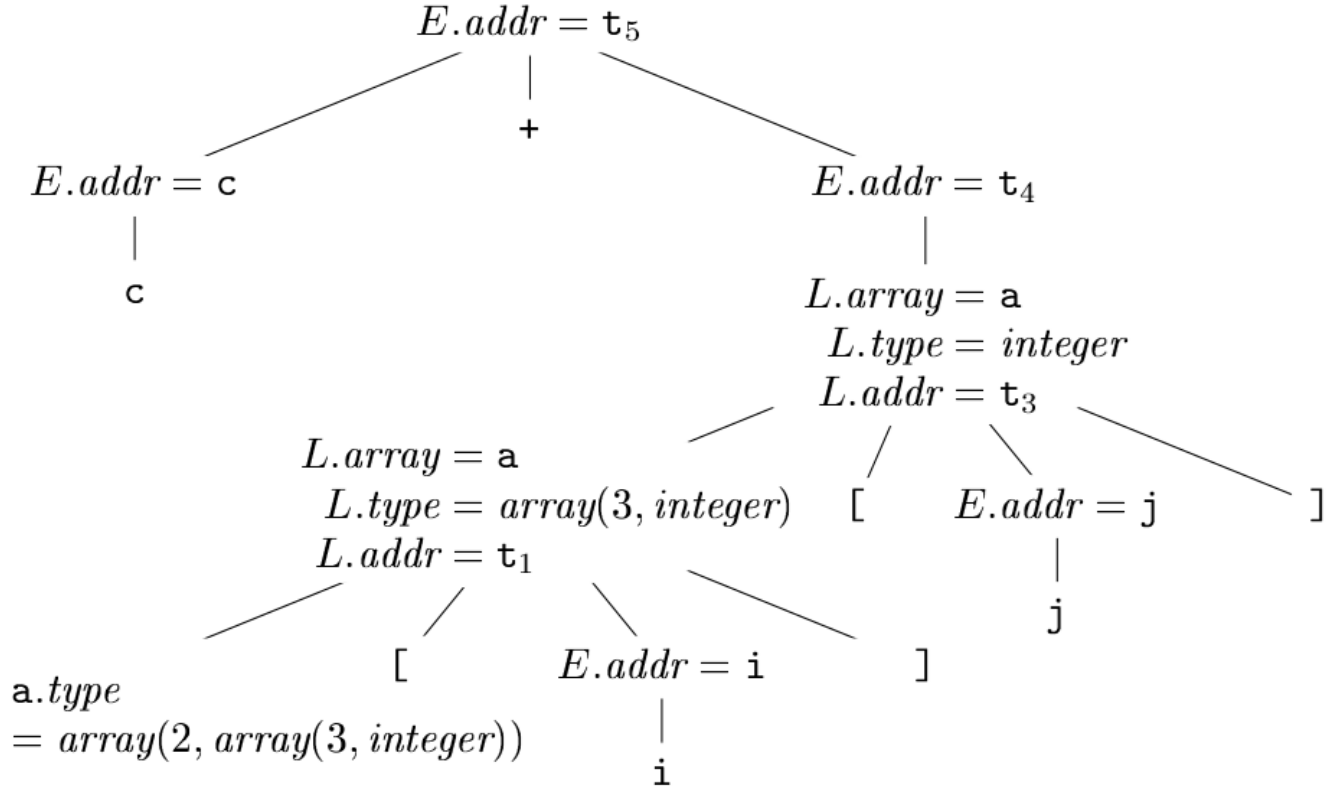
- Geração de código para atribuições
- **Extensão para arrays**
  - Cálculo dos endereços

$$
\begin{array}{lcll}
S & \rightarrow & \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\
 & | & L = E\ ; & \{\ gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
 & | & \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \} \\
 & | & L & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\
L & \rightarrow & \mathbf{id}\ [\ E\ ] & \{\ L.array = top.get(\mathbf{id}.lexeme);\ L.type = L.array.type.elem;\ L.addr = \mathbf{new}\ Temp\,();\ gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
 & | & L_1\ [\ E\ ] & \{\ L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \mathbf{new}\ Temp\,();\ L.addr = \mathbf{new}\ Temp\,();\ gen(t\ '='\ E.addr\ '*'\ L.type.width);\ gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

Comandos de Atribuição/Expressões

“c + a[i][j]”

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```



Verificação de tipos

- Relembrando:
  - Para fazer a verificação de tipo, um compilador precisa atribuir uma expressão de tipo a cada componente do programa de origem
  - O compilador deve então determinar que estes expressões de tipo estão em conformidade com uma coleção de regras lógicas que é chamada de sistemas de tipos
  - A verificação de tipo pode assumir duas formas: síntese e inferência

Verificação de tipos

- Síntese:
  - Constrói o tipo de uma expressão a partir dos tipos de suas subexpressões
  - Requer que os nomes sejam declarados antes de serem usados
  - O tipo de $E_1 + E_2$ é definido em termos dos tipos de E_1 e E_2
  - Uma regra típica para síntese de tipo tem a forma:

    if *f has type s→ t* ***and*** *x has type s,* ***then*** *expression f(x) has type t*

  - A regra pode ser adaptada para $E_1 + E_2$ enxergando isso como uma função *soma*(E_1,E_2)

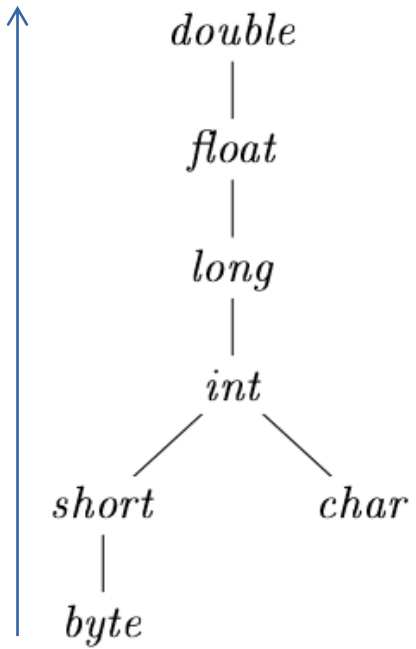
Verificação de tipos

- Inferência:
  - Determina o tipo de construção de uma linguagem a partir do modo como é usado
  - Depende da semântica da linguagem

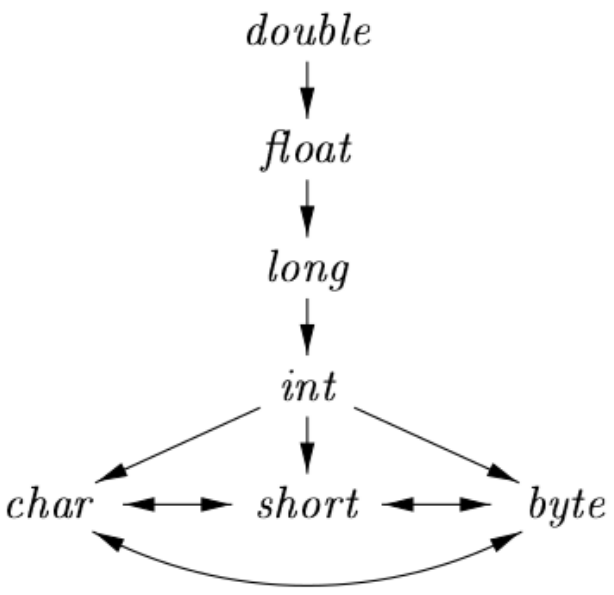
Verificação de tipos: conversões

- Código de 3-endereços
  - Conversão explícita de tipos:
    2 * 3.14 → t1 = **(float)** 2
    t2 = t1 * 3.14
- As regras de conversão de tipo variam para cada linguagem
  - Java:
    - Conversões de expansão (*widening*), que se destinam a preservar informações
    - Conversões de restrição (*narrowing*), que podem perder informações

Verificação de tipos: conversões



(a) Widening conversions



(b) Narrowing conversions

Verificação de tipos: conversões

- Ação semântica de conversão de tipos para $E \rightarrow E_1 + E_2$ utiliza duas funções:

max(t1,t2): retorna o máximo dos dois tipos (t1, t2) na hierarquia de expansão de tipos

- Erro se t1 ou t2 não estiver na hierarquia

widen(a,t,w) gera conversões de tipo, se necessário, para ampliar o conteúdo de um endereço '*a*' do tipo '*t*' em um valor do tipo '*w*'

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Verificação de tipos: conversões

- Ação semântica de expressões é estendida com as regras para conversão de tipos

$$
\begin{aligned}
E \rightarrow E_1 + E_2 \quad \{\ & E.type = max(E_1.type, E_2.type); \\
& a_1 = widen(E_1.addr, E_1.type, E.type); \\
& a_2 = widen(E_2.addr, E_2.type, E.type); \\
& E.addr = \textbf{new}\ Temp\,(); \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{aligned}
$$

2 * 3.14 → t1 = **(float)** 2
t2 = t1 * 3.14

Expressões lógicas

- Expressões lógicas podem ser avaliadas em analogia com expressões aritméticas usando instruções de três endereços com operadores lógicos
  - Compostas de operadores lógicos aplicados a elementos que são variáveis booleanas ou expressões relacionais

- Vamos considerar expressões booleanas geradas pela seguinte gramática:

```
B →   B || B |
      B && B |
      !B |
      ( B ) |
      E rel E |
      true |
      false
```

Expressões lógicas

- Dois métodos de tradução
  - *Representação Numérica:*
    - Tratamento como expressões aritméticas
    - Falso é 0, Verdadeiro é diferente de 0
  - *Fluxo de controle*
    - Tratamento por desvios
    - Não calcula os valores explicitamente

Expressões lógicas: Representação Numérica

- Neste método, as operações lógicas são avaliadas numericamente:
  1 → true
  0 → false

- Exemplos:

  x = a || b && !c

  é convertido para:

```
t1 = not c
t2 = b and t1
t3 = a or t2
x = t3
```

  a < b

  é convertido para:

```
if a<b goto L1
t1=0
goto L2
L1:          t1=1
L2:          ...
```

Expressões lógicas: Representação Numérica

- Esquema de tradução:

*Faltam regras de precedência e associatividade

B → B || B *{B.addr=new Temp(); gen(B.addr"="B_1.addr "or" B_2.addr)}*

B → B && B *{B.addr=new Temp(); gen(B.addr"="B_1.addr "and" B_2.addr)}*

B → !B *{B.addr=new Temp(); gen(B.addr"= not "B1.addr)}*

B → (B_1) *{B.addr=B_1.addr}*

B → id_1 rel id_2 *{B.addr=new Temp(); B.true=new Label(); B.next=new Label();*

gen("if" id_1.addr rel.op id_2.addr "goto" B.true);

gen(B.addr"=0"); gen("goto" B.next);

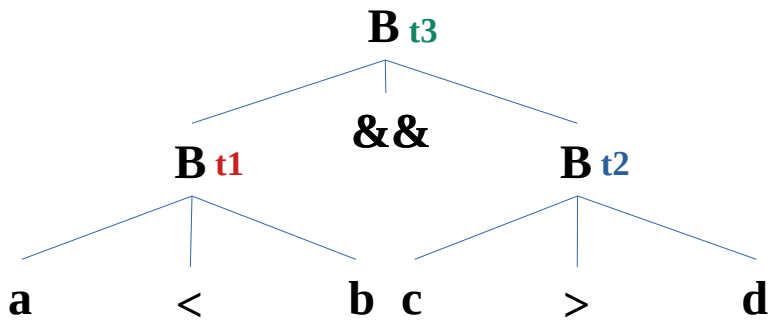
gen(B.true":" B.addr"=1"); gen(B.next":")}

B → true *{B.addr=new Temp(); gen(B.addr"=1")}*

B → false *{B.addr=new Temp(); gen(B.addr"=0")}*

Expressões lógicas: Representação Numérica

- Exemplo:
- “a < b && c > d”
- é transformado no seguinte código de 3 endereços:




```
if a < b goto L1
t1=0
goto L2
L1:t1=1
L2:
        if c > d goto L3
   t2=0
   goto L4
L3:t2=1
L4:
        t3=t1 and t2
```

Controle de Fluxo

- Tradução de comandos de controle de fluxo:
  - if-then
  - if-then-else
  - while-do

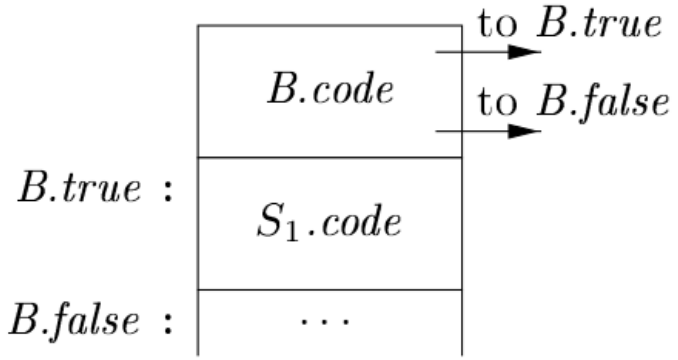
- Gramática

  $S \rightarrow$ if (B) S_1

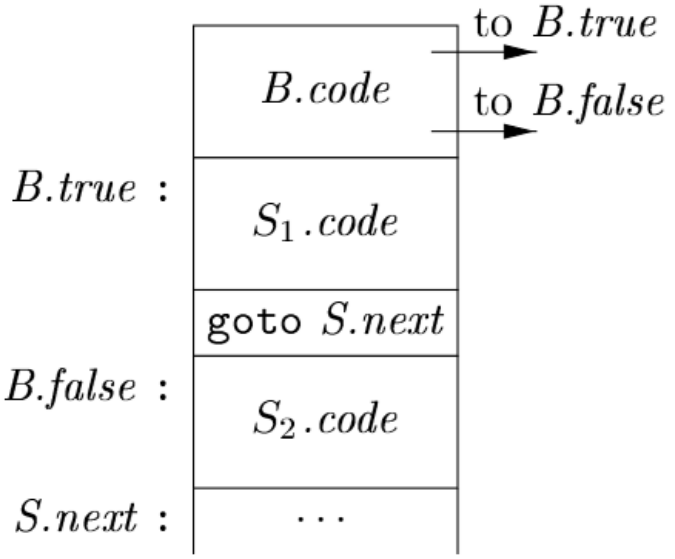
  $S \rightarrow$ if (B) S_1 else S_2

  $S \rightarrow$ while (B) S_1

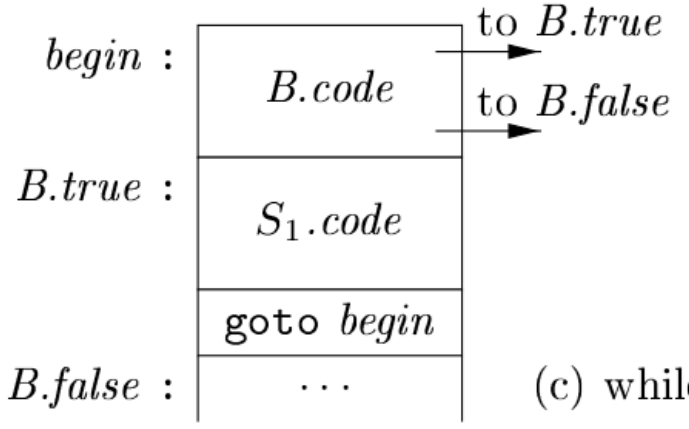
Controle de Fluxo: ideia geral



(a) if



(b) if-else



(c) while

Controle de Fluxo:
Gramática de atributos

- Tradução de comandos de controle de fluxo:
  - if-then
  - if-then-else
  - while-do

- Código gerado é armazenado no atributo .code
- Operações lógicas ‘B’ tem código gerado à parte (produções específicas)

- ‘||’ → concatenação

- **Esquema original apresentado no livro texto (Dragão) não é nem S-atribuído e nem L-Atribuído**

Controle de Fluxo:

Gramática de atributos adaptada do material do Prof. Lucas Schnorr (UFRGS): http://www.inf.ufrgs.br/~schnorr/inf1147/

Gramática de atributos – Esquema L-Atribuído (Análise Sintática Top-Down)

S → **if** *{B.t=newlabel(); B.f=S.next}*

(B) *{S1.next=S.next}*

S1 *{S.code=B.code||gen(B.t:)||S1.code||gen(S1.next:)}*

S → **if** *{B.t=newlabel(); B.f=newlabel()}*

(B) *{S1.next=S.next}*

S1

else *{S2.next=S.next}*

S2 *{S.code=B.code||gen(B.t:)||S1.code||gen(B.f:)||S2.code||gen(S2.next:)}*

S → **while** *{B.f=S.next; B.t=newlabel()}*

(B) *{S.begin=newlabel(); S1.next=S.begin}*

S1 *{S.code=gen(S.begin:)||B.code||gen(B.t:)||S1.code||gen(goto S.begin)||gen(B.f)}*

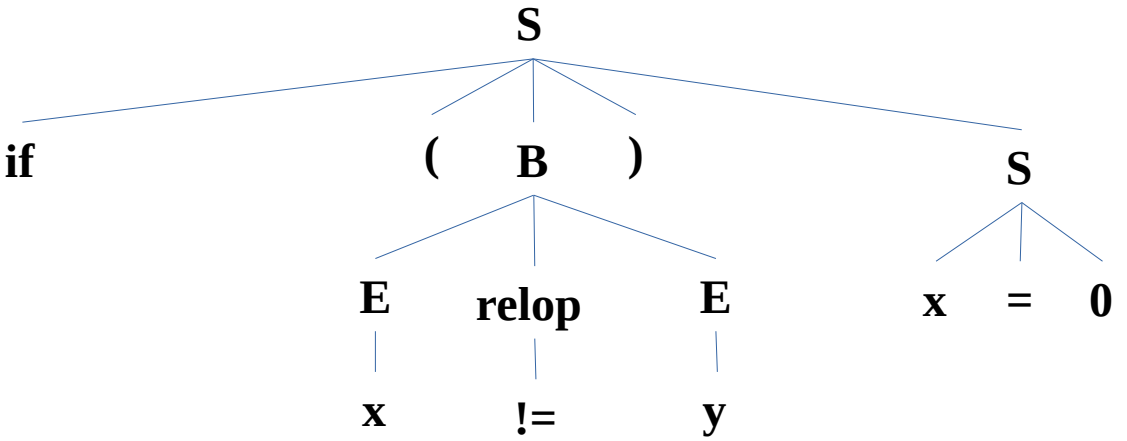
B → *{B1.t=newlabel();B1.f=B.f}* **B1 &&** *{B2.t=B.t;B2.f=B.f}* **B2** *{B.code=B1.code||label(B1.t)||B2.code}*

B → **E1 relop E2** *{B.code=E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto B.t) || gen(goto B.f)}*

Controle de Fluxo

.Exemplo:

.if(x != y) x = 0



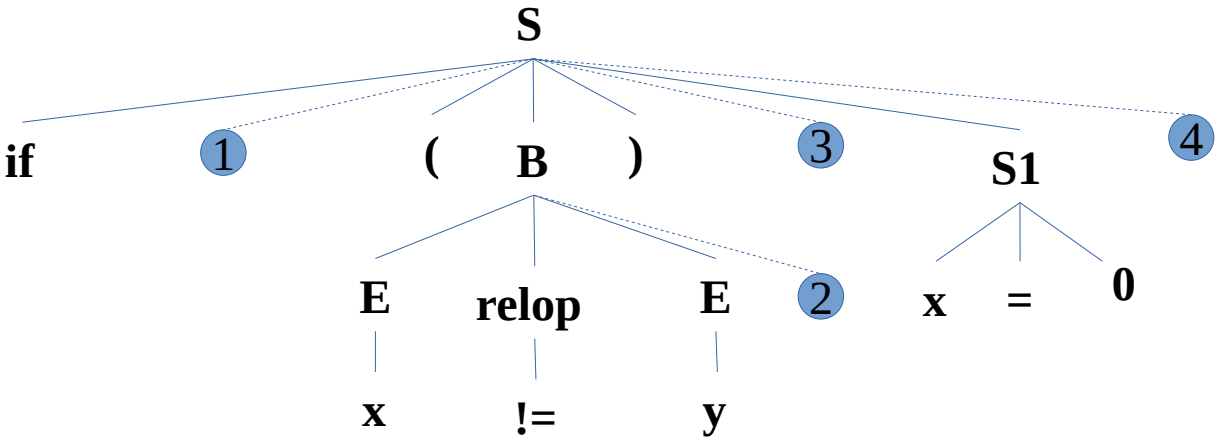
S → **if** *{B.t=newlabel(); B.f=S.next}*
(B) *{S1.next=S.next}*
S1 *{S.code=B.code||gen(B.t:)||S1.code||gen(S1.next:)}*

B → **E1 relop E2** *{B.code=E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto B.t) || gen(goto B.f)}*

Controle de Fluxo

.**Exemplo:**

.**if(x != y) x = 0**



S → **if** *{B.t=newlabel(); B.f=S.next}* 1
(B) *{S1.next=S.next}* 3
S1 *{S.code=B.code||gen(B.t:)||S1.code||gen(S1.next:)}* 4

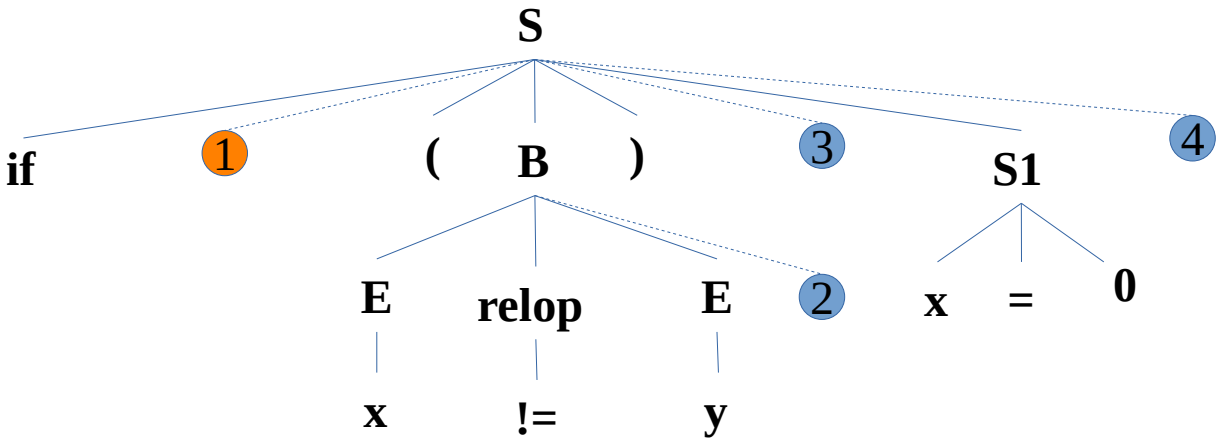
Pesquisa em profundidade!

B → **E1 relop E2** *{B.code=E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto B.t) || gen(goto B.f)}* 2

Controle de Fluxo

.Exemplo:

.if(x != y) x = 0



Execução de 1:

{B.t=newlabel(); B.f=S.next}

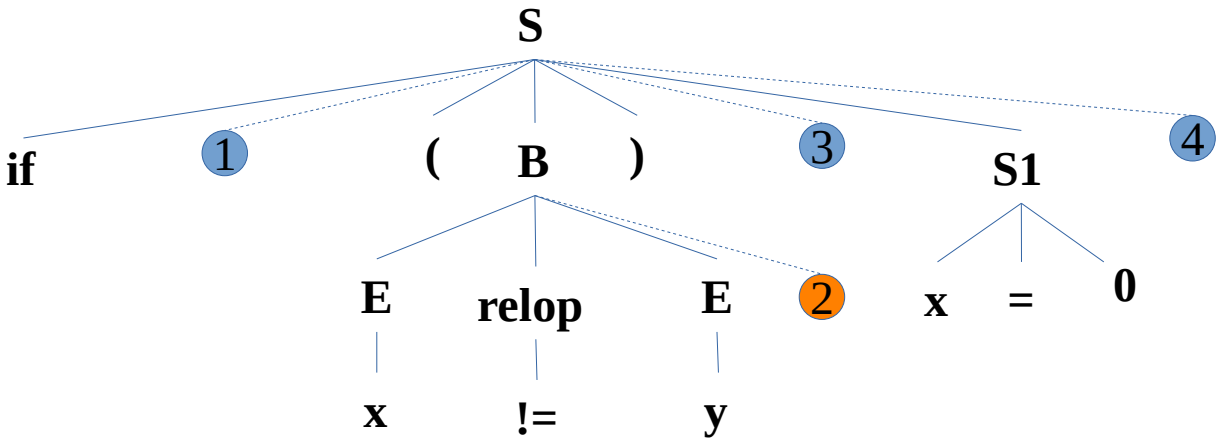
B.t=L1

B.f=Prox (Não sabemos o que está vindo de S)

Controle de Fluxo

.**Exemplo:**

.**if(x != y) x = 0**



Execução de 2:

{B.code=E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto B.t) || gen(goto B.f)}

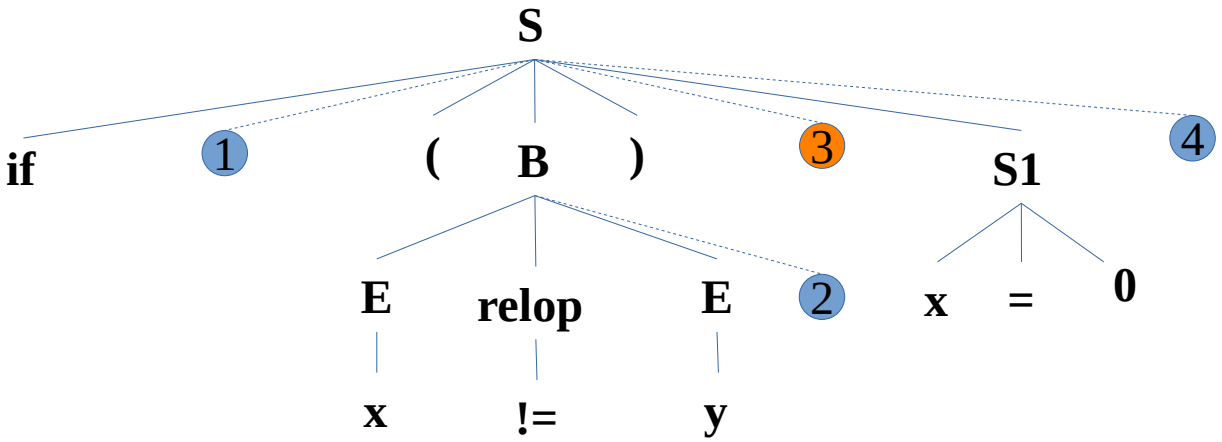
B.code = ‘if x != y goto L1

goto Prox’

Controle de Fluxo

.Exemplo:

.if(x != y) x = 0



Execução de 3:

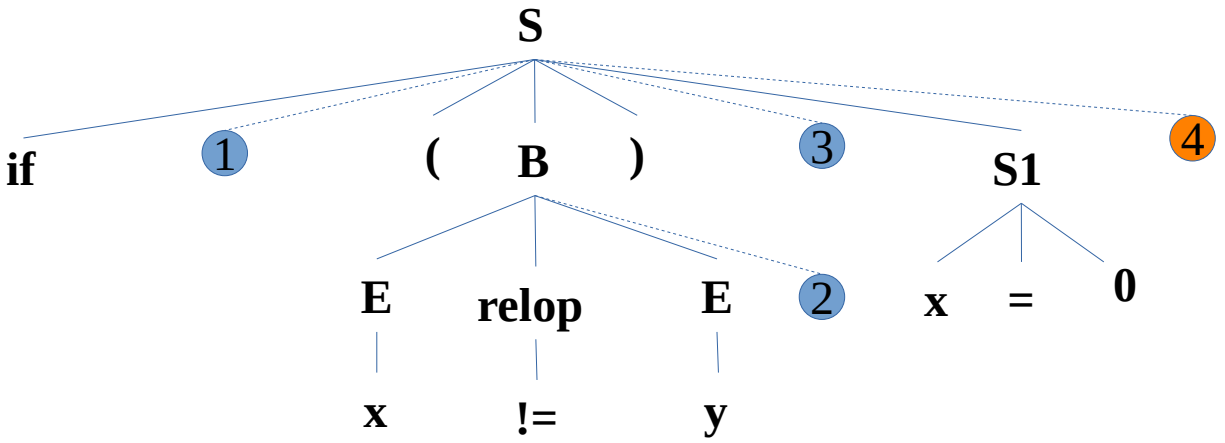
{S1.next=S.next}

S1.next = Prox

Controle de Fluxo

.**Exemplo:**

.**if(x != y) x = 0**



Execução de 4:

{S.code=B.code||gen(B.t:)||S1.code||gen(S1.prox:)}

S.code = ‘if x != y goto L1

goto Prox

L1: x = 0

Prox:’

Código de 3 endereços gerado:

if x != y goto L1

goto Prox

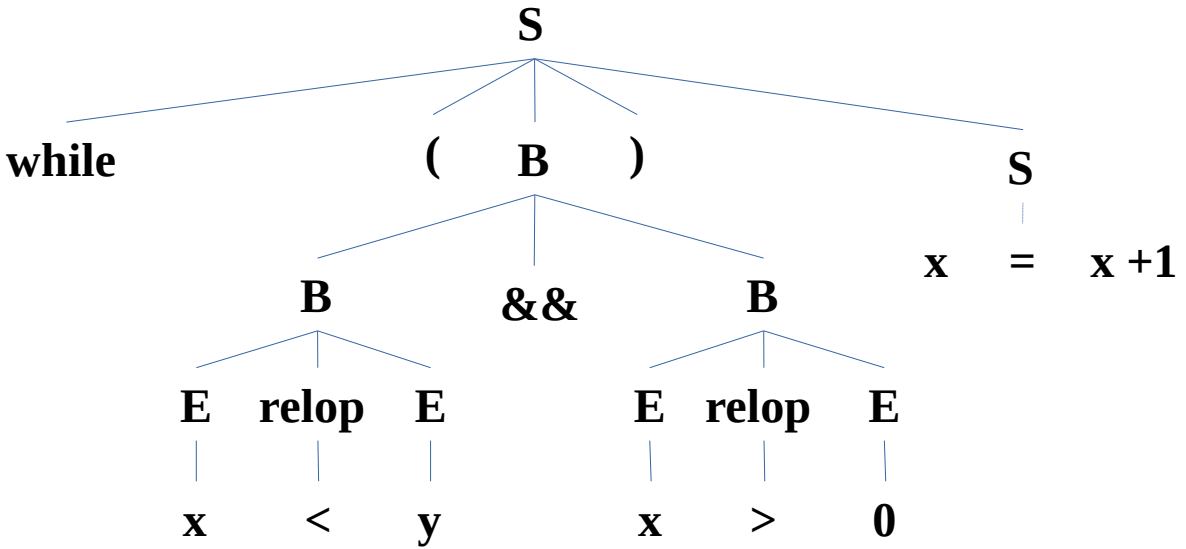
L1: x = 0

Prox:

Controle de Fluxo

.Exemplo 2:

.while(x<y && x<0) x=x + 1



S → **while** {B.f=S.next; B.t=newlabel()}

(B) {S.begin=newlabel(); S1.next=S.begin}

S1 {S.code=gen(S.begin:)||B.code||gen(B.t:)||S1.code||gen(goto S.begin)||gen(B.f)}

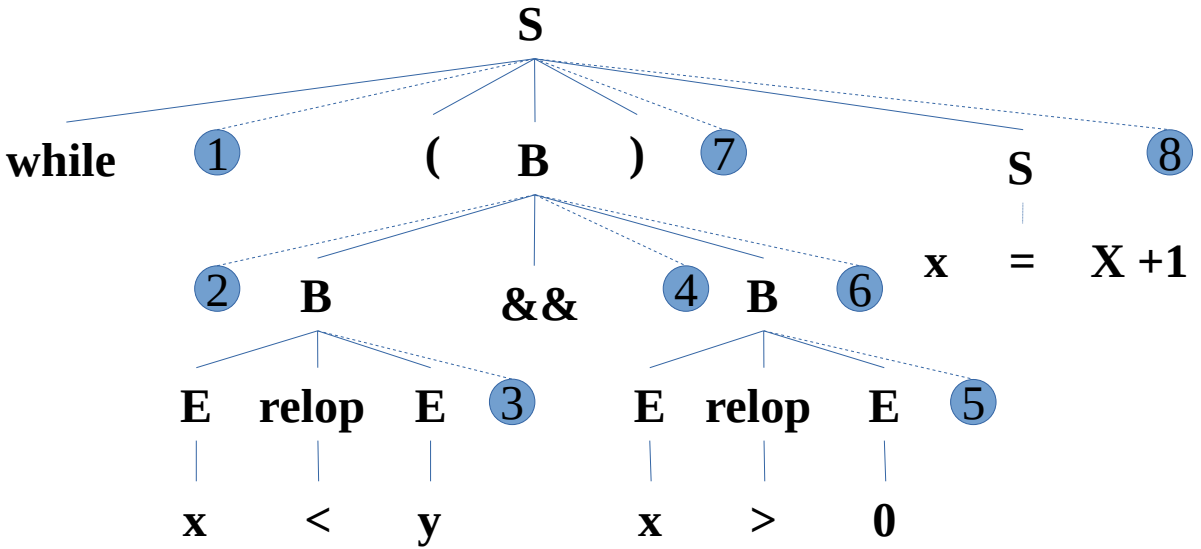
B → {B1.t=newlabel();B1.f=B.f} **B1 &&** {B2.t=B.t;B2.f=B.f} **B2** {B.code=B1.code||label(B1.t)||B2.code}

B → **E1 relop E2** {B.code=E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto B.t) || gen(goto B.f)}

Controle de Fluxo

.Exemplo 2:

.while(x<y && x<0) x=x + 1



S → **while** {B.f=S.next; B.t=newlabel()} 1

(B) {S.begin=newlabel(); S1.next=S.begin} 7

S1 {S.code=gen(S.begin:)||B.code||gen(B.t:)||S1.code||gen(goto S.begin)||gen(B.f)} 8

B → {B1.t=newlabel();B1.f=B.f} **B1 &&** {B2.t=B.t;B2.f=B.f} **B2** {B.code=B1.code||label(B1.t)||B2.code} 2 4 6

B → **E1 relop E2** {B.code=E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto B.t) || gen(goto B.f)} 3 5

Controle de Fluxo

S → **while** {B.f=S.next; B.t=newlabel()} 1

(B) {S.begin=newlabel(); S1.next=S.begin} 7

S1 {S.code=gen(S.begin:)||B.code||gen(B.t:)||S1.code||gen(goto S.begin)||gen(B.f)} 8

B → {B1.t=newlabel();B1.f=B.f} 2 **B1 &&** {B2.t=B.t;B2.f=B.f} 4 **B2** {B.code=B1.code||label(B1.t)||B2.code} 6

B → **E1 relop E2** {B.code=E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto B.t) || gen(goto B.f)} 3 5

Executando na ordem prevista, obtemos:

```
L3:     if x<y goto L2
        goto Prox:
L2:     if x>0 goto L1
        goto Prox
L1: x=x+1
        goto L3
Prox:
```

Controle de Fluxo:

Backpatching - Esquema S-Atribuído (Análise Sintática Bottom-Up)

- Necessário também adaptar o esquema para análise Bottom-Up
- Usa-se uma abordagem, chamada de *backpatching*, na qual listas de saltos são passadas como atributos sintetizados
  - Especificamente, quando um salto é gerado, o alvo do salto é temporariamente deixado não especificado
  - Cada salto é colocado em uma lista de saltos cujos rótulos devem ser preenchidos quando o devido o rótulo puder ser determinado
  - Todos os saltos em uma lista têm o mesmo rótulo de destino

Controle de Fluxo:

Backpatching - Esquema S-Atribuído (Análise Sintática Bottom-Up)

- Funções auxiliares
  - remendo()
  - concat(lista1, lista2)
  - remenda (lista, rotulo)
    - todos os remendos da lista serão resolvidos com o rótulo

- Atributos sintetizados:
  - B.tl (lista de remendos caso verdadeiro)
  - B.fl (lista de remendos caso falso)

Controle de Fluxo:

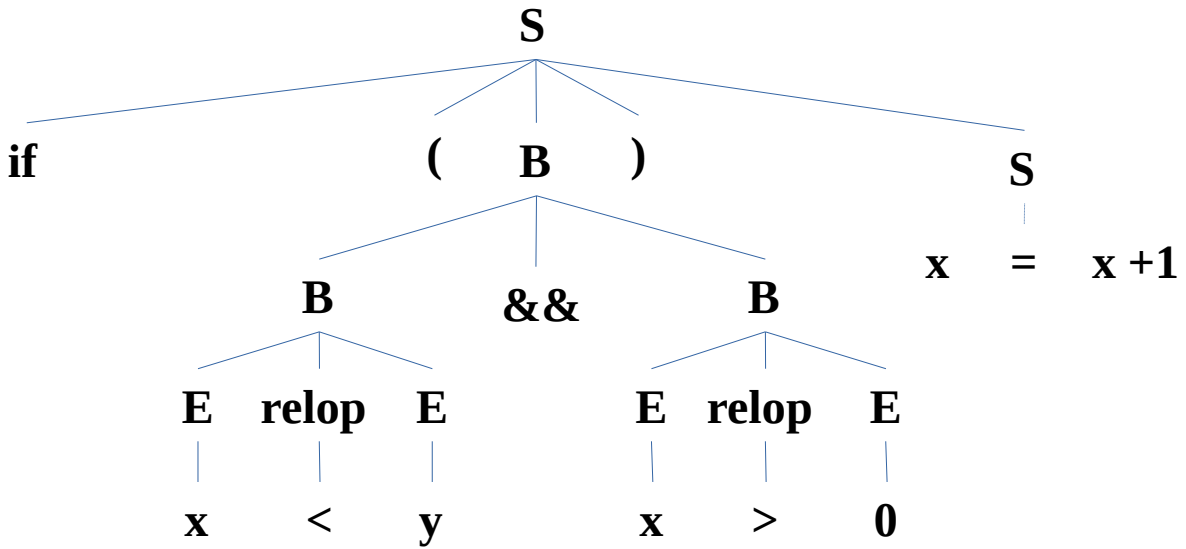
Backpatching - Esquema S-Atribuído (Análise Sintática Bottom-Up)

- Exemplo:

B → E1 relop E2	*x = remendo()* *y = remendo()* *B.code = E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto x)* || *gen(goto y)* *B.tl = lista(x)* *B.fl = lista(y)*
B → B1 && B2	*x = newlabel()* *remenda (B1.tl, x)* *B.tl = B2.tl* *B.fl = concat(B1.fl, B2.fl)* *B.code = B1.code* || *gen(x": ")* || *B2.code*
S → if (B) S1	x = newlabel y = newlabel remenda(B.tl, x) remenda(B.tf, y) S.next = y S.code = B.code || gen(B.t:) || S1.code || gen(S.next:)

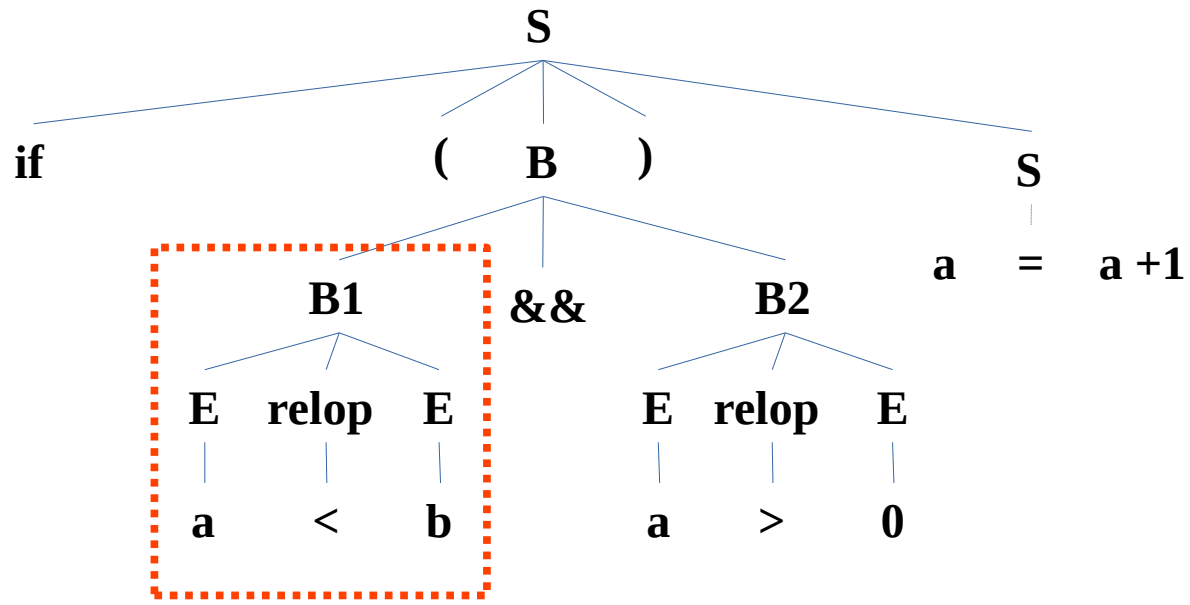
Exemplo:

if(a<b && a>0) a=a+1



Exemplo:

if(a<b && a>0) a=a+1



Código de B1:

if a<b goto x1
goto y1

Aqui, x e y são as labels que não estão

Estados das listas:

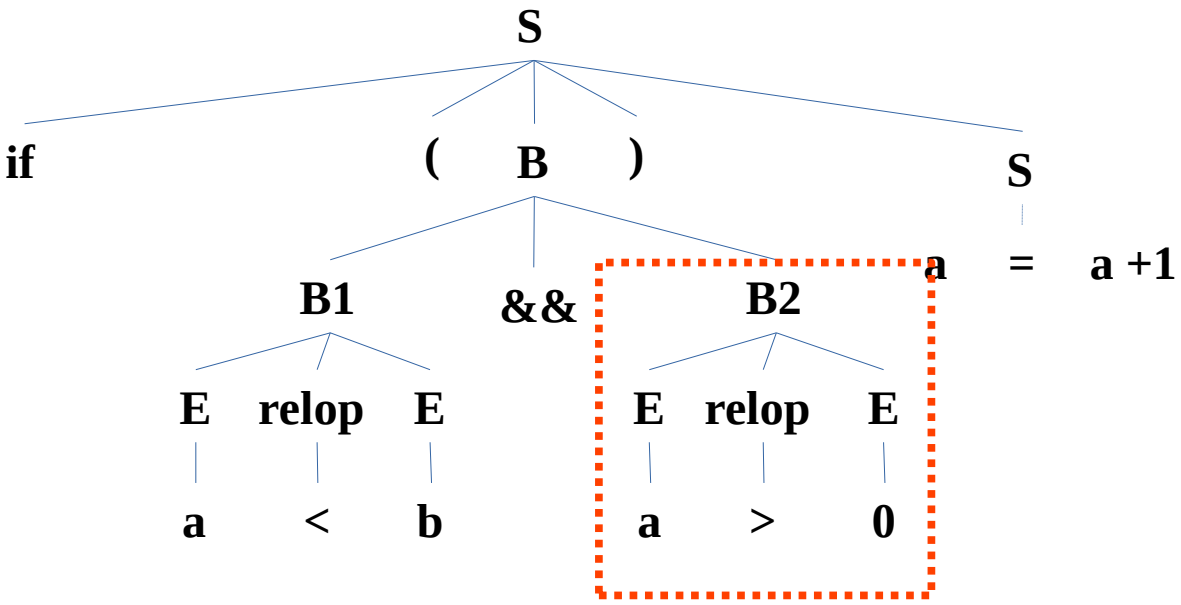
B1.tl → x1
B1.fl → y1

A redução causa a execução de:

```
x = remendo()
y = remendo()
B.code = E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto x) ||
gen(goto y)
B.tl = lista(x)
B.fl = lista(y)
```

Exemplo:

if(a<b && a>0) a=a+1



Código de B2:

if a>0 goto x2
goto y2

Aqui, x e y são as labels que não estão

Estados das listas:

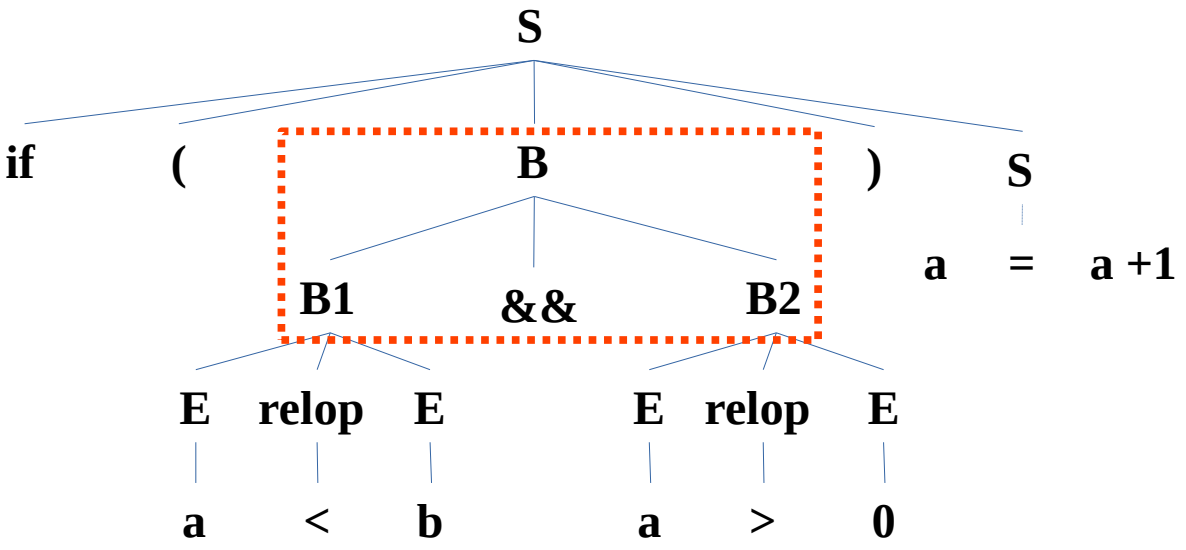
B2.tl → x2
B2.fl → y2

A redução causa a execução de:

```
x = remendo()
y = remendo()
B.code = E1.code||E2.code||gen(if E1.local relop.lexval E2.local goto x) ||
gen(goto y)
B.tl = lista(x)
B.fl = lista(y)
```

Exemplo:

if(a<b && a>0) a=a+1



A redução causa a execução de:

```
x = newlabel()
remenda (B1.tl, x)
B.tl = B2.tl
B.fl = concat(B1.fl, B2.fl)
B.code = B1.code || gen(x": ") || B2.code
```

x=L1
B1.tl=L1 (remendado)
B.tl=x2
B.fl=y1,y2

Código de B:

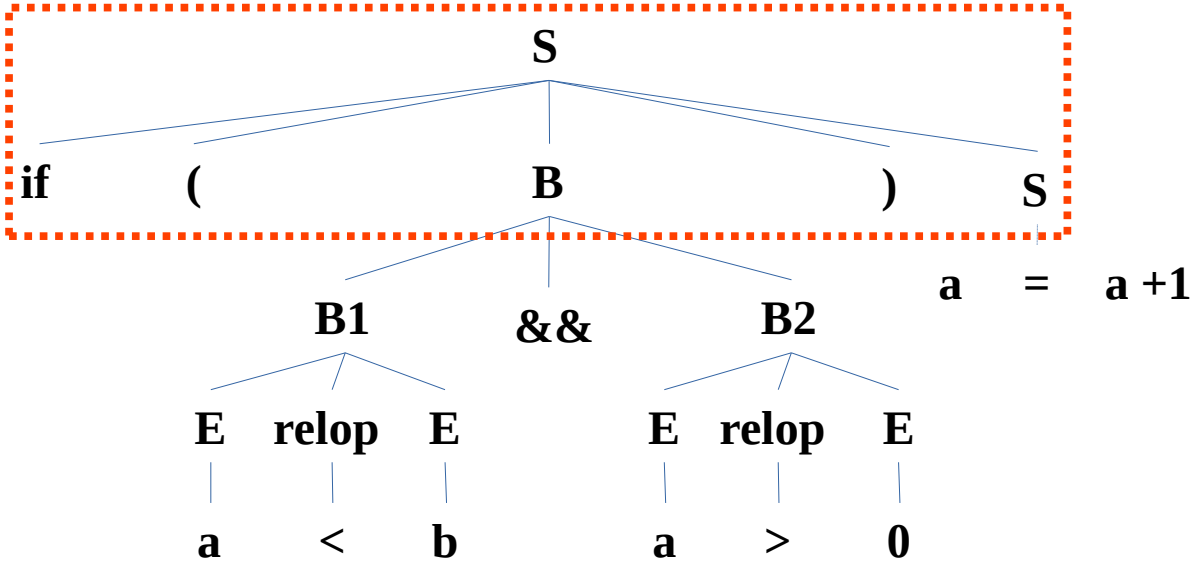
```
        if a<b goto L1
        goto y1
L1:     if a>0 goto x2
        goto y2
```

Estados das listas:

B.tl → x2
B.fl → y1, y2

Exemplo:

if(a<b && a>0) a=a+1



A redução causa a execução de:

```
x = newlabel
y = newlabel
remenda(B.tl, x)
remenda(B.tf, y)
S.next = y
S.code = B.code || gen(B.t:) || S1.code || gen(S.next:)
```

x=L2
y=L3
B.tl=L2 (x2 remendado)
B.fl=L3 (y1,y2 remendado)
S.next=L3

Código de S:

```
        if a<b goto L1
        goto L3
L1:     if a>0 goto L2
        goto L3
L2:     a=a+1
L3: ...
```

Controle de Fluxo:

Comandos de Seleção Múltipla: Switch-Case

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn-1: Sn-1
        default: Sn
}
```

```
                code to evaluate E into t
                goto test
L1:             code for S1
                goto next
L2:             code for S2
                goto next
                ...
Ln-1:           code for Sn-1
                goto next
Ln:             code for Sn
                goto next
test:           if t = V1 goto L1
                if t = V2 goto L2
                ...
                if t = Vn-1 goto Ln-1
                goto Ln
next:
```

Geração de Código Intermediário:

Leitura Recomendada

- **Livro Dragão:**
  - **Capítulo 8 (1.a ed)**
  - **Capítulo 8 (2.a ed)**
  - **Conteúdos são um pouco diferentes nas duas versões**
    - **Preferencialmente 2.a ed**