



Tradução de Fluxo de Controle & Expressões Booleanas

ÉFREN L. SOUZA

Fluxo de Controle & Expressões Booleanas

- A tradução de comandos como **if**, **if-else** e **while** está relacionada com a tradução de expressões booleanas
- Expressões booleanas podem ser usadas para
 - Alterar o fluxo de controle
 - Computar valores lógicos

Usos de Expressões Booleanas

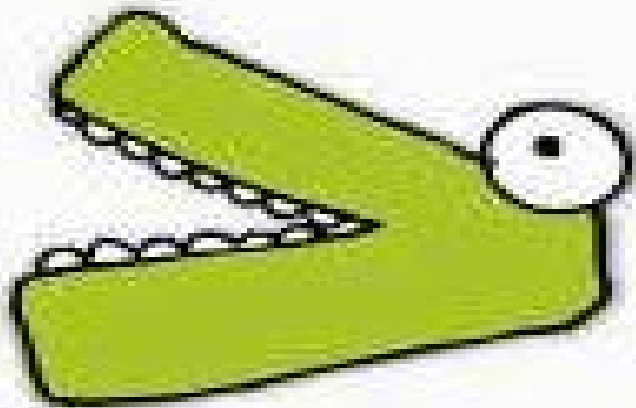
```
inteiro a;  
se (a <= 0)  
    v → a = 1;  
    F → a = a * a;
```

```
inteiro a;  
booleano b;  
b = (a <= 0);
```

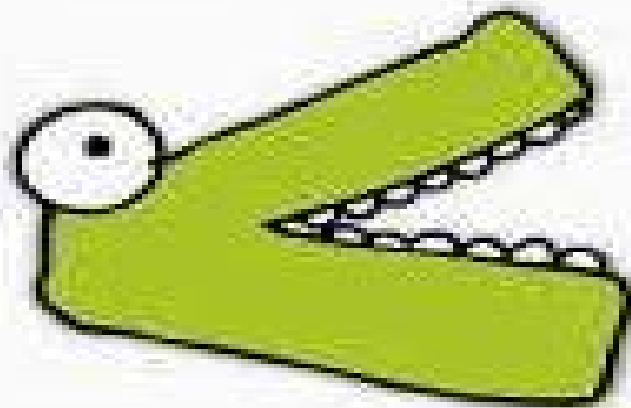
Expressões Booleanas

- São formadas por operadores lógicos aplicados a variáveis lógicas ou expressões relacionais

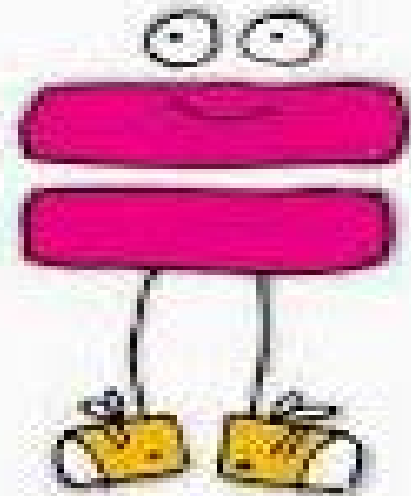
Categoria	Operadores
Operadores Lógicos	(OR); & (AND); ! (NOT)
Operadores Relacionais	==; !=; <; <=; >; >=
Literais Lógicos	verdadeiro; falso
Variáveis Lógicas	ID



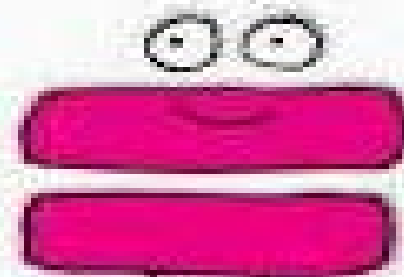
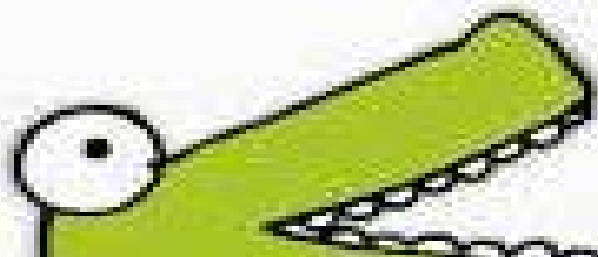
greater than



less than



equal to



Tradução de Expressões Relacionais

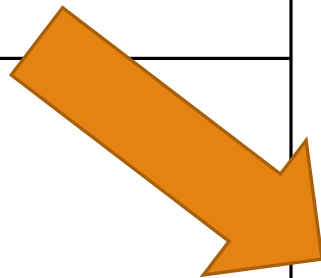
Gramática da DL

- Gramática resumida
- Apenas o operador lógico OU
- E os operadores relacionais `==`, `<` e `<=`

```
IF      ::= se (EXPR) STMT
EXPR    ::= EXPR " | " REL | REL
REL     ::= REL < ARITH | REL <= ARITH |
           REL > ARITH | ARITH
ARITH   ::= ARITH + TERM | ARITH - TERM |
           TERM
TERM    ::= TERM * FACTOR | FACTOR
FACTOR  ::= (EXPR) | ID | LIT_INT |
           LIT_REAL | LIT_BOOL
```

Computando Valores Lógicos

```
booleano b;  
b = ( 0 < 10 );  
escreva(b);
```



```
%b = alloca i1  
store i1 0, i1* %b  
%2 = icmp slt i32 0, 10  
store i1 %2, i1* %b  
...
```

Operadores Relacionais em LLVM-IR

Operador da DL	Equivalente em LLVM-IR	
	Inteiro/Booleano	Ponto flutuante
<code>==</code>	<code>icmp eq</code>	<code>fcmp oeq</code>
<code>!=</code>	<code>icmp ne</code>	<code>fcmp one</code>
<code><</code>	<code>icmp slt</code>	<code>fcmp olt</code>
<code><=</code>	<code>icmp sle</code>	<code>fcmp ole</code>
<code>></code>	<code>icmp sgt</code>	<code>fcmp ogt</code>
<code>>=</code>	<code>icmp sge</code>	<code>fcmp oge</code>

Emitter.java

- No switch do método `codeOperation()` precisamos incluir as novas instruções

PARA O TIPO REAL

```
case LT:    return "fcmp olt";  
case LE:    return "fcmp ole";  
case GT:    return "fcmp ogt";
```

PARA O TIPO INTEIRO

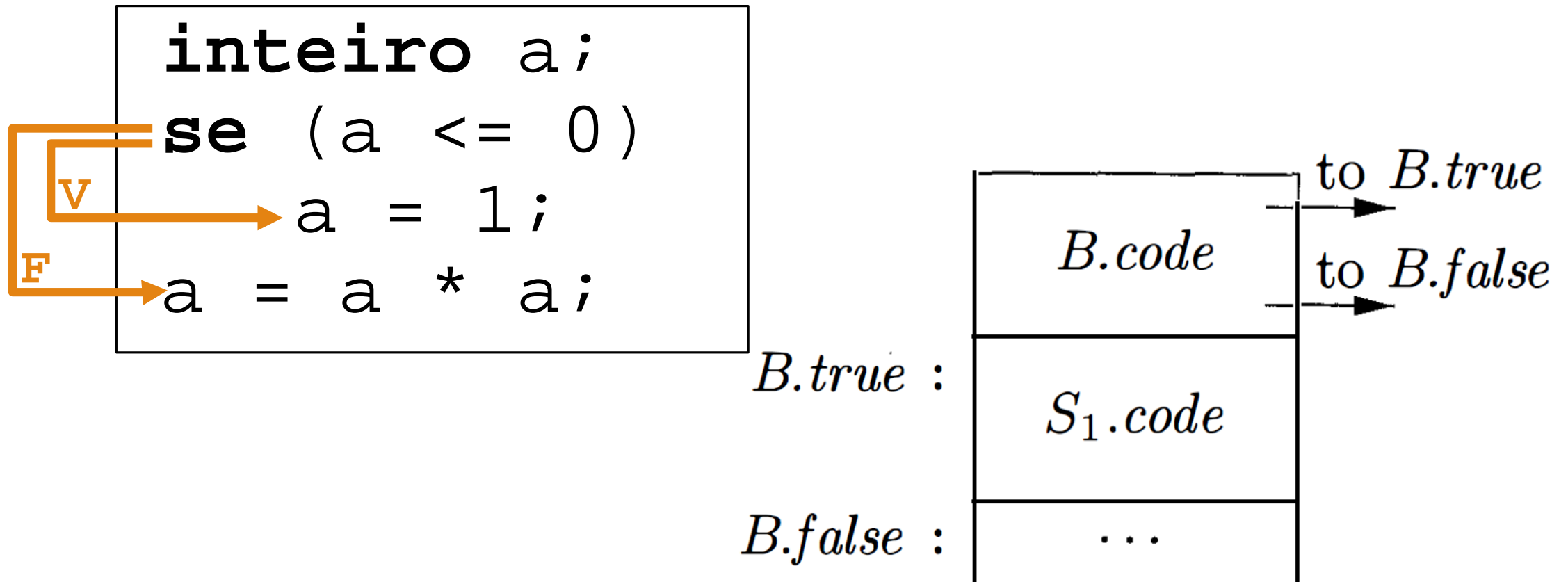
```
case LT:    return "icmp slt";  
case LE:    return "icmp sle";  
case GT:    return "icmp sgt";
```

Rel.java


- Muito similar à tradução dos operadores aritméticos
- O temporário sempre será booleano
- **Teste o código!**

```
@Override
public Expr gen() {
    Expr e1 = expr1.gen();
    Expr e2 = expr2.gen();
    Expr op1 = widen(e1, e2.type() );
    Expr op2 = widen(e2, e1.type() );
    Temp d = new Temp(Tag.BOOL);
    code.emitOperation(
        d, op1, op2, op.tag());
    return d;
}
```

Desvio de Fluxo



```
inteiro a;  
se (a <= 0)  
    a = 1;  
a = a * a;
```



```
%a = alloca i32  
store i32 0, i32* %a  
%2 = load i32, i32* %a  
%3 = icmp sle i32 %2, 0  
br i1 %3, label %L1, label %L2  
L1:  
store i32 1, i32* %a  
br label %L2  
L2:  
...
```

Uso de Expressões Booleanas

- Temos que diferenciar o uso das expressões booleanas
 - Um código é gerado para computar valores lógicos
 - Mas um código diferente é gerado para alterar o fluxo de controle
- A solução é criar um comando específico para gerar o código que controla o fluxo (`jumping`)

Expr.java

- Toda expressão terá um método `jumping()`
- Mas apenas as booleanas farão uso da mesma
- Ela recebe dois rótulos como parâmetro
 - Uma para verdadeiro
 - E outro para falso

```
public void jumping(  
    int t, int f) {  
}
```

Emitter.java

- Vamos adicionar um atributo para controlar a criação dos *labels*
- Sempre que um novo *label* for necessário, esse atributo será incrementado

```
→ private int label;  
  
    public Emitter() {  
        code = new StringBuffer();  
→     label = 0;  
    }
```

Emitter.java

- `newLabel()` será usado sempre que precisarmos criar um novo *label*
- `emitLabel()` gera o código para um *label*

L1 :

```
public int newLabel() {  
    return ++label;  
}
```

```
public void emitLabel(int l) {  
    emit("L" + l + ":");  
}
```


Emitter.java

- O comando `br` é similar ao `goto`
- Ele pode ter duas formas (métodos sobrecarregados)

```
br label %L2
```

```
public void emitBreak(int l) {  
    emit("br label %L" + l);  
}
```

```
br i1 %3, label %L1, label %L2
```

```
public void emitBreak(  
    Expr cond, int lt, int lf) {  
    emit("br i1 " + cond  
        + ", label %L" + lt  
        + ", label %L" + lf);  
}
```

Rel.java

- `jumping()` gera o código que será usado quando a expressão `for` aplicada para controlar o fluxo do programa

```
%2 = load i32, i32* %1
%3 = icmp sle i32 %2, 0
br i1 %3, label %L1, label %L2
```

```
@Override
public void jumping(int t, int f) {
    Expr cond = this.gen();
    code.emitBreak(cond, t, f);
}
```

Literal.java

- Um literal sozinho pode ser uma expressão lógica, se ele for do tipo lógico
- Então ele também precisa de um `jumping`

```
@Override  
public void jumping(int t, int f) {  
    code.emitBreak(this, t, f);  
}
```

Id.java

- Um id sozinho pode ser uma expressão lógica, se ele for do tipo lógico
- Então ele também precisa de um jumping

```
@Override  
public void jumping(int t, int f) {  
    Expr e = this.gen();  
    code.emitBreak(e, t, f);  
}
```

If.java

- O `br` é gerado pelo `jumping()` da expressão lógica
- O *label* `l1` manda o fluxo para o corpo do `if`
- O *label* `l2` manda o fluxo para fora do `if`
- Teste o código usando literal, id e relacional!

```
br i1 %3, label %L1, label %L2
L1:
    ...
    br label %L2
L2:
    ...
```

@Override

```
public void gen() {
    int l1 = code.newLabel();
    int l2 = code.newLabel();
    expr.jumping(l1, l2);
    code.emitLabel(l1);
    stmt.gen();
    code.emitBreak(l2);
    code.emitLabel(l2);
}
```



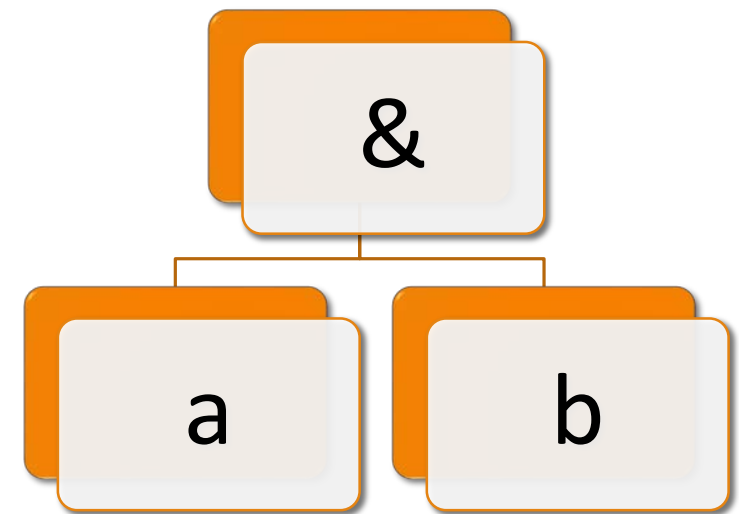
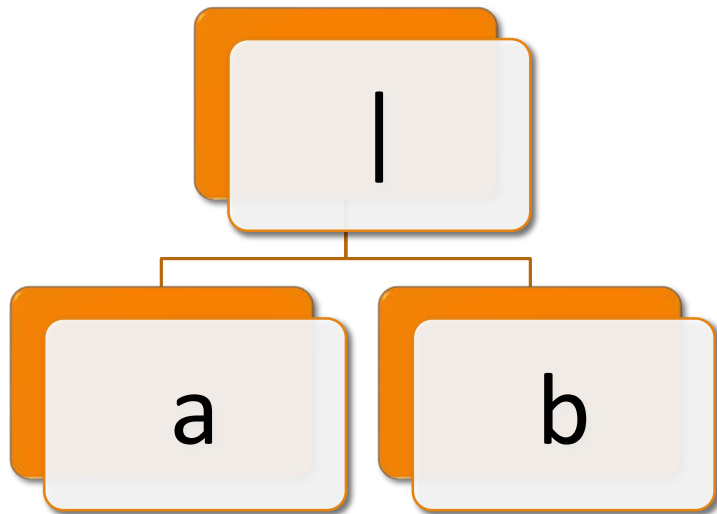
"NO!
Try not!
DO or DO NOT,
There is no try."

Tradução de Operadores Lógicos

Operadores Lógicos

- As expressões booleanas (ou lógicas) são formadas por operadores booleanos (ou lógicos)
 - *Or*
 - *And*
 - *Not*
- Aplicados a operandos do tipo booleano


Operadores Lógicos Binários e Unário



Or (|)

- Dada a expressão lógica $(a \mid b)$
 - Se considerarmos que a é verdadeiro
 - Concluimos que a expressão inteira é verdadeira
 - Sem precisar verificar b
- Assim, o compilador pode otimizar a avaliação de expressões booleanas avaliando apenas o suficiente para determinar seu valor

```
inteiro a;  
inteiro b;  
a = 0; b = 10;  
se ( a <= 0 | b <= 0 )  
    a = 100;  
escreva(a);
```

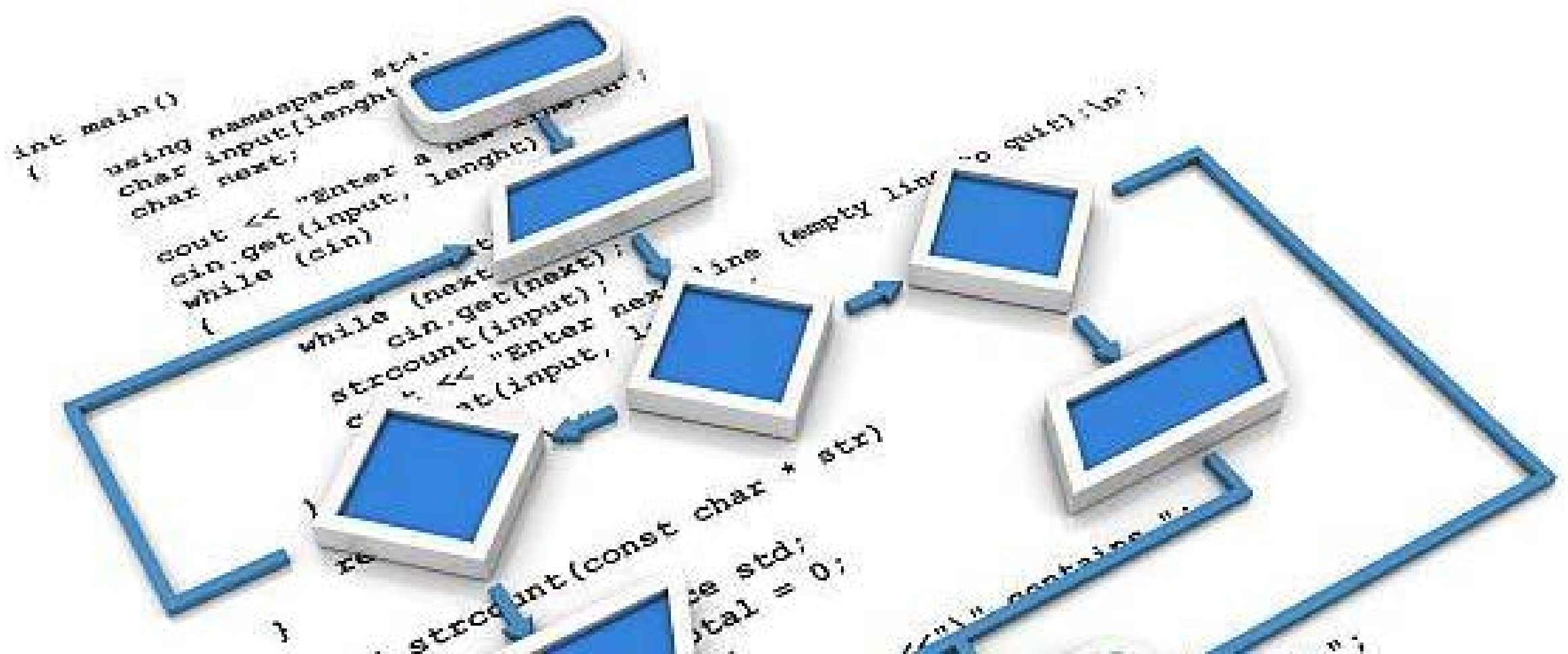


```
%3 = load i32, i32* %a  
%4 = icmp sle i32 %3, 0  
br i1 %4, label %L1, label %L3  
L3:  
%5 = load i32, i32* %b  
%6 = icmp sle i32 %5, 0  
br i1 %6, label %L1, label %L2  
L1:  
store i32 100, i32* %a  
br label %L2  
L2:  
%7 = load i32, i32* %a  
...
```

Or.java

- Os *labels* *t* e *f* são criados pelo comando *if*, por exemplo
- Um outro *label* é criado para o caso da segunda expressão precisar ser avaliada
- **Teste o código!**

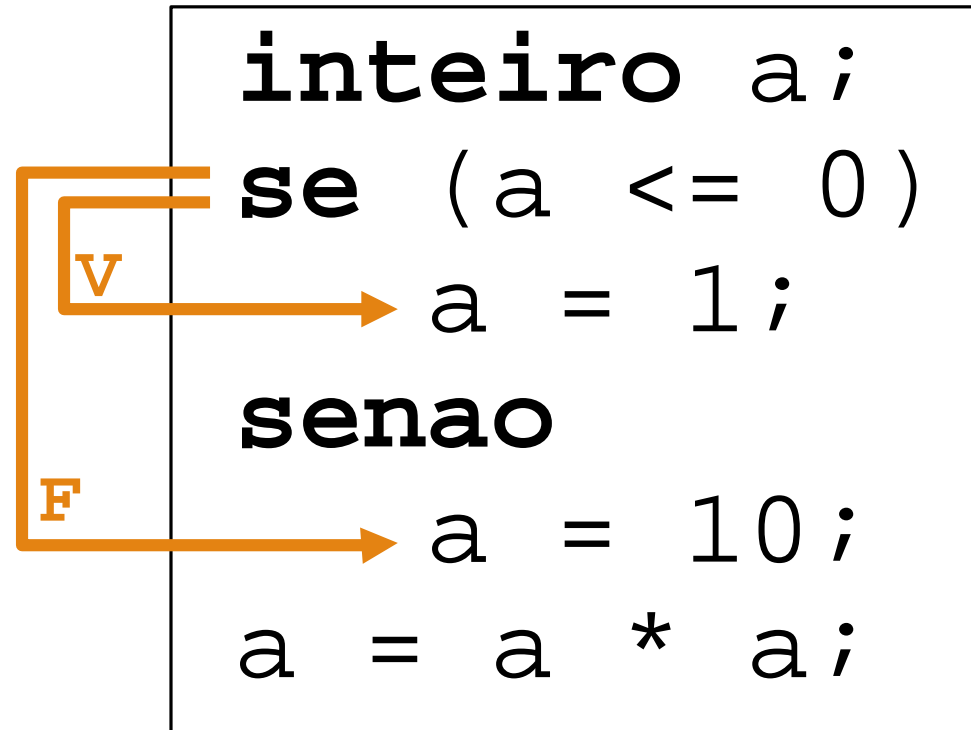
```
@Override
public void jumping(int t, int f) {
    int label = code.newLabel();
    expr1.jumping(t, label);
    code.emitLabel(label);
    expr2.jumping(t, f);
}
```



Outros Comandos de Controle de Fluxo

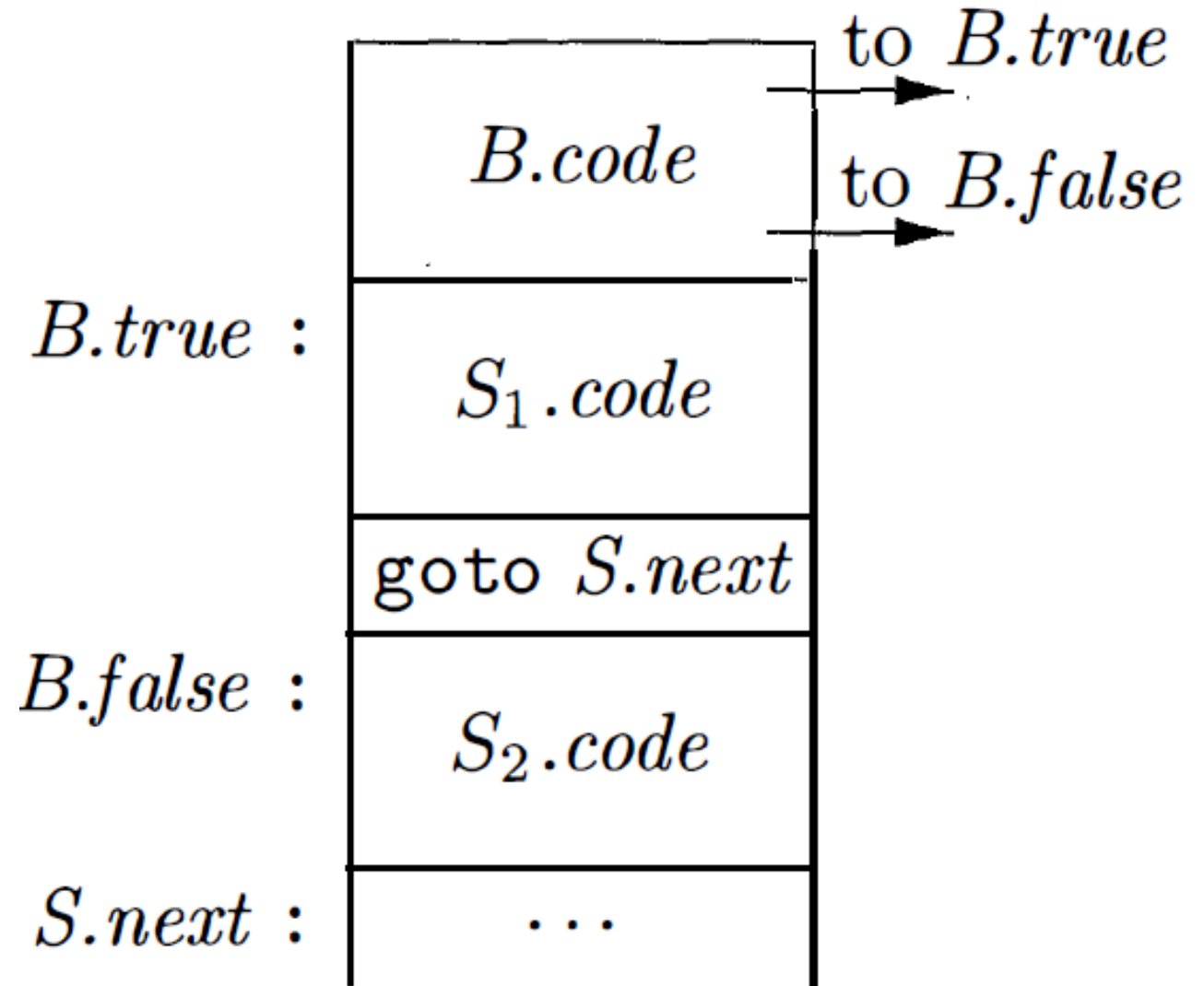
Desvio de Fluxo com If-Else

- O if-else possui dois comandos que podem ser executados



Desvio de Fluxo com If-Else

- O if-else possui dois comandos que podem ser executados



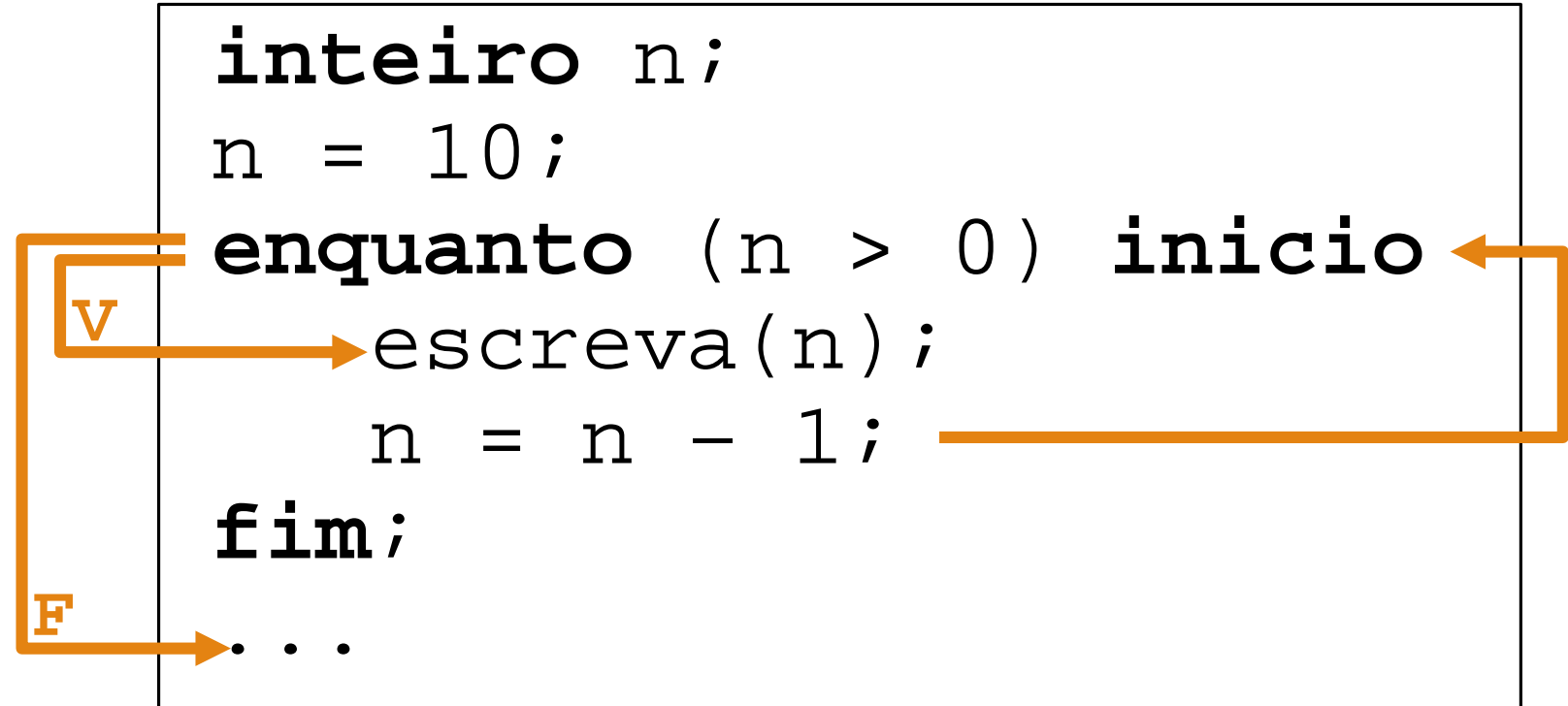
```
booleano aprovado;  
real n1; real n2;  
real n3;  
real media;  
n1 = 10.0; n2 = 5.0;  
n3 = 9.0;  
media = (n1+n2+n3)/3;  
se ( media >= 6.0 )  
    aprovado = verdadeiro  
senao  
    aprovado = falso;  
escreva(media);  
escreva(aprovado);
```



```
...  
%7 = load double, double* %media  
%8 = fcmp oge double %7, 6.0  
br i1 %8, label %L1, label %L2  
L1:  
store i1 true, i1* %aprovado  
br label %L3  
L2:  
store i1 false, i1* %aprovado  
br label %L3  
L3:  
...
```

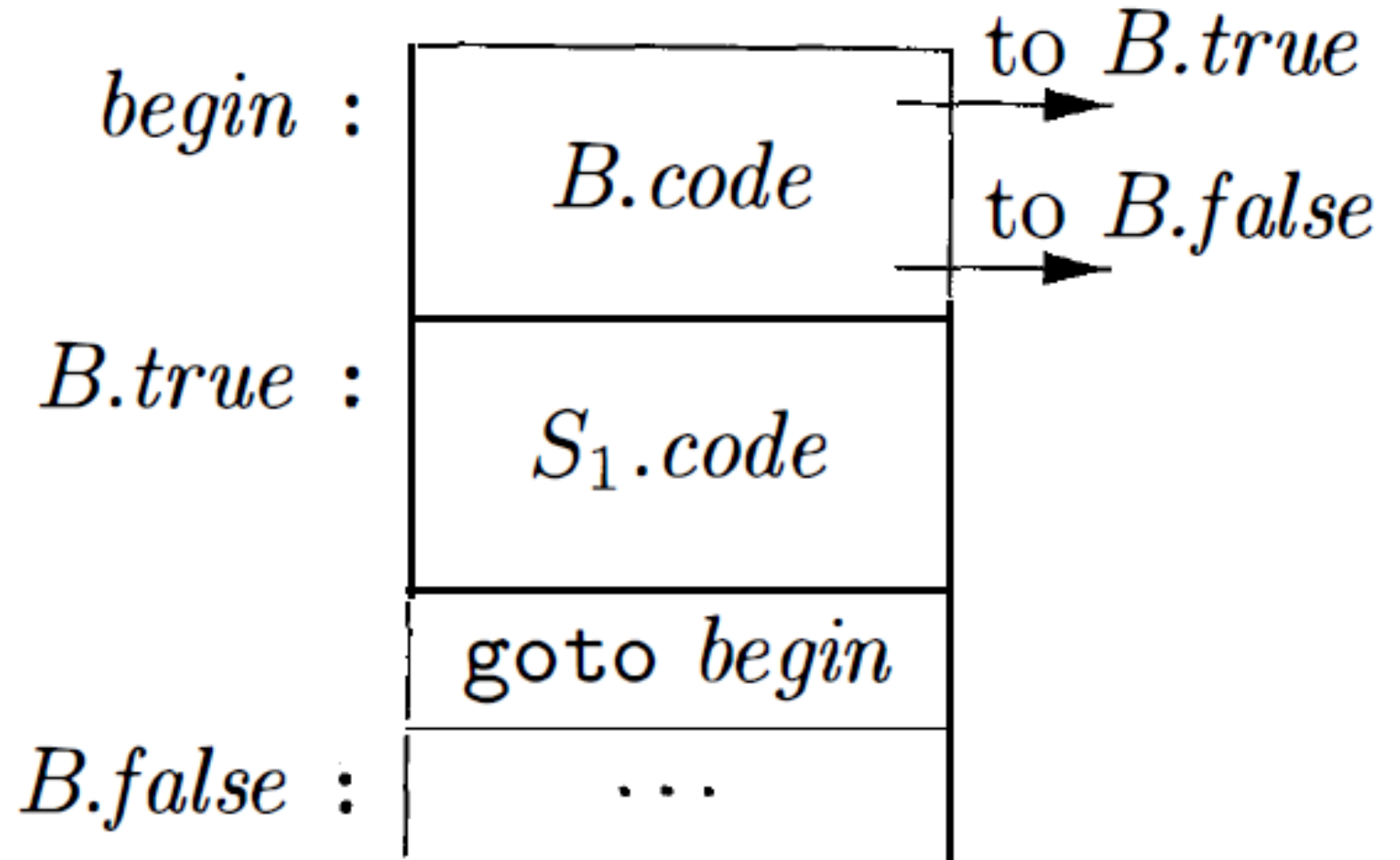
Desvio de Fluxo com while

- O while deve criar um *label* no seu topo
- Após terminar de executar seus comandos, um goto é direcionado a esse *label*



Desvio de Fluxo com while

- O while deve criar um *label* no seu topo
- Após terminar de executar seus comandos, o fluxo é direcionado para o início



```

inteiro b; inteiro e;
inteiro p; inteiro c;
b = 2; e = 1;
p = 1; c = 1;
enquanto ( c <= e )
inicio
    p = p * b;
    c = c + 1;
fim;
escreva(p);

```



```

br label %L1
L1:
%1 = load i32, i32* %c
%2 = load i32, i32* %e
%3 = icmp sle i32 %1, %2
br i1 %3, label %L2, label %L3
L2:
%4 = load i32, i32* %p
%5 = load i32, i32* %b
%6 = mul i32 %4, %5
store i32 %6, i32* %p
%7 = load i32, i32* %c
%8 = add i32 %7, 1
store i32 %8, i32* %c
br label %L1
L3:
...

```

Bibliografia

- AHO, A. V.; SETHI, R.; ULLMAN, J. D. Compiladores: princípios, técnicas e ferramentas. Rio de Janeiro: LTC, 1995.
- CAMPBELL, B.; LYER, S.; AKBAL-DELIBAS, B. Introduction to Compiler Construction in a Java World. CRC Press, 2013.
- APPEL, A. W. Modern compiler implementation in C. Cambridge. Cambridge University Press, 1998.

