Decision Structures - The Syntax Tree

Lecture 22

Sections 8.4, 8.6

Robb T. Koether

Hampden-Sydney College

Fri, Apr 10, 2009

Outline

`if` Statements

- We will consider two forms of the `if` statement.

$$\begin{aligned} stmt &\rightarrow \textbf{if}\ (\ cexpr\ )\ stmt\ \texttt{;} \\ stmt &\rightarrow \textbf{if}\ (\ cexpr\ )\ stmt\ \textbf{else}\ stmt\ \texttt{;} \end{aligned}$$

  where $cexpr$ is a conditional expression.
- Recall that we have the productions

$$\begin{aligned} stmt &\rightarrow \{\ stmts\ \} \\ cexpr &\rightarrow expr \end{aligned}$$

`if` Statements

- For the time being, we will assume that $cexpr$ is a numerical expression.
- Zero is interpreted as false.
- Any nonzero value is interpreted as true.
- Thus, we do not need to be concerned with relational operators (==, !=, <, >, <=, >=) or boolean operators (`&&`, `||`, !) yet.

Flow of Control

- Consider the flow of control for the production

$$stmt \quad \rightarrow \quad \textbf{if} \ (\ cexpr \ ) \ stmt_1 \ ;$$

- If $cexpr$ is non-zero, then execution must jump to $stmt_1$.
- If $cexpr$ is zero, then execution must jump to whatever follows $stmt_1$.
- Problem: We do not yet know where that it.

Flow of Control

- We need an object associated with $cexpr$ that will contain two destinations.
  - The "true" destination.
  - The "false" destination.
- Such an object is called a backpatch node.

The `BackpatchNode` Class

The `BackpatchNode` Class

```
public class BackpatchNode
{
    LinkedList trueList;
    LinkedList falseList;
}
```

Backpatch Nodes

- Each backpatch node contains
  - A "true" reference to a linked list of labels.
  - A "false" reference to a linked list of labels.
- The nonterminal $cexpr$ will represent a backpatch node.
- From $cexpr$ we set up a "true" destination label and a "false" destination label.

Backpatching

- The “true” and “false” destination labels are labels to be resolved once we know where the destinations are.
- An unresolved label is called a backpatch label.
- When we figure out the destination, we will “backpatch” the backpatch label to an actual label.

Backpatching

An Equate Statement

```
B6=L8
```

- An assembly language equate statement will accomplish this.

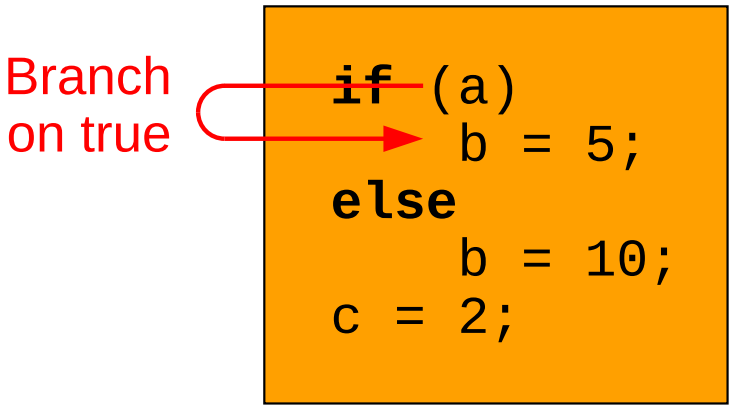
Backpatching

- The reason we keep a *list* of such labels is that there may be several branches in the program that all have the same destination.

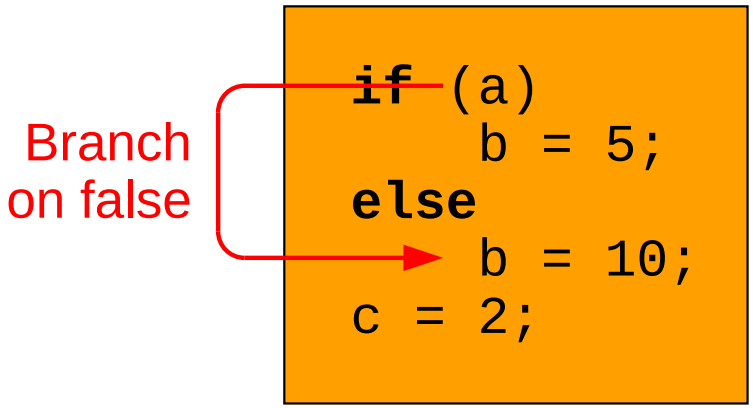
Example


```
if (a)
    b = 5;
else
    b = 10;
c = 2;
```

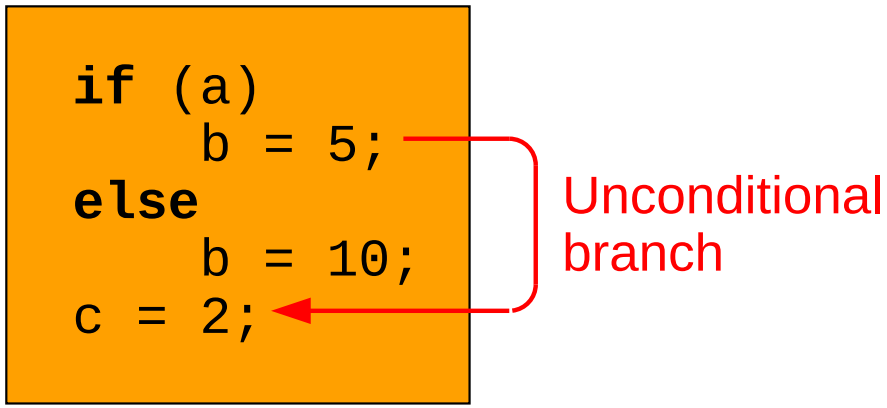
Example



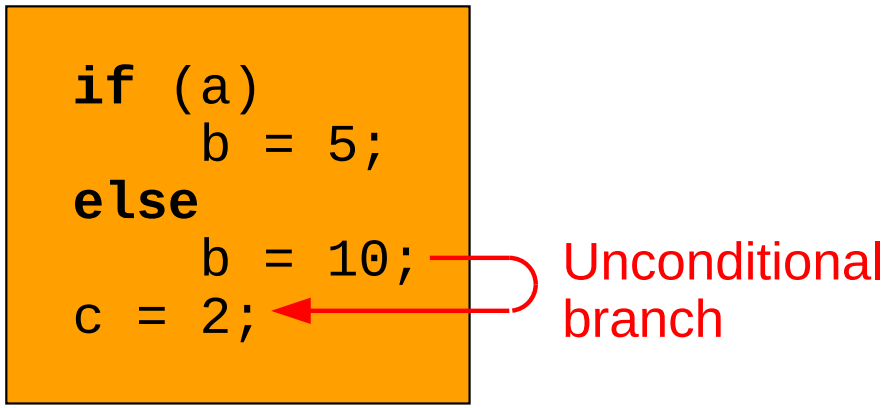
Example



Example



Example



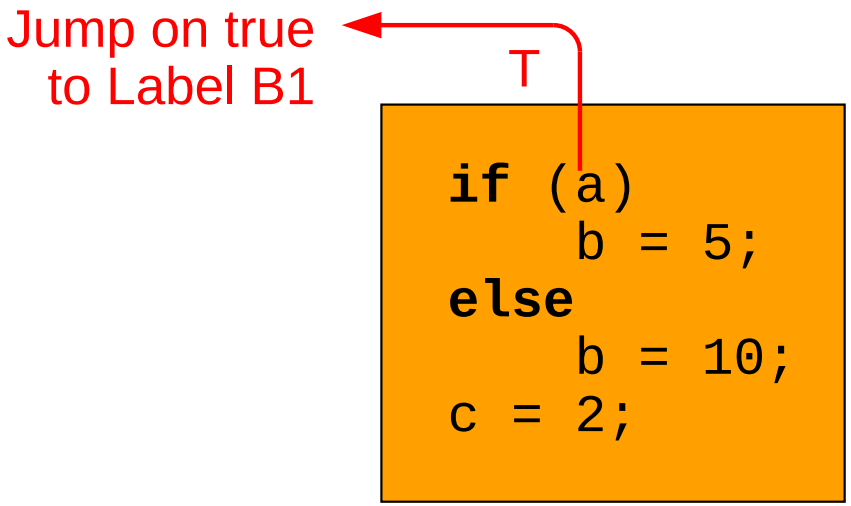
Example

- Notice that the unconditional branch from the true part and the unconditional branch from the false part have the same destination.
- Thus, we build a list of two labels that will be resolved to the same destination.

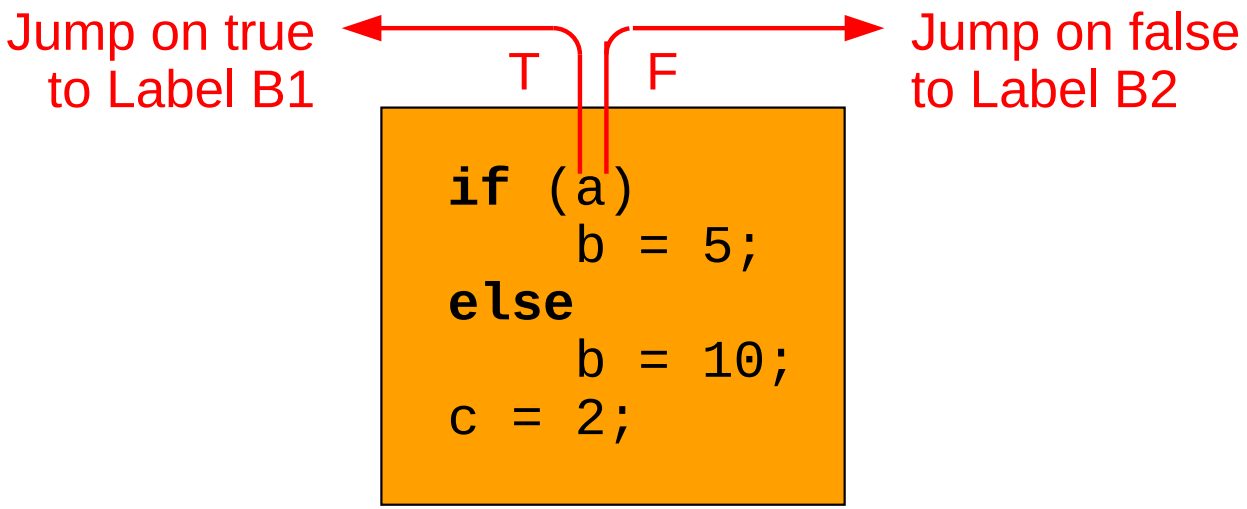
Example


```
if (a)
    b = 5;
else
    b = 10;
c = 2;
```

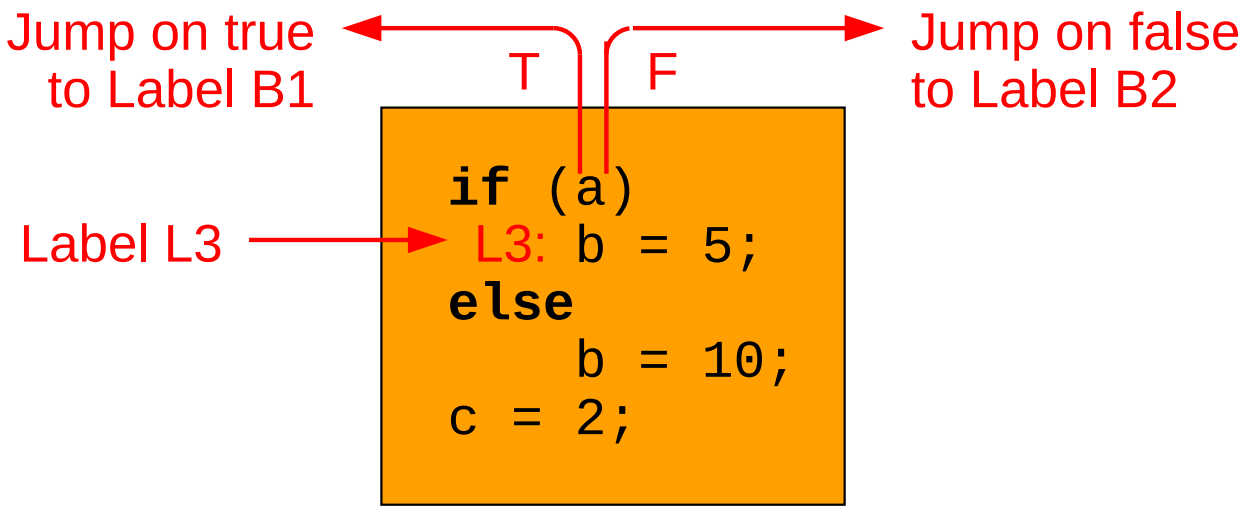
Example



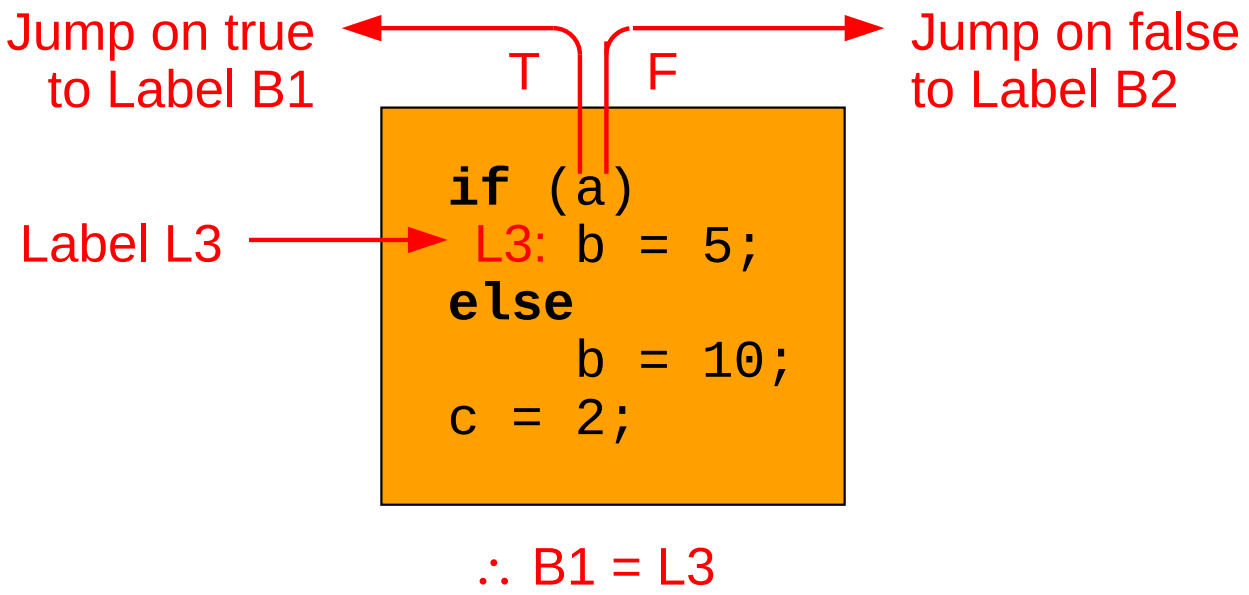
Example



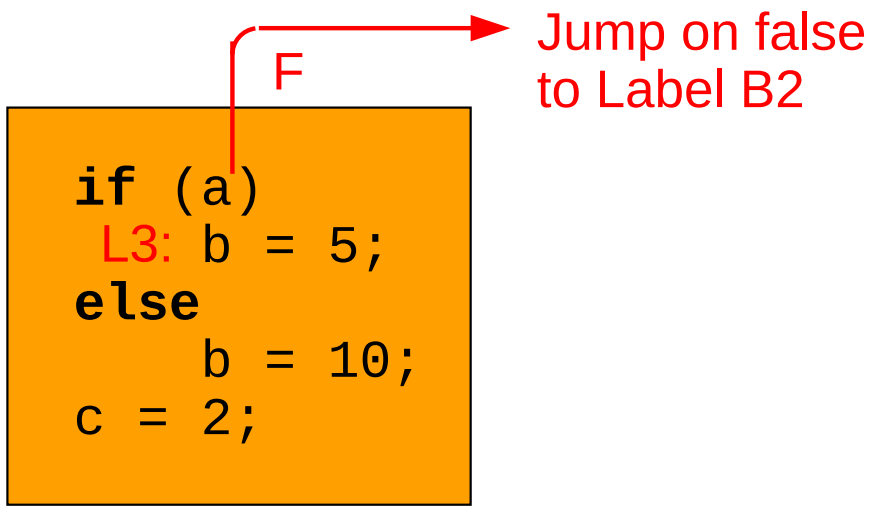
Example



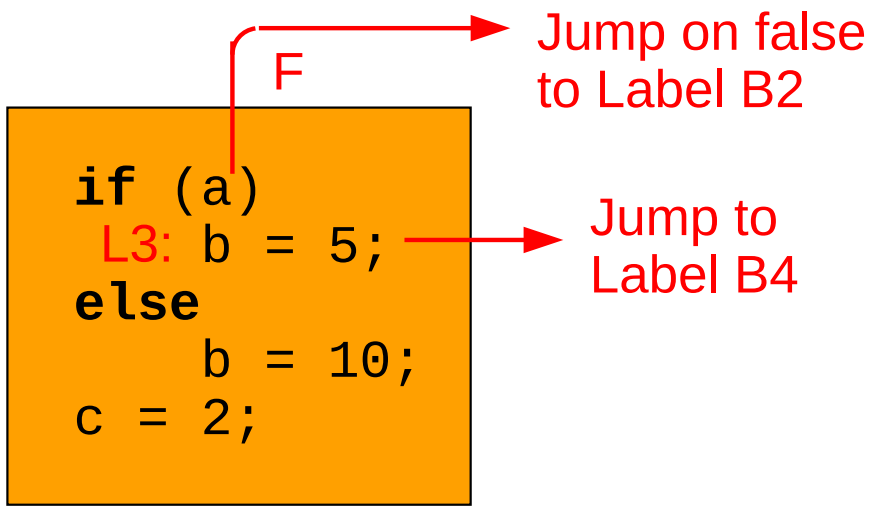
Example



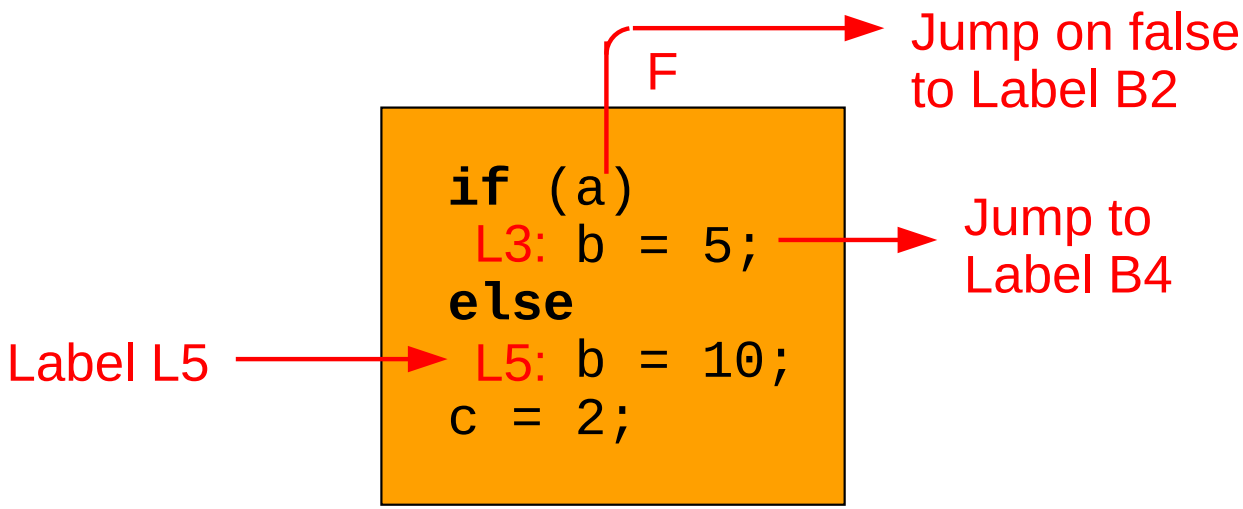
Example



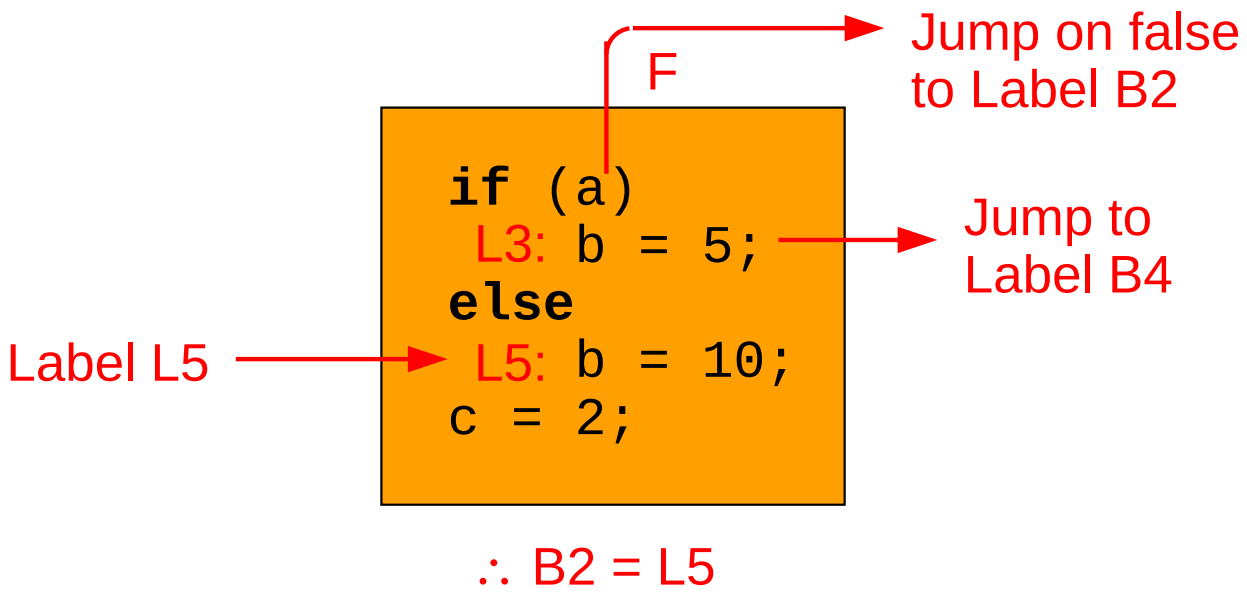
Example



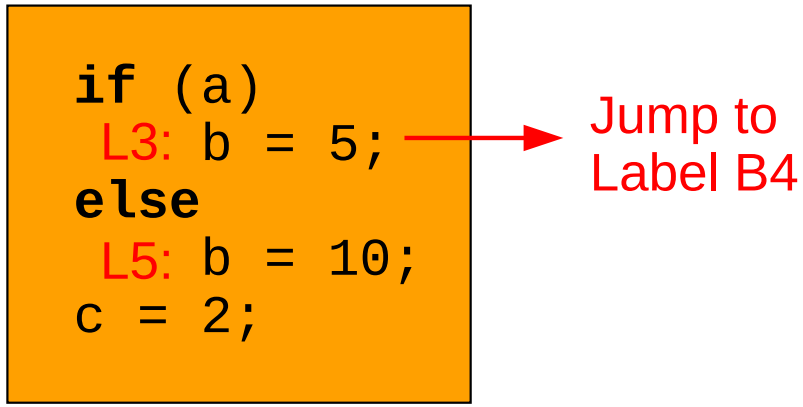
Example


F
Jump on false
to Label B2
if (a)
L3: b = 5;
else
L5: b = 10;
c = 2;
Jump to
Label B4
Label L5

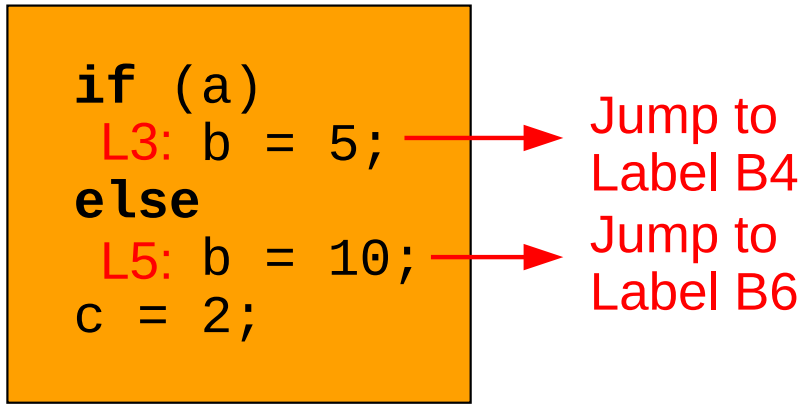
Example


F
Jump on false
to Label B2
if (a)
L3: b = 5;
else
L5: b = 10;
c = 2;
Jump to
Label B4
Label L5
∴ B2 = L5

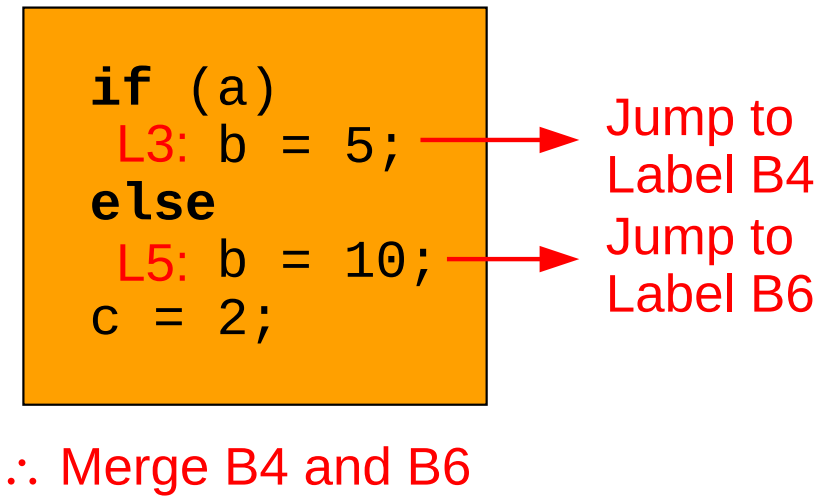
Example



Example

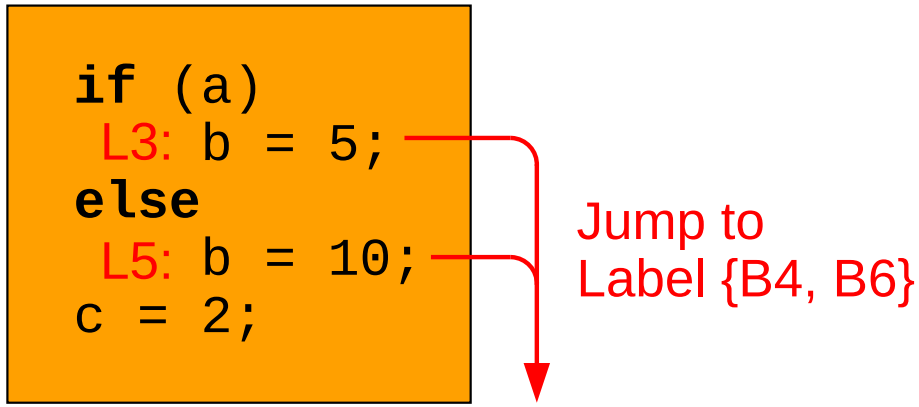


Example

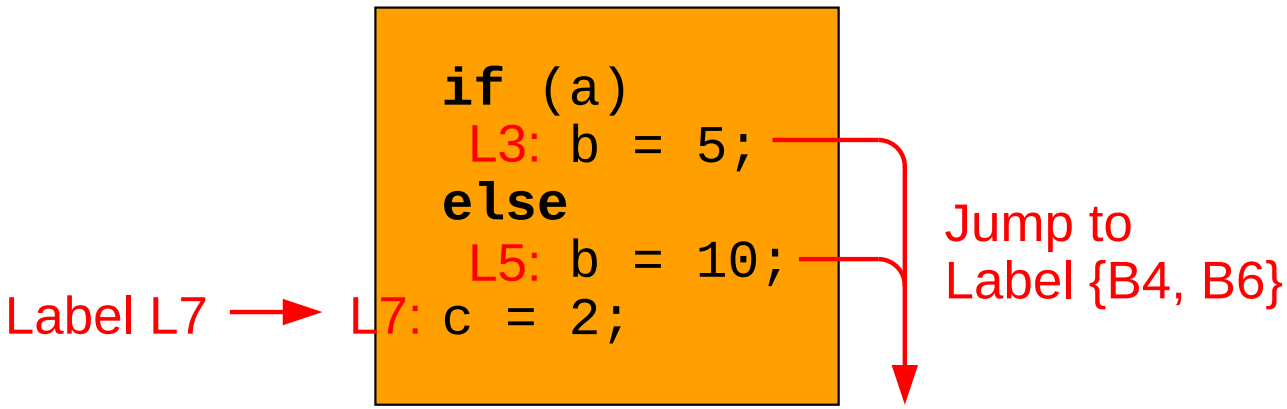


$\therefore$ Merge B4 and B6

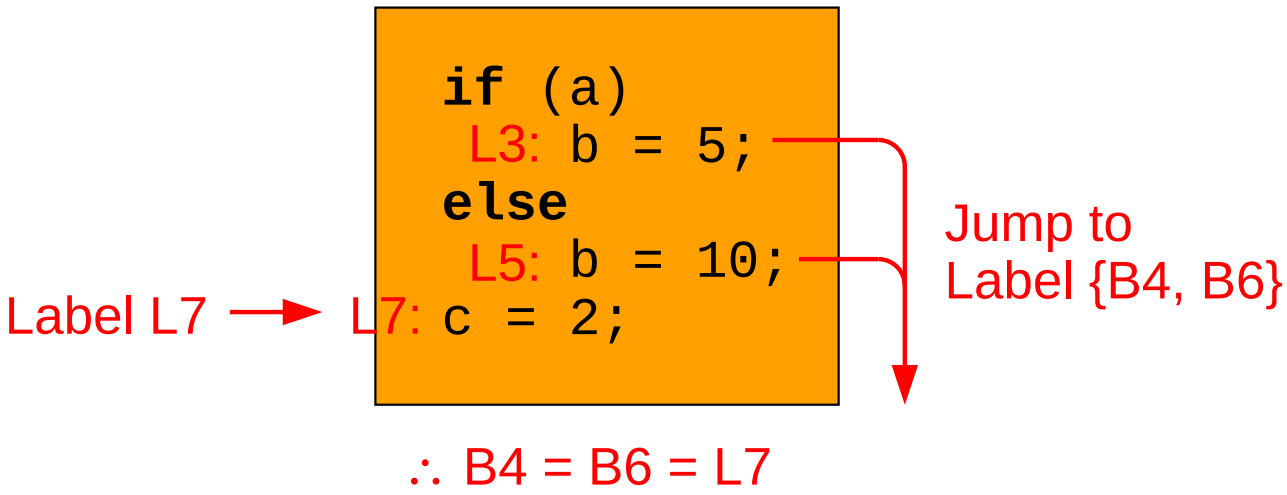
Example



Example



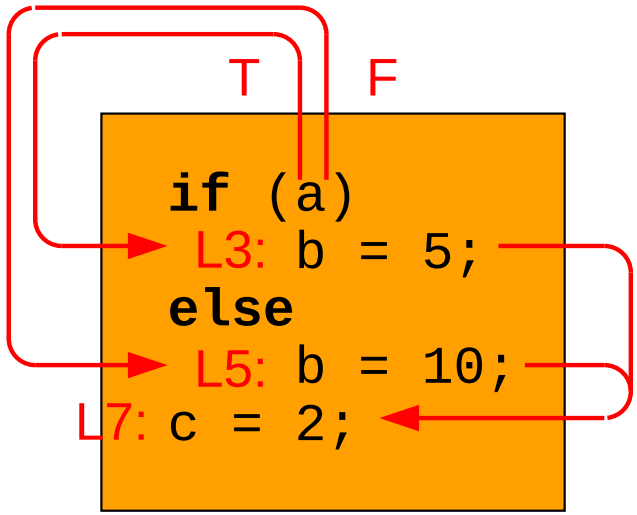
Example



Example


```
  if (a)
   L3: b = 5;
  else
   L5: b = 10;
L7: c = 2;
```

Example



Example

- Within the **`if`** statement, there are two "known" destinations and one unknown destination.
- We can see that labels `B4` and `B6` will be resolved to the same destination eventually, but within the **`if`** statement that destination is unknown.
- Therefore, in the meantime, we merge them together in a list of backpatch nodes to be resolved later to the same destination.
- Once we learn the destination, all labels in the list are resolved to that location.

Backpatch-Related Functions

Backpatching Functions

- `makeList(label);`

  Creates a `LinkedList` containing the single `Integer` `label`. Returns a reference to the list.
- `merge(list1, list2);`

  Merges the elements of `list1` and `list2`. Returns a reference to the merged list.
- `backpatch(list, label);`

  Equates `label` as the target label for each backpatch label in `list`.

Labels and Jumps in the Grammar

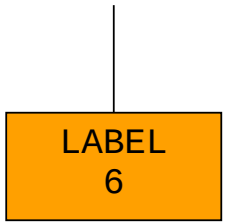
- We saw in the example that actual labels are needed within the **`if`** statement to serve as destinations.
- We will incorporate the generation of these labels into the grammar.
- The nonterminal m will create a label node which will serve as a known destination.
- The nonterminal n will create a jump to an as-yet unknown destination.

Labels and Jumps in the Grammar

- m represents a destination.
- n represents an unconditional branch.
- The productions that involve these nonterminals are

$$
\begin{aligned}
stmts &\rightarrow stmts\ m\ stmt \\
stmt &\rightarrow \textbf{if}\ (\ cexpr\ )\ m\ stmt \\
stmt &\rightarrow \textbf{if}\ (\ cexpr\ )\ m\ stmt\ n\ \textbf{else}\ m\ stmt \\
func &\rightarrow fbeg\ stmts\ m\ \}
\end{aligned}
$$

Label Trees

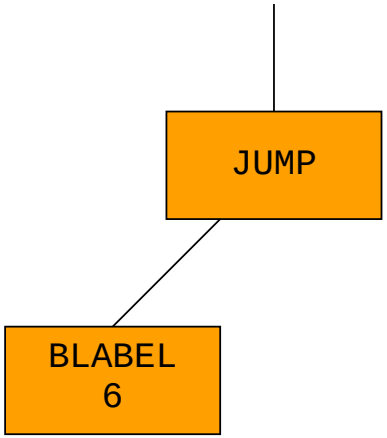


A Label Tree

A Printed Label Tree

An Assembly Code Label

Jump Trees



A Jump Tree

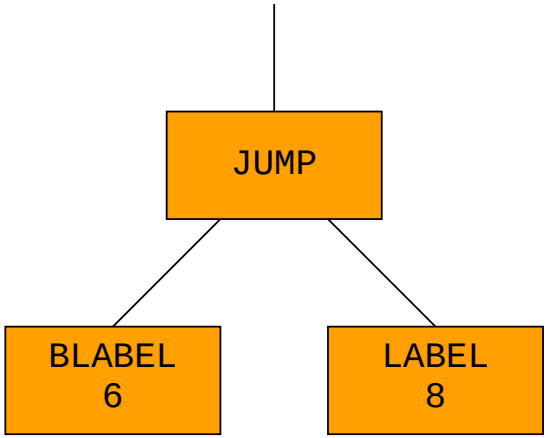
```
JUMP INT
    BLABEL blabel=6
```

A Printed Jump Tree

```
jmp  B6
```

An Assembly Code Jump

Equate Trees



An Equate Tree

```
EQU
    BLABEL blabel=6
    LABEL label=8
```

A Printed Equate Tree

```
B6=L8
```

An Assembly Code Equate

Assignment

Homework

- Read Section 8.4, pages 491 - 493.
- Read Section 8.6, pages 504 - 506.