6.6 Control Flow

The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions. In programming languages, boolean expressions are often used to

1. *Alter the flow of control.* Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expressions is implicit in a position reached in a program. For example, in **if** (E) S, the expression E must be true if statement S is reached.

2. *Compute logical values.* A boolean expression can represent *true* or *false* as values. Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

The intended use of boolean expressions is determined by its syntactic context. For example, an expression following the keyword **if** is used to alter the flow of control, while an expression on the right side of an assignment is used to denote a logical value. Such syntactic contexts can be specified in a number of ways: we may use two different nonterminals, use inherited attributes, or set a flag during parsing. Alternatively we may build a syntax tree and invoke different procedures for the two different uses of boolean expressions.

This section concentrates on the use of boolean expressions to alter the flow of control. For clarity, we introduce a new nonterminal B for this purpose. In Section 6.6.6, we consider how a compiler can allow boolean expressions to represent logical values.

6.6.1 Boolean Expressions

Boolean expressions are composed of the boolean operators (which we denote `&&`, `||`, and `!`, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 **rel** E_2, where E_1 and

E_2 are arithmetic expressions. In this section, we consider boolean expressions generated by the following grammar:

$$B \rightarrow B \mid\mid B \mid B \,\&\&\, B \mid\ !B \mid (B) \mid E \textbf{ rel } E \mid \textbf{true} \mid \textbf{false}$$

We use the attribute **rel**.*op* to indicate which of the six comparison operators <, <=, =, ! =, >, or >= is represented by **rel**. As is customary, we assume that || and && are left-associative, and that || has lowest precedence, then &&, then !.

Given the expression B_1 || B_2, if we determine that B_1 is true, then we can conclude that the entire expression is true without having to evaluate B_2. Similarly, given $B_1 \&\& B_2$, if B_1 is false, then the entire expression is false.

The semantic definition of the programming language determines whether all parts of a boolean expression must be evaluated. If the language definition permits (or requires) portions of a boolean expression to go unevaluated, then the compiler can optimize the evaluation of boolean expressions by computing only enough of an expression to determine its value. Thus, in an expression such as B_1 || B_2, neither B_1 nor B_2 is necessarily evaluated fully. If either B_1 or B_2 is an expression with side effects (e.g., it contains a function that changes a global variable), then an unexpected answer may be obtained.

6.6.2 Short-Circuit Code

In *short-circuit* (or *jumping*) code, the boolean operators &&, ||, and ! translate into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Example 6.21: The statement

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

might be translated into the code of Fig. 6.34. In this translation, the boolean expression is true if control reaches label L_2. If the expression is false, control goes immediately to L_1, skipping L_2 and the assignment `x = 0`. □

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Figure 6.34: Jumping code

6.6.3 Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of statements such as those generated by the following grammar:

$$\begin{array}{lcl} S & \rightarrow & \textbf{if} \ (\ B \ ) \ S_1 \\ S & \rightarrow & \textbf{if} \ (\ B \ ) \ S_1 \ \textbf{else} \ S_2 \\ S & \rightarrow & \textbf{while} \ (\ B \ ) \ S_1 \end{array}$$

In these productions, nonterminal B represents a boolean expression and nonterminal S represents a statement.

This grammar generalizes the running example of while expressions that we introduced in Example 5.19. As in that example, both B and S have a synthesized attribute *code*, which gives the translation into three-address instructions. For simplicity, we build up the translations $B.code$ and $S.code$ as strings, using syntax-directed definitions. The semantic rules defining the *code* attributes could be implemented instead by building up syntax trees and then emitting code during a tree traversal, or by any of the approaches outlined in Section 5.5.

The translation of **if** (B) S_1 consists of $B.code$ followed by $S_1.code$, as illustrated in Fig. 6.35(a). Within $B.code$ are jumps based on the value of B. If B is true, control flows to the first instruction of $S_1.code$, and if B is false, control flows to the instruction immediately following $S_1.code$.

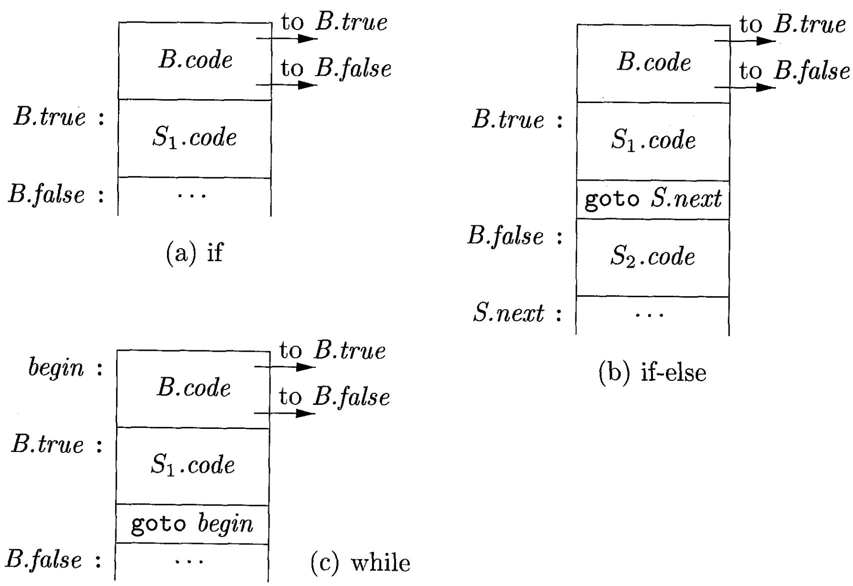

Figure 6.35: Code for if-, if-else-, and while-statements

The labels for the jumps in $B.code$ and $S.code$ are managed using inherited attributes. With a boolean expression B, we associate two labels: $B.true$, the label to which control flows if B is true, and $B.false$, the label to which control flows if B is false. With a statement S, we associate an inherited attribute $S.next$ denoting a label for the instruction immediately after the code for S. In some cases, the instruction immediately following $S.code$ is a jump to some label L. A jump to a jump to L from within $S.code$ is avoided using $S.next$.

The syntax-directed definition in Fig. 6.36-6.37 produces three-address code for boolean expressions in the context of if-, if-else-, and while-statements.

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Figure 6.36: Syntax-directed definition for flow-of-control statements.

We assume that $newlabel()$ creates a new label each time it is called, and that $label(L)$ attaches label L to the next three-address instruction to be generated.[8]

[8]If implemented literally, the semantic rules will generate lots of labels and may attach more than one label to a three-address instruction. The backpatching approach of Section 6.7

A program consists of a statement generated by $P \rightarrow S$. The semantic rules associated with this production initialize *S.next* to a new label. *P.code* consists of *S.code* followed by the new label *S.next*. Token **assign** in the production $S \rightarrow$ **assign** is a placeholder for assignment statements. The translation of assignments is as discussed in Section 6.4; for this discussion of control flow, *S.code* is simply **assign**.*code*.

In translating $S \rightarrow$ **if** (B) S_1, the semantic rules in Fig. 6.36 create a new label *B.true* and attach it to the first three-address instruction generated for the statement S_1, as illustrated in Fig. 6.35(a). Thus, jumps to *B.true* within the code for B will go to the code for S_1. Further, by setting *B.false* to *S.next*, we ensure that control will skip the code for S_1 if B evaluates to false.

In translating the if-else-statement $S \rightarrow$ **if** (B) S_1 **else** S_2, the code for the boolean expression B has jumps out of it to the first instruction of the code for S_1 if B is true, and to the first instruction of the code for S_2 if B is false, as illustrated in Fig. 6.35(b). Further, control flows from both S_1 and S_2 to the three-address instruction immediately following the code for S — its label is given by the inherited attribute *S.next*. An explicit `goto` *S.next* appears after the code for S_1 to skip over the code for S_2. No goto is needed after S_2, since S_2*.next* is the same as *S.next*.

The code for $S \rightarrow$ **while** (B) S_1 is formed from *B.code* and S_1*.code* as shown in Fig. 6.35(c). We use a local variable *begin* to hold a new label attached to the first instruction for this while-statement, which is also the first instruction for B. We use a variable rather than an attribute, because *begin* is local to the semantic rules for this production. The inherited label *S.next* marks the instruction that control must flow to if B is false; hence, *B.false* is set to be *S.next*. A new label *B.true* is attached to the first instruction for S_1; the code for B generates a jump to this label if B is true. After the code for S_1 we place the instruction `goto` *begin*, which causes a jump back to the beginning of the code for the boolean expression. Note that S_1*.next* is set to this label *begin*, so jumps from within S_1*.code* can go directly to *begin*.

The code for $S \rightarrow S_1\ S_2$ consists of the code for S_1 followed by the code for S_2. The semantic rules manage the labels; the first instruction after the code for S_1 is the beginning of the code for S_2; and the instruction after the code for S_2 is also the instruction after the code for S.

We discuss the translation of flow-of-control statements further in Section 6.7. There we shall see an alternative method, called "backpatching," which emits code for statements in one pass.

6.6.4 Control-Flow Translation of Boolean Expressions

The semantic rules for boolean expressions in Fig. 6.37 complement the semantic rules for statements in Fig. 6.36. As in the code layout of Fig. 6.35, a boolean expression B is translated into three-address instructions that evaluate B using

---

creates labels only when they are needed. Alternatively, unnecessary labels can be eliminated during a subsequent optimization phase.

conditional and unconditional jumps to one of two labels: $B.true$ if B is true, and $B.false$ if B is false.

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1$ && B_2	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow\ !\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen($'`if`' $E_1.addr$ **rel**.op $E_2.addr$ '`goto`' $B.true)$ $\mid\mid gen($'`goto`' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'`goto`' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'`goto`' $B.false)$

Figure 6.37: Generating three-address code for booleans

The fourth production in Fig. 6.37, $B \rightarrow E_1$ **rel** E_2, is translated directly into a comparison three-address instruction with jumps to the appropriate places. For instance, B of the form $a < b$ translates into:

```
if a < b goto B.true
goto B.false
```

The remaining productions for B are translated as follows:

1. Suppose B is of the form $B_1 \mid\mid B_2$. If B_1 is true, then we immediately know that B itself is true, so $B_1.true$ is the same as $B.true$. If B_1 is false, then B_2 must be evaluated, so we make $B_1.false$ be the label of the first instruction in the code for B_2. The true and false exits of B_2 are the same as the true and false exits of B, respectively.

2. The translation of B_1 && B_2 is similar.

3. No code is needed for an expression B of the form $!B_1$: just interchange the true and false exits of B to get the true and false exits of B_1.

4. The constants **true** and **false** translate into jumps to $B.true$ and $B.false$, respectively.

Example 6.22: Consider again the following statement from Example 6.21:

```
if( x < 100 || x > 200 && x != y ) x = 0;
```
(6.13)

Using the syntax-directed definitions in Figs. 6.36 and 6.37 we would obtain the code in Fig. 6.38.

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

Figure 6.38: Control-flow translation of a simple if-statement

The statement (6.13) constitutes a program generated by $P \rightarrow S$ from Fig. 6.36. The semantic rules for the production generate a new label L_1 for the instruction after the code for S. Statement S has the form **if** (B) S_1, where S_1 is `x = 0;`, so the rules in Fig. 6.36 generate a new label L_2 and attach it to the first (and only, in this case) instruction in $S_1.code$, which is `x = 0`.

Since `||` has lower precedence than &&, the boolean expression in (6.13) has the form B_1 `||` B_2, where B_1 is $x < 100$. Following the rules in Fig. 6.37, $B_1.true$ is L_2, the label of the assignment `x = 0;`. $B_1.false$ is a new label $\mathtt{L}_3$, attached to the first instruction in the code for B_2.

Note that the code generated is not optimal, in that the translation has three more instructions (goto's) than the code in Example 6.21. The instruction `goto` $\mathtt{L}_3$ is redundant, since L_3 is the label of the very next instruction. The two `goto` $\mathtt{L}_1$ instructions can be eliminated by using `ifFalse` instead of `if` instructions, as in Example 6.21. □

6.6.5 Avoiding Redundant Gotos

In Example 6.22, the comparison $x > 200$ translates into the code fragment:

```
        if x > 200 goto L4
        goto L1
L4:     ...
```

Instead, consider the instruction:

```
        ifFalse x > 200 goto L1
L4:     ...
```

This `ifFalse` instruction takes advantage of the natural flow from one instruction to the next in sequence, so control simply "falls through" to label L_4 if $x > 200$ is false, thereby avoiding a jump.

In the code layouts for if- and while-statements in Fig. 6.35, the code for statement S_1 immediately follows the code for the boolean expression B. By using a special label *fall* (i.e., "don't generate any jump"), we can adapt the semantic rules in Fig. 6.36 and 6.37 to allow control to fall through from the code for B to the code for S_1. The new rules for $S \rightarrow \textbf{if}\ (B)\ S_1$ in Fig. 6.36 set $B.true$ to *fall*:

$$\begin{aligned} B.true &= \mathit{fall} \\ B.\mathit{false} &= S_1.next \;=\; S.next \\ S.code &= B.code \;||\; S_1.code \end{aligned}$$

Similarly, the rules for if-else- and while-statements also set $B.true$ to *fall*.

We now adapt the semantic rules for boolean expressions to allow control to fall through whenever possible. The new rules for $B \rightarrow E_1$ **rel** E_2 in Fig. 6.39 generate two instructions, as in Fig. 6.37, if both $B.true$ and $B.\mathit{false}$ are explicit labels; that is, neither equals *fall*. Otherwise, if $B.true$ is an explicit label, then $B.\mathit{false}$ must be *fall*, so they generate an `if` instruction that lets control fall through if the condition is false. Conversely, if $B.\mathit{false}$ is an explicit label, then they generate an `ifFalse` instruction. In the remaining case, both $B.true$ and $B.\mathit{false}$ are *fall*, so no jump in generated.[9]

In the new rules for $B \rightarrow B_1 \;||\; B_2$ in Fig. 6.40, note that the meaning of label *fall* for B is different from its meaning for B_1. Suppose $B.true$ is *fall*; i.e, control falls through B, if B evaluates to true. Although B evaluates to true if B_1 does, $B_1.true$ must ensure that control jumps over the code for B_2 to get to the next instruction after B.

On the other hand, if B_1 evaluates to false, the truth-value of B is determined by the value of B_2, so the rules in Fig. 6.40 ensure that $B_1.\mathit{false}$ corresponds to control falling through from B_1 to the code for B_2.

The semantic rules are for $B \rightarrow B_1 \;\&\&\; B_2$ are similar to those in Fig. 6.40. We leave them as an exercise.

Example 6.23: With the new rules using the special label *fall*, the program (6.13) from Example 6.21

[9] In C and Java, expressions may contain assignments within them, so code must be generated for the subexpressions E_1 and E_2, even if both $B.true$ and $B.\mathit{false}$ are *fall*. If desired, dead code can be eliminated during an optimization phase.

$$
\begin{aligned}
test &= E_1.addr\ \mathbf{rel}.op\ E_2.addr \\
s &= \mathbf{if}\ B.true \neq fall\ \mathbf{and}\ B.false \neq fall\ \mathbf{then} \\
&\qquad gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true)\ ||\ gen(\texttt{'goto'}\ B.false) \\
&\quad \mathbf{else\ if}\ B.true \neq fall\ \mathbf{then}\ gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true) \\
&\quad \mathbf{else\ if}\ B.false \neq fall\ \mathbf{then}\ gen(\texttt{'ifFalse'}\ test\ \texttt{'goto'}\ B.false) \\
&\quad \mathbf{else}\ '' \\
B.code &= E_1.code\ ||\ E_2.code\ ||\ s
\end{aligned}
$$

Figure 6.39: Semantic rules for $B \rightarrow E_1$ **rel** E_2

$$
\begin{aligned}
B_1.true &= \mathbf{if}\ B.true \neq fall\ \mathbf{then}\ B.true\ \mathbf{else}\ newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \mathbf{if}\ B.true \neq fall\ \mathbf{then}\ B_1.code\ ||\ B_2.code \\
&\quad\ \mathbf{else}\ B_1.code\ ||\ B_2.code\ ||\ label(B_1.true)
\end{aligned}
$$

Figure 6.40: Semantic rules for $B \rightarrow B_1$ `||` B_2

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

translates into the code of Fig. 6.41.

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Figure 6.41: If-statement translated using the fall-through technique

As in Example 6.22, the rules for $P \rightarrow S$ create label L_1. The difference from Example 6.22 is that the inherited attribute $B.true$ is *fall* when the semantic rules for $B \rightarrow B_1$ `||` B_2 are applied ($B.false$ is L_1). The rules in Fig. 6.40 create a new label L_2 to allow a jump over the code for B_2 if B_1 evaluates to true. Thus, $B_1.true$ is L_2 and $B_1.false$ is *fall*, since B_2 must be evaluated if B_1 is false.

The production $B \rightarrow E_1$ **rel** E_2 that generates $x < 100$ is therefore reached with $B.true = L_2$ and $B.false = fall$. With these inherited labels, the rules in Fig. 6.39 therefore generate a single instruction `if x < 100 goto L2`. □

6.6.6 Boolean Values and Jumping Code

The focus in this section has been on the use of boolean expressions to alter the flow of control in statements. A boolean expression may also be evaluated for its value, as in assignment statements such as `x = true;` or `x = a<b;`.

A clean way of handling both roles of boolean expressions is to first build a syntax tree for expressions, using either of the following approaches:

1. *Use two passes.* Construct a complete syntax tree for the input, and then walk the tree in depth-first order, computing the translations specified by the semantic rules.

2. *Use one pass for statements, but two passes for expressions.* With this approach, we would translate E in **while** (E) S_1 before S_1 is examined. The translation of E, however, would be done by building its syntax tree and then walking the tree.

The following grammar has a single nonterminal E for expressions:

$$\begin{array}{lcl} S & \rightarrow & \textbf{id} \; = \; E \; ; \;\mid\; \textbf{if} \; (\; E \;) \; S \;\mid\; \textbf{while} \; (\; E \;) \; S \;\mid\; S \; S \\ E & \rightarrow & E \,||\, E \;\mid\; E \,\&\&\, E \mid E \; \textbf{rel} \; E \mid E + E \mid (E) \mid \textbf{id} \mid \textbf{true} \mid \textbf{false} \end{array}$$

Nonterminal E governs the flow of control in $S \rightarrow$ **while** (E) S_1. The same nonterminal E denotes a value in $S \rightarrow \textbf{id} \; = \; E\,;$ and $E \rightarrow E \; + \; E$.

We can handle these two roles of expressions by using separate code-generation functions. Suppose that attribute $E.n$ denotes the syntax-tree node for an expression E and that nodes are objects. Let method *jump* generate jumping code at an expression node, and let method *rvalue* generate code to compute the value of the node into a temporary.

When E appears in $S \rightarrow$ **while** (E) S_1, method *jump* is called at node $E.n$. The implementation of *jump* is based on the rules for boolean expressions in Fig. 6.37. Specifically, jumping code is generated by calling $E.n.jump(t, f)$, where t is a new label for the first instruction of $S_1.code$ and f is the label $S.next$.

When E appears in $S \rightarrow \textbf{id} \; = \; E\,;$, method *rvalue* is called at node $E.n$. If E has the form $E_1 + E_2$, the method call $E.n.rvalue()$ generates code as discussed in Section 6.4. If E has the form $E_1 \,\&\&\, E_2$, we first generate jumping code for E and then assign true or false to a new temporary `t` at the true and false exits, respectively, from the jumping code.

For example, the assignment `x = a < b && c < d` can be implemented by the code in Fig. 6.42.

6.6.7 Exercises for Section 6.6

Exercise 6.6.1: Add rules to the syntax-directed definition of Fig. 6.36 for the following control-flow constructs:

a) A repeat-statment **repeat** S **while** B.

```
        ifFalse a < b goto L1
        ifFalse c > d goto L1
        t = true
        goto L2
L1:     t = false
L2:     x = t
```

Figure 6.42: Translating a boolean assignment by computing the value of a temporary

! b) A for-loop **for** $(S_1;\ B;\ S_2)\ S_3$.

Exercise 6.6.2: Modern machines try to execute many instructions at the same time, including branching instructions. Thus, there is a severe cost if the machine speculatively follows one branch, when control actually goes another way (all the speculative work is thrown away). It is therefore desirable to minimize the number of branches. Notice that the implementation of a while-loop in Fig. 6.35(c) has two branches per interation: one to enter the body from the condition B and the other to jump back to the code for B. As a result, it is usually preferable to implement **while** (B) S as if it were **if** (B) $\{$ **repeat** S **until** $!(B)$ $\}$. Show what the code layout looks like for this translation, and revise the rule for while-loops in Fig. 6.36.

! **Exercise 6.6.3:** Suppose that there were an "exclusive-or" operator (true if and only if exactly one of its two arguments is true) in C. Write the rule for this operator in the style of Fig. 6.37.

Exercise 6.6.4: Translate the following expressions using the goto-avoiding translation scheme of Section 6.6.5:

a) `if (a==b && c==d || e==f) x == 1;`

b) `if (a==b || c==d || e==f) x == 1;`

c) `if (a==b && c==d && e==f) x == 1;`

Exercise 6.6.5: Give a translation scheme based on the syntax-directed definition in Figs. 6.36 and 6.37.

Exercise 6.6.6: Adapt the semantic rules in Figs. 6.36 and 6.37 to allow control to fall through, using rules like the ones in Figs. 6.39 and 6.40.

! **Exercise 6.6.7:** The semantic rules for statements in Exercise 6.6.6 generate unnecessary labels. Modify the rules for statements in Fig. 6.36 to create labels as needed, using a special label *deferred* to mean that a label has not yet been created. Your rules must generate code similar to that in Example 6.21.

!! **Exercise 6.6.8:** Section 6.6.5 talks about using fall-through code to minimize the number of jumps in the generated intermediate code. However, it does not take advantage of the option to replace a condition by its complement, e.g., replace `if a < b goto` L_1`; goto` L_2 by `if b >= a goto` L_2`; goto` L_1. Develop a SDD that does take advantage of this option when needed.