1. Dealing with Boolean conditions

1.1 Introduction

There are three main problems in compiling conditions:

1. Many of the branch instructions produced are forward branches, and the compiler does not know the destination of the branch until some time later. (For example, to jump around the ELSE part of an IF statement, the entire ELSE part must be compiled before you know how far to jump.) Thus, some form of backpatching is needed.
2. In Modula-2 and similar languages, there is actually no language construct called "condition". Instead, there are *expressions*, some of which turn out to have type BOOLEAN and which can therefore be used in constructs like the IF statement.
3. You don't always know, at the time of processing a condition, whether a "branch on true" or a "branch on false" type of jump will be needed. As a result, compilers often produce inefficient code sequences like

```
            bgt lab03
            br lab04
lab03:
```

(For clarity, I am using assembly language in this example. In practice, labels like `lab03` would of course be replaced by addresses.) The problem is that it is not obvious how to produce more efficient code, since there appear to be too many special cases. (The example just given looks simple enough, but things get messy once you start using the logical AND and OR operators.) It would be worth having a method which was not too complicated and which still produced efficient code.

1.2 The Backpatching Problem

The way to handle backpatching is to put out a dummy instruction, and to record the location of that instruction and what is supposed to be there. Then, once the compiler has worked out what the instruction should really have been, it needs to modify the output file appropriately. However, the obvious solution of actually going and rewriting part of the output file turns out to be hopelessly inefficient.

The solution is to keep a buffer of, for example, about 100 instructions in the output module. The caller of the output module does not need to know about this buffer. Normally the code produced is in sequence, and a call like

```
PutCode (code)
```

simply adds the new code to buffer, and sends some or all of the buffer to the output file as the buffer fills up. When a backpatch is desired, the caller can specify

```
BackPatch (location, code)
```

and then the output module can check whether `location` falls within the range covered by the current buffer contents. If so, `code` can be placed in the appropriate buffer location, overwriting what was already there, and the problem is solved. This covers a very high proportion of the cases which occur in practice.

If `location` does not fall within the buffer range, then genuine backpatching is needed. Some possible ways of handling this are:

1. Design the output file as a sequence of variable-length records, where each record header shows the starting address (or triple number, etc., depending on what sort of intermediate code you are putting into the output file) for that record. Then whenever out-of-sequence code has to be produced, start a new record. This passes the backpatching problem on to the next pass of the compiler, which is not necessarily a bad thing. If the buck is passed all the way down the line, so that eventually there is out-of-sequence code in the machine language file, the loader can handle this quite easily.

2. Use two output files, one for the "normal" case and one for out-of-sequence code. The next pass of the compiler can put things back in order by reading the two files in parallel. (This is equivalent to using a merging algorithm to merge two sorted files into a single sorted file.) This method works very well in practice.
3. Use a separate sorting pass, or passes, between the front end and back end of the compiler. This method is *not* recommended, because file I/O tends to be the major factor affecting the speed of a compiler, and extra passes can slow it down to the point where nobody wants to use it.
4. Keep all the code in buffers in memory until you are sure that all backpatching has been completed and everything is in sequence. This is equivalent to using an output buffer of unlimited size. I don't much like this method because it can lead to mysterious crashes when you run out of memory (and note that it is relatively common for a long procedure to consist of a single IF statement, which means that a lot of code has to be kept in memory before the backpatching has been completed). However, it could be argued that modern computers have a lot of spare memory and that therefore this method will work on 99% of all programs which occur in practice.

1.3 Distinguishing Between Expressions and Conditions

Given a statement like

```
a := b AND c
```

most compilers would produce code something like

```
          load r1, b
          and r1, c
          store r1, a
```

(using a hypothetical assembly language). But for a statement like

```
IF b AND c THEN
     Group1
ELSE
     Group2
END (*IF*)
```

the more obvious code to produce is

```
          test b
          bfalse elsepart
          test c
          bfalse elsepart
          (code for Group1)
          br next
elsepart: (code for Group2)
next:
```

This suggests that a distinction needs to be made between a Boolean *expression* and an expression used as a Boolean *condition*.

This distinction is, in fact, easy to make, since during parsing you always know – before looking at the expression – whether the expression is going to be used as a condition. Thus you could, if you wished, give the procedure which compiles expressions an input parameter which specifies what sort of code you want.

However, this is a red herring. If you look at the issue more carefully, you will see that the decision on what sort of code to use is a question of code optimisation, and is quite independent of how the expression is going to be used. A simple solution to this whole problem is to treat *every* Boolean expression using the "condition" approach. For example, the first example of this section can be compiled as if it had been written

```
a := FALSE;
IF b AND c THEN
     a := TRUE
END (*IF*)
```

and then the decisions about whether the code can be improved can be left to the code optimizer (if present) in the back end of the compiler.

1.4 A Simple Method for Producing Efficient Code

Now, let us look at the problem of putting in the right sort of conditional branches. In general, the code to look after a condition works out to be something like

```
code to evaluate the condition
btrue truepart
bfalse falsepart
```

where `btrue` and `bfalse` stand for "branch if true" and "branch if false" respectively. However, it always turns out that either the true part or the false part immediately follows this code, so that one of the branches is redundant. The catch is that you do not know, in general, *which* of the two branches will be redundant. You might think that you could predict this by knowing what sort of statement (an IF statement, a WHILE statement, etc.) is currently being processed, but a simple rule like that turns out to be inadequate when dealing with complicated conditions linked by AND and OR.

The solution which I recommend is to maintain two lists, which I shall call TrueList and FalseList, while compiling a condition. Each element of such a list is a record saying the location of that branch instruction, and what sort of branch is desired. (Note well that we are recording where the branch is *from*. There is no point in recording the destination of the branch, since we don't know that until later. In any case, all branches on a given list will have the same destination.) TrueList keeps track of all branches which will jump to the point we want to reach if the condition is true; and similarly for FalseList.

In the case of a simple condition, we produce the code

```
     code to evaluate the condition
b1:  dummy branch instruction
```

and then set up TrueList with the single record <b1, btrue> and FalseList with the single record <b1, bfalse>. Ultimately, the dummy branch will be overwritten by one of the branch instructions in the two lists, and then that in turn will be overwritten by the other branch instruction. That is, only one of the branch instructions will eventually survive; but we do not know in advance which one it will be.

Let's see how this works for compiling the statement

```
IF x > y THEN
     Group1
ELSE
     Group2
END (*IF*)
```

To begin with, the code produced will look like

```
     cmp x, y
b1:  dummy branch
b2:
```

and we have TrueList = (<b1, bgt>) and FalseList = (<b1, ble>). Now we know the starting location of the code for Group1, so we can call a patching procedure FixBranches(TrueList,b2) whose job is to produce a "conditional branch to b2" for each element of the list. After doing this, we have the code

```
     cmp x, y
b1:  bgt b2
b2:
```

and TrueList is empty. After compiling the code for Group1, we insert a dummy branch and record its location for later patching. That is, we have the code

```
     cmp x, y
b1:  bgt b2
b2:  code for Group1
```

```
b3:  dummy branch
b4:
```

and a new list EndList = (<b3, br>). By now, we know where the code for Group2 will start, so we can perform the operation FixBranches(FalseList,b4). Because FalseList has an entry for location b1, the original `bgt` instruction will be overwritten, to give the object code

```
     cmp x, y
b1:  ble b4
b2:  code for Group1
b3:  dummy branch
b4:
```

It should by now be obvious how to continue the example. With a little extra thought, you will easily discover how to handle constructs like ELSIF, WHILE, and so on.

By the way, the labels b1, b2, and so on are used here only to make the example easier to follow. If you were actually producing machine language, the labels would not exist; if you were producing triples as intermediate code, the labels would be triple numbers; and so on.

What is probably not obvious to you at this stage is why the same result could not have been achieved with a much simpler method. The discussion so far has only considered the case where TrueList and FalseList have one element each. It is when the logical connectives AND, OR, and NOT are introduced that the strength of the method becomes evident. Consider the compilation of

```
IF ((a=b) OR (c<d)) AND NOT ((e>=f) OR (g<>h)) THEN ...
```

To begin with, the code produced is

```
     cmp a, b
b1:  dummy branch
b2:
```

and we have TrueList = (<b1,beq>), FalseList = (<b1,bne>). To process the OR, we patch the branches from FalseList (why?), to give the code

```
     cmp a, b
b1:  bne b2
b2:
```

Next, we compile the `(c>d)` part, generating two new lists which we shall call TrueList2 and FalseList2. (The original TrueList is not thrown away; it will be dealt with later. The original FalseList is by now irrelevant since we have already dealt with it.) This brings us to the point where the code is

```
     cmp a, b
b1:  bne b2
b2:  cmp c, d
b3:  dummy branch
b4:
```

and we have TrueList = (<b1,beq>), FalseList = (), TrueList2 = (<b3,blt>), FalseList2 = (<b3,bge>).

Let us, at this point, pause and consider what all these lists mean.

(a) TrueList contains the branches for the case where (a=b) is true – which means that (a=b) OR (anything) must be true.

(b) FalseList is by now irrelevant.

(c) TrueList2 contains the branches for the case where (c>d) is true – which means that the condition (a=b) OR (c>d) must be true.

(d) FalseList2 contains the branches for the case where (c>d) is false – which means that the entire expression so far must be false, since we would not be testing for (c>d) unless the part before the OR was known to be false.

The conclusion from all of this is that the lists TrueList and TrueList2 should be combined. This gives us TrueList = (<b1,beq>,<b3,blt>) and FalseList = (<b3,bge>).

On seeing an AND, what should we do? Notice that if the part before the AND is false then the entire expression is false, whereas if the part before the AND is true then we should test the part after the AND. This implies that we ought to do the backpatching for TrueList at this point, giving the code

```
     cmp a, b
b1:  beq b4
b2:  cmp c, d
b3:  blt b4
b4:
```

The part after the NOT is processed in the same way as the first OR condition. (Forget about the NOT for now, we'll handle it later.) This leads eventually to the code

```
     cmp a, b
b1:  beq b4
b2:  cmp c, d
b3:  blt b4
b4:  cmp e, f
b5:  blt b6
b6:  cmp g, h
b7:  dummy branch
b8:
```

at which point we have TrueList2 = (<b5,bge>,<b7,bne>), FalseList2 = (<b7,beq>), where these lists refer to the sub-condition we are currently processing, which is the part after the NOT. Now we handle the NOT by swapping these two lists. Recalling that we still have a TrueList and FalseList from the part before the AND, the situation becomes TrueList = (), FalseList = (<b3,bge>). TrueList2 = (<b7,beq>), FalseList2 = (<b5,bge>,<b7,bne>). To finish processing the AND, we combine the two FalseLists. (If either of X or Y is false, then X AND Y is false.) The result is TrueList = (<b7,beq>), FalseList = (<b3,bge>,<b5,bge>,<b7,bne>). These two lists are the ones which will be used in constructing the branches to the THEN part and the ELSE part of the IF statement.

Although this explanation was rather long, the actual implementation is simple. If you work through the general case for yourself, you will discover some elementary rules for dealing with an arbitrary Boolean expression. It should also be clear that the technique is independent of what sort of code the compiler is generating − the same method works for machine language or any of the popular forms of intermediate code.

Notice that technique often produces a redundant "conditional branch to next instruction", but this does not matter because the redundant instruction is overwritten. The one case where it is not overwritten is where the redundant instruction is an *unconditional* branch. It is probably worth checking for this special case (which occurs, for example, when compiling an IF statement with no ELSE part) in the FixBranches procedure.