Compilers

Exercise 6: Optimisation (and supplementary questions)

Exercise 6.1 Back-edges

In the notes I show a single loop with two backedges. Show an example in C or Java of two source-code loops that lead to two backedges with the same target node.

Exercise 6.2 Immediate dominators

The *immediate dominator* of a node n is a node that dominates n but does not dominate any other dominator of n. Show that immediate dominators form a tree.

Exercise 6.3 Reducible flow graphs

A Depth-first spanning tree for a graph arises when the edges are traversed in a depth-first search. Categorizing edges in graph:

- Advancing edges: from ancestor to proper descendant.
- Retreating edges: from descendant to ancestor (not necessarily proper)
- Cross edges: all other edges

A backedge is always a retreating edge.

Construct a graph containing a retreating edge which is not a backedge (Thanks to Monica Lam for this one).

Exercise 6.4 Structured control flow

Structured control flow constructs enforce the property that loops have a single entry node. Can you write a loop with multiple entry nodes in Java without using goto? Can you see a way to do it in C? (it's called "Duff's Device").

Supplementary questions

The following questions develop some of the ideas presented in the lectures, connects them to textbook material and introduce some more advanced topics. They are not primarily intended to be a guide to likely exam questions.

Exercise 6.5: Sethi-Ullman weights and graph colouring

If we use the graph colouring scheme for register allocation, we can simplify the code for transExp. TransExp walks the AST, producing instructions which use "temporaries" — that is, pseudo-registers. The pseudo-registers are assigned to real registers by the graph-colouring register allocator. TransExp doesn't have to handle running out of pseudo-registers.

Question: Should transExp use the Sethi-Ullman "weights" algorithm to choose the order of evaluation of sub-expressions? What would happen if it didn't?

Exercise 6.6: addressing modes

In Exercise 3, you were encouraged to use instructions which allow immediate and absolute operands for arithmetic instructions. The 68000's instruction set is much more powerful, and there are further extensions in the 68020 and later versions.

Question: Use Haskell's pattern matching to design a code generator which uses some of these more complex instructions where applicable.

(Look up "maximal munch" in Appel's book).

Does your pattern-matching approach lead to the minimum possible number of instructions? Look up "optimum" versus "optimal" in Appel's book.

(Incidentally, Motorola have also developed ColdFire, a processor for system-on-a-chip applications which executes only a simplified subset of the 68000 instruction set, omitting some of these difficult-to-implement (and difficult-to-use?) features).

Exercise 6.7: Variations on register targeting

In the Intel IA-32 instruction set, the 32-bit (signed and unsigned) integer divide instructions divide a 64-bit operand by a 32-bit number, producing a 32-bit quotient and a 32-bit remainder:

```
idiv   operand       ; 2's complement division
                     ; does edx||eax / operand
                     ; eax <- quotient
                     ; edx <- remainder
```

The quotient is delivered to register eax, the remainder to edx.

Question: How would you modify the code generation for arithmetic expressions shown in the lecture notes to accommodate instructions like this?

Incidentally, unsigned multiply has a similar problem:

```
mul   eax, operand   ; unsigned multiplication
                     ; edx||eax <- eax * operand
```

However, the IA-32 designers took pity on compiler writers and provided three variants of signed integer multiply:

```
imul   operand       ; 2's comp. multiplication
                     ; edx||eax <- eax * operand
       reg, operand  ; reg <- reg * operand
       reg, immed    ; reg <- reg * immed
```

Exercise 6.8: Traces

Consider the following loop:

```
for (i=0; i<=98; i++) {
  if (i == a) {
    b = 1;
  }
}
```

If you implement the algorithms presented so far in this course, you get code like this:

```
.data
; Variable a has been allocated to D4, b has been allocated to D0
.text
move.l #98, D0
clr.l  D1
bra    L5
L1:
move.l D1, D2
move.l D4, D3
cmp.l  D2, D3
beq    L2
bra    L3
L2:
move.l #1, D0
bra    L4
L3:
L4:
add.l  #1, D1
L5:
cmp.l  D0, D1
bgt    L6
bra    L1
L6:
move.l D1, i
```

This code has lots of redundant branches. They result from the simple, syntax-directed translation scheme we used - being syntax-directed, it fails to take advantage of the context in which the code occurs.

There are two ways to improve this. One way might be to modify transStat to take account of the context better. Another way, which promises better results, is to start from the control flow graph for the code above.

Question: Design an algorithm to minimise the number of branches in a CFG.

See Appel Section 8.2 ("Traces").

Exercise 6.9: coalescing registers

The code above includes a number of redundant register-register moves.

Question: How can these be avoided?

Again, you can try to avoid introducing them (in transExp, transStat etc). Alternatively, you can try to eliminate them later. Can the graph colouring scheme be extended to do this?

Exercise 6.10: parallelism

The Intel IA-64 instruction set is explicitly parallel, an idea called "EPIC" — which stands for 'explicit parallel instruction computing'. Each 128-bit "packet" holds three IA-64 instructions, each 41 bits long. The Itanium implementation of IA-64 can issue up to six instructions per cycle - provided they are independent. The main purpose of the extra five bits ($41 * 3 + 5 = 128$) is to indicate whether there is a "stop" present in this packet, and if so, where. Obviously, we get better performance if we can arrange the code so there are six instructions between each stop (a group of parallel instructions, which may span several packets, is called a "bundle").

The challenge with IA-64 is to schedule instructions to maximise parallelism.

Question: Design an algorithm which traverses the instructions of a basic block (ie a branch-free sequence) and schedules each instruction to be executed as early as possible. This is not an optimal schedule — how could it be improved?

Question: What effect does this transformation have on the number of registers needed? Why?

Question: Now consider if-then-else. Why is it sometimes a good idea to execute instructions whose results might not be needed?

Question: How could you extend this idea to handle loops?

See Appel Chapter 20 (instruction scheduling).

Exercise 6.11: Using parallel multimedia instructions

Many recent architectures include special instructions which operate on several short-word operands packed into a single register. Examples include Intel's MMX and SSE, AMD's 3DNow!, and Motorola's PowerPC Altivec. They are often called "SIMD" — single instruction, multiple data.

For example, consider this C code:

```
typedef struct {float f [4];} VECTOR4F;
VECTOR4F add4f (VECTOR4F a, VECTOR4F b)
 { VECTOR4F r;
   r.f [0] = a.f [0] * b.f [0];
   r.f [1] = a.f [1] * b.f [1];
   r.f [2] = a.f [2] * b.f [2];
   r.f [3] = a.f [3] * b.f [3];
   return r; }
```

This can be packed into just one SSE instruction:

```
   addps xmm0,xmm1
```

Now consider a loop:

```
void addv(size, float A[], float B[], float C[])
{ int i;
  for (i=0;i<size;i++)
    C[i] = A[i]+B[i];
  }}
```

Question: What would a compiler have to do to generate SSE code for this?

Compilers

Exercise 6: Optimisation - Sample Solutions

Exercise 6.1 Back-edges

In the notes I show a single loop with two backedges. Show an example in C or Java of two source-code loops that lead to two backedges with the same target node.

It's actually easier with just one loop:

```
while (a < b) {
  if (a == 0) {
    a = b;
  } else {
    a = -a;
  }
}
```

To do it with two takes a little more thought:

```
do {
  do {
    a += 1;
  } while (a<c);
} while (a<b);
```

Exercise 6.2 Immediate dominators

The *immediate dominator* of a node n is a node that dominates n but does not dominate any other dominator of n. Show that immediate dominators form a tree.

We have to show that each node has a unique immediate dominator (except for the start node). Suppose instead there are two: d_1 dominates n, and d_2 dominates n. So all paths from start to n pass through both d_1 and d_2. Then there must be a path from start to d_1 then *d_2 (or vice versa). But all paths from d_1 pass through d_2, so $d1$ dominates d_2.*

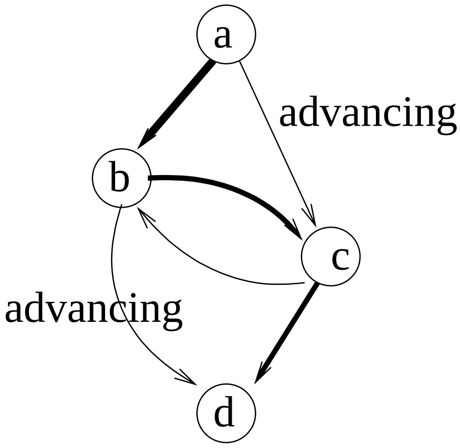
Exercise 6.3 Reducible flow graphs

A Depth-first spanning tree for a graph arises when the edges are traversed in a depth-first search. Categorizing edges in graph:

- Advancing edges: from ancestor to proper descendant.
- Retreating edges: from descendant to ancestor (not necessarily proper)
- Cross edges: all other edges

A backedge is always a retreating edge.

Construct a graph containing a retreating edge which is not a backedge (Thanks to Monica Lam for this one).



The arc from c to b is retreating (since b is c's parent in the spanning tree), but it is not a backedge, since b does not dominate c.

Exercise 6.4 Structured control flow

Structured control flow constructs enforce the property that loops have a single entry node. Can you write a loop with multiple entry nodes in C without using goto?

Consider this:

```
switch(i)
  do {
  case 1:
    printf(``(%d) One\n'',i);
    i+=5;
    continue;
  default:
    printf(``(%d) More!\n'',i);
    i+=1;
    continue;
  } while (i<10);
```

Try it if you don't believe this is valid C!

Paul Kelly, Imperial College November 2009