Loop optimizations

Most programs spend most time executing loops.

So loop optimization is important:

- Code motion: avoid recomputing in loop.
- Induction variables: reduce number of loop counters.
- Loop unfolding: reduce amount of branching.

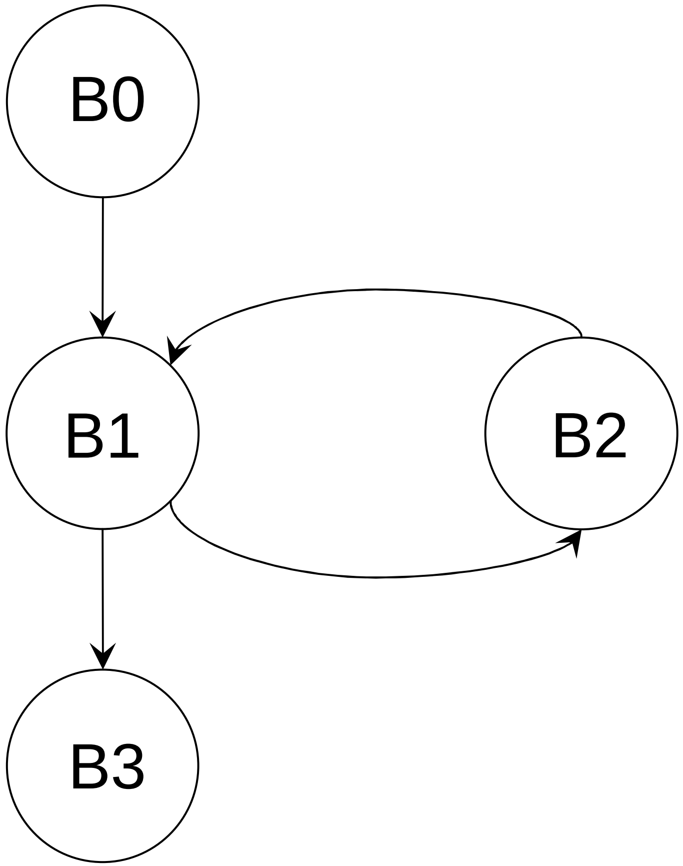
Example

```
size = 2;
i = 0;
while (i <= size*5-1) {
   sum = sum + a[i];
   i = i + 1;
}
write(sum);
```

Quadruples:

	size = 2 i = 0 **goto 3**	B0
3:	t0 = size * 5 t1 = t0 – 1 if (i <= t1) goto 6 else goto 11	B1
6:	t2 = i * 4 t3 = M[a+t2] sum = sum + t3 i = i + 1 goto 3	B2
11:	write(sum)	B3

Flow graph:



Code motion

Code in a loop that always computes same value can be moved (*hoisted*) to before the loop.

Example:

Quadruples

```
    size = 2
    i = 0
3:  t0 = size * 5
    t1 = t0 - 1
    if (i <= t1) goto 6 else goto 11
6:  t2 = i * 4
    t3 = M[a+t2]
    sum = sum + t3
    i = i + 1
    goto 3
11: write(sum)
```

Optimized

```
    size = 2
    i = 0
    t0 = size * 5
    t1 = t0 - 1
5:  if (i <= t1) goto 6 else goto 11
6:  t2 = i * 4
    t3 = M[a+t2]
    sum = sum + t3
    i = i + 1
    goto 5
11: write(sum)
```

Loop-invariant computations

Loop-invariant computation:

A quadruple

d: `t = a₁ op a₂` → d: $t = a_1 \text{ op } a_2$

where for each a_i:

- a_i is a constant
- *or* all definitions of a_i reaching *d* are outside the loop
- *or* only one definition of a_i, *e*, reaches *d*, and *e* is loop-invariant.

Hoisting

Loop-invariant computation

```
d:  t = a1 op a2
```

can be moved before the loop only if:

1. `t` is not live on entry to the loop

2. *and d* is the only definition of `t` in the loop

3. *and d* "dominates" all loop exits at which `t` is live