Control flow optimization

- Control flow (if, while, etc.) = JUMPs and CJUMPs in intermediate code
- How can we reduce the number of JUMPs and CJUMPs executed?

Basic blocks

Intermediate code can be divided into basic blocks.

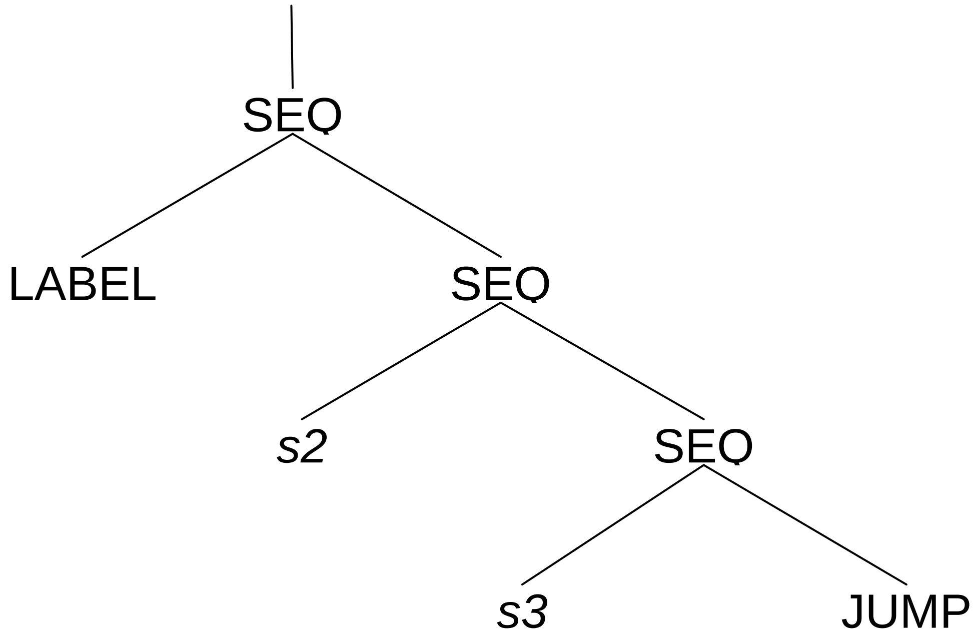
Basic block:

a sequence of statements with no branching
to any statement in the block (except the first) or
from any statement in the block (except the last).

In IR tree form:

Basic block:

a sequence of statements
beginning with a `LABEL`
statement and ending with a
`JUMP` or `CJUMP` statement.



Dividing an IR tree program into basic blocks

```
add new label to first statement of program;
put this in new basic block;

for each statement in program {
  LABEL(L):
    if (current basic block doesn’t end with JUMP or CJUMP) {
      add a  JUMP(L)  statement to end of current basic block;
    }
    start a new basic block;
    add this LABEL statement to current basic block;
  JUMP or CJUMP statement:
    add this statement to end of current basic block;
    start a new basic block;
  else:
    add this statement to the current basic block;
}
```

Basic block example

```
z = 0;
n = y;
while (n > 0) {
   z = z + x;
   n = n - 1;
}
prod = z;
```

Intermediate (IR tree) code:

MOVE(z, 0) MOVE(n, y) **JUMP(NAME(L1))**	B0
LABEL(L1) CJUMP(>, n, 0, NAME(L2), NAME(L3));	B1
LABEL(L2) MOVE(z, +(z, x)) MOVE(n, -(n, 1)) JUMP(NAME(L1))	B2
LABEL(L3) MOVE(prod, z) **JUMP(end)**	B3

(Control) flow graphs

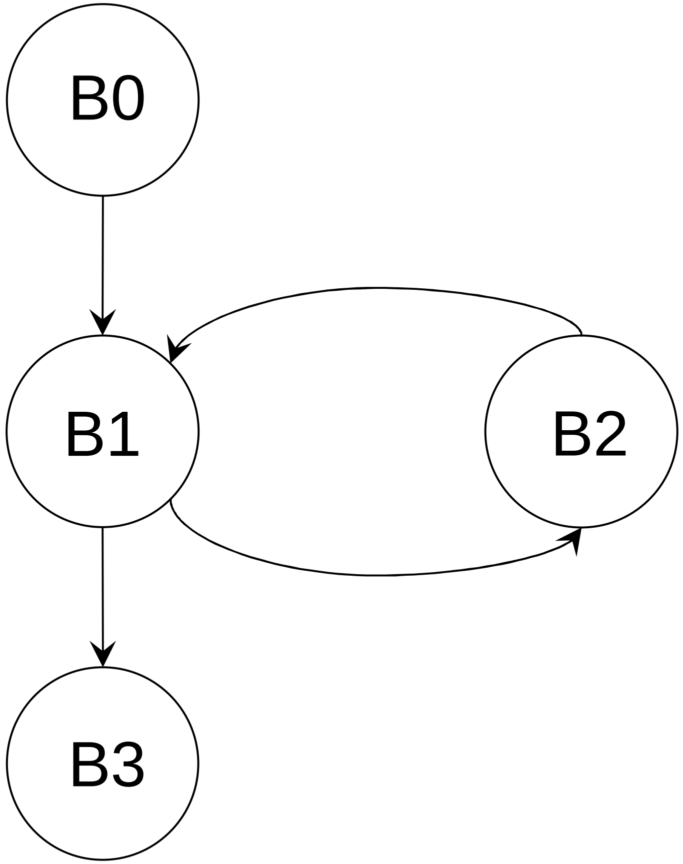
Show structure of a program composed of basic blocks.

Flow graph: directed graph whose

- nodes are basic blocks
- edges show control flow between basic blocks:

  edge from A to B if A ends with `JUMP` or `CJUMP` to label that begins B.

Flow graph for example program:



Basic blocks can be in **any** order.

But for efficiency:

- each block should be followed by a successor in the flow graph:
  - block ending with `JUMP(L)` should be followed by block beginning with `LABEL(L)`
  - block ending with `CJUMP(…, …, …, L1, L2)` should be followed by block beginning with `LABEL(L1)` *or* block beginning with `LABEL(L2)`

Traces

Trace:

sequence of statements executed by a program

= sequence of basic blocks executed by a program

= sequence of basic blocks $b_1, \ldots, b_n$ in which each b_i's successor is b_{i+1}.

To order basic blocks efficiently:

1. find a small set of nonoverlapping traces that cover the program
2. place traces in any order in final program

Dividing flow graph into nonoverlapping traces

```
put all basic blocks into list q;
create new empty trace t;
move first block b from q to t;
while (q not empty) {
  if (q contains a successor of b)
    c = any successor of b in q;
  or {
    end trace t;
    c = any block in q;
    create new empty trace t;
  }
  move c from q to t;
}
end trace t;
```

Traces example

Example flow graph: three ways to divide into two traces:

Trace 1

```
[B0, B1, B2]
[B3]
```

```
MOVE(z, 0)
MOVE(n, y)
~~JUMP(NAME(L1))~~
LABEL(L1)
CJUMP(>,n,0,NAME(L2),NAME(L3))
LABEL(L2)
MOVE(z, +(z, x))
MOVE(n, -(n, 1))
JUMP(NAME(L1))
LABEL(L3)
MOVE(prod, z)
JUMP(end)
```

Trace 2

```
[B0, B1, B3]
[B2]
```

```
MOVE(z, 0)
MOVE(n, y)
~~JUMP(NAME(L1))~~
LABEL(L1)
CJUMP(>,n,0,NAME(L2),NAME(L3))
LABEL(L3)
MOVE(prod, z)
JUMP(end)
LABEL(L2)
MOVE(z, +(z, x))
MOVE(n, -(n, 1))
JUMP(NAME(L1))
```

Trace 3

```
[B0]
[B2, B1, B3]

MOVE(z, 0)
MOVE(n, y)
JUMP(NAME(L1))
LABEL(L2)
MOVE(z, +(z, x))
MOVE(n, -(n, 1))
JUMP(NAME(L1))
LABEL(L1)
CJUMP(>,n,0,NAME(L2),NAME(L3))
LABEL(L3)
MOVE(prod, z)
JUMP(end)
```

One jump is eliminated by canonicalization.

Last set of traces is the most efficient. Why?

Quadruple optimizations

Several optimizations can be done at the quadruples level:

- **dead code elimination**
- **constant propagation**
- **copy propagation**
- **common subexpression elimination**
- **algebraic optimizations**
- **loop optimizations**

Dead code elimination

Dead code:

quadruple

s: `a = b op c`

such that `a` is not subsequently used.

Do liveness analysis on quadruples:

- s is dead code if `a` $\notin$ $out(s)$.
- dead code can be deleted

Optimization: constant propagation

If we have two quadruples

```
d:  t = c
u:  y = t op x
```

where `c` is a constant, maybe we can replace *u*:

```
u:  y = c op x
```

But how do we know that `t` (in *u*) has value `c`?

Constant propagation

```
d:  t = c
u:  y = t op x
```

where c is a constant.

We can replace *u*:

```
u:  y = c op x
```

Conditions:

1. Definition *d* reaches *u*
2. No other definitions of t reach *u*

Reaching definitions

A definition

```
d:  t = …
```

reaches statement *u* if there is a path (of control flow) from *d* to *u* that does not contain a definition of `t`.

in(*s*) = set of definitions that reach the beginning of (statement) *s*

out(*s*) = set of definitions that reach the end of (statement) *s*

In a program, each statement

- **generates** some definitions
- **kills** some definitions

$gen(s)$ = set of definitions generated by statement s

$kill(s)$ = set of definitions killed by statement s

For any assignment (to a temporary) `s: t =` ...:

$$gen(s) = \{s\} \qquad kill(s) = defs(t) - \{s\}$$

For any other quadruple:

$$gen(s) = \{\} \qquad kill(s) = \{\}$$

Algorithm:

1. For each statement n:

   $out(n) = in(n) = \{\}$

2. Repeat

   For each statement n:

   $in'(n) = in(n)$

   $out'(n) = out(n)$

   $in(n) = \cup_{p \in pred(n)} out(p)$

   $out(n) = gen(n) \cup in(n) - kill(n)$

   until $in'(n) == in(n)$ && $out'(n) == out(n)$ for all n

Example: Fibonacci program

s		*gen*(*s*)	*kill*(*s*)
0:	max = 1000	{0}	{}
1:	x = 0	{1}	{5}
2:	y = 1	{2}	{6}
3:	if (y > max) goto 8	{}	{}
4:	z = x + y	{4}	{}
5:	x = y	{5}	{1}
6:	y = z	{6}	{2}
7:	goto 3	{}	{}
8:	write(y)	{}	{}

1st iteration:

```
in(0) = {}
out(0) = {0}
in(1) = {0}
out(1) = {0, 1}
in(2) = {0, 1}
out(2) = {0, 1, 2}
in(3) = out(2) ∪ out(7) = {0, 1, 2} ∪ {}
out(3) = {0, 1, 2}
in(4) = {0, 1, 2}
out(4) = {0, 1, 2, 4}
in(5) = {0, 1, 2, 4}
out(5) = {0, 2, 4, 5}
in(6) = {0, 2, 4, 5}
out(6) = {0, 4, 5, 6}
in(7) = {0, 4, 5, 6}
out(7) = {0, 4, 5, 6}
in(8) = {0, 1, 2}
out(8) = {0, 1, 2}
```

2nd iteration:

```
in(0) = {}
out(0) = {0}
in(1) = {0}
out(1) = {0, 1}
in(2) = {0, 1}
out(2) = {0, 1, 2}
in(3) = out(2)∪out(7) = {0, 1, 2} ∪ {0, 4, 5, 6} = {0, 1, 2, 4, 5, 6}
out(3) = {0, 1, 2, 4, 5, 6}
in(4) = {0, 1, 2, 4, 5, 6}
out(4) = {0, 1, 2, 4, 5, 6}
in(5) = {0, 1, 2, 4, 5, 6}
out(5) = {0, 2, 4, 5, 6}
in(6) = {0, 2, 4, 5, 6}
out(6) = {0, 4, 5, 6}
in(7) = {0, 4, 5, 6}
out(7) = {0, 4, 5, 6}
in(8) = {0, 1, 2, 4, 5, 6}
out(8) = {0, 1, 2, 4, 5, 6}
```

Constant propagation (contd.)

```
0:  max = 1000
```

is the only definition of max, and *0* reaches

```
3:  if (y > max) goto 8
```

so `max` can be replaced by 1000 in *3*.

Can't replace `y` in *3* because both definitions of `y` (*2* & *6*) reach *3*.

Optimization: copy propagation

If there is a quadruple

```
d:  t = z
```

where `z` is a variable, and a quadruple

```
u:  y = t op x
```

u can be replaced by

```
u:  y = z op x
```

Conditions:

1. Definition *d* reaches *u*
2. No other definitions of `t` reach *u*
3. No definition of `z` on path from *d* to *u*