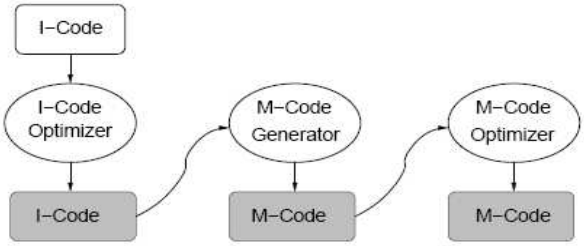
Optimization Strategies



Optimization is a misnomer here: we mean just code improvement.

Two different kinds of optimization can be performed:

- Machine independent based on control-flow and data-flow of the intermediate code.
- Machine dependent - based on the register structure and instruction set of the target machine.

In either case, the goal is to improve the efficiency of the code by **meaning-preserving transformations**.

The literature of the subject is vast, we will only do a general introduction.

Example. Given the following intermediate three address code statements:

```
(1)     i := 0
(2)     i := i + 1
```

it is not always possible to replace them by the single statement:

```
(1)     i := 1
```

as this code may represent the start of a loop whose last statement is:

```
(8)     if i < 10 goto (2)
```

This *optimization* would transform a loop to an infinite loop!

We assume that the intermediate code is a sequence of three-address statements each involving

```
(label)     (operator, op1, op2, target)
```

written in variety of forms, where some of the entries can be dummy.

First, we define the concept of basic block.

A **basic block** is a sequence of consecutive statements in which control flow enters at the first statement and leaves at the last statement.

Here is an algorithm for partitioning three-address statements into basic blocks:

A. Construct the leader set
   1) The first statement is a leader.
   2) Any statement that is the target of a branch is a leader.
   3) Any statement that immediately follows a branch statement is a leader.

B. Construct the Basic Blocks

   For each leader, construct one basic block consisting of the leader and all statements following the leader, up to but not including the next leader or the end of the program.

The function of a basic block is to compute a set of expressions.

The expressions are the values of the names which are **live** on exit from the block.

Two basic blocks are **equivalent** if they compute the same set of expressions.

Find the basic blocks: (4*i because of 4 byte integers)

```
(1)     i : = m-1
(2)     j : = n
(3)     t1: = 4*n
(4)     v : = a[t1]
(5)     i : = i+1
(6)     t2 : = 4*i
(7)     t3 : = a[t2]
(8)     if t3 < v goto (5)
(9)     j : = j-1
(10)    t4 : = 4*j
(11)    t5 : = a[t4]
(12)    if t5 > v goto (9)
(13)    if i >= j goto (23)
(14)    t6 : = 4*i
(15)    x : = a[t6]
(16)    t7 : = 4*i
(17)    t8 : = 4*j
(18)    t9 : = a[t8]
(19)    a[t7] : = t9
(20)    t10 : = 4*j
(21)    a[t10] : = x
(22)    goto (5)
(23)    t11: = 4*i
(24)    x : = a[t11]
(25)    t12 : = 4*i
(26)    t13 : = 4*n
(27)    t14 : = a[t13]
(28)    a[t12] : = t14
(29)    t15 : = 4*n
(30)    a[t15] : = x
```

Basic blocks and their leaders:

B1 : (1), B2 : (5), B3 : (9), B4 : (13), B5 : (14), B6 : (23)

Draw the control flow graph.

Machine independent optimization can be divided into two parts:

local transformations: looking only within a basic block

global optimization: otherwise

Two categories of local transformations applicable to basic blocks

- structure-preserving
- algebraic transformations.

The primary structure-preserving transformations are:

- common sub-expression elimination,
- copy propagation,
- dead-code elimination.

Common Sub-expression Elimination

Example. Remove repeated sub-expressions from B5:

	B5	B5'
	`t6 := 4*i`	`t6 := 4*i`
`t7 -> t6`	`x := a[t6]`	`x := a[t6]`
	`t7 := 4*i`	
	`t8 := 4*j`	`t8 := 4*j`
	`t9 := a[t8]`	`t9 := a[t8]`
`t10 -> t8`	`a[t7] := t9`	`a[t6] := t9`
	`t10 := 4*j`	
	`a[t10] := x`	`a[t8] := x`
	`goto B2`	`goto B2`

A repeated sub-expression is a **common sub-expression** only if no variables in the sub-expression are re-assigned between the repeated occurrences.
The following blocks

```
t1 := 2*k          t1 := 2*k
k := k+1           k := k+1
t2 := 2*k          t2 := t1
```

are not equivalent since the value of `k` which occurs in the repeated sub-expression, `2*k` is re-assigned between the two occurrences.

Common sub-expression elimination can also occur between blocks.

For example, the common sub-expressions between blocks B2, B3 and B5':

```
4*i    in  t2 := 4*i    (B2)  and  t6 := 4*i   (B5)   t6    -> t2
a[t2]  in  t3 := a[t2]  (B2)  and  x := a[t6]  (B5)   a[t6] -> t3
4*j    in  t4 := 4*j    (B3)  and  t8 := 4*j   (B5)   t8    -> t4
a[t4]  in  t5 := a[t4]  (B3)  and  t9 := a[t8] (B5)   t9    -> t5
```

can be removed from block B5' to produce block B5" (note that B5 can only be entered after going through B2, B3 and B4).

```
i := i+1              B2
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2
```

```
j := j-1              B3
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
```

```
t6 := 4*i             B5'
x := a[t6]
t8 := 4*j
t9 := a[t8]
a[t6] := t9
a[t8] := x
goto B2
```

```
                      B5''
x := t3


a[t2] := t5
a[t4] := x
goto B2
```


Common SEs can be eliminated from block B6 to produce an equivalent B6'.

```
t11:= 4*i             B6
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x
```

```
                      B6'
x := t3


t14 := a[t1]
a[t2] := t14

a[t1] := x
```

Notice that `a[t1]` in `B1` and `B6'` is not common. Why?

```
i := m-1              B1
j := n
t1:= 4*n
v := a[t1]
```

```
x := t3               B6'
t14 := a[t1]
a[t2] := t14
a[t1] := x
```

Although `a[t1]` occurs in `v := a[t1]` in `B1` and `t14 := a[t1]` in `B6`, after control leaves `B1` and before it enters `B6`, it can pass through `B5` where there are assignments to `a` in `a[t2] := t5` and `a[t4] := x`.

Elimination of common SEs has reduced the code from 30 to 21 statements.

Copy Propagation & Dead-code Elimination

Copy propagations do not reduce code size; they are used in locating dead-code transformations which reduce code size.

A statement of the form `x := y` is called a **copy statement**.

In a local copy propagation, all future references to the LHS variable of a copy statement are replaced by references to the RHS variable.

Example, a copy transformation applied to `B5''` to produce `B5'''`.

```
x := t3          B5''
a[t2] := t5
a[t4] := x
goto B2
```

```
x := t3          B5'''
a[t2] := t5
a[t4] := t3
goto B2
```

A statement, `x := y + z` is said to define `x` and to use (reference) `y` and `z`.

A name is **live** at a point if its value is used anywhere after that point. A name is **dead** if it is not live.

If `x` is dead at the point where an assignment statement to `x` appears, then the statement can be eliminated (this requires global view).

For example, `x` is dead in the first statement of `B5'''` and can be eliminated.

```
x := t3          B5'''
a[t2] := t5
a[t4] := t3
goto B2
```

```
                 B5''''
a[t2] := t5
a[t4] := t3
goto B2
```

A similar copy propagation and dead-code elimination removes the first statement of `B6'` to produce `B6''`.

```
x := t3          B6'
t14 := a[t1]
a[t2] := t14
a[t1] := x
```

```
                 B6''
t14 := a[t1]
a[t2] := t14
a[t1] := t3
```

Copy and dead-code transformations have eliminated 2 of the remaining 21 statements.

Algebraic Transformations

An algebraic transformation changes the set of expressions computed in a basic block into an algebraically equivalent set. Examples.

- **simplification** transformations, examples
  - Removal of statements `x := x+0` and `x := x*1`.
  - Constant folding, transforming `x := y+2+3` to `x := y+5`.
- **commutativity** and **associativity** transformations
  - `x := 2+y+3` transforms to `x := y+2+3` and then to `x := y+5`.
  - The following are equivalent

```
a := b + c                    a := b + c
t := c + d
e := t + b                    e := a + d
```

- **strength reduction** transformations

  Replacement of `x := y**2` by `x := y*y`, or `x := 2*y` by `x := y+y` with the assumption that the first statement requires longer to execute than the second (this is not always true).

  A more interesting example of strength reduction later.

Flow Graphs and Loops

To perform global optimizations we introduce a notation for control flow.

We construct a **flow graph** whose nodes are basic blocks of the program and whose directed edges represent the control flow.

The **initial node** is the block whose leader is the first statement.

There is a directed edge from block B_i to block B_j if either:

- The last statement of B_i branches to the first statement of B_j.
- B_j immediately follows B_i in the program and B_i does not end in an unconditional branch.

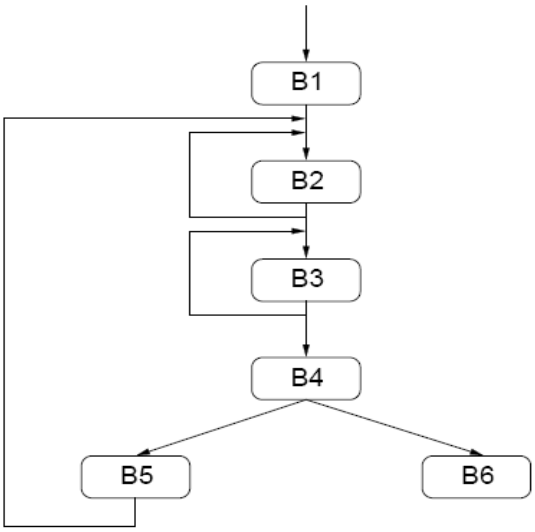
A loop is a collection of nodes in a flow graph such that

- All nodes in the collection are strongly connected.
- The collection of nodes has a unique entry from outside.

Any directed edge represents a possible flow of control, not a required flow.

A loop that contains no other loops as a sub-collection is called an inner loop.

The flow graph for our initial code:



There are three loops

- B2
- B3
- B2-B3-B4-B5

Loop Optimizations

The most common techniques: code motion and induction variable elimination.

A **loop invariant** is an expression which occurs inside of a loop and whose value is independent of the number of times the loop is executed.

Note. *Do not confuse this loop invariant with a different notion of a loop invariant used when proving properties of loops.*

Code motion is the movement of loop invariants to outside of the loop.

A compiler might produce the following blocked three-address code

```
while (i <= k-2) do
        i := i+j
od
```

```
t1 := k-2              B6
if i > t1 goto B7
i := i+j
goto B6

     ...               B7
```

Since the computation of `t1` in `t1 := k-2` is independent of the number of times the loop `B6` is executed, it can be removed either into the preceding block, `B5`, or a new block, `B5.5`.

A variable is an **induction variable** if its value in the loop is related to another variable in a way independent of the number of loop executions. An induction variable can be removed from a loop when it can be expressed by another induction variable.

Computation of an induction variable may be simplified using a strength reduction transformation. Consider

```
j := j-1                B3
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
-------------------------
if i > = j goto B6      B4
```

`j` and `t4` are induction variables: `t4 = 4*j` holds after assignment to `t4` in `B3`.

Unfortunately, neither `j` nor `t4` can be eliminated from `B3` since `t4` is used in `B3` and `j` is used in `B4`.

However, we can apply a strength reduction transformation and replace `t4 := 4*j` with the less "expensive" `t4 := t4-4`.

If `t4 = 4*j` at the end of the loop and we execute the loop once more then after `j := j-1`, we have to assure that `t4 := 4*j-4`.

We can replace `t4 := 4*j` by `t4 := t4 - 4`.

We need to add an initialization statement, `t4 := 4*j` before `B3` is executed for the first time. We can place this statement in `B1` just after `j` is initialized

```
i := m-1          B1
j := n
t1:= 4*n
v := a[t1]
t4 := 4*j
```

```
j := j-1          B3
t4 := t4 - 4
t5 := a[t4]
if t5 > v goto B3
```

The replacement of multiplication by subtraction (in a loop) will speed up the execution time on most machines (on some it will not), at the expense of the time to execute the extra instruction once.

(Note that `4*x` is typically implemented by fast bit shifting).

Similarly, `i` and `t2` are induction variables in `B2` and a similar change can be made in blocks `B2` and `B1`.

Now, consider the outer loop of B2-B3-B4-B5. We have two pairs of induction variables, (i, t2) and (j, t4). Since the only uses of i and j are in the statement

```
if i >= j goto B6      B4
```

this statement can be replaced by

```
if t2 >= t4 goto B6
```

and the variables i and j can be eliminated from the loop.

This completes the optimization and we get:

We started with 30 statements and ended up with 19. More importantly, the length of blocks in loops decreased. B1 is now longer but it is executed only once.

```
i := m-1                    B1
j := n
t1:= 4*n
v := a[t1]
t2 := 4*i
t4 := 4*j
-------------------------------
t2 := t2 + 4                B2
t3 := a[t2]
if t3 < v goto B2
-------------------------------
t4 := t4 - 4                B3
t5 := a[t4]
if t5 > v goto B3
-------------------------------
if t2 >= t4 goto B6         B4
-------------------------------
a[t7] := t5                 B5
a[t10] := t3
goto B2
-------------------------------
t14 := a[t11]               B6
a[t2] := t14
a[t1] := t3
```


Explain this: The code from a Pascal compiler for

for i := 1 **to** n **do** x[i] = 1

runs 48.4 μs, while the code for

for i := 1 **to** n **do** x[i] = 1.0

runs 5.4 μs.

Why?

Peephole Optimization

Peephole optimization is a simple multi-pass technique for improving the local structure of intermediate or machine code in which a small moving window is used to view the code. Some typical uses:

- **redundant instruction elimination**
  Consider a register machine instruction sequence

```
MOV     R0, a
MOV     a, R0
```

  in which the second instruction can be deleted or the ASC sequence

```
PUSHA   3[1]    push addr: 3rd var of level 1
PUSHI   0       dereference top of stack
```

  which can be replaced by:

```
PUSH    3[1]    push cont: 3rd var of level 1.
```

- **flow-of-control optimizations**
  - *elimination of jumps over jumps*
  - *removal of unreachable code*

For example, assume that the high-level code:

```
        #define debug 0
. . .
        if (debug) {
                print debug information
        }
```

produces the following intermediate code:

```
                if debug = 1 goto L1
                goto L2
        L1:     print debug information
        L2:
```

On the first pass, the jump over jump could be removed:

```
                if debug != 1 goto L2
                print debug information
        L2:
```

Do constant folding by transforming `0 != 1` to `true` and then `if` would be transformed to `goto L2` yielding

```
                goto L2
                print debug information
        L2:
```

Now, all of the print debug information statements could be eliminated, as well as the jump itself and there would be no statements left.

– *simplification of jumps to jumps*

```
            goto L1
        . . .
        L1: goto L2
```

can be replaced by

```
            goto L2
        . . .
        L1: goto L2
```

which is more efficient.

- **algebraic simplifications**

  remove statements like `x := x+0` or `x := x*1`

- **special machine characteristics**

  Replace machine instructions by more powerful ones. For example, standard intermediate code for the high-level statement `i := i+1` could be replaced by code using an auto-increment addressing mode.

What is going on here?

```
unsigned int i;
...
for (i = N; i >= 0; i--)
  a += i;
```

And here the real puzzle. In some old gcc

```
unsigned int i;
...
for (i = N; i >= 0; i--)
  X[i] = 0;
```

No optimization: crashes. Why?

With optimization works fine.

```
v1 = &X[N];
v2 = &X[0];
for (v1 = v1; v1 >= v2; v1 = v1-4)
  *v1 = 0;
```

One thing to remember: When a program uses doubles, after optimization we do not get an equivalent program. Machine arithmetic is just different than what we are used to.