Code optimization

Have we achieved optimal code? Impossible to answer!

We make improvements to the code.

Aim: faster code and/or less space

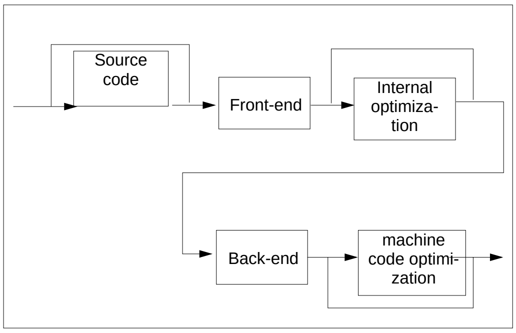
Types of optimization

- machine-independent

  In source code or internal form.

- machine dependent

  In the object program (peephole-optimization).



Code transformations

What are the disadvantages?

- Debugging is made difficult because of code optimization (e.g. moving code around). Important to be able to switch off optimization.
- The compiler runs more slowly.
- Unpleasant effects!
  Example:

  `A:=B*C-D+E` ⇒ `A:=B*C+E-D`

  Can lead to *overflow*.

The effects of optimization:

- Register use and choice of instruction
- Inner loops (locality, 90-10 rule: 90% of the time goes on 10% of the code).

Types of optimizations

1. Algorithm optimization
   - Replace a slow algorithm with a quicker one e.g.

     bubble sort ⇒ quick sort.

   - Poor algorithms are the main source of inefficiency but difficult to optimize.
   - Machine and compiler independence.
2. Intermediate code optimization
   - Performed on intermediate code
   - Examples of optimizations:
     - Local optimization, within *basic blocks*.
     - Loop optimization
     - Address calculations on arrays and records
     - *Global* optimization
     - *Inter-procedural* optimization
   - Compiler-dependent but machine-independent.
3. *Peephole* optimization
   - Transformations performed on machine code
   - machine-dependent

Basic block

A *basic block* is a sequence of operations with an entry and an exit.

No jump instructions may appear within the block (except for the very last instruction).

Exercise:

Divide the quadruples on the enclosed paper into groups of *basic blocks* and provide the corresponding control flow graph.

Local optimization

Is performed within a basic block without information from any other block.

Example:

1. *Constant folding*

   Constant expressions are calculated during compilation.

   Example

```
const NN = 4;
...
i:=2+NN;     (* ⇒ i := 6 *)
j:=i*5+a;    (* ⇒ j := 30 + a *)
             (* constant propagation *)
```

   Example:: Constant folding if we know that `b = -1`

```
a:=5+b*c-3+d   ⇒   a:=2-c+d
```


2. Elimination of common sub-expressions

   Example:

```
A[I+1] := B[I+1]
```

   is transformed to

```
T := I+1;
A[T] := B[T];
```

   Example:

```
D := D + C*B;
A := D + C*B;
```

   is transformed to:

```
T := C*B;
D := D + T;
A := D + T;
```


3. *Reduction in strength*

   Replace an expensive operation by a cheaper one.

   Example:

```
x:=y**2   ⇒   x:=y*y

x:=2.0*y  ⇒   x:=y+y
```

   Example: Concatenation in Snobol

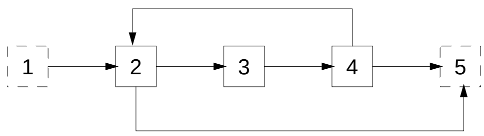
```
L := Length(S1||S2)

        ⇓

L := Length(S1)+Length(S2)
```


Loop optimization

- Loop optimization aims to minimise the time spent in the loop, often by reducing the number of operations in the loop.

Control flow graph:

A number of nodes and edges where the nodes are basic blocks and edges are jumps.



Definition: **loop**

A number of nodes which

1. are *strongly connected*, i.e. all nodes in a loop can be reached from the others.
2. has a **unique** entry.

The loop in the diagram is {2, 3, 4}

Example of loop optimizations:

1. Move loop invariants

   Example:

```
for i := 1 to 10 do  begin
        z := i + b/c
        .
        .
end;
```

   can be rewritten

```
t := b/c;
for i := 1 to 10 do begin
        z := i + t;
        .
        .
end;
```


2. Elimination of induction variables

   Has the greatest effect on the intermediate form.
   Can remove variables which makes debugging more difficult:

   ”What is the value of I?”

```
debug> I=
  "Excuse me, optimizer killed me."
```


3. *Loop unrolling*

   Example:

```
i:=1;
while (i<=50) do begin
   a[i]:= b[i];
   i:= i + 1
end;
```

   can be written as

```
i:=1;
while (i<=50) do begin
   a[i]:= b[i];
   i:= i + 1;
   a[i]:= b[i];
   i:= i + 1;
end;
```

   Reduce the number of tests and jumps by doubling the code:

   \+ more efficient in time
   \- increased memory load

4. Loop fusion

   Merge several loops to one loop:

```
for i:= 1 to n do
    for j:= 1 to m  do
         a[i,j]:= 1;
```

   can be written as

```
 for i:=1 to n*m do
{at the internal form level }
     a[i]:= 1;
```

Global data flow analysis: on whole procedures

A higher level of optimization can be achieved if the whole procedure is analysed.

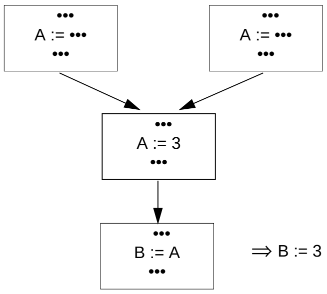
(Inter-procedural analysis deals with the whole program)

Concepts:

Definition: `A := 5` `A` is defined
Use: `B := A*C` `A` is used

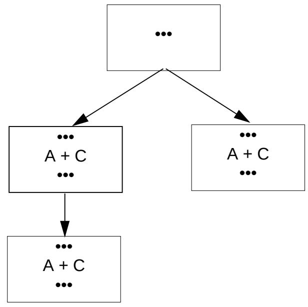
The analysis is performed in two phases:

1. **Forwards**
   Reaching definitions
   Which definitions apply at a point *p* in a flow graph?



Available expression

For example, to manage to eliminate common sub-expressions over block borders.



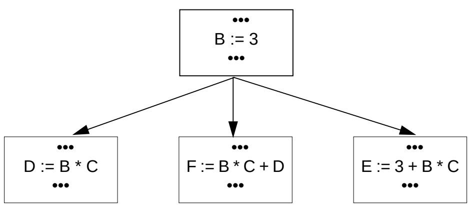
2. **Backward analysis**

   Live variables
   A variable `v` is *live* at point *p* if its value is used after *p* before any new definition of `v` is made.

```
  .
  .
v := A;
 ...           ← is there a new definition of v?
C := v;
```

Example: If variable `A` is in a register and it is dead (will not be referenced) the register can be released.

Very busy expressions
An expression is *very busy* if all paths from the expression use it later in the program.



Global data flow analysis provides optimization:

1. Remove variables which are never referenced.
2. Do not make calculations whose results are not used.
3. Remove code which is not called or reachable (*dead code elimination*).
4. *Code motion*
5. Find uninitialised variables

Examples of other machine-independent optimizations

1. Array-references

   ```
   C := A[I,J] + A[I,J+1]
   ```

   Elements are beside each other in memory.
   Ought to be "*give me the next element*".

2. Expand the code for small routines

   `x := sqr(y)` ⇒ `x := y * y`

3. Short-circuit evaluation of tests

   ```
   while (a > b) and (c-b < k) and ...
          ↑
   ```

   if `false` the rest does not need to be evaluated.

4. Fixed references

   ```
   A[3,5] := B[1,5,4]
   ```

   Calculate the addresses during compilation as if:

   ```
   C := D
   ```

5. Exploit algebraic manipulations

   `K := -C*(B-A)` ⇒ `K := C*(A-B)`

   Eliminate multiplication by `1` and addition with `0`.

Machine-dependent optimization: Peephole optimization

- We have a window of 3-4 instructions.
- We try to optimize within the window and then move the window one instruction forwards.
- Several passes over the code are often required.

```
____________
____________
____________
MOV A, R0
ADD 1,R0
MOV R0,A
MOV A,R0
ADD 2,R0
```

Redundant `load` and `store` instructions

```
MOV R0,A
MOV A,R0
```

equivalent to `MOV R0,A`

Recognize a pattern which is appropriate for a special instruction

```
MOV A,R0
ADD 1,R0
MOV R0,A
```

equivalent to

```
INC A
```

Algebraic simplifications

`ADD 0,R0` Eliminate!
`MUL 1,R0` Eliminate!

Reduction in strength

```
MUL 2,R0
```

are written

```
SHIFT 1,R0
```

Improvement of jump over jump

```
      GOTO  L1    { is changed to GOTO L2 }
        •
        •
        •
L1:   GOTO  L2
```