Iteration Variable:

\( i \) is an iteration variable if the only definition of \( i \) inside the loop occurs at an edge which separates the body and is of the form:

\[
i = i + h;
\]

for some loop constant \( h \).

A loop constant is simply a constant (e.g., 42), or slightly more liberal, an expression which only depends on variables which are not modified during the loop :-)
(3) Differences for Sets

Consider the fixpoint computation:

\[
x = \emptyset; \\
\text{for } (t = F x; t \nsubseteq x; t = F x; ) \\
x = x \cup t;
\]

If \( F \) is distributive, it could be replaced by:

\[
x = \emptyset; \\
\text{for } (\Delta = F x; \Delta \neq \emptyset; \Delta = (F \Delta) \setminus x; ) \\
x = x \cup \Delta;
\]

The function \( F \) must only be computed for the smaller sets \( \Delta \) :-) semi-naive iteration
Instead of the sequence:  \( \emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots \)
we compute:  \( \Delta_1 \cup \Delta_2 \cup \ldots \)
where:
\[
\Delta_{i+1} = F(F^i(\emptyset)) \setminus F^i(\emptyset)
= F(\Delta_i) \setminus (\Delta_1 \cup \ldots \cup \Delta_i) \quad \text{with} \quad \Delta_0 = \emptyset
\]

Assume that the costs of  \( F \times \) is  \( 1 + \#x \).
Then the costs may sum up to:

<table>
<thead>
<tr>
<th>naive</th>
<th>1 + 2 + \ldots + n + n = ( \frac{1}{2}n(n + 3) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>semi-naive</td>
<td>2n</td>
</tr>
</tbody>
</table>

where  \( n \)  is the cardinality of the result.

\[\Rightarrow\]  A linear factor is saved  :-)
2.2 Peephole Optimization

Idea:

- Slide a small window over the program.
- Optimize aggressively inside the window, i.e.,
  -> Eliminate redundancies!
  -> Replace expensive operations inside the window by cheaper ones!
Examples:

\[ y = M[x]; x = x + 1; \quad \implies \quad y = M[x++] ; \]
\[
// \quad \text{given that there is a specific post-increment instruction} \quad :-) 
\]
\[ z = y - a + a; \quad \implies \quad z = y; \]
\[
// \quad \text{algebraic simplifications} \quad :-) 
\]
\[ x = x; \quad \implies \quad \;
\]
\[ x = 0; \quad \implies \quad x = x \oplus x; 
\]
\[ x = 2 \cdot x; \quad \implies \quad x = x + x; 
\]
Important Subproblem: \textit{nop}-Optimization

\[ (v_1, ;, v) \text{ is an edge}, \quad v_1 \text{ has no further out-going edge.} \]

\[ \text{Consequently, we can identify } v_1 \text{ and } v :-(:) \]

\[ \text{The ordering of the identifications does not matter } :-(:)) \]
Implementation:

- We construct a function \( \text{next} : \text{Nodes} \rightarrow \text{Nodes} \) with:

\[
\text{next } u = \begin{cases} 
\text{next } v & \text{if } (u, ;, v) \text{ edge} \\
 u & \text{otherwise}
\end{cases}
\]

**Warning:** This definition is only recursive if there are \( ; \)-loops.

- We replace every edge:

\[
(u, lab, v) \implies (u, lab, \text{next } v)
\]

... whenever \( lab \neq ; \)

- All \( ; \)-edges are removed \( ;-) \)
Example:

\[
\begin{align*}
\text{next } 1 &= 1 \\
\text{next } 3 &= 4 \\
\text{next } 5 &= 6
\end{align*}
\]
Example:

\[
\begin{align*}
\text{next } 1 &= 1 \\
\text{next } 3 &= 4 \\
\text{next } 5 &= 6
\end{align*}
\]
2. Subproblem: Linearization

After optimization, the CFG must again be brought into a linearly arranged arrangement of instructions :-)

Warning:

Not every linearization is equally efficient !!!
Example:

```
0:  
1:  if (e_1) goto 2; 
2:  goto 1; 
3:  if (e_2) goto 4; 
4:  halt
```

**Bad:** The loop body is jumped into  :-(
Example:

0:
1: if (!e_1) goto 4;
2: Rumpf
3: if (!e_2) goto 1;
4: halt

// better cache behavior  :-)
Idea:

- Assign to each node a temperature!
- always jumps to
  
  (1) nodes which have already been handled;
  
(2) colder nodes.

- Temperature $\approx$ nesting-depth

For the computation, we use the pre-dominator tree and strongly connected components ...
... in the Example:

The sub-tree with back edge is hotter ...
... in the Example:
More Complicated Example:
More Complicated Example:
More Complicated Example:
Our definition of Loop implies that (detected) loops are necessarily nested :-) 

It is also meaningful for do-while-loops with breaks ...
Our definition of \textbf{Loop} implies that (detected) loops are necessarily nested :-

Is is also meaningful for \texttt{do-while}-loops with \texttt{breaks} ...
Summary: The Approach

(1) For every node, determine a temperature;

(2) Pre-order-DFS over the CFG;

→ If an edge leads to a node we already have generated code for, then we insert a jump.

→ If a node has two successors with different temperature, then we insert a jump to the colder of the two.

→ If both successors are equally warm, then it does not matter ;-)

→ At a join point, detect whether the current edge is a back edge (corresponding to a loop) or a cross edge (corresponding to a conditional).
2.3 Procedures

We extend our mini-programming language by procedures without parameters and procedure calls.

For that, we introduce a new statement:

\[ f(); \]

Every procedure \( f \) has a definition:

\[ f() \{ \text{stmt*} \} \]

Additionally, we distinguish between global and local variables.

Program execution starts with the call of a procedure \( \text{main}() \).
Example:

```c
int  a, ret;
main() {
    a = 3;
f();
M[17] = ret;
ret = 0;
}

f() {
    int  b;
    if (a ≤ 1) {ret = 1; goto exit; }
b = a;
a = b − 1;
f();
ret = b · ret;
exit :
}
```

Such programs can be represented by a set of CFGs: one for each procedure ...
... in the Example:

```c
int main() {
    int a = 3;
    f();
    M[17] = ret;
    ret = 0;
    ret = 1;
    f();
    a = b - 1;
    f();
    ret = b * ret;
    f();
}
```

```

```
In order to optimize such programs, we require an extended operational semantics ;-) 

Program executions are no longer paths, but forests: