Important Subproblem: \textit{nop}-Optimization

\[ (v_1, :, v) \] is an edge, \( v_1 \) has no further out-going edge.

Consequently, we can identify \( v_1 \) and \( v \) :-)

The ordering of the identifications does not matter :-))
Implementation:

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

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

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

- We replace every edge:

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

... whenever \( \text{lab} \neq ; \)

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

\[
\begin{align*}
\text{next } 1 & \quad = \quad 1 \\
\text{next } 3 & \quad = \quad 4 \\
\text{next } 5 & \quad = \quad 6
\end{align*}
\]
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: Rumpf 
4: halt 
3: if $(e_2)$ goto 4; 
goto 1; 

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 \text{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 :-) Is it also meaningful for do-while-loops with breaks ...
Our definition of Loop implies that (detected) loops are necessarily nested :-)
Is is also meaningful for do-while-loops with 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 ;-)
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:

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

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
main()

a = 3;
f();
M[17] = ret;
ret = 0;

f();
a = b - 1;
ret = b * ret;
```

```
f()
Neg (a ≤ 1)
  b = a;
  a = b - 1;
  f();
  ret = b * ret;

Pos (a ≤ 1)
  ret = 1;
```
In order to optimize such programs, we require an extended operational semantics ;-) 

Program executions are no longer paths, but forests:
... in the Example:
The function $\llbracket . \rrbracket$ is extended to computation forests:

$$w : \llbracket w \rrbracket : (Vars \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z}) \to (Vars \to \mathbb{Z}) \times (\mathbb{N} \to \mathbb{Z})$$

For a call $k = (u, f(); v)$ we must:

- determine the initial values for the locals:

$$\text{enter } \rho = \{ x \mapsto 0 \mid x \in Locals \} \oplus (\rho|_{Global})$$

- ... combine the new values for the globals with the old values for the locals:

$$\text{combine } (\rho_1, \rho_2) = (\rho_1|_{Locals}) \oplus (\rho_2|_{Global})$$

- ... evaluate the computation forest inbetween:

$$\llbracket k \langle w \rangle \rrbracket (\rho, \mu) = \text{let } (\rho_1, \mu_1) = \llbracket w \rrbracket (\text{enter } \rho, \mu) \text{ in } (\text{combine } (\rho, \rho_1), \mu_1)$$
Warning:

- In general, $\llbracket w \rrbracket$ is only partially defined :-)
- Dedicated global/local variables $a_i, b_i, \text{ret}$ can be used to simulate specific calling conventions.
- The standard operational semantics relies on configurations which maintain a call stack.
- Computation forests are better suited for the construction of analyses and correctness proofs :-)
- It is an awkward (but useful) exercise to prove the equivalence of the two approaches ...
Configurations:

\[
\begin{align*}
\text{configuration} & = stack \times store \\
\text{store} & = globals \times (\mathbb{N} \rightarrow \mathbb{Z}) \\
\text{globals} & = (\text{Globals} \rightarrow \mathbb{Z}) \\
\text{stack} & = \text{frame} \cdot \text{frame}^* \\
\text{frame} & = \text{point} \times \text{locals} \\
\text{locals} & = (\text{Locals} \rightarrow \mathbb{Z})
\end{align*}
\]

A \textit{frame} specifies the local state of computation inside a procedure call :-)

The \textcolor{red}{\textbf{leftmost}} frame corresponds to the current call.
Computation steps refer to the current call  :-)

The novel kinds of steps:

\[
\text{call } k = (u, f(); v) : \quad \left( (u, \rho) \cdot \sigma, \langle \gamma, \mu \rangle \right) \quad \Rightarrow \quad \left( (u_f, \{x \to 0 \mid x \in \text{Locals}\}) \cdot (v, \rho) \cdot \sigma, \langle \gamma, \mu \rangle \right)
\]

\(u_f\) entry point of \(f\)

\[
\text{return:} \quad \left( (r_f, \_ ) \cdot \sigma, \langle \gamma, \mu \rangle \right) \quad \Rightarrow \quad (\sigma, \langle \gamma, \mu \rangle)
\]

\(r_f\) return point of \(f\)
The call stack explicitly implements the DFS traversal through the computation forest  :-)

... in the Example:
The call stack explicitly implements the DFS traversal through the computation forest  :-)

... in the Example:
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:
The call stack explicitly implements the DFS traversal through the computation forest  :-)

... in the Example:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>b</td>
<td>⇝</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>⇝</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

```
<table>
<thead>
<tr>
<th></th>
<th>b ➝ 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b ➝ 3</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```
The call stack explicitly implements the DFS traversal through the computation forest  

... in the Example:

| 5 | b ← 0 |
| 9 | b ← 2 |
| 9 | b ← 3 |
| 2 |      |
The call stack explicitly implements the DFS traversal through the computation forest  

... in the Example:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>b → 0</td>
</tr>
<tr>
<td>9</td>
<td>b → 2</td>
</tr>
<tr>
<td>9</td>
<td>b → 3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:

<table>
<thead>
<tr>
<th>9</th>
<th>b ↦ 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>b ↦ 3</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
The call stack explicitly implements the DFS traversal through the computation forest  :-) 

... in the Example: 

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>$b \mapsto 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$b \mapsto 3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The call stack explicitly implements the DFS traversal through the computation forest  :-)

... in the Example:

<table>
<thead>
<tr>
<th>9</th>
<th>b → 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:
The call stack explicitly implements the DFS traversal through the computation forest :-)

... in the Example:
This operational semantics is quite realistic :-) 

Costs for a Procedure Call:

Before entering the body:  • Creating a stack frame;
  • assigning of the parameters;
  • Saving the registers;
  • Saving the return address;
  • Jump to the body.

At procedure exit:  • Freeing the stack frame.
  • Restoring the registers.
  • Passing of the result.
  • Return behind the call.

⇒ ... quite expensive !!!
1. Idea: **Inlining**

Copy the procedure body at every call site !!!

Example:

```c
abs () { 
    a_2 = -a_1;  
    max ();  
} 

max () {  
    if (a_1 < a_2) {  
        ret = a_2;  
        goto _exit;  
    }  
    ret = a_1;  
} 

_exit :  
}
```
... yields:

```c
abs () {
    a2 = −a1;
    if (a1 < a2) { ret = a2; goto _exit; }
    ret = a1;
    _exit :
}
```
Problems:

- The copied block may modify the locals of the calling procedure
- More general: Multiple use of local variable names may lead to errors.
- Multiple calls of a procedure may lead to code duplication :-(
- How can we handle recursion ???