... in the Example:

main()

0

\[ t = 0; \]

1

Neg \( t \)  \quad Pos \( t \)

2

\[ M[17] = 3; \]

3

\[ a_1 = t; \]

4

work();

5

work();

6

\[ \text{ret} = 1 - \text{ret}; \]

7

work();

8

Neg \( a_1 \)  \quad Pos \( a_1 \)

9

\[ \text{ret} = a_1; \]

10
... in the Example:

```
main()

0

\[ t = 0; \]

1

\[ M[17] = 3; \]

2

\[ a_1 = t; \]

3

4

\[ \text{combine} \]

5

6

\[ \text{ret} = 1 - \text{ret}; \]

\[ \text{ret} = a_1; \]

7

8

\[ \text{Pos} (a_1) \]

9

\[ \text{Neg} (a_1) \]

10

\[ \text{combine} \]

\[ \text{enter} \]
```

575
The conditions for $5, 7, 10$, e.g., are:

\[ R[5] \supseteq \text{combine}^\# (R[4], R[10]) \]
\[ R[7] \supseteq \text{enter}^\# (R[4]) \]
\[ R[7] \supseteq \text{enter}^\# (R[8]) \]
\[ R[9] \supseteq \text{combine}^\# (R[8], R[10]) \]

**Warning:**

The resulting super-graph contains obviously impossible paths ...
... in the Example this is:

main()

0

$\textit{\textbf{t}} = 0$; 

1

Neg ($t$) 

Pos ($t$)

2

$M[17] = 3$; 

3

$a_1 = t$; 

4

5

6

ret = 1 - ret;

7

work ()

Neg ($a_1$) 

Pos ($a_1$)

8

9

ret = $a_1$;

10

combine

combine

enter

enter
... in the Example this is:

\begin{align*}
\text{main()} &= \quad t = 0; \\
\text{Neg}(t) &= \quad \text{Pos}(t) \\
\text{Neg}(a_1) &= \quad \text{Pos}(a_1) \\
M[17] &= 3; \\
a_1 &= t; \\
\text{combine} &= \quad \text{combine} \\
\text{ret} &= a_1; \\
\text{combine} &= \quad \text{combine} \\
\text{ret} &= 1 - \text{ret}; \\
\end{align*}
Note:

→ In the example, we find the same results: more paths render the results less precise.

In particular, we provide for each procedure the result just for one (possibly very boring) argument :-(

→ The analysis terminates — whenever $D$ has no infinite strictly ascending chains :-)

→ The correctness is easily shown w.r.t. the operational semantics with call stacks.

→ For the correctness of the functional approach, the semantics with computation forests is better suited :-)
3 Exploiting Hardware Features

Question: How can we optimally use:

... Registers
... Pipelines
... Caches
... Processors ???
3.1 Registers

Example:

```plaintext
read();
x = M[A];
y = x + 1;
if (y) {
    z = x \cdot x;
    M[A] = z;
} else {
    t = -y \cdot y;
    M[A] = t;
}
```
The program uses 5 variables ...

**Problem:**

What if the program uses more variables than there are registers  :-(

**Idea:**

Use one register for *several* variables  :-)  
In the example, e.g., one for  $x, t, z ...$
read();
\[ x = M[A]; \]
\[ y = x + 1; \]
if (y) {
  \[ z = x \cdot x; \]
  \[ M[A] = z; \]
} else {
  \[ t = -y \cdot y; \]
  \[ M[A] = t; \]
}
read();

\( R = M[A] \);

\( y = R + 1 \);

if (y) {

\[ R = R \cdot R; \]

\[ M[A] = R; \]

} else {

\[ R = -y \cdot y; \]

\[ M[A] = R; \]

}
Warning:

This is only possible if the live ranges do not overlap.

The (true) live range of $x$ is defined by:

$$\mathcal{L}[x] = \{ u \mid x \in \mathcal{L}[u] \}$$

... in the Example:
read();

1

x = M[A];

2

y = x + 1;

3

Neg (y) Pos (y)

4

t = −y · y;

5

M[A] = t;

6

z = x · x

7

M[A] = z;

8

L

<table>
<thead>
<tr>
<th></th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>∅</td>
</tr>
<tr>
<td>7</td>
<td>{A, z}</td>
</tr>
<tr>
<td>6</td>
<td>{A, x}</td>
</tr>
<tr>
<td>5</td>
<td>{A, t}</td>
</tr>
<tr>
<td>4</td>
<td>{A, y}</td>
</tr>
<tr>
<td>3</td>
<td>{A, x, y}</td>
</tr>
<tr>
<td>2</td>
<td>{A, x}</td>
</tr>
<tr>
<td>1</td>
<td>{A}</td>
</tr>
<tr>
<td>0</td>
<td>∅</td>
</tr>
</tbody>
</table>
read();

\[ x = M[A]; \]

\[ y = x + 1; \]

\[ z = x \cdot x \]

\[ t = -y \cdot y; \]

\[ M[A] = t; \]

\[ M[A] = z; \]

\[ \text{Neg}(y) \]

\[ \text{Pos}(y) \]

<table>
<thead>
<tr>
<th>( L )</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
read();

\[ x = M[A]; \]

\[ y = x + 1; \]

\[ t = -y \cdot y; \]

\[ z = x \cdot x \]

\[ M[A] = t; \]

\[ M[A] = z; \]

Live Ranges:

<table>
<thead>
<tr>
<th></th>
<th>( A )</th>
<th>( x )</th>
<th>( y )</th>
<th>( t )</th>
<th>( z )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{1, \ldots, 7}</td>
<td>{2, 3, 6}</td>
<td>{2, 4}</td>
<td>{5}</td>
<td>{7}</td>
</tr>
</tbody>
</table>
In order to determine sets of compatible variables, we construct the
Interference Graph $I = (\text{Vars}, E_I)$ where:

$$E_I = \{\{x, y\} \mid x \neq y, \mathcal{L}[x] \cap \mathcal{L}[y] \neq \emptyset\}$$

$E_I$ has an edge for $x \neq y$ iff $x, y$ are jointly live at some program point :-)

... in the Example:
read();

\[ x = M[A]; \]

\[ y = x + 1; \]

\[ z = x \cdot x \]

\[ t = -y \cdot y; \]

\[ M[A] = t; \]

\[ M[A] = z; \]

Interference Graph:
Variables which are not connected with an edge can be assigned to the same register :-(
Variables which are not connected with an edge can be assigned to the same register  :-)

Color  ===  Register
Sviatoslav Sergeevich Lavrov,
Russian Academy of Sciences  (1962)
Gregory J. Chaitin, University of Maine  (1981)
Abstract Problem:

Given: Undirected Graph \((V, E)\).

Wanted: Minimal coloring, i.e., mapping \(c : V \rightarrow \mathbb{N}\) mit

\[
(1) \quad c(u) \neq c(v) \text{ for } \{u, v\} \in E;
\]

\[
(2) \quad \bigcup\{c(u) \mid u \in V\} \text{ minimal!}
\]

- In the example, 3 colors suffice :-)
- But:
- In general, the minimal coloring is not unique :-(
- It is NP-complete to determine whether there is a coloring with at most \(k\) colors :-(

\[\Rightarrow\]

We must rely on heuristics or special cases :-)
Greedy Heuristics:

- Start somewhere with color 1;
- Next choose the smallest color which is different from the colors of all already colored neighbors;
- If a node is colored, color all neighbors which not yet have colors;
- Deal with one component after the other ...
... more concretely:

forall (v ∈ V) c[v] = 0;  
forall (v ∈ V) color (v);

void color (v) {
    if (c[v] ≠ 0) return;
    neighbors = {u ∈ V | {u, v} ∈ E};
    c[v] = ∏{k > 0 | ∀ u ∈ neighbors : k ≠ c(u)};
    forall (u ∈ neighbors)
        if (c(u) == 0) color (u);
}

The new color can be easily determined once the neighbors are sorted according to their colors  :-)

Discussion:

→ Essentially, this is a Pre-order DFS 😊
→ In theory, the result may arbitrarily far from the optimum 😞
→ ... in practice, it may not be as bad 😊
→ ... Anecdote: different variants have been patented !!!
Discussion:

→ Essentially, this is a Pre-order DFS  :-) 
→ In theory, the result may arbitrarily far from the optimum  :-(
→ ... in practice, it may not be as bad  :-)
→ ... Anecdote: different variants have been patented !!!

The algorithm works the better the smaller life ranges are ...

Idea: Life Range Splitting
Special Case: Basic Blocks

<table>
<thead>
<tr>
<th>Equation</th>
<th>( L )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 = x + y );</td>
<td>( x, y, z )</td>
</tr>
<tr>
<td>( M[A_1] = z );</td>
<td>( x, z )</td>
</tr>
<tr>
<td>( x = x + 1 );</td>
<td>( x )</td>
</tr>
<tr>
<td>( z = M[A_1] );</td>
<td>( x, z )</td>
</tr>
<tr>
<td>( t = M[x] );</td>
<td>( x, z, t )</td>
</tr>
<tr>
<td>( A_2 = x + t );</td>
<td>( x, z, t )</td>
</tr>
<tr>
<td>( M[A_2] = z );</td>
<td>( x, t )</td>
</tr>
<tr>
<td>( y = M[x] );</td>
<td>( y, t )</td>
</tr>
<tr>
<td>( M[y] = t );</td>
<td></td>
</tr>
</tbody>
</table>

Diagram: A directed acyclic graph with nodes labeled \( x, y, z, t \) and edges connecting them.
### Special Case: Basic Blocks

<table>
<thead>
<tr>
<th>( A_1 = x + y; )</th>
<th>( x, y, z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M[A_1] = z; )</td>
<td>( x, z )</td>
</tr>
<tr>
<td>( x = x + 1; )</td>
<td>( x )</td>
</tr>
<tr>
<td>( z = M[A_1]; )</td>
<td>( x, z )</td>
</tr>
<tr>
<td>( t = M[x]; )</td>
<td>( x, z, t )</td>
</tr>
<tr>
<td>( A_2 = x + t; )</td>
<td>( x, z, t )</td>
</tr>
<tr>
<td>( M[A_2] = z; )</td>
<td>( x, t )</td>
</tr>
<tr>
<td>( y = M[x]; )</td>
<td>( y, t )</td>
</tr>
<tr>
<td>( M[y] = t; )</td>
<td></td>
</tr>
</tbody>
</table>

![Graph Representation of Basic Blocks](image-url)
The live ranges of $x$ and $z$ can be split:

<table>
<thead>
<tr>
<th></th>
<th>$\mathcal{L}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1 = x + y;$</td>
<td>$x, y, z$</td>
</tr>
<tr>
<td>$M[A_1] = z;$</td>
<td>$x, z$</td>
</tr>
<tr>
<td>$x_1 = x + 1;$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$z_1 = M[A_1];$</td>
<td>$x_1, z_1$</td>
</tr>
<tr>
<td>$t = M[x_1];$</td>
<td>$x_1, z_1, t$</td>
</tr>
<tr>
<td>$A_2 = x_1 + t;$</td>
<td>$x_1, z_1, t$</td>
</tr>
<tr>
<td>$M[A_2] = z_1;$</td>
<td>$x_1, t$</td>
</tr>
<tr>
<td>$y_1 = M[x_1];$</td>
<td>$y_1, t$</td>
</tr>
<tr>
<td>$M[y_1] = t;$</td>
<td></td>
</tr>
</tbody>
</table>
The live ranges of $x$ and $z$ can be split:

\[
\begin{array}{|l|c|}
\hline
\text{ } & L \\
\hline
A_1 = x + y; & x, y, z \\
M[A_1] = z; & x, z \\
x_1 = x + 1; & x \\
z_1 = M[A_1]; & x_1 \\
t = M[x_1]; & x_1, z_1 \\
A_2 = x_1 + t; & x_1, z_1, t \\
M[A_2] = z_1; & x_1, t \\
y_1 = M[x_1]; & y_1, t \\
M[y_1] = t; & \\
\hline
\end{array}
\]
Interference graphs for minimal live ranges on basic blocks are known as interval graphs:

vertex = interval
edge = joint vertex
The covering number of a vertex is given by the number of incident intervals.

**Theorem:**

maximal covering number

\[ \text{size of the maximal clique} \]

\[ \text{minimally necessary number of colors} \quad :-) \]

Graphs with this property (for every sub-graph) are called *perfect* ...

A minimal coloring can be found in polynomial time \( :-) \)
Idea:

→ Conceptually iterate over the vertices $0, \ldots, m - 1$.
→ Maintain a list of currently free colors.
→ If an interval starts, allocate the next free color.
→ If an interval ends, free its color.

This results in the following algorithm:
free = [1, \ldots, k];

for (i = 0; i < m; i++) {
    init[i] = []; exit[i] = [];
}

forall (I = [u, v] \in \text{Intervals}) {
    init[u] = (I :: init[u]); exit[v] = (I :: exit[v]);
}

for (i = 0; i < m; i++) {
    forall (I \in \text{init}[i]) {
        color[I] = \text{hd free}; free = \text{tl free};
    }
    forall (I \in \text{exit}[i]) free = color[I] :: free;
}
Discussion:

→ For basic blocks we have succeeded to derive an optimal register allocation  

→ The same problem for simple loops (circular arc graphs) is already NP-hard  

→ For arbitrary programs, we thus may apply some heuristics for graph coloring ...

→ which always works better the less live ranges overlap  

→ If the number of real register does not suffice, the remaining variables are spilled into a fixed area on the stack.

→ Generally, variables from inner loops are preferably held in registers.