Contents of Lecture 8

- Purpose of Operator Strength Reduction, OSR
- Original algorithm for OSR
- The SSA Graph
- Strongly Connected Components
- Tarjan’s Algorithm for computing the strongly connected components
- OSR on SSA Form
double  a[N];

for (i = 0; i < N; ++i)
    x += a[i];

while (p < end)
    x += *p++;

- The most important purpose is to rewrite the code to the left into the code to the right.
- C/C++ compilers are required to make it possible to use the address of the array element after the last declared element.
- Typically, in total one extra byte might be wasted in memory due to this.
- It’s **not** one extra byte per array but rather per memory segment.
Invalid C Code

double a[N];

double* p = &a[N];

for (i = N-1; i >= 0; --i)
   x += a[i];

while (--p >= a)
   x += *p;

- In the last iteration $p == a[-1]$ in the comparison.
- The compiler is not required to make that address valid.
- The code to the right triggers undefined behavior.
OSR is also known as Induction Variable Elimination

```c
do {
    x = x + a[i];
    i = i + 1;
} while (i < N);
```

```c
do {
    s = i * 4;
    t = load a+s;
    x = x + t;
    i = i + 1;
} while (i < N);
```
The primary goal is to get rid of the multiplication

```c
do {
    s = i * 4;
    t = load a+s;
    x = x + t;
    i = i + 1;
} while (i < N);
```

- i is a *basic* induction variable
- Classes of *dependent* induction variables: \( j \leftarrow b \times i + c \), i is a basic IV
- s \leftarrow 4 \times i + 0

Classes of dependent induction variables:
- \( j \leftarrow b \times i + c \)
- \( s \leftarrow 4 \times i + 0 \)
do {
  s = i * 4;
  t = load a+s;
  x = x + t;
  i = i + 1;
} while (i < N);

s = 4 * i;

do {
  t = load a+s;
  x = x + t;
  i = i + 1;
  s = s + 4;
} while (i < N);

- Initialize the dependent IV before the loop
- Increment the dependent IV just after the basic IV is incremented
- Maybe we can get rid of the basic IV now?
Linear function test replacement

\[ s = 4 \times i; \]
\[ \text{do } \{ \]
\[ \quad t = \text{load a} + s; \]
\[ \quad x = x + t; \]
\[ \quad i = i + 1; \]
\[ \quad s = s + 4; \]
\[ \} \text{ while (i < N);} \]

\[ m = 4 \times N; \]
\[ s = 4 \times i; \]
\[ \text{do } \{ \]
\[ \quad t = \text{load a} + s; \]
\[ \quad x = x + t; \]
\[ \quad s = s + 4; \]
\[ \} \text{ while (s < m);} \]

- \[ s = i \times b + c \text{ (we have } b = 4 \text{ and } c = 0) \]
- \[ i = \frac{s-c}{b} \]
- \[ i < N \Rightarrow \frac{s-c}{b} < N \Rightarrow s < N \times b + c, \text{ if } b > 0 \]
procedure operator_strength_reduce(ssa_graph) 
    dfnum ← 0
    empty stack
    for each vertex v ∈ ssa_graph do
        visited(v) ← false
    for each vertex v ∈ ssa_graph do
        if (not visited(v))
            strong_connect(v)
    end
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
     dfn(v) ← dfnum
     lowlink(v) ← dfnum
     visited(v) ← true
     push(v)
     dfnum ← dfnum + 1

     for each w ∈ succ(v) do /* operands(v) = succ(v) */
          if (not visited(w)) {
               strong_connect(w)
               lowlink(v) ← min(lowlink(v), lowlink(w))
          } else if (dfn(w) < dfn(v) and w is on stack)
               lowlink(v) ← min(lowlink(v), dfn(w))

     if (lowlink(v) = dfn(v))
          scc ← ∅
          do
               w ← pop()
               add w to scc
               while (w ≠ v)
          end
          process_scc(scc)

end

int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
int dfnum

/* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
Tarjan's Algorithm: Initial Processing of 3

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
  dfn(v) ← dfnum
  lowlink(v) ← dfnum
  visited(v) ← true
  push(v)
  dfnum ← dfnum + 1
  for each w ∈ succ(v) do /* operands(v) = succ(v) */
    if (not visited(w)) {
      strong_connect(w)
      lowlink(v) ← min(lowlink(v), lowlink(w))
    } else if (dfn(w) < dfn(v) and w is on stack)
      lowlink(v) ← min(lowlink(v), dfn(w))
  if (lowlink(v) = dfn(v))
    scc ← ∅
    do
      w ← pop()
      add w to scc
      while (w ≠ v)
    process_scc(scc)
end
```

int \texttt{dfnum} \quad \text{/* Depth-first search number. */}

procedure \texttt{strong_connect(v)}
    \texttt{dfn(v) \leftarrow dfnum}
    \texttt{lowlink(v) \leftarrow dfnum}
    \texttt{visited(v) \leftarrow true}
    \texttt{push(v)}
    \texttt{dfnum \leftarrow dfnum + 1}

    for each w \in \text{succ(v)} do /* operands(v) = succ(v) */
        if (not \texttt{visited(w)})
            \texttt{strong_connect(w)}
            \texttt{lowlink(v) \leftarrow min(lowlink(v), lowlink(w))}
        else if (\texttt{dfn(w) < dfn(v) and w is on stack})
            \texttt{lowlink(v) \leftarrow min(lowlink(v), dfn(w))}

    if (\texttt{lowlink(v) = dfn(v)})
        \texttt{scc \leftarrow \emptyset}
        \texttt{do}
            \texttt{w \leftarrow pop()}
            \texttt{add w to scc}
        \texttt{while (w \neq v)}
        \texttt{process_scc(scc)}
    \texttt{end}

\begin{center}
\begin{tikzpicture}
    \node (0) at (0,0) {0, 0};
    \node (1) at (1,1) {1, 1};
    \node (2) at (2,2) {2, 2};
    \node (3) at (3,3) {3, 3};
    \node (4) at (4,4) {4, 4};

    \draw (0) -- (1);
    \draw (0) -- (2);
    \draw (0) -- (3);
    \draw (0) -- (4);
    \draw (1) -- (2);
    \draw (1) -- (3);
    \draw (1) -- (4);
    \draw (2) -- (3);
    \draw (2) -- (4);
    \draw (3) -- (4);

    \node[rectangle, draw, fill=white, thin, rounded corners] at (5,0) {stack};
    \node[rectangle, draw, fill=white, thin, rounded corners] at (5,1) {4};
    \node[rectangle, draw, fill=white, thin, rounded corners] at (5,2) {3};
    \node[rectangle, draw, fill=white, thin, rounded corners] at (5,3) {2};
    \node[rectangle, draw, fill=white, thin, rounded corners] at (5,4) {1};
    \node[rectangle, draw, fill=white, thin, rounded corners] at (5,5) {0};
\end{tikzpicture}
\end{center}
int \text{dfnum} /* Depth-first search number. */

procedure strong_connect(v)
    \text{dfn}(v) \leftarrow \text{dfnum}
    \text{lowlink}(v) \leftarrow \text{dfnum}
    \text{visited}(v) \leftarrow \text{true}
    \text{push}(v)
    \text{dfnum} \leftarrow \text{dfnum} + 1

    for each w \in \text{succ}(v) do /* operands(v) = succ(v) */
        if (not \text{visited}(w)) {
            strong_connect(w)
            \text{lowlink}(v) \leftarrow \min(\text{lowlink}(v), \text{lowlink}(w))
        } else if (\text{dfn}(w) < \text{dfn}(v) and w is on stack)
            \text{lowlink}(v) \leftarrow \min(\text{lowlink}(v), \text{dfn}(w))

    if (\text{lowlink}(v) = \text{dfn}(v))
        \text{scc} \leftarrow \emptyset
        do
            w \leftarrow \text{pop}()
            add w to \text{scc}
        while (w \neq v)
        process_scc(\text{scc})

end
**Tarjan’s Algorithm: Initial Processing of 6**

```plaintext
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```
(6, 2) \Rightarrow 6 \text{ in same scc as 2.}

int \ dfnum /* Depth-first search number. */

procedure \texttt{strong\_connect}(v)
\begin{align*}
\text{dfn}(v) &\leftarrow dfnum \\
\text{lowlink}(v) &\leftarrow dfnum \\
\text{visited}(v) &\leftarrow \text{true} \\
\text{push}(v) &\leftarrow \text{false} \\
\text{dfnum} &\leftarrow \text{dfnum} + 1
\end{align*}

for each \( w \in \text{succ}(v) \) do /* operands(v) = succ(v) */
\begin{align*}
\text{if (not visited}(w)) \{ \\
\text{strong\_connect}(w) \\
\text{lowlink}(v) &\leftarrow \text{min}(\text{lowlink}(v), \text{lowlink}(w)) \\
\} \text{ else if (dfn}(w) < \text{dfn}(v) \text{ and } w \text{ is on stack}) \\
\text{lowlink}(v) &\leftarrow \text{min}(\text{lowlink}(v), \text{dfn}(w))
\end{align*}

if (\text{lowlink}(v) = \text{dfn}(v))
\begin{align*}
\text{scc} &\leftarrow \emptyset \\
\text{do} \\
\text{w} &\leftarrow \text{pop}() \\
\text{add w to scc} \\
\text{while (w \neq v) } \\
\text{process\_scc(scc)}
\end{align*}
\end{align*}

end
(6, 3). no action.

```c
int dfnum /* Depth-first search number. */

void strong_connect(v)
  dfn(v) ← dfnum
  lowlink(v) ← dfnum
  visited(v) ← true
  push(v)
  dfnum ← dfnum + 1
  for each w ∈ succ(v) do /* operands(v) = succ(v) */
    if (not visited(w)) {
      strong_connect(w)
      lowlink(v) ← min(lowlink(v), lowlink(w))
    } else if (dfn(w) < dfn(v) and w is on stack)
      lowlink(v) ← min(lowlink(v), dfn(w))
  if (lowlink(v) = dfn(v))
    scc ← ∅
    do
      w ← pop()
      add w to scc
    while (w ≠ v)
    process_scc(scc)
end
```
6 remains on the stack.

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```
New lowlink and remains.

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```
Tarjan’s Algorithm: More Processing of 4

- New lowlink and remains.

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
```

- Diagram:
  - Graph with nodes labeled 0, 1, 2, 3, 4, 5, 6.
  - Edges connecting nodes.
  - Stack labeled 0, 1, 2, 3, 4, 5, 6.

```c
int dfnum

/* Depth-first search number. */

procedure strong_connect(v)
  dfn(v) ← dfnum
  lowlink(v) ← dfnum
  visited(v) ← true
  push(v)
  dfnum ← dfnum +1

  for each w ∈ succ(v) do /* operands(v) = succ(v) */
    if (not visited(w)) {
      strong_connect(w)
      lowlink(v) ← min(lowlink(v), lowlink(w))
    } else if (dfn(w) < dfn(v) and w is on stack)
      lowlink(v) ← min(lowlink(v), dfn(w))

  if (lowlink(v) = dfn(v))
    scc ← ∅
    do
      w ← pop()
      add w to scc
    while (w ≠ v)
    process_scc(scc)
end
```
Lowlink is set.

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
            while (w ≠ v)
                process_scc(scc)
        end
```
Tarjan’s Algorithm: More Processing of 2

Remove SCC from stack

int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
Tarjan’s Algorithm: Initial Processing of 8

No path from 2 to 8.

```plaintext
int dfnum

/* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
            while (w ≠ v)
                process_scc(scc)
```

**Tarjan’s Algorithm: More Processing of 8**

- 8 is its own SCC.

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```
1 is its own SCC.

```c
int dfnum; /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```
0 is its own SCC.

```
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
  dfn(v) ← dfnum
  lowlink(v) ← dfnum
  visited(v) ← true
  push(v)
  dfnum ← dfnum + 1

  for each w ∈ succ(v) do /* operands(v) = succ(v) */
    if (not visited(w)) {
      strong_connect(w)
      lowlink(v) ← min(lowlink(v), lowlink(w))
    } else if (dfn(w) < dfn(v) and w is on stack)
      lowlink(v) ← min(lowlink(v), dfn(w))

  if (lowlink(v) = dfn(v))
    scc ← ∅
    do
      w ← pop()
      add w to scc
    while (w ≠ v)
    process_scc(scc)
```
Consider the edge \((v, w)\).

When \(w\) is not yet visited we must visit it by calling \texttt{strong_connect}(w).

If \(w\) has been visited, we have two main cases:

1. \(w\) is not on the stack, because it has already found its SCC.
2. \(w\) is on the stack, because it's waiting for being popped.
   
   - If \(dfn(w) < dfn(v)\) then \(v\) must set it's lowlink so it does not think it is its own SCC.
   - If \(dfn(w) \geq dfn(v)\) then no more interesting information for \(v\) is available.
double a[N];

for (i = 0; i < N; ++i)
    x += a[i];
We first find all strongly connected components of the SSA graph.

We want to copy the SCC of \( i \) and modify the copy for \( t_1 \).

Therefore we want to have processed \( i \) before processing \( t_1 \).

Let us start with \( x \).
Processing of $x_0$

- $\text{SCC}_0 = \{x_0\}$. Empty stack.
- Nodes processed in a SCC are green.
- Next processing $x_1$. 
Processing of $x_1$ and $x_2$

- $x_1$ and $x_2$ are pushed and then the search continues with $t_2$.
- Nodes on the stack are red.
- Next processing $t_2$. 
Processing of $t_2$

- Next processing $t_1$. 

\[ \phi(x_0, x_2) \]
\[ x_1 \]
\[ x_2 + t_2 \]
\[ t_1 \]
\[ i_1 \times 8 \]
\[ M[a + t_1] \]
\[ t_2 \]
\[ \phi(i_0, i_2) \]
\[ i_0 \]
\[ i_1 + 1 \]
\[ i_2 \]
Processing of $t_1$

- Next processing $i_2$. 
Processing of $i_2$ and $i_1$

Next processing $i_0$. 
Processing of $i_0$

- $\text{SCC}_1 = \{i_0\}$
- Next more processing in $i_2$. 
Classifying $SCC_2 = \{i_1, i_2\}$

- $SCC_2 = \{i_1, i_2\}$
- $SCC_2$ is an **induction variable** due it consists of a $\phi$-function and an add with a **region constant**.

- A region constant is not modified in a loop, i.e. it’s a number or its definition strictly dominates the loop header.
Replacing $i_1 \times 8$

- $SCC_3 = \{ t_1 \}$
- $SCC_3$ is a multiplication of an induction variable and a region constant.
- Therefore $SCC_3$ is replaced by a modified copy of $SCC_2$ with $\phi(i)$. 
Modifying a Copy of $SCC_2$ to Compute $t_1$

- $SCC_4 = \{y_1, y_2\}$
- Due to the replacement, the assignment to $t_1$ becomes dead code.
- There is a very beautiful algorithm to remove $t_1$ and other dead code that we will look at during the next lecture.
Also $a + y_1$ can be Replaced

- Due to Tarjan’s algorithm we can start in any node and be sure we have already processed the operand nodes, when a variable’s definition is going to be replaced.
- Not only multiplications but also some additions can be replaced, but we don’t show this in the example.
Processing of a new SCC

- When nodes have been popped from the stack and collected in a SCC, the following is performed.
- A SCC has the attribute `header` which is the header of a loop in the control flow graph.

```plaintext
procedure process_scc(scc)
    if (scc has a single member n)
        if (valid_form(n))
            replace(n, iv, rc)
        else
            header(n) ← ⊥
    else
        classify(scc)
end
```
Valid Forms of Definition for Replacement

- $iv$ is induction variable
- $rc$ is region constant

```plaintext
function valid_form(n)
    if (n is of form $x \leftarrow iv \times rc$
        or n is of form $x \leftarrow rc \times iv$
        or n is of form $x \leftarrow iv \pm rc$
        or n is of form $x \leftarrow rc + iv$)
        return true
    else
        return false
end
```
**Definition of Region Constant**

```plaintext
function region_const(x, header)
    return x is constant or vertex(x) strictly dominates header
end
```

```plaintext
scanf("%d %d", &a, &b);
while (i < n) {
    x += u[a * i + b];
    i += 1;
}
```

- The variables `a` and `b` are region constants in the loop.
Reverse Post Order

int i

procedure dfs(v)
    visited(v) ← true
    for each w ∈ succ(v) do
        if (not visited(w))
            dfs(w)
    i ← i - 1
    rpo(v) ← i
end

procedure compute_rpo(CFG)
    i ← |V|
    for each vertex v do
        visited(v) ← false
        dfs(v)
    end
procedure classify(scc)
    for each \( n \in scc \) do
        if (\( \text{rpo}(\text{vertex}(n)) < \text{rpo}(\text{header}) \))
            header \( \leftarrow \) vertex(n)
    for each \( n \in scc \) do
        if (\( \text{operator}(n) \notin \{\phi, +, -, \text{move}\} \))
            scc is not an induction variable
        else
            for each operand \( \omega \in \text{operands}(n) \) do
                if (\( \omega \notin scc \) and not \( \text{region}\_\text{const}(\omega, header) \))
                    scc is not an induction variable
    if (scc is an induction variable)
        for each \( n \in scc \) do
            header(n) \( \leftarrow \) header
    else
        for each \( n \in scc \) do
            if (valid_form(n))
                replace(n, iv, rc)
            else
                header(n) \( \leftarrow \) \( \bot \)
end
procedure replace(operation, iv, rc)
    result ← reduce(opcode(operation), iv, rc)
    replace operation with mov using result as source
    header(operation) ← header(iv)
end
function reduce(operation, iv, rc)
    result ← lookup(opcode, iv, rc)
    if (result is not found)
        result ← new_temp()
        install(opcode, iv, rc, result)
        new_def ← copy_def(iv, result)
        for each operand ω in new_def do
            if (ω is an induction variable)
                replace ω with reduce(opcode, ω, rc)
            else if (opcode = × or new_def is a φ)
                replace ω with apply(opcode, ω, rc)
    return result
end
Apply

**function** `apply(opcode, op1, op2)`

`result ← lookup(opcode, op1, op2)`

**if** (`result` is not found)

**if** (`op1` is an induction variable and `op2` is a region constant)

`result ← reduce(opcode, op1, op2)`

**else if** (`op2` is an induction variable and `op1` is a region constant)

`result ← reduce(opcode, op2, op1)`

**else**

`result ← new_temp()`

`install(opcode, op1, op2, result)`

choose the location where the operation will be inserted

perform constant folding if possible

create a new operation at the chosen location

**return** `result`

end