*For RC 2012*

# Synthesizing Loops For Program Inversion

Cong Hou, Daniel Quinlan,
David Jefferson, Richard Fujimoto, Richard Vuduc

Georgia Tech College of Computing
Computational Science and Engineering

hpcgarage

Lawrence Livermore National Laboratory

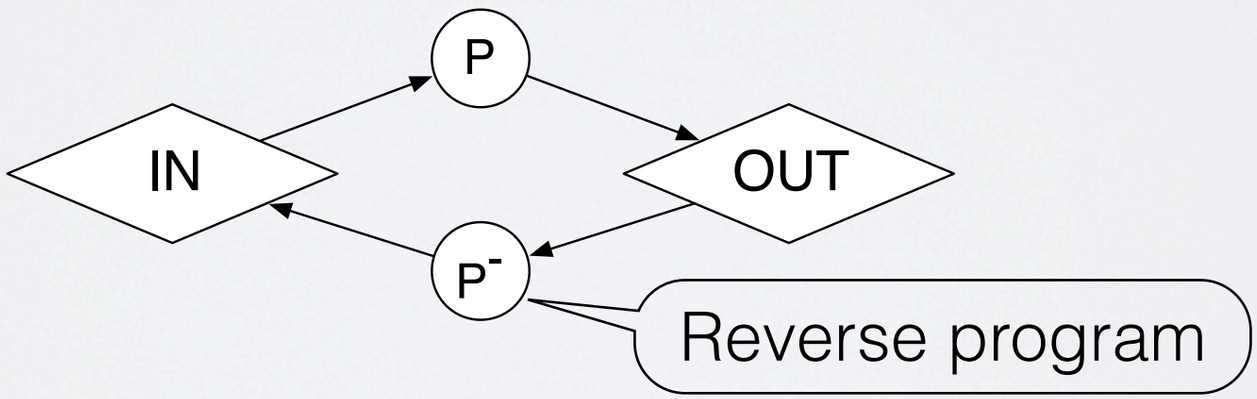
# Program Inversion

- Given a program *P*, and its inverse $P^-$, then we have

$$P \; ; \; P^- = \text{no-op}$$

- Examples: `++a`/`--a`, **`swap(a,b)`**/**`swap(a,b)`**, compression/decompression, encryption/decryption, etc..

- Assume the input and output of *P* are *IN* and *OUT*:



# Program Inversion

- What if *P* is not reversible?

```
IN: a

a = 0;

OUT: a
```

*P*

# Program Inversion

- What if *P* is not reversible?

```
IN: a

a = 0;

OUT: a
```

*P*

- Make it reversible!

```
IN: a

s = a;
a = 0;

OUT: a, s
```

*P+*

```
IN: a, s

a = s;

OUT: a
```

*P-*

# Program Inversion

- What if *P* is not reversible?

```
IN: a

a = 0;

OUT: a
```

*P*

- Make it reversible!

State Saving

```
IN: a

s = a;
a = 0;

OUT: a, s
```

*P+*

```
IN: a, s

a = s;

OUT: a
```

*P-*

# Program Inversion

- What if *P* is not reversible?

```
IN: a

a = 0;

OUT: a
```

*P*

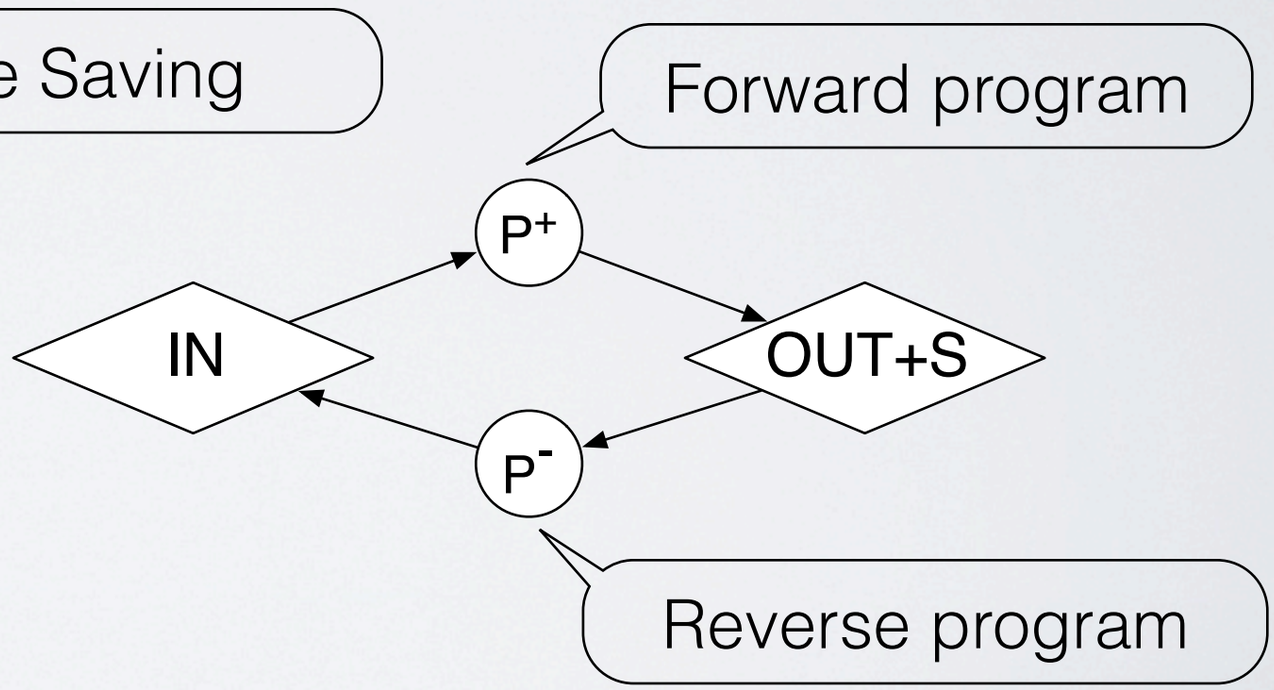
- Make it reversible!

```
IN: a

s = a;
a = 0;

OUT: a, s
```

*P+*

```
IN: a, s

a = s;

OUT: a
```

*P-*



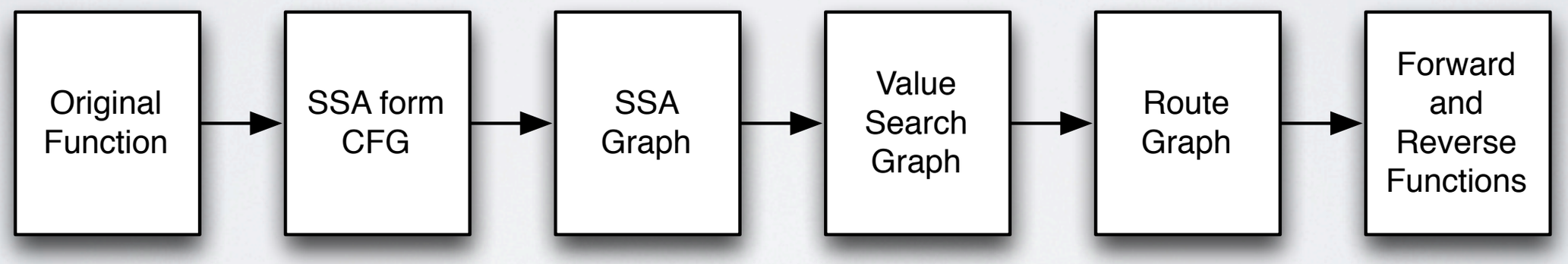
# Our Previous Work

- We have built a framework that can generate the forward and reverse programs for loop-free programs.
- We have implemented this framework into a compiler called Backstroke.
- For more details please refer to our CC paper:

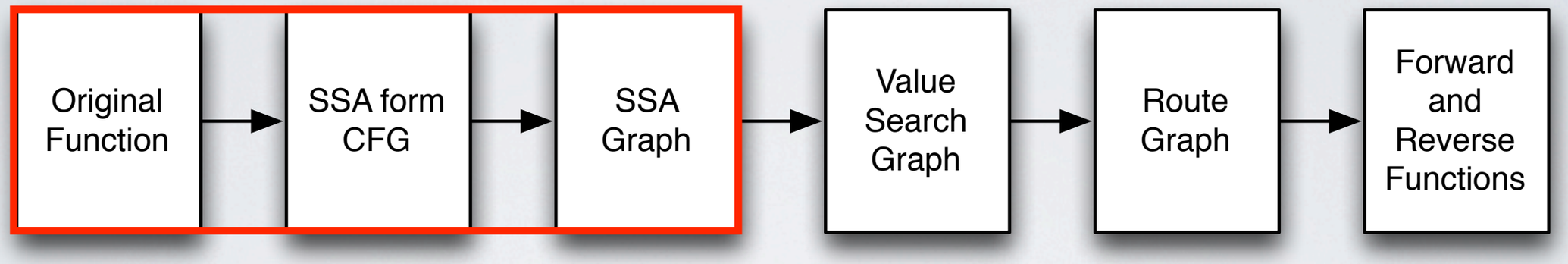
  C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc. **A new method for program inversion.** International Conference on Compiler Construction, 2012.

# Overview Of Our Previous Work

- Our approach:

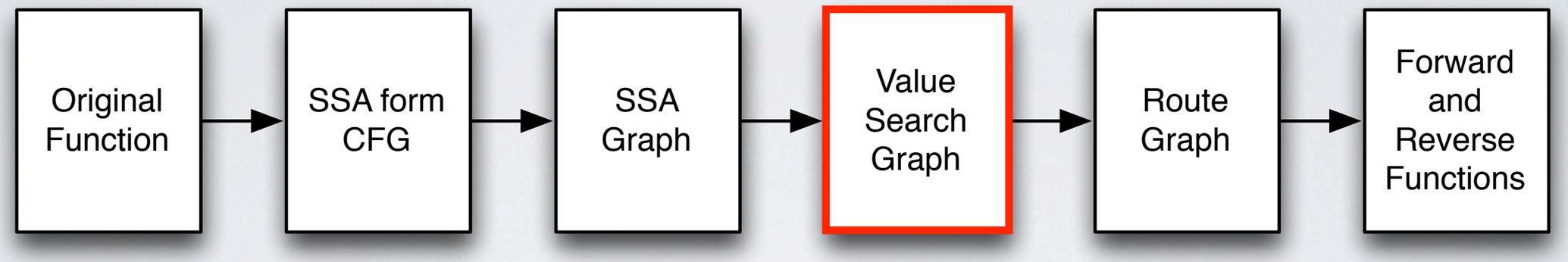


# Overview Of Our Previous Work



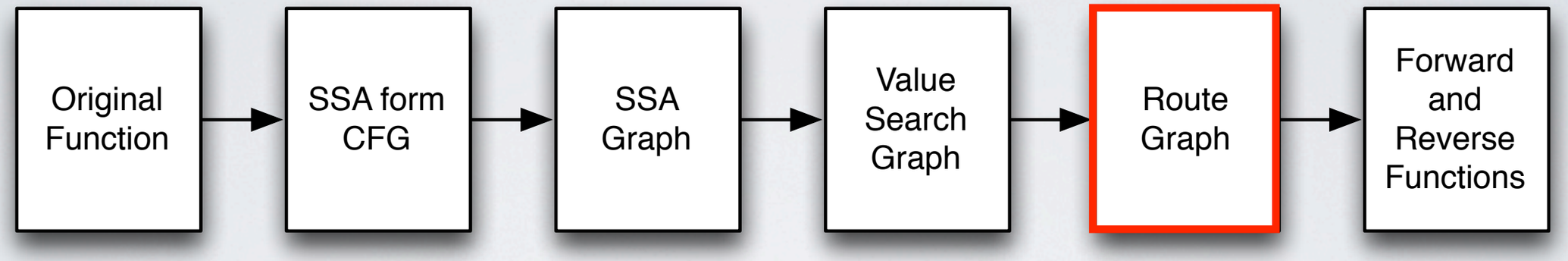

- We turn the program into the **SSA (Static Single Assignment) form CFG (Control Flow Graph)**, so that each variable is defined only once and can represent a distinct value. An SSA graph is then built to show the data dependencies between different values.

# Overview Of Our Previous Work



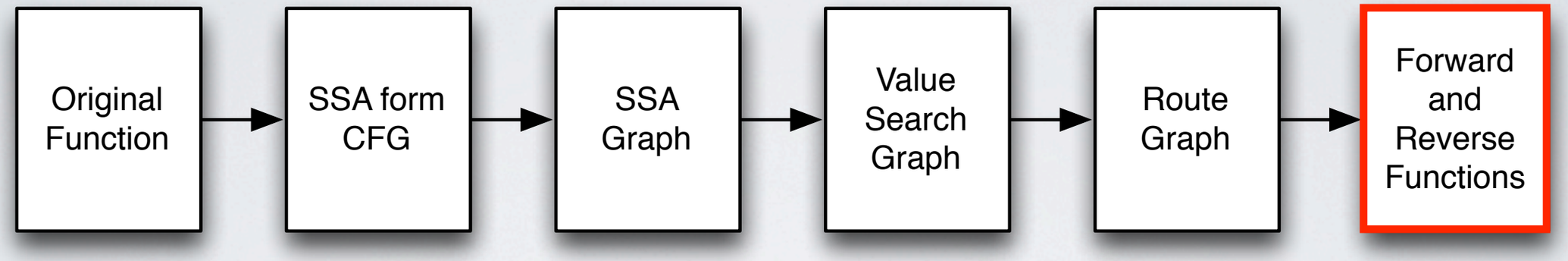

- We build a **Value Search Graph** (VSG) showing all **equality relations** between values in the program. Finding the inverse becomes a search problem in this graph. Each equality is constrained by a condition represented by a set of CFG paths.

# Overview Of Our Previous Work



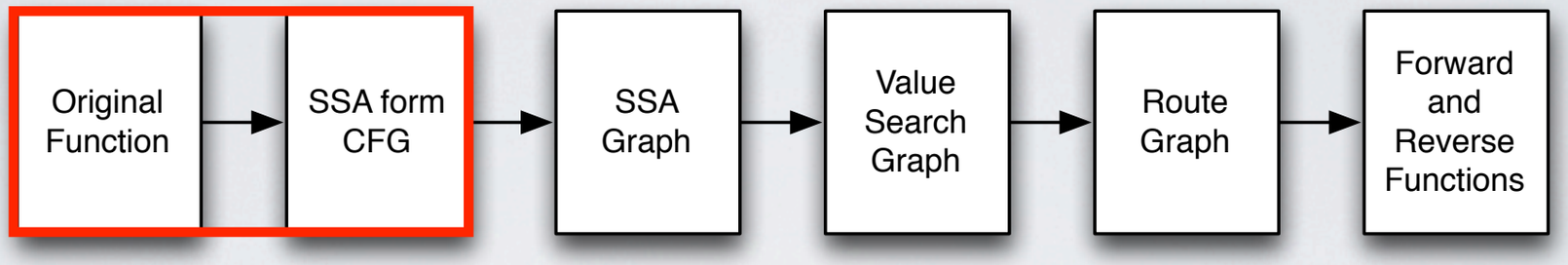

- We then search for the desired values by **following the equalities** until available values are reached. The search should guarantee that each value is retrieved for all CFG paths. The search result which we call a **Route Graph (RG)** shows valid data dependences in reverse program.

# Overview Of Our Previous Work

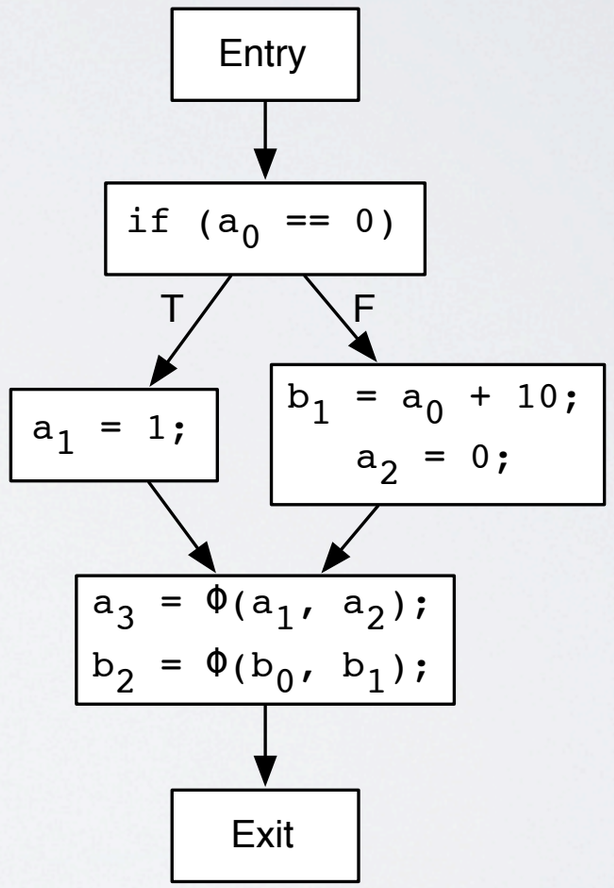



- The forward and reverse programs are built based on the search result.

# Example: Building SSA Form CFG

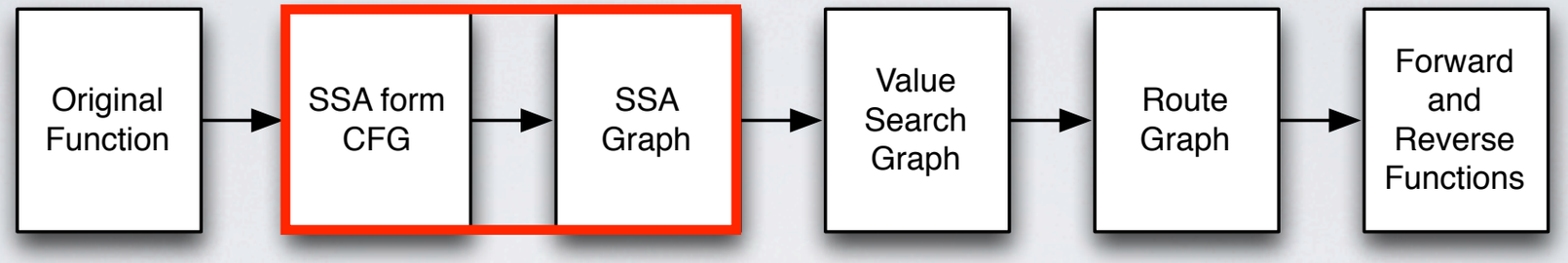


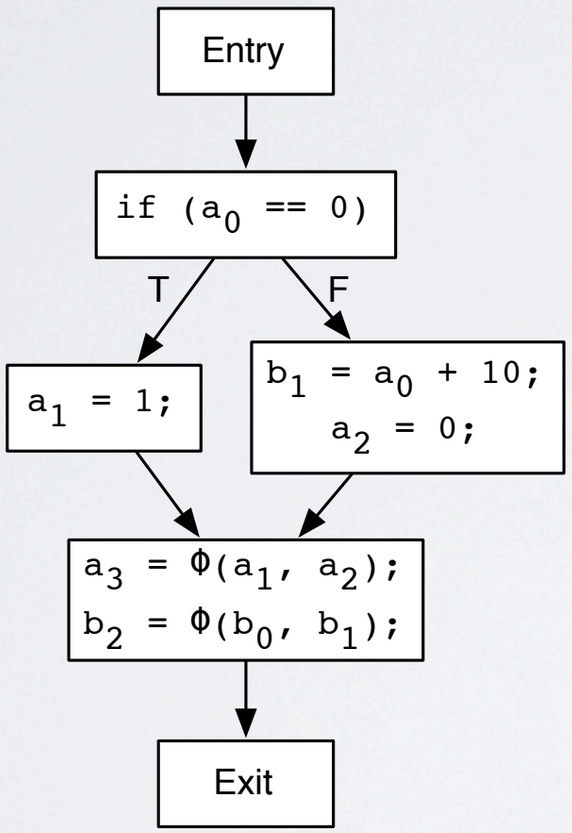

```
IN: a, b

void foo()
{
  if (a == 0)
    a = 1;
  else
  {
    b = a + 10;
    a = 0;
  }
}

OUT: a, b
```



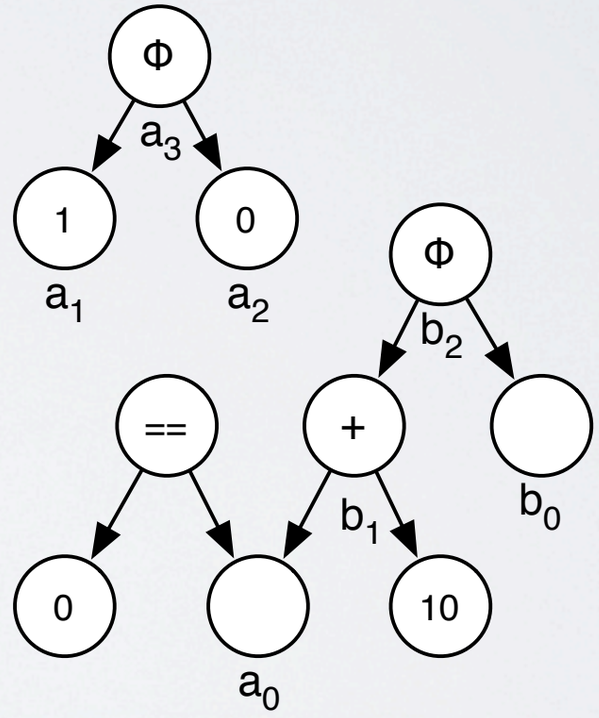

SSA form CFG

# Example: Building SSA Graph



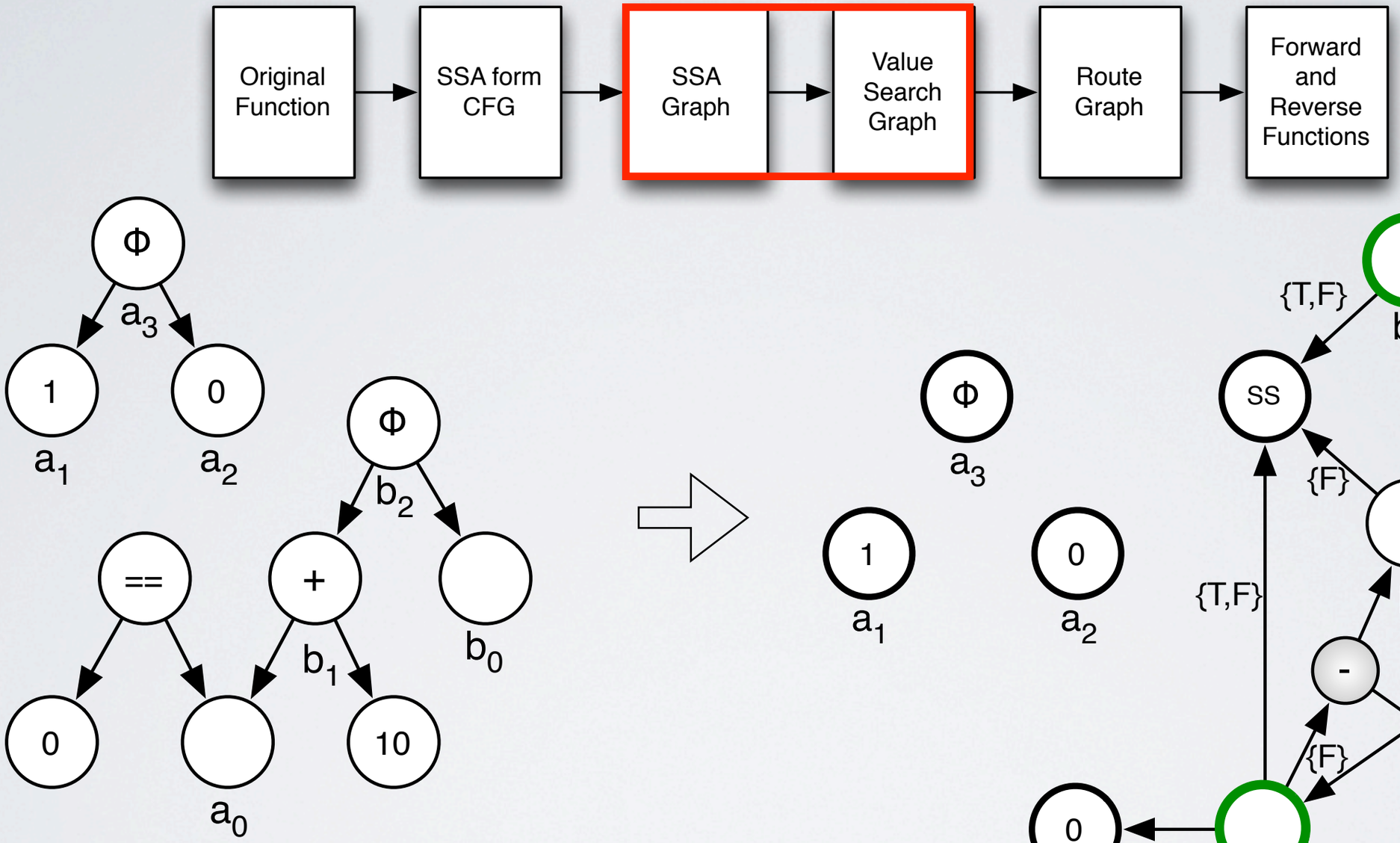




SSA form CFG

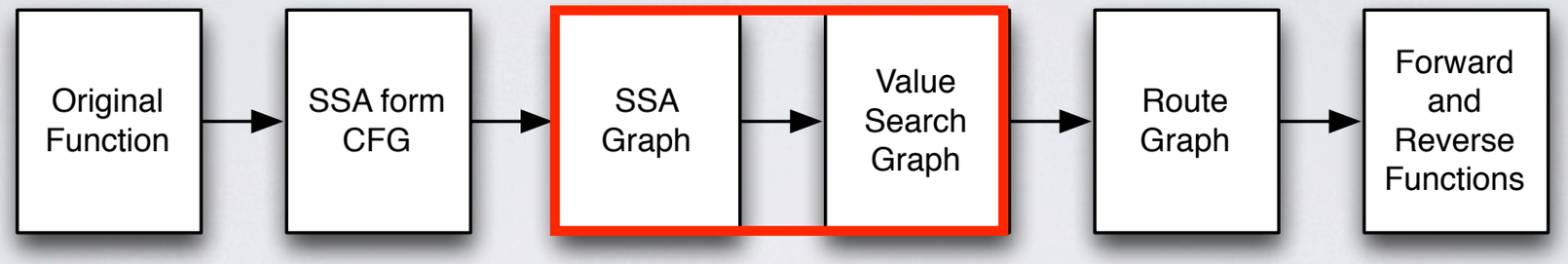


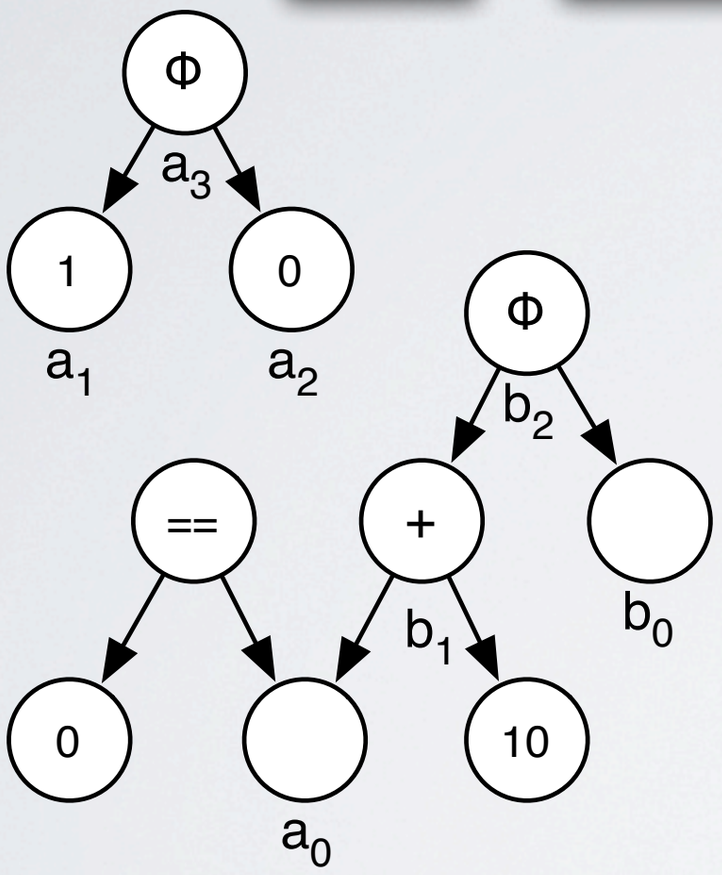

SSA Graph

# Example: Building Value Search Graph



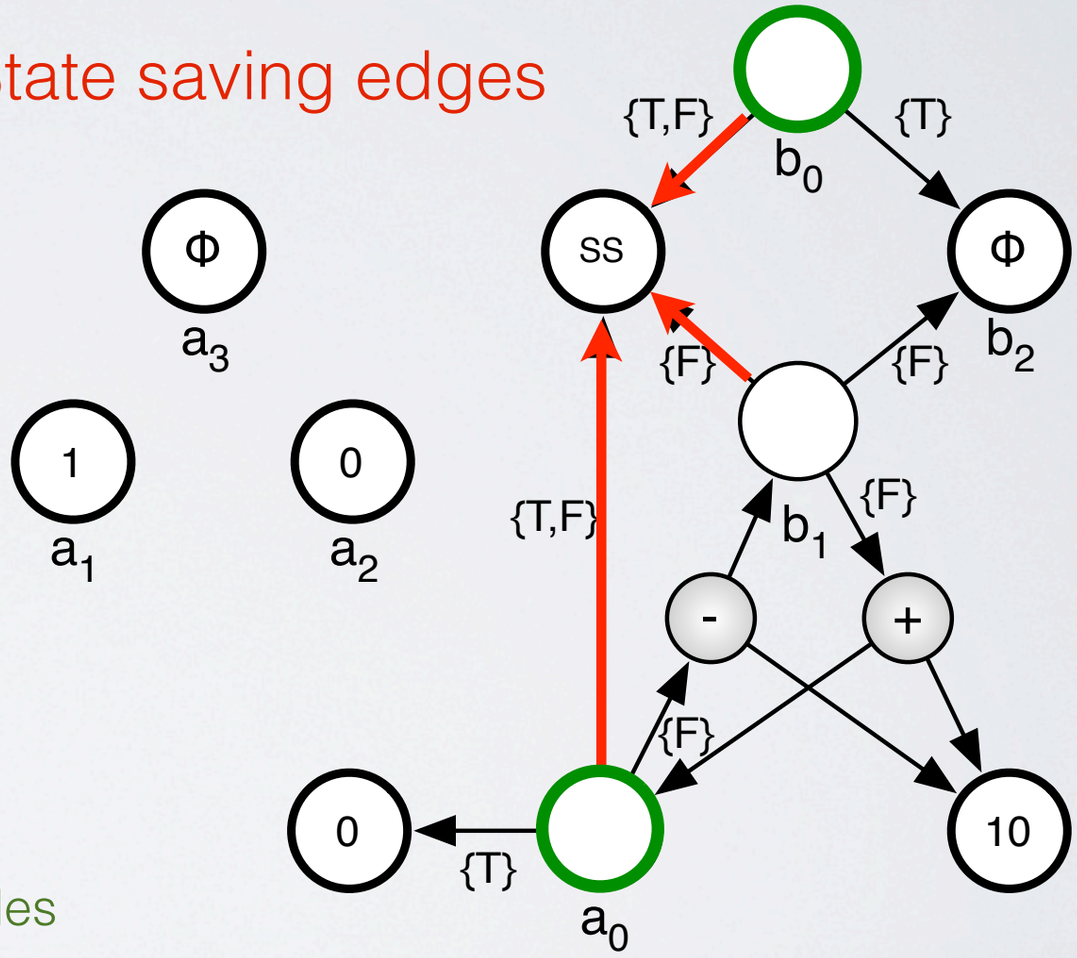
# Example: Building Value Search Graph




SSA Graph

State saving edges

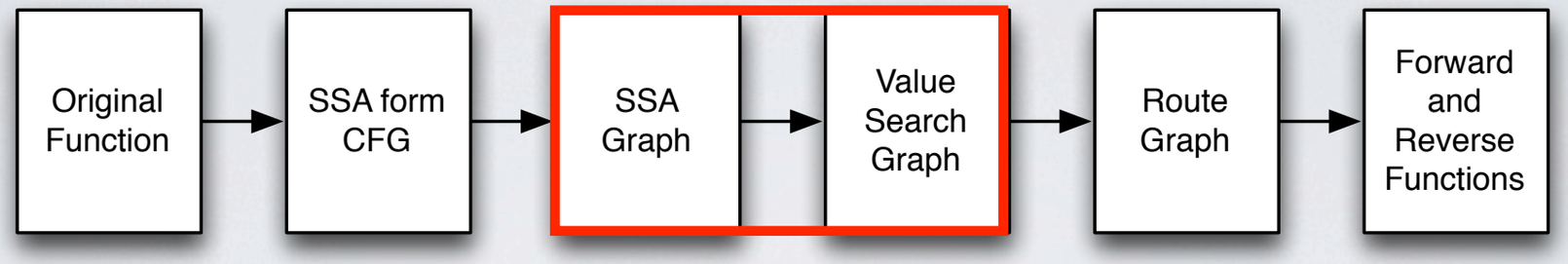


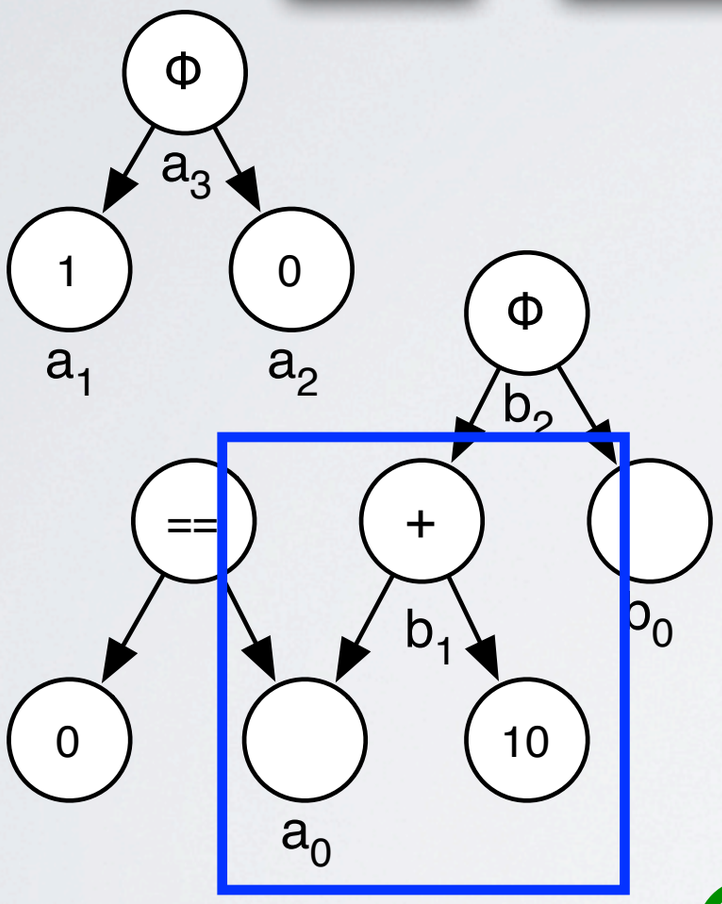

Value Search Graph

Target nodes

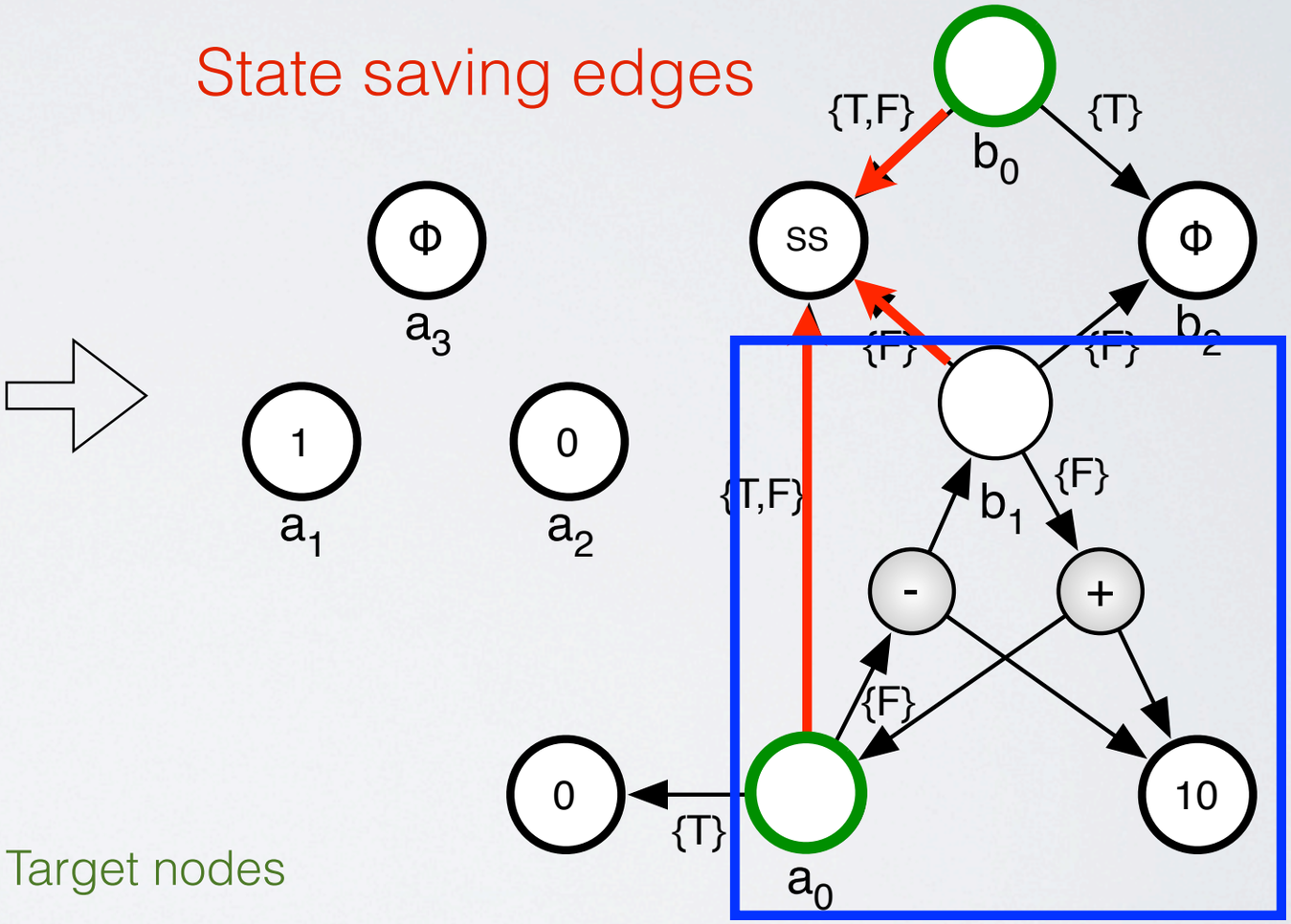
Available nodes

# Example: Building Value Search Graph




SSA Graph

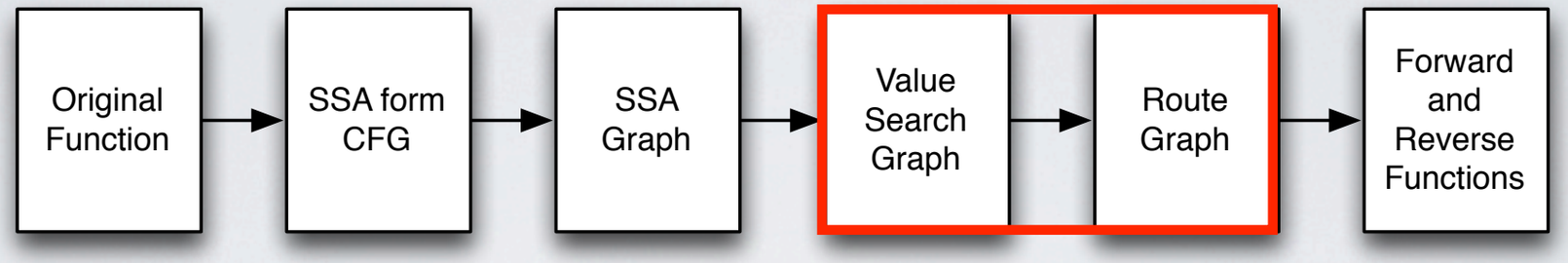


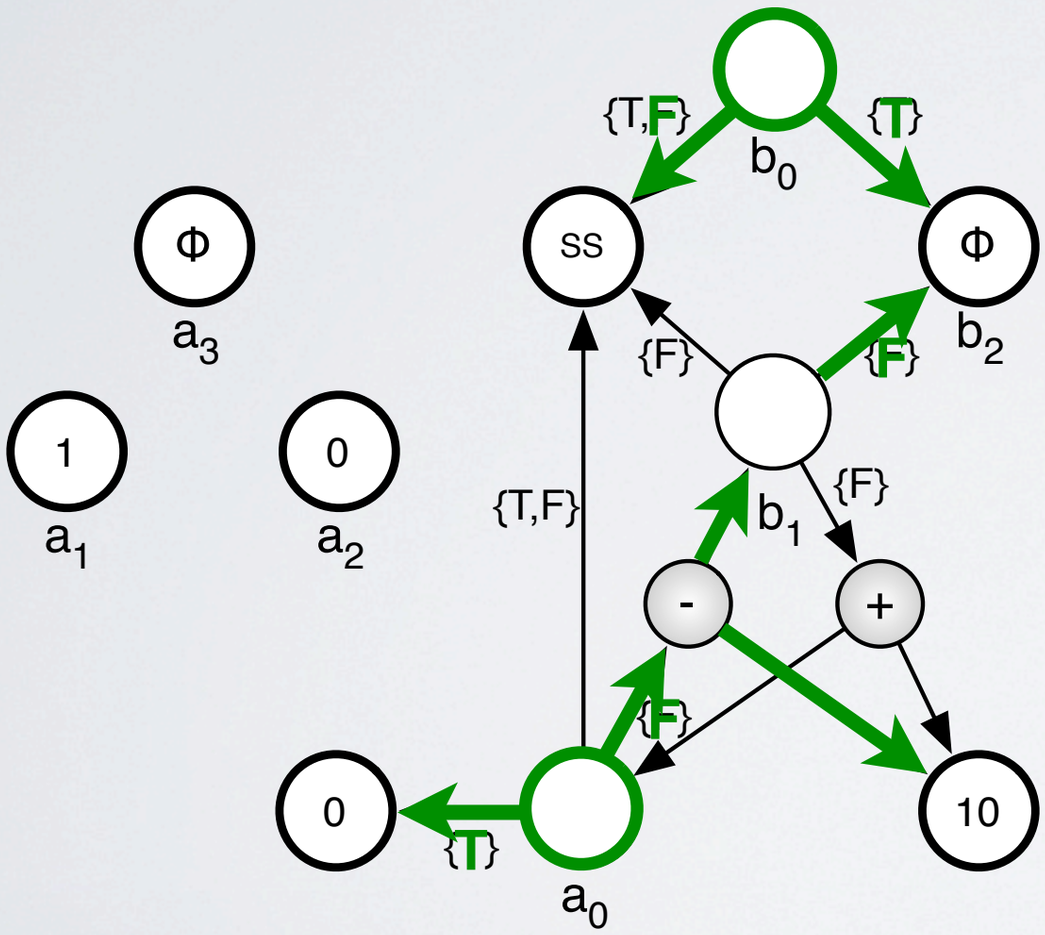

Value Search Graph

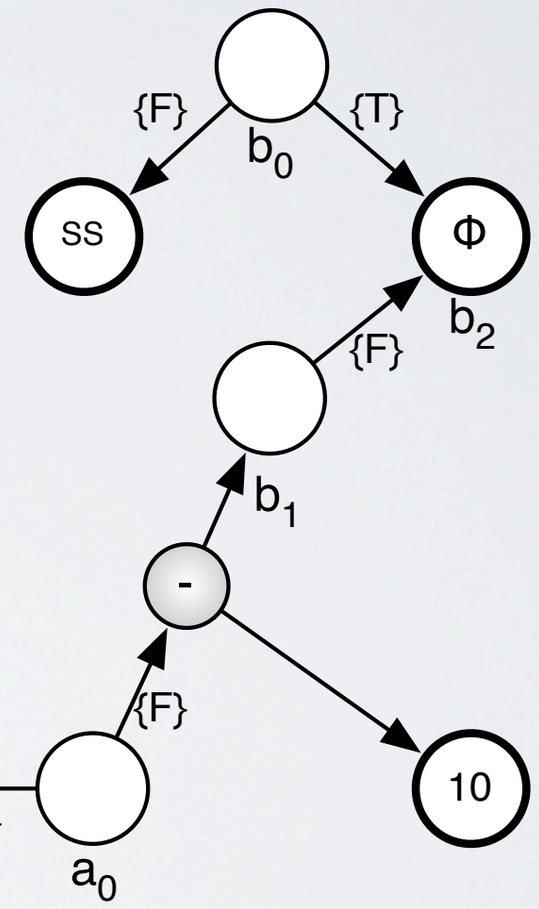
Target nodes

Available nodes

# Example: Building Route Graph

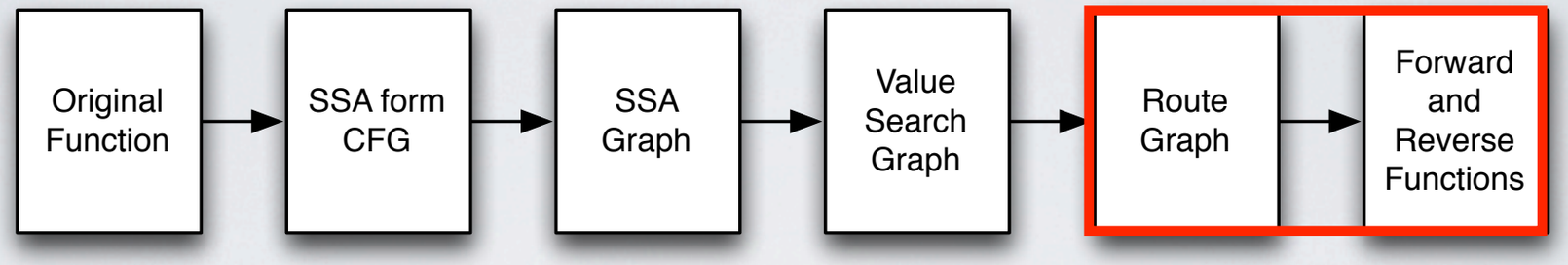


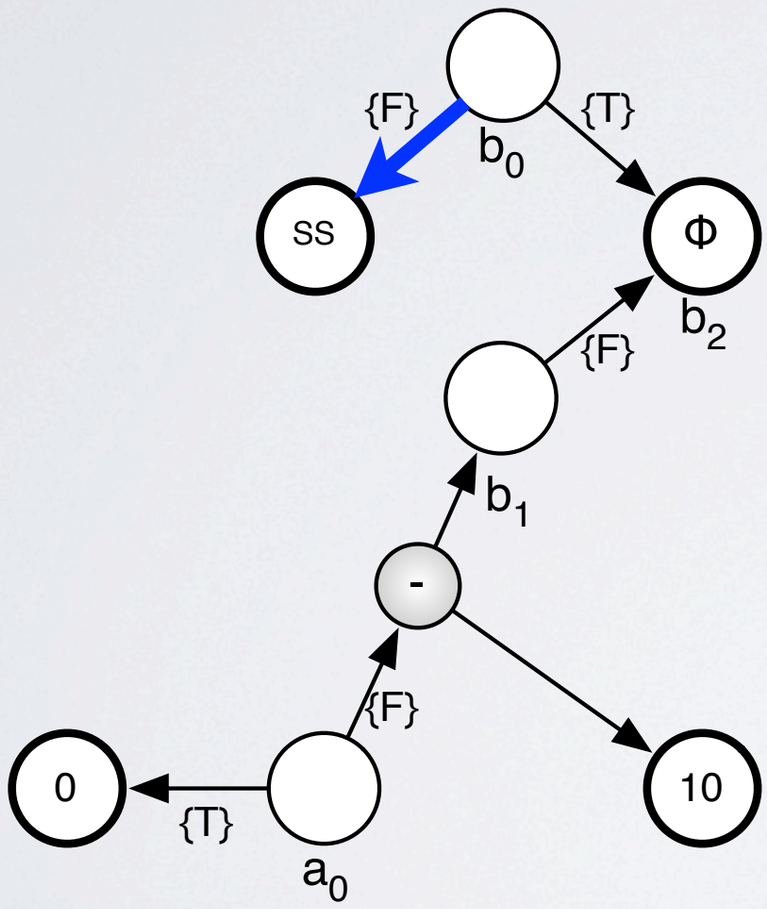




Route Graph

# Example: Generating The Forward Program



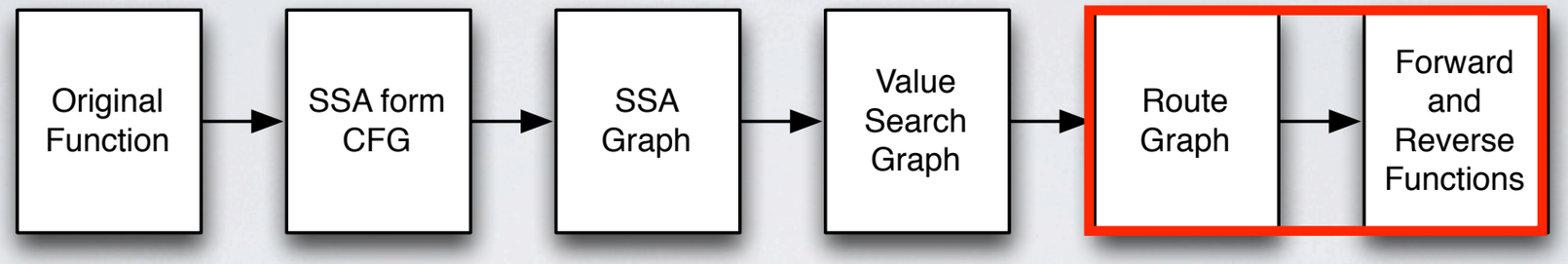



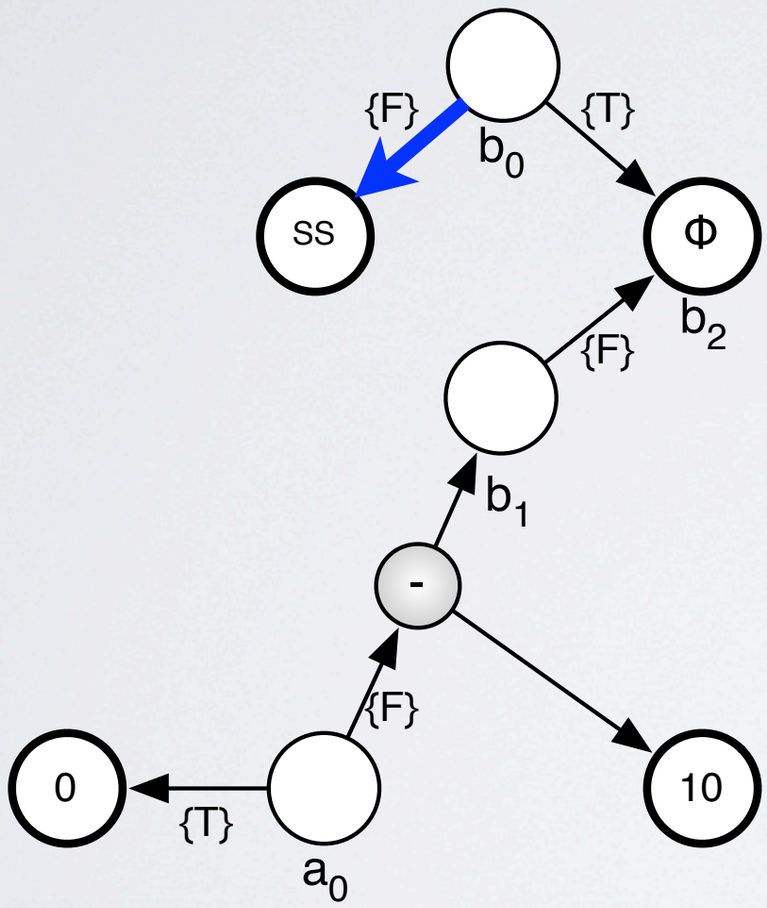

Route Graph

```
void foo_forward()
{
  int trace = 0;
  if (a == 0)
  {
    trace |= 1;
    a = 1;
  }
  else
  {
    store(b);
    b = a + 10;
    a = 0;
  }
  store(trace);
}
```

Red: Path recording. Blue: State saving.

# Example: Generating The Reverse Program




Route Graph

```
void foo_reverse()
{
  int trace;
  restore(trace);
  if ((trace & 1) == 1)
    a = 0;
  else
  {
    a = b - 10;
    restore(b);
  }
}
```

# Example: Generated Forward And Reverse Programs

```
void foo()
{
  if (a == 0)
    a = 1;
  else
  {
    b = a + 10;
    a = 0;
  }
}
```

```
void foo_forward()
{
  int trace = 0;
  if (a == 0)
  {
    trace |= 1;
    a = 1;
  }
  else
  {
    store(b);
    b = a + 10;
    a = 0;
  }
  store(trace);
}
```

```
void foo_reverse()
{
  int trace;
  restore(trace);
  if ((trace & 1) == 1)
    a = 0;
  else
  {
    a = b - 10;
    restore(b);
  }
}
```

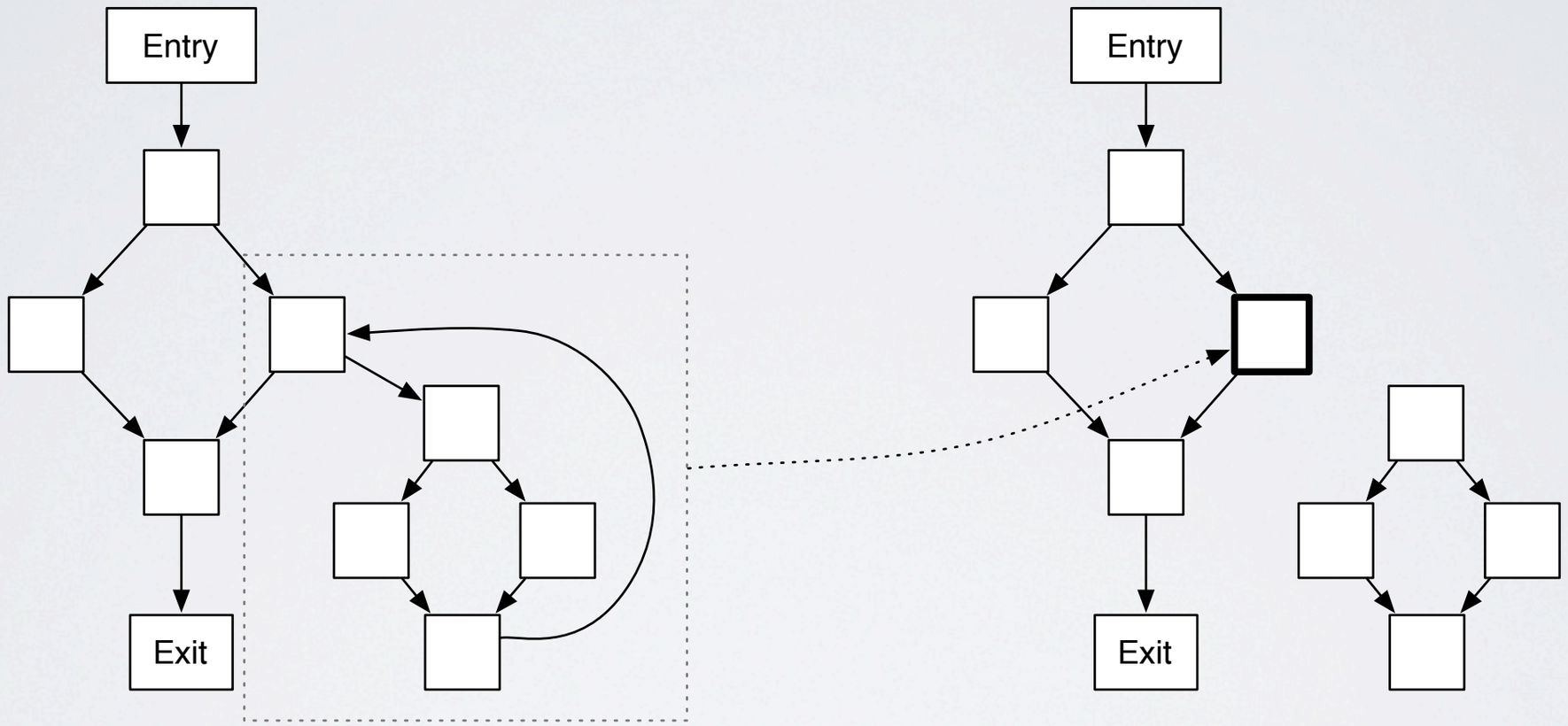
Red: Path recording. Blue: State saving.

# Handling Loops

- What problems do loops bring?
  - Cyclic control flow paths.
    - Solution: In the CFG, we collapse the loop into a single node and remove cycles. Also, we record the control flows in the loop body separately, where the loop body is treated as another loop-free program.

# Handling Loops

- Recording CFG paths for Programs with Loops

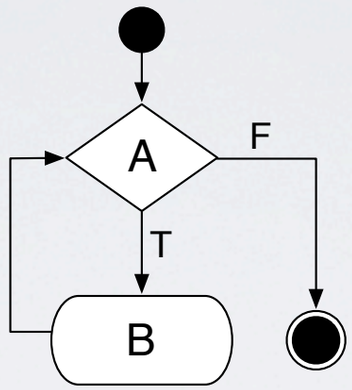


# Handling Loops

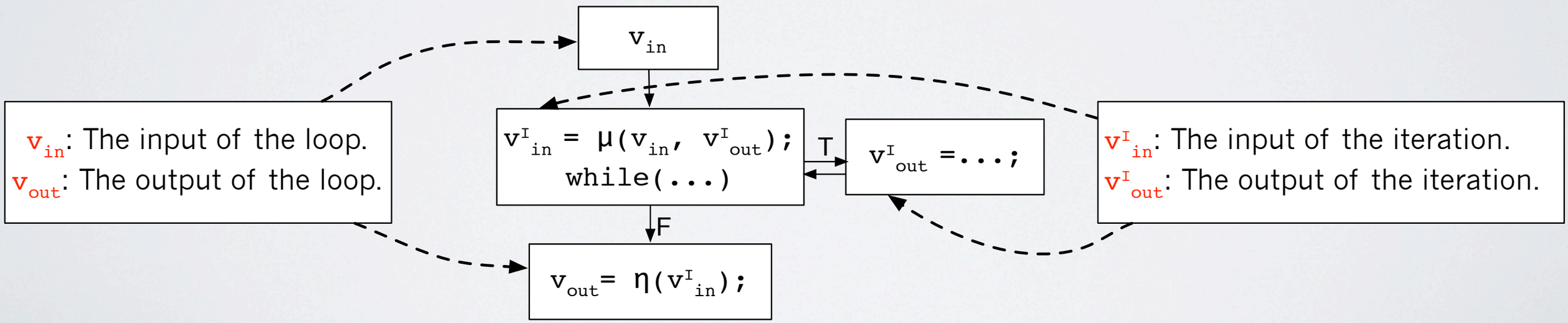
- What problems do loops bring?
  - Cyclic control flow paths.
    - Solution: In the CFG, we collapse the loop into a single node and remove cycles. Also, we record the control flows in the loop body separately, where the loop body is treated as another loop-free program.
- If we want to build loops in the reverse program, cycles may be formed in the Route Graph.
  - Solution: we build special constructs in VSG for loops and also develop special searching rules.

# Handling While Loops

- While loop: a special single-entry single-exit loop.

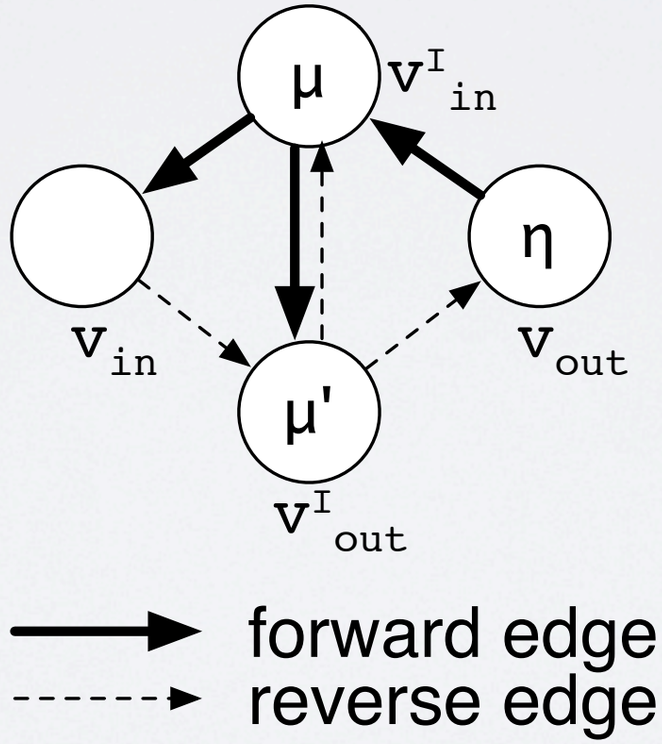



- For each variable modified in a while loop, we define four special definitions of it.



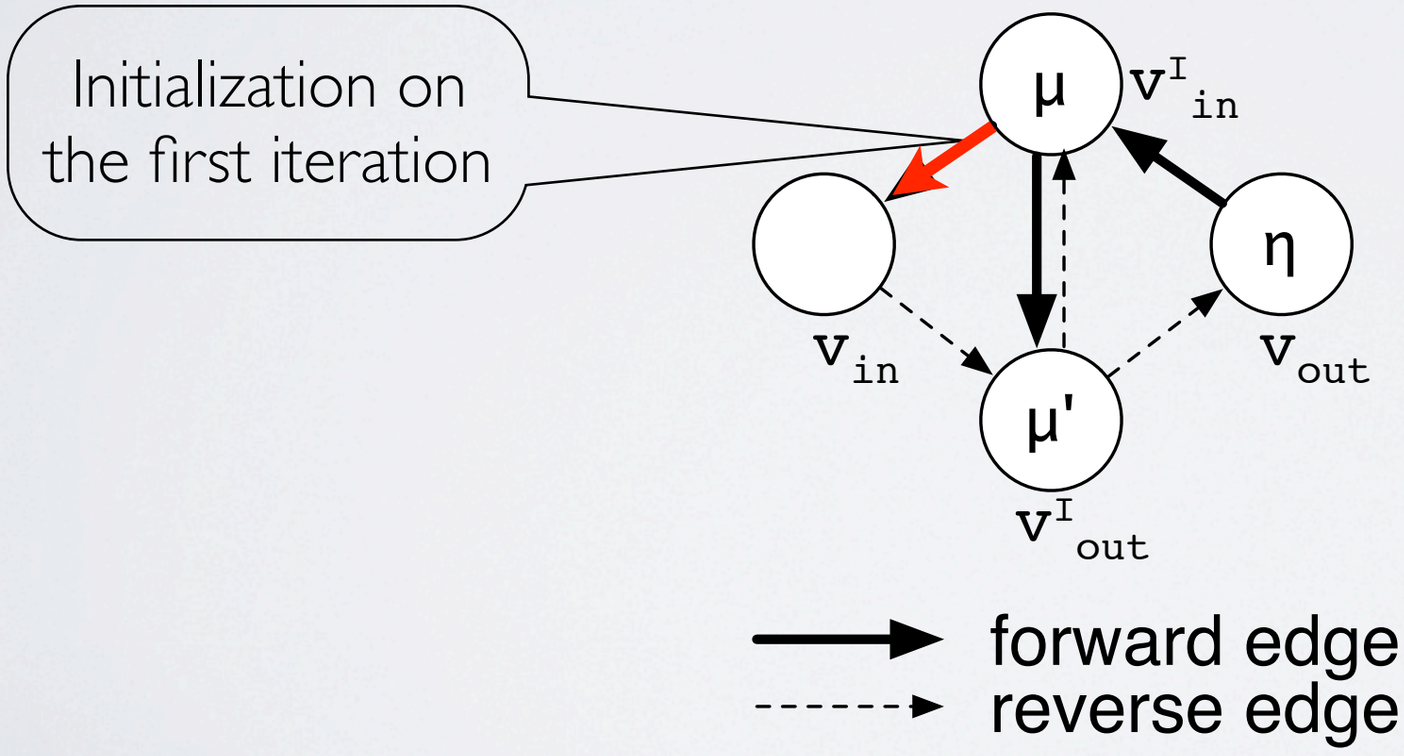
# Handling While Loops

- Those four definitions in the VSG:
  - Forward edges represent data flows in the original (forward) programs.
  - Reverse edges represent data flows in the reverse programs.



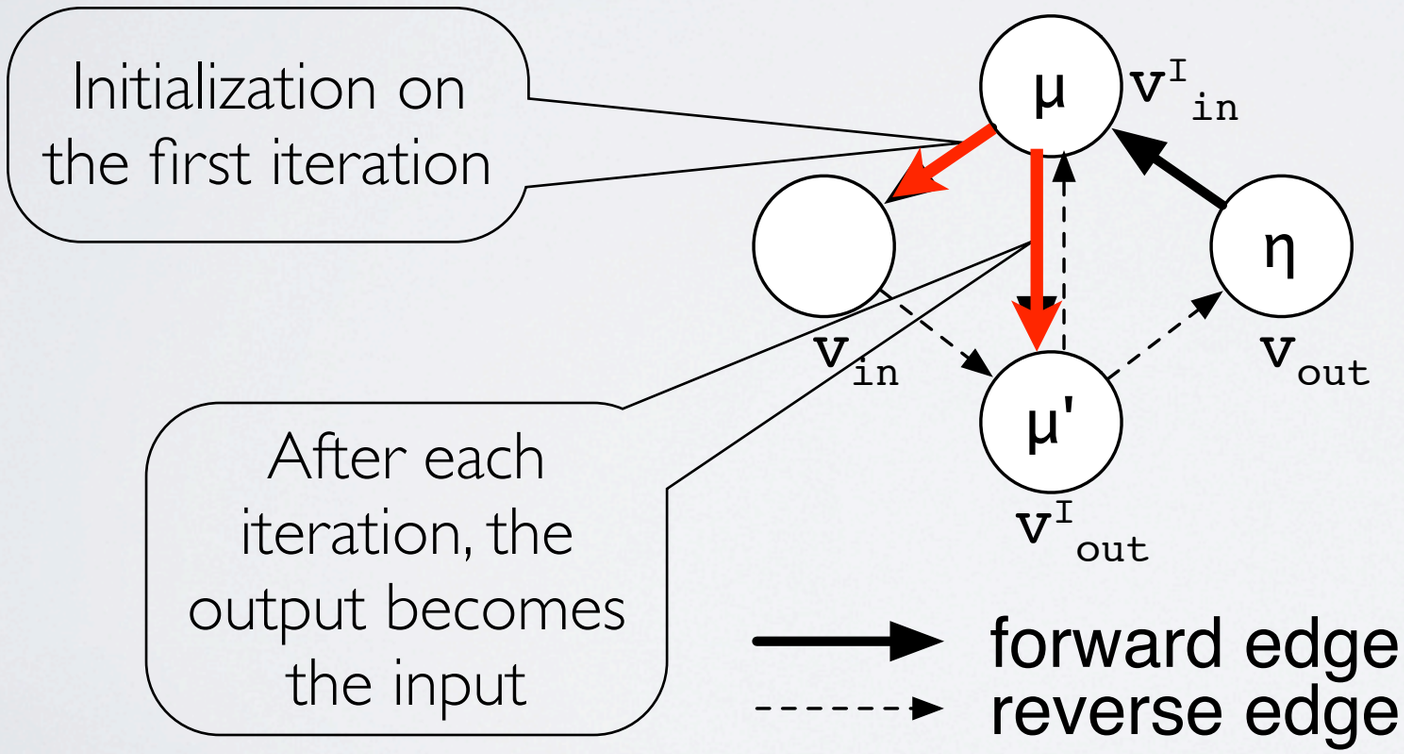
# Handling While Loops

- Those four definitions in the VSG:
  - Forward edges represent data flows in the original (forward) programs.
  - Reverse edges represent data flows in the reverse programs.



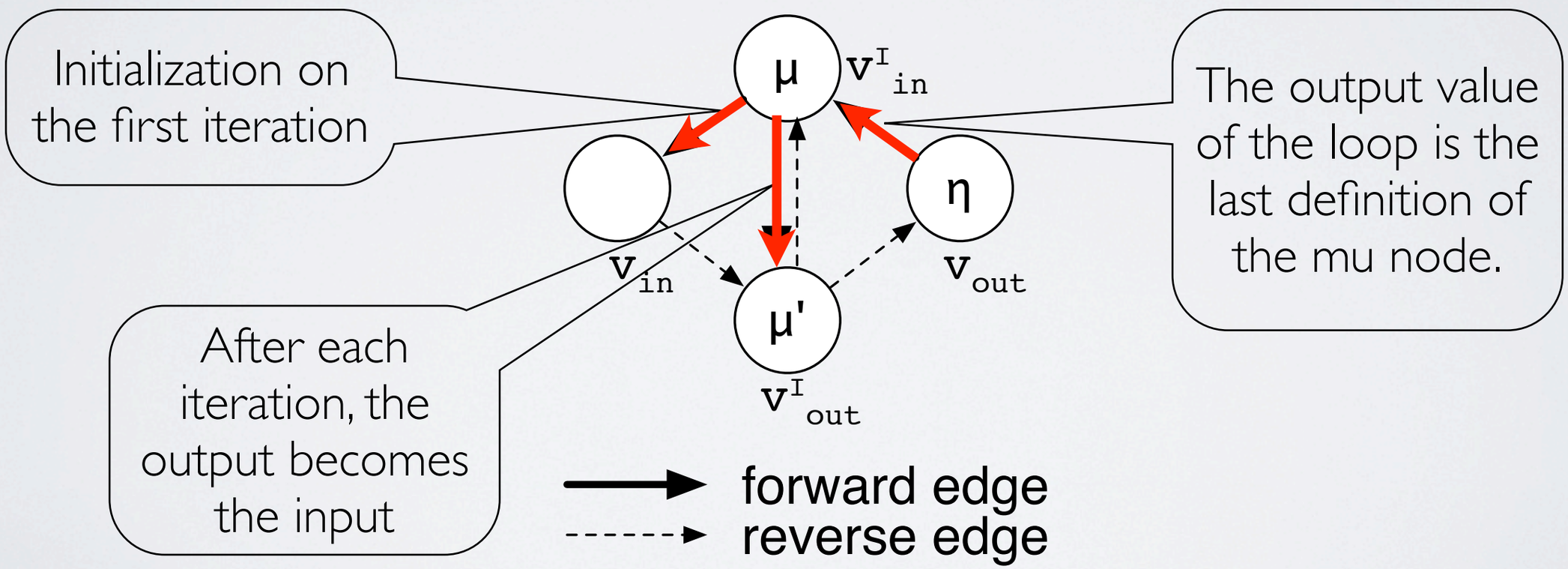
# Handling While Loops

- Those four definitions in the VSG:
  - Forward edges represent data flows in the original (forward) programs.
  - Reverse edges represent data flows in the reverse programs.



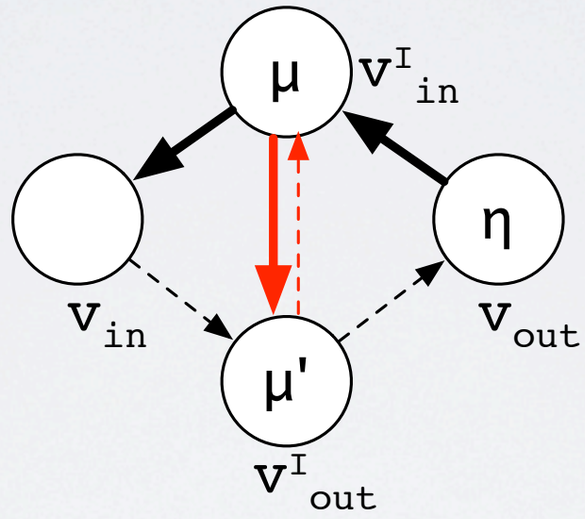
# Handling While Loops

- Those four definitions in the VSG:
  - Forward edges represent data flows in the original (forward) programs.
  - Reverse edges represent data flows in the reverse programs.



# Handling While Loops

- Special search rules on the VSG:
  - Allows cycles to be formed. But each cycle must contain a forward/reverse edge between the input and output of the iteration.




  - During the search for a given value, the forward and reverse edges cannot coexist in the search result.

# Handling While Loops

- Building the loop body.
  - Using the same method we build the reverse code for loop-free programs.
- Building the loop predicate.
  - Approach 1: Building the same loop predicate as in the original loop.
  - Approach 2: If there is a monotonic variable in a loop, build the loop predicate from it.
  - Approach 3: Insert a counter counting the number of iterations of the loop in the forward program, and use this counter to build the loop predicate in the reverse program.

# Handling While Loops

- Building the loop predicate.
  - Approach 1: Building the same loop predicate as in the original loop.

```
i = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
}
```

Original loop

```
i = 0;
while (A[i] > 0) {
    /* generated loop body */
    i = i + 2;
}
```

Generated loop

# Handling While Loops

- Building the loop predicate.
  - Approach 2: If there is a monotonic variable in a loop, build the loop predicate from it.

```
i = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
}
/* i == i1 */
```

Original loop

```
i = 0;
while (i != i1) {
    /* generated loop body */
    i = i + 2;
}
```

Generated loop

# Handling While Loops

- Building the loop predicate.
  - Approach 3: Insert a counter counting the number of iterations of the loop in the forward program, and use this counter to build the loop predicate in the reverse program.

```
i = 0;
count = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
    count = count + 1;
}
store(count);
```

Original loop

```
restore(count);
while (count > 0) {
    /* generated loop body */
    count = count - 1;
}
```

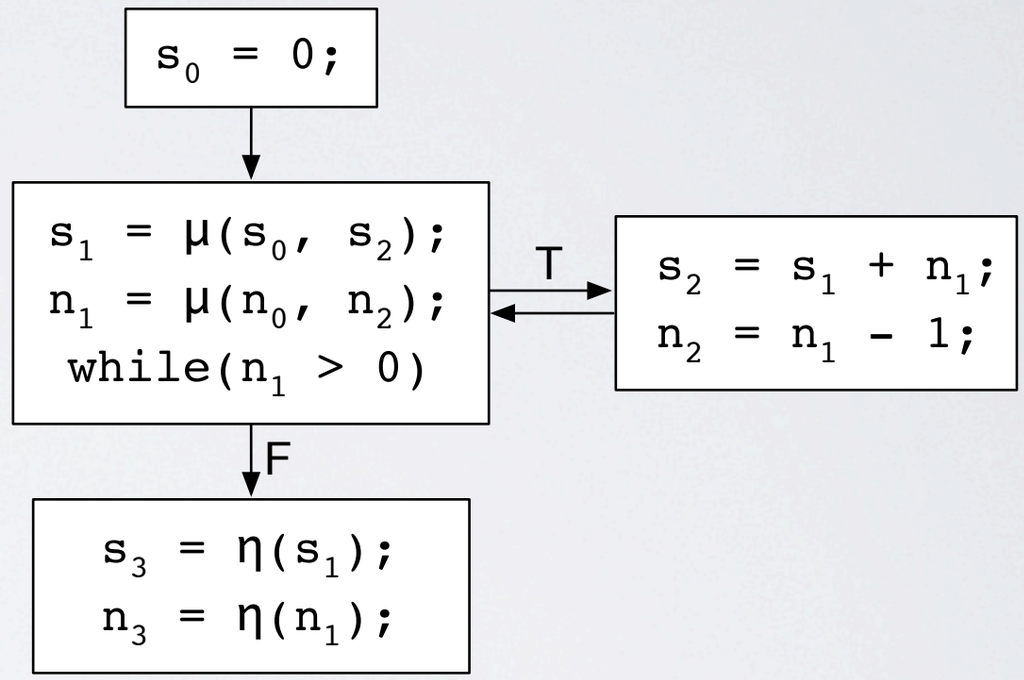
Generated loop

# An Example

- Our example: Given an integer *n (n > 0)*, get *1+2+…+n*.

```
// input: n (n > 0)
s = 0;
while (n > 0) {
     s = s + n;
     n = n - 1;
}
// output: s
```

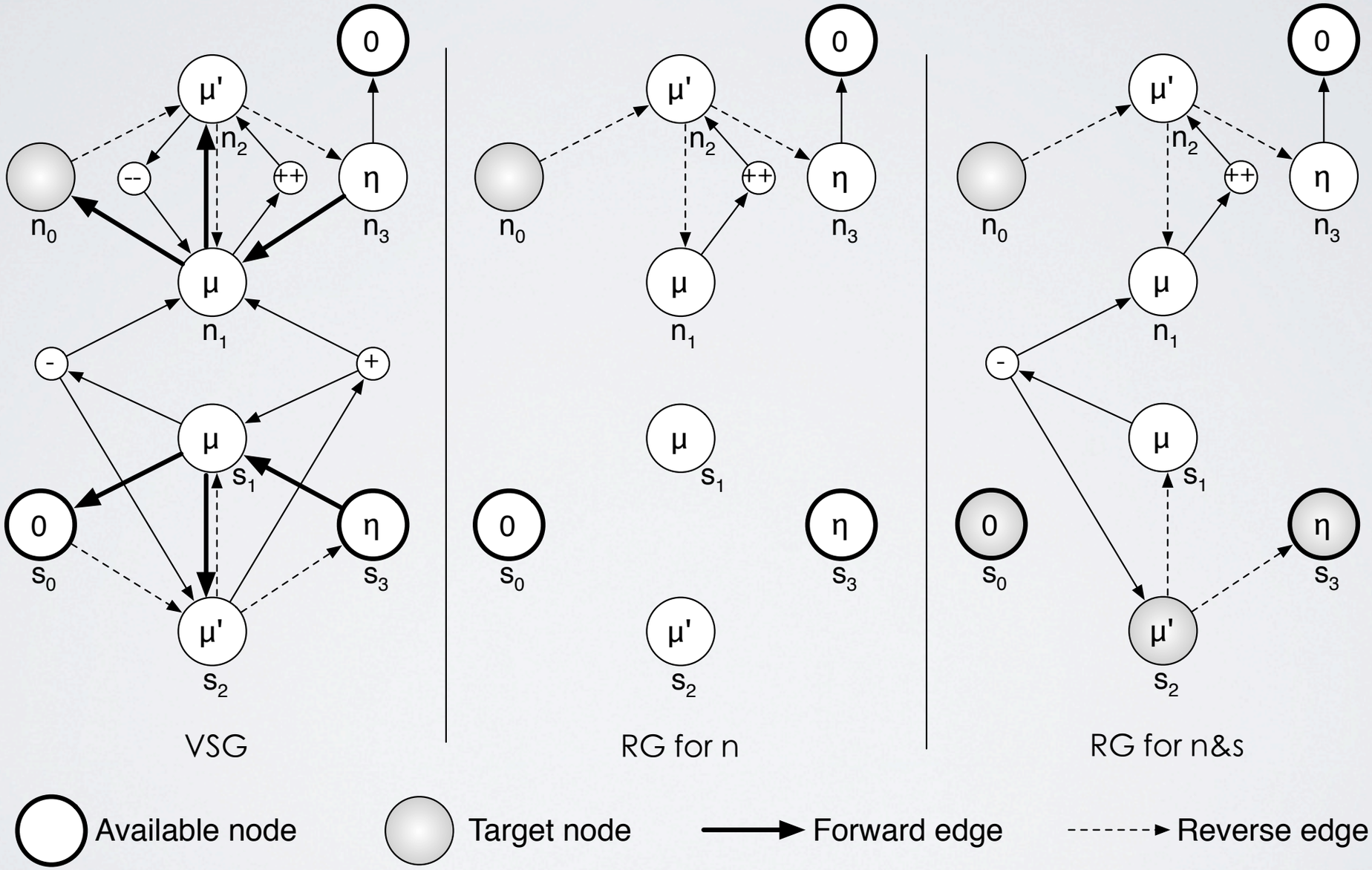
The loop example




CFG in SSA form

# An Example

- The search result of our example.



# An Example

- The original and generated loop:

```
s = 0;
while (n > 0) {
    s = s + n;
    n = n - 1;
}
```

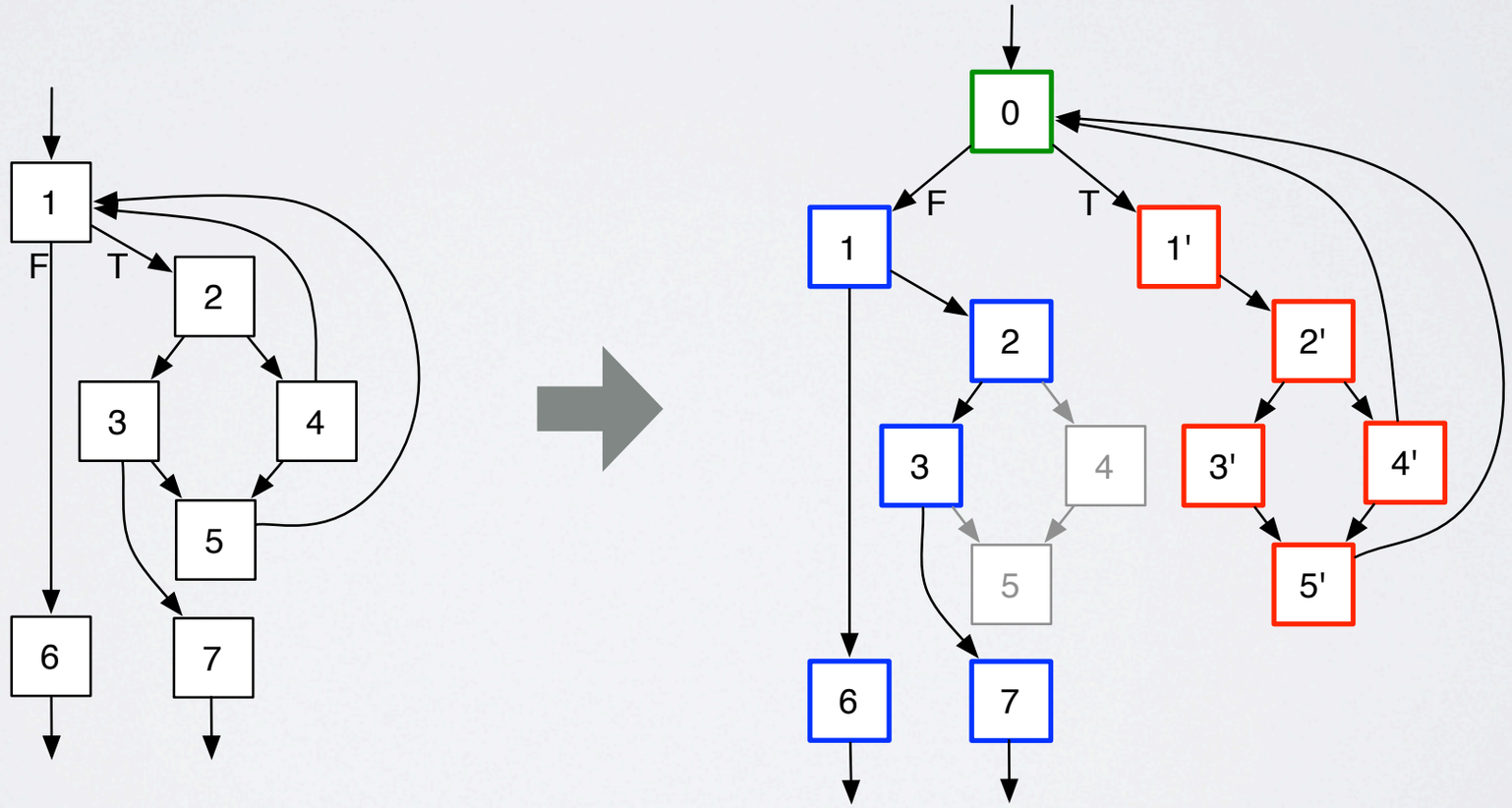
The original loop

```
n = 0;
while (s != 0) {
    n = n + 1;
    s = s - n;
}
```

The generated loop

# Handling Other Loops

- We only consider natural loops (loops with single-entry).
- A non-while loops may be:
  - A loop with several exits.
  - The entry and exit are at different nodes.



# Current And Future Work

- We are working on reversing programs with arrays. Specifically, we are interested in automatically reverse some injective programs like compression/decompression programs.

- We will research on how to rebuild the control flows in the reverse program without path recording.

- Questions?