CS251: Introduction to Language Processing

Machine Independent Optimizations

Vishwesh Jatala

Assistant Professor

Department of CSE

Indian Institute of Technology Bhilai

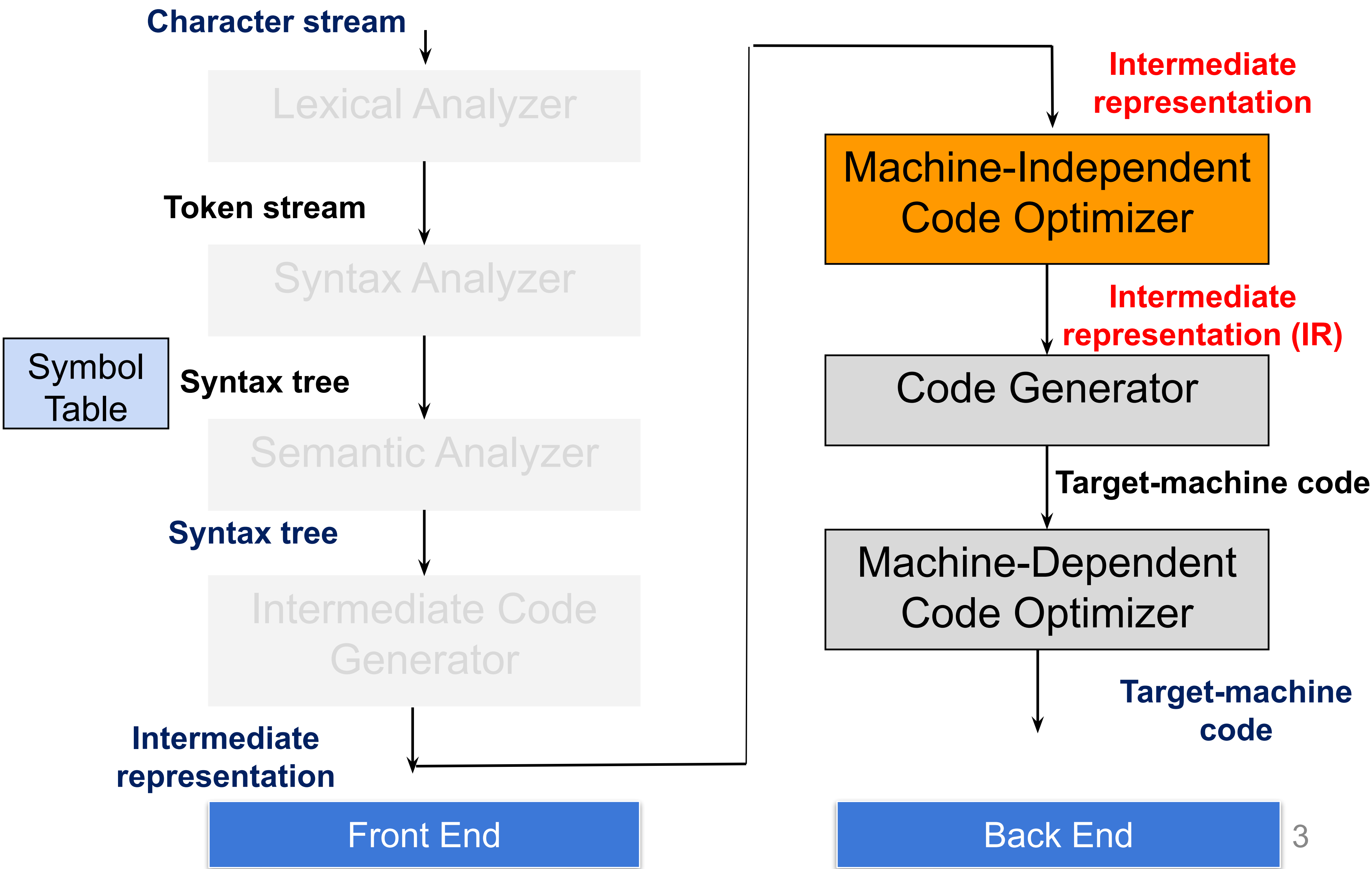
vishwesh@iitbhilai.ac.in

2023-24 M

Acknowledgement

- References for today's slides
  - *Stanford University*
    https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/
  - *Prof. Y. N Srikant, IISc Bangalore*
    https://iith.ac.in/~ramakrishna/Compilers-Aug14/slides/
  - *http://sei.pku.edu.cn/~yaoguo/ACT11/slides/lect2-opt.ppt*
  - *Course textbook*

Compiler Design



Why IR Optimization?

- In order to optimize our IR, we need to understand why it can be improved in the first place.
- **Reason one:** IR generation introduces redundancy.
  - A naïve translation of high-level language features into IR often introduces sub computations.
  - Those sub computations can often be speeded up or eliminated.
- **Reason two:** Programmers are lazy.
  - Code executed inside of a loop can often be factored out of the loop.

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x * x + y
b2 = x * x - y
b3 = x * x * y
```

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x * x + y
b2 = x * x - y
b3 = x * x * y
```

```
t0 = x * x;
t1 = t0 + y;
b1 = t1

t2 = x * x;
t3 = t2 - y;
b2 = t3

t4 = x * x;
t5 = t4 * y;
b3 = t4
```

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x * x + y
b2 = x * x - y
b3 = x * x * y
```

```
t0 = x * x;
t1 = t0 + y;
b1 = t1

t2 = x * x;
t3 = t2 - y;
b2 = t3

t4 = x * x;
t5 = t4 * y;
b3 = t4
```

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x * x + y
b2 = x * x - y
b3 = x * x * y
```

```
t0 = x * x;
t1 = t0 + y;
b1 = t1

t2 = t0 - y;
b2 = t2

t3 = t0 * y;
b3 = t3
```

```
while (x < y + z) {
    x = x - y;
}
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
L0:
    t0 = y + z;
    If x < t0 Goto L1;
    t1 = x - y;
    x = t1
    Goto L0;
L1:
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
L0:
    t0 = y + z;
    If x < t0 Goto L1;
    t1 = x - y;
    x = t1
    Goto L0;
L1:
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
    t0 = y + z;
L0:
    If x < t0 Goto L1;
    t1 = x - y;
    x = t1
    Goto L0;
L1:
```

A Note on Terminology

- The term "optimization" implies looking for an "optimal" piece of code for a program.
- This is, in general, undecidable.
- Our goal will be IR *improvement* rather than IR *optimization*.

The Challenge of Optimization

- A good optimizer
  - Should never change the observable behavior of a program.
  - Should produce IR that is as efficient as possible.
  - Should not take too long to process inputs.
- Unfortunately:
  - Optimizers often miss "easy" optimizations due to limitations of their algorithms.
  - Almost all interesting optimizations are NP-hard or undecidable.

What are we Optimizing?

- Optimizers can try to improve code usage with respect to many observable properties.
- What are some quantities we might want to optimize?

What are we Optimizing?

- Optimizers can try to improve code usage with respect to many observable properties.
- What are some quantities we might want to optimize?
- **Runtime** (make the program as fast as possible at the expense of time and power)
- **Memory usage** (generate the smallest possible executable at the expense of time and power)
- **Power consumption** (choose simple instructions at the expense of speed and memory usage)
- Plus a lot more (minimize function calls, reduce use of floating-point hardware, etc.)

Overview of IR Optimization

- **Formalisms and Terminology** (Today)
  - Control-flow graphs.
  - Basic blocks.
- **Optimizations** (Today)
  - Examples
- **Data flow analysis** (Next lecture)
  - Implementation point of view in compiler

Formalisms and Terminology

Semantics-Preserving Optimizations

- An optimization is **semantics-preserving** if it does not alter the semantics of the original program.
- Examples:
  - Eliminating unnecessary temporary variables.
  - Computing values that are known statically at compile-time instead of runtime.
  - Evaluating constant expressions outside of a loop instead of inside.
- Non-examples:
  - Replacing bubble sort with quicksort.
- The optimizations we will consider in this class are all semantics-preserving.

A Formalism for IR Optimization

- Every phase of the compiler uses some new abstraction:
  - Scanning uses regular expressions.
  - Parsing uses CFGs.
  - Semantic analysis semantic actions and symbol tables.
  - Intermediate code generation uses IRs.
- In optimization, we need a formalism that captures the structure of a program in a way amenable to optimization.

Examples

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

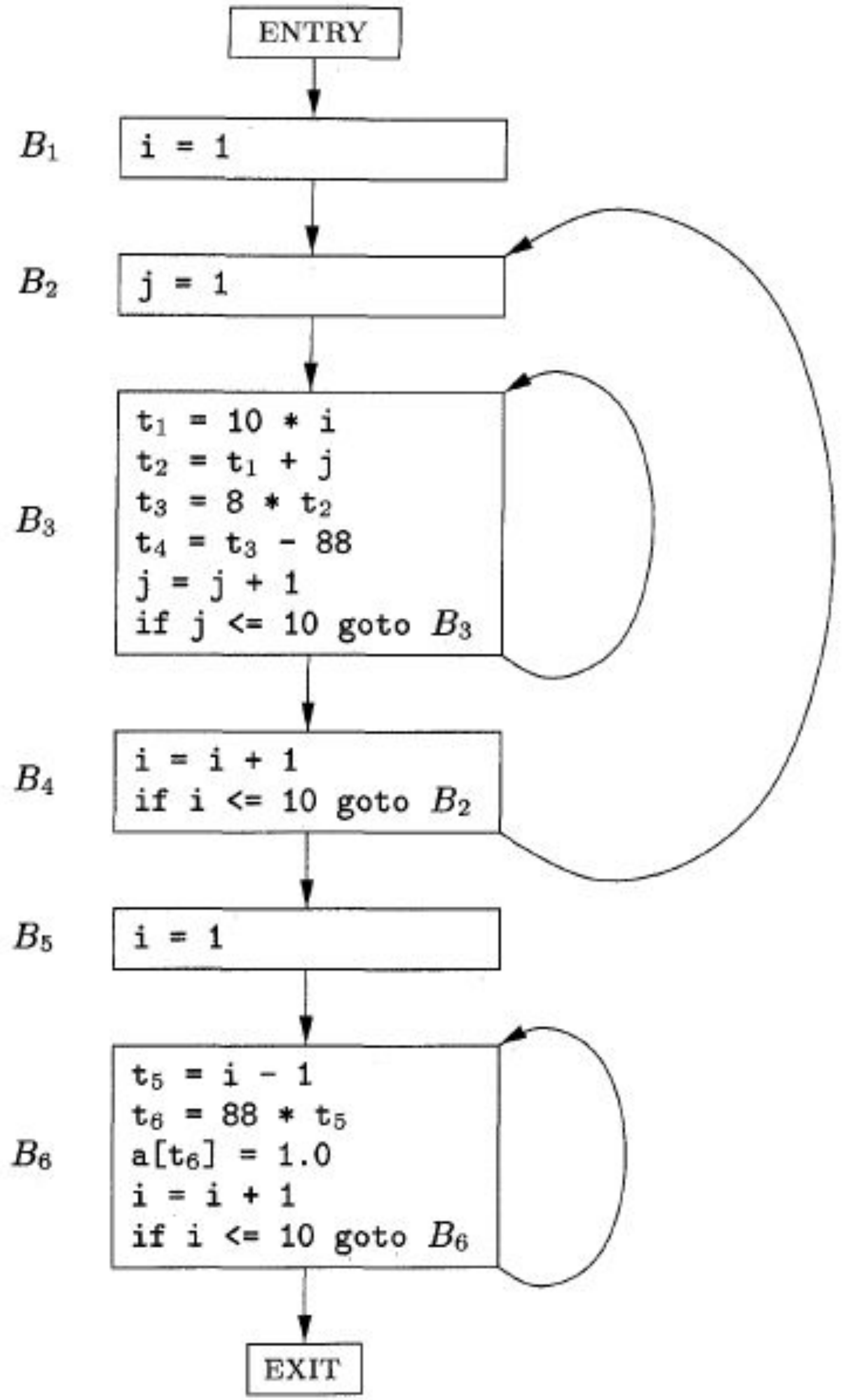
```
for i from 1 to 10 do
  for j from 1 to 10 do
     a[i,j]=0.0

for i from 1 to 10 do
     a[i,i]=0.0
```

Control Flow

```
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i,j]=0.0

for i from 1 to 10 do
    a[i,i]=0.0
```



Basic Blocks

- A **basic block** is a maximal sequence of consecutive three-address instructions with the following properties:
  - The flow of control can only enter the basic block thru the 1st instr.
  - There is exactly one spot where control leaves the sequence, which must be at the end of the sequence.
- Basic blocks become the nodes of a **flow graph**, with edges indicating the order.

Identifying Basic Blocks

- Input: sequence of instructions *instr(i)*
- Output: A list of basic blocks
- Method:
  - Identify **leaders**:

    the first instruction of a basic block
  - Iterate: add subsequent instructions to basic block until we reach another leader

Identifying Leaders

- Rules for finding leaders in code
  - First instr in the code is a leader
  - Any instr that is the target of a (conditional or unconditional) jump is a leader
  - Any instr that immediately follow a (conditional or unconditional) jump is a leader

Basic Block Partition Algorithm

```
leaders = {1}              // start of program
for i = 1 to |n|           // all instructions
    if instr(i) is a branch
     leaders = leaders U  targets of instr(i) U  instr(i+1)
worklist = leaders
While worklist not empty
    x = first instruction in worklist
    worklist = worklist – {x}
    block(x) = {x}
    for i = x + 1; i <= |n| && i not in leaders; i++
     block(x) = block(x) U {i}
```

Basic Block Example

```
1.  i = 1                    A
2.  j = 1                    B
3.  t1 = 10 * i
4.  t2 = t1 + j
5.  t3 = 8 * t2
6.  t4 = t3 - 88             C
7.  a[t4] = 0.0
8.  j = j + 1
9.  if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)      D
12. i = 1                    E
13. t5 = i - 1
14. t6 = 88 * t5
15. a[t6] = 1.0              F
16. i = i + 1
17. if i <= 10 goto (13)
```

Leaders

Basic Blocks

Control Flow Graphs

- **Control-flow graph**:
  - Node: an instruction or sequence of instructions (a basic block)
    - Two instructions i, j in same basic block *iff* execution of i *guarantees* execution of j
  - Directed edge: *potential* flow of control
  - Distinguished start node *Entry & Exit*
    - First & last instruction in program

Control Flow Edges

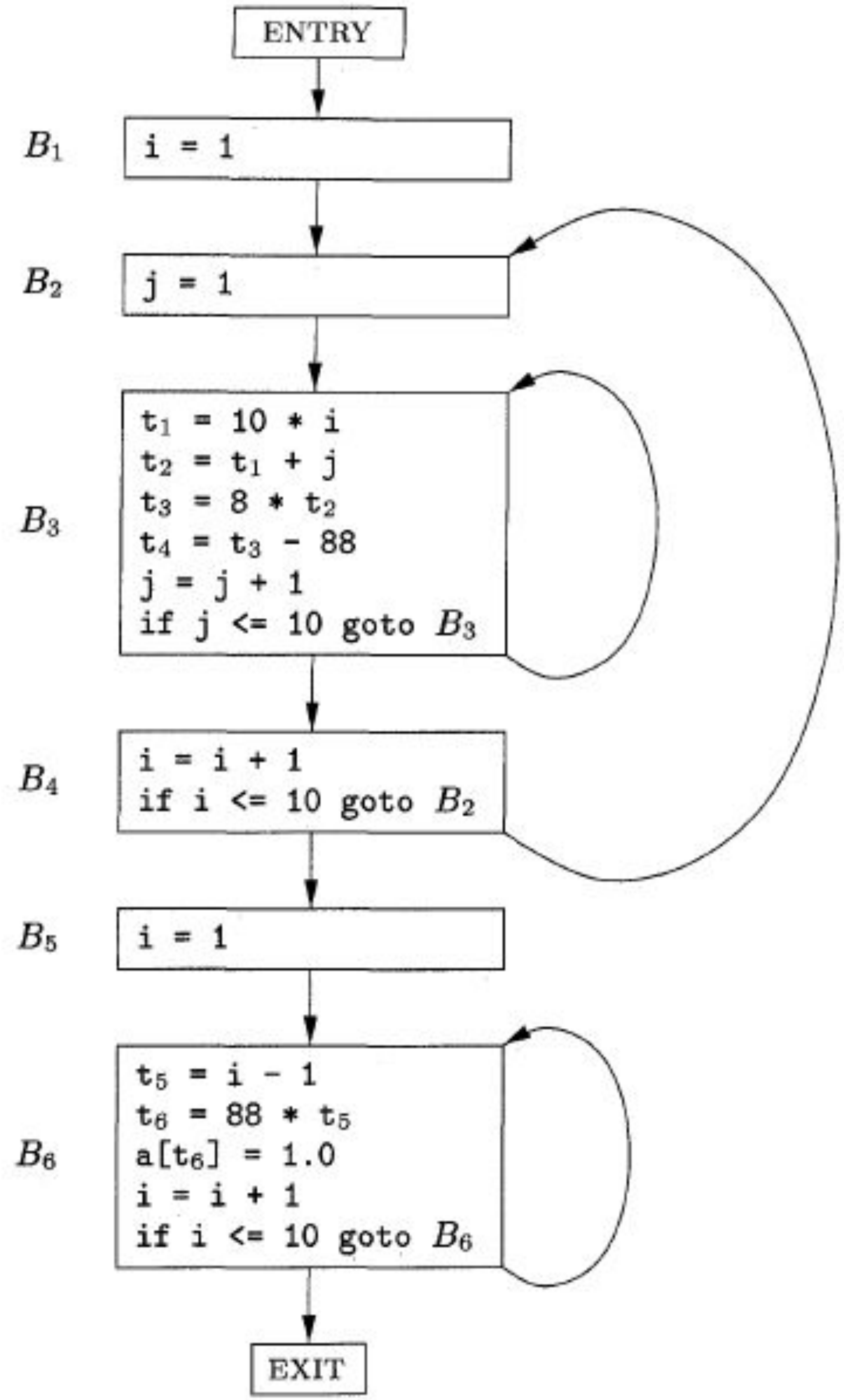
- Basic blocks = nodes
- Edges:
  - Add directed edge between B1 and B2 if:
    - Branch from last statement of B1 to first statement of B2 (B2 is a leader), or
    - B2 immediately follows B1 in program order and B1 does not end with unconditional branch (goto)
  - Definition of predecessor and successor
    - B1 is a predecessor of B2
    - B2 is a successor of B1

Control Flow Algorithm

Input: block(i), sequence of basic blocks
Output: CFG where nodes are basic blocks

```
for i = 1 to the number of blocks
    x = last instruction of block(i)
    if instr(x) is a branch
     for each target y of instr(x),
          create edge (i -> y)
    if instr(x) is not unconditional branch,
     create edge (i -> i+1)
```

CFG Example



ENTRY
B_1
i = 1
B_2
j = 1
B_3
t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6
EXIT

Optimizations

- Global common subexpression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction

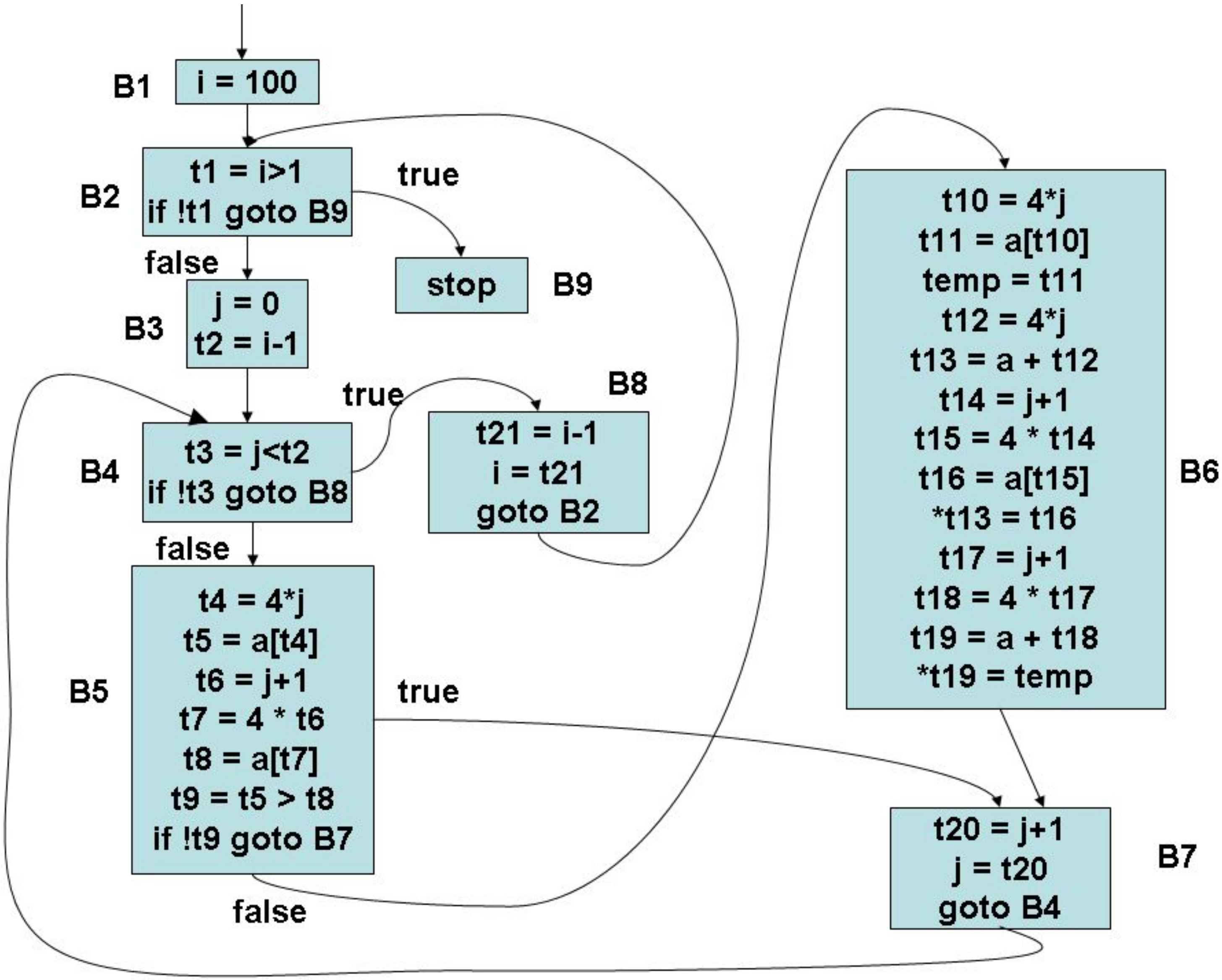
Example

Bubble Sort

```
for (i=100; i>1; i--) {
    for (j=0; j<i-1; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j+1] = a[j];
            a[j] = temp;
        }
    }
}
```

- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted

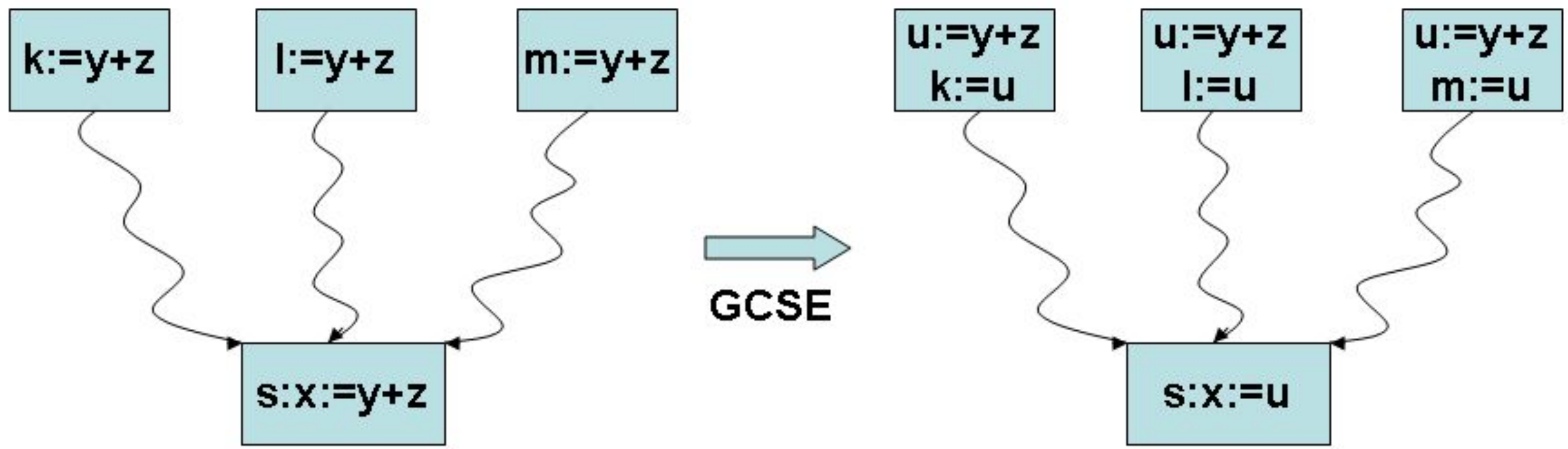
Control Flow Graph of Bubble Sort



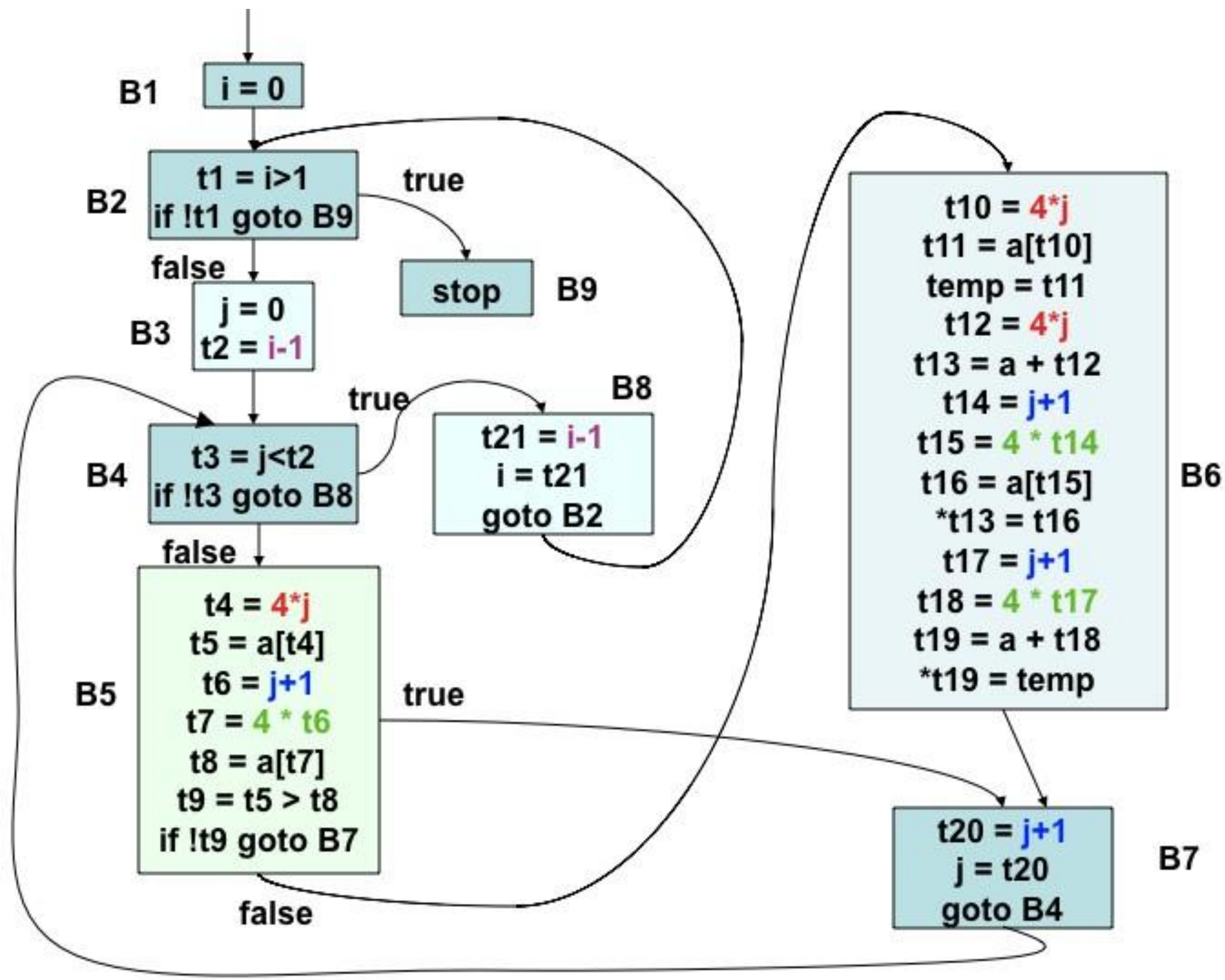
Optimizations

- Global common subexpression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction

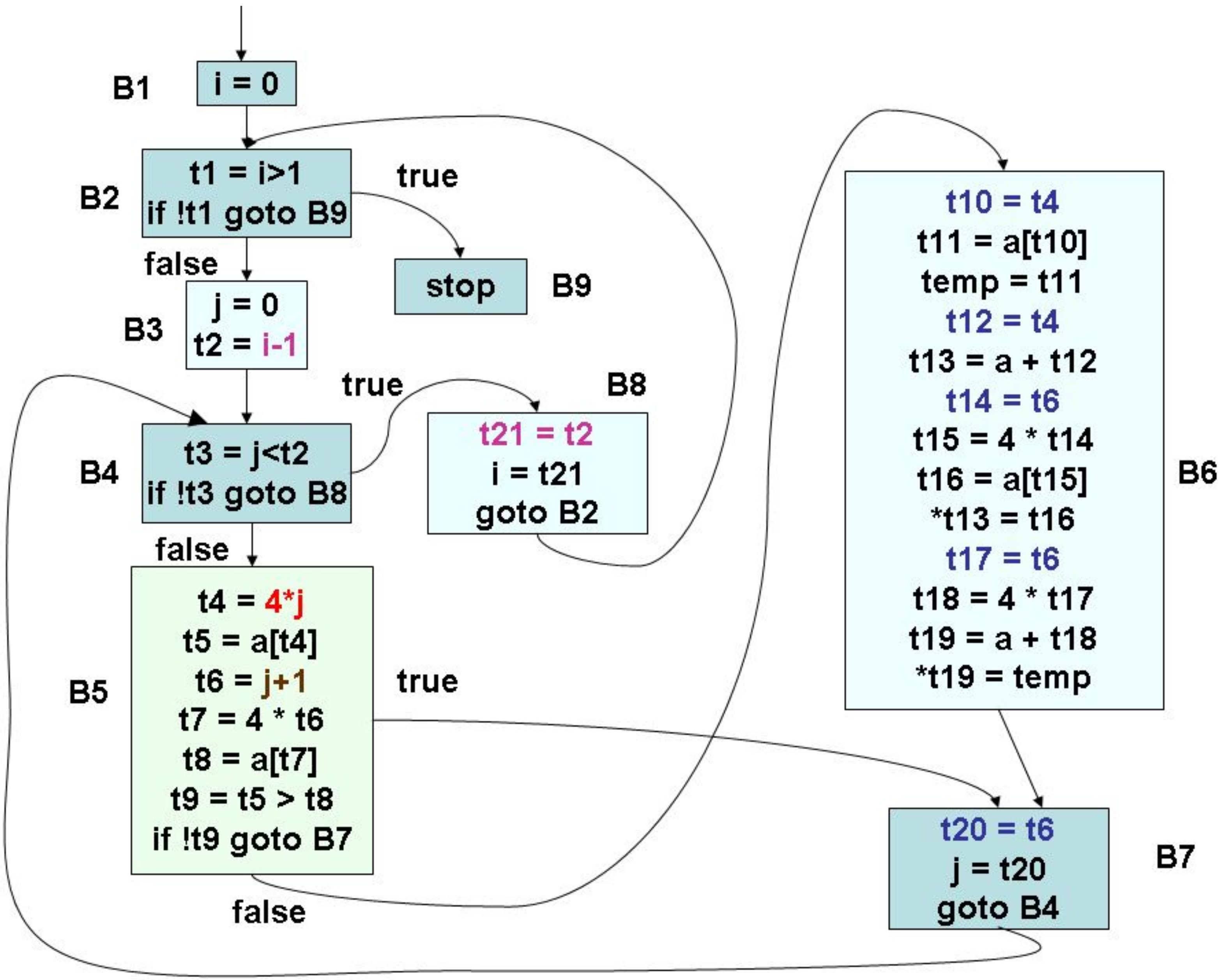
Global Common Subexpression Elimination (GCSE)



Global Common Subexpression Elimination (GCSE)



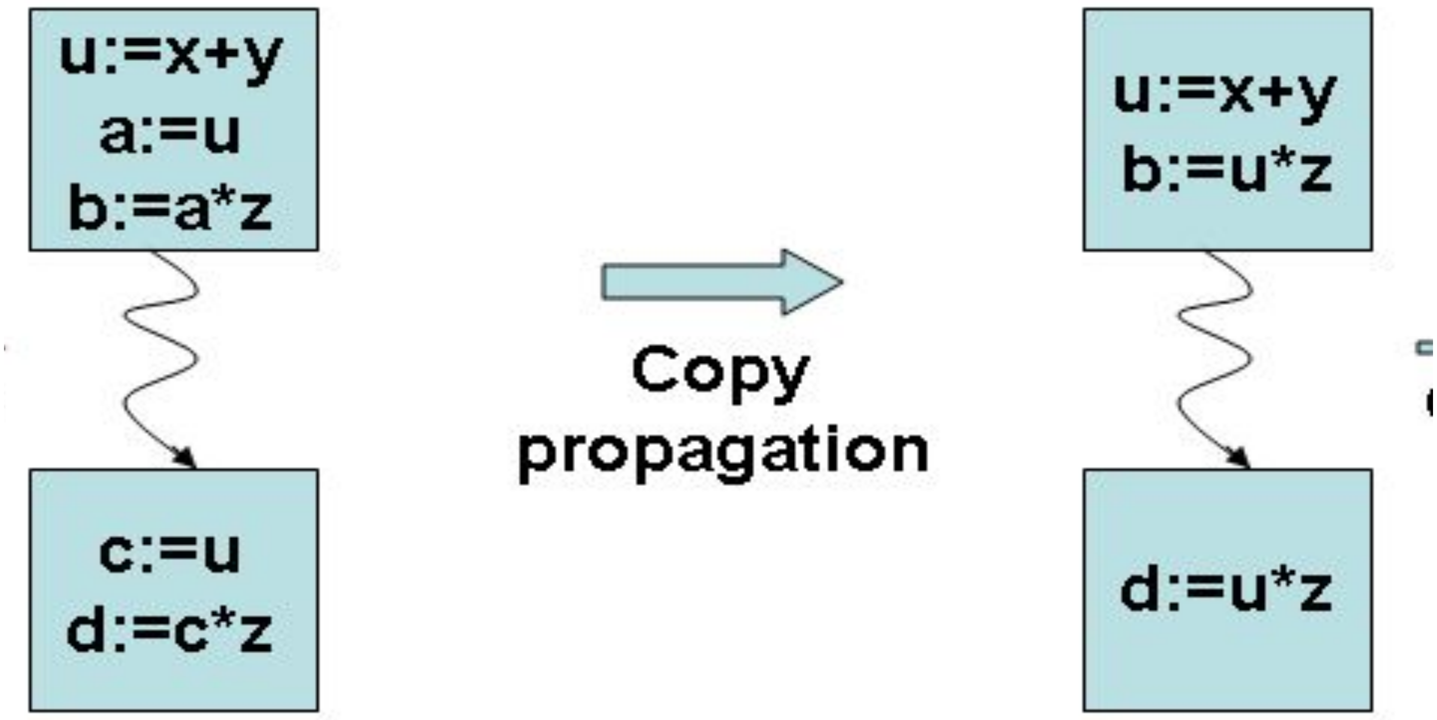
Global Common Subexpression Elimination (GCSE)



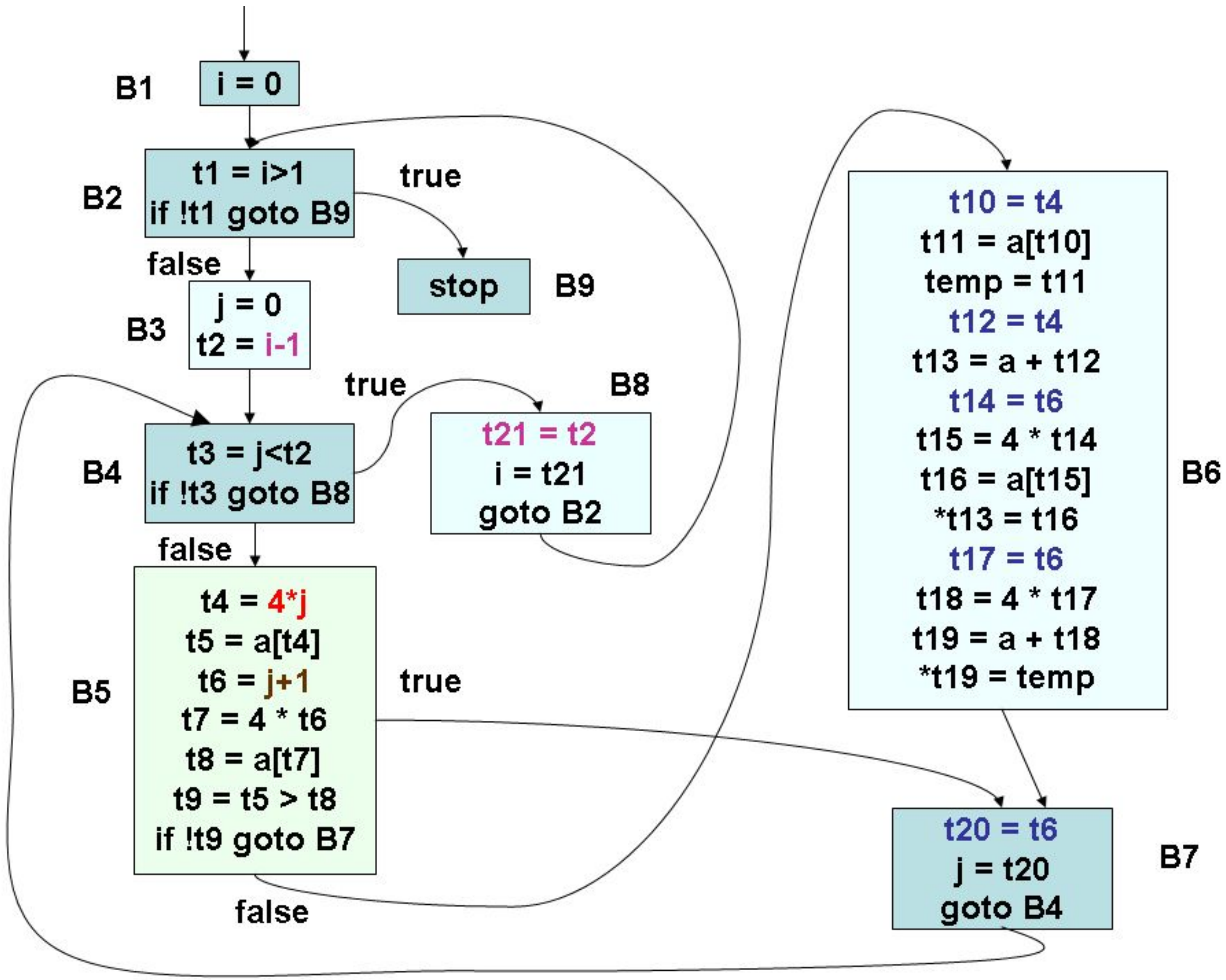
Optimizations

- Global common subexpression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction

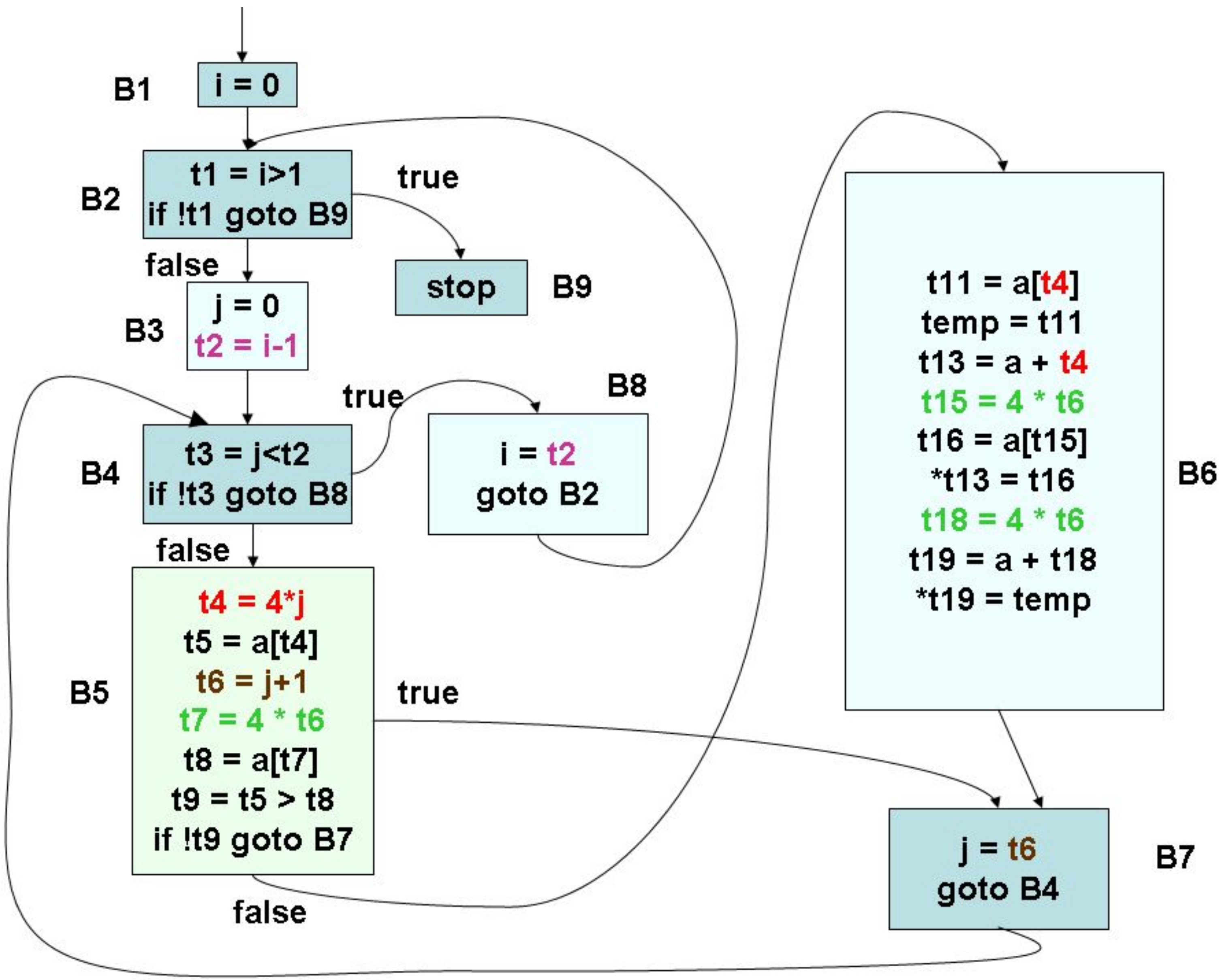
Copy Propagation



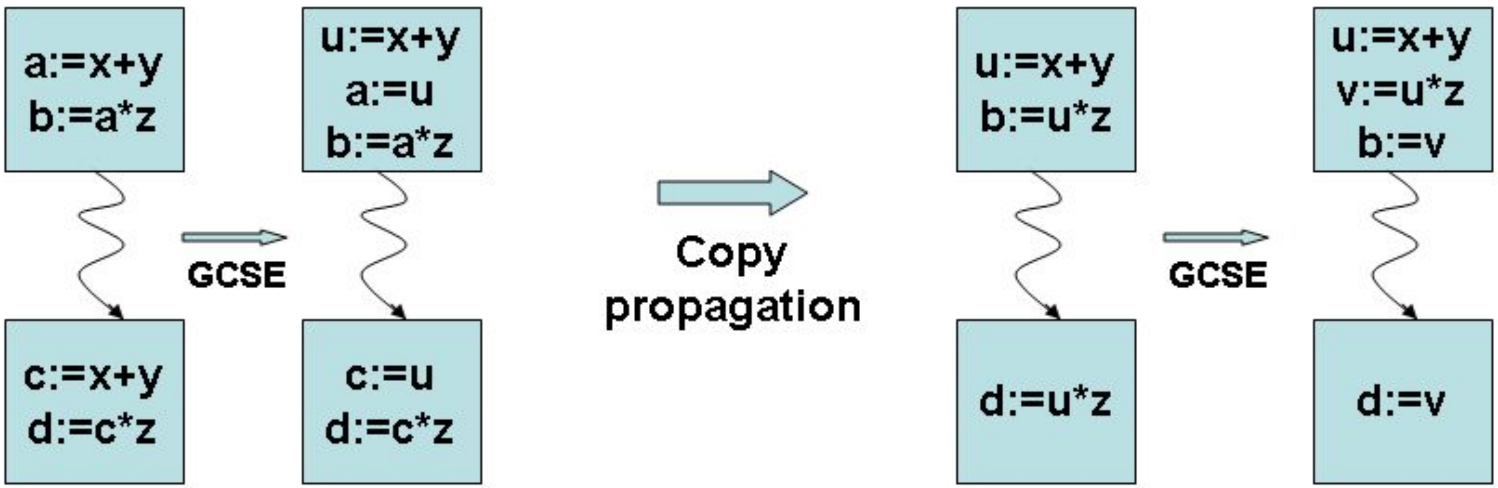
Global Common Subexpression Elimination (GCSE)



Copy Propagation

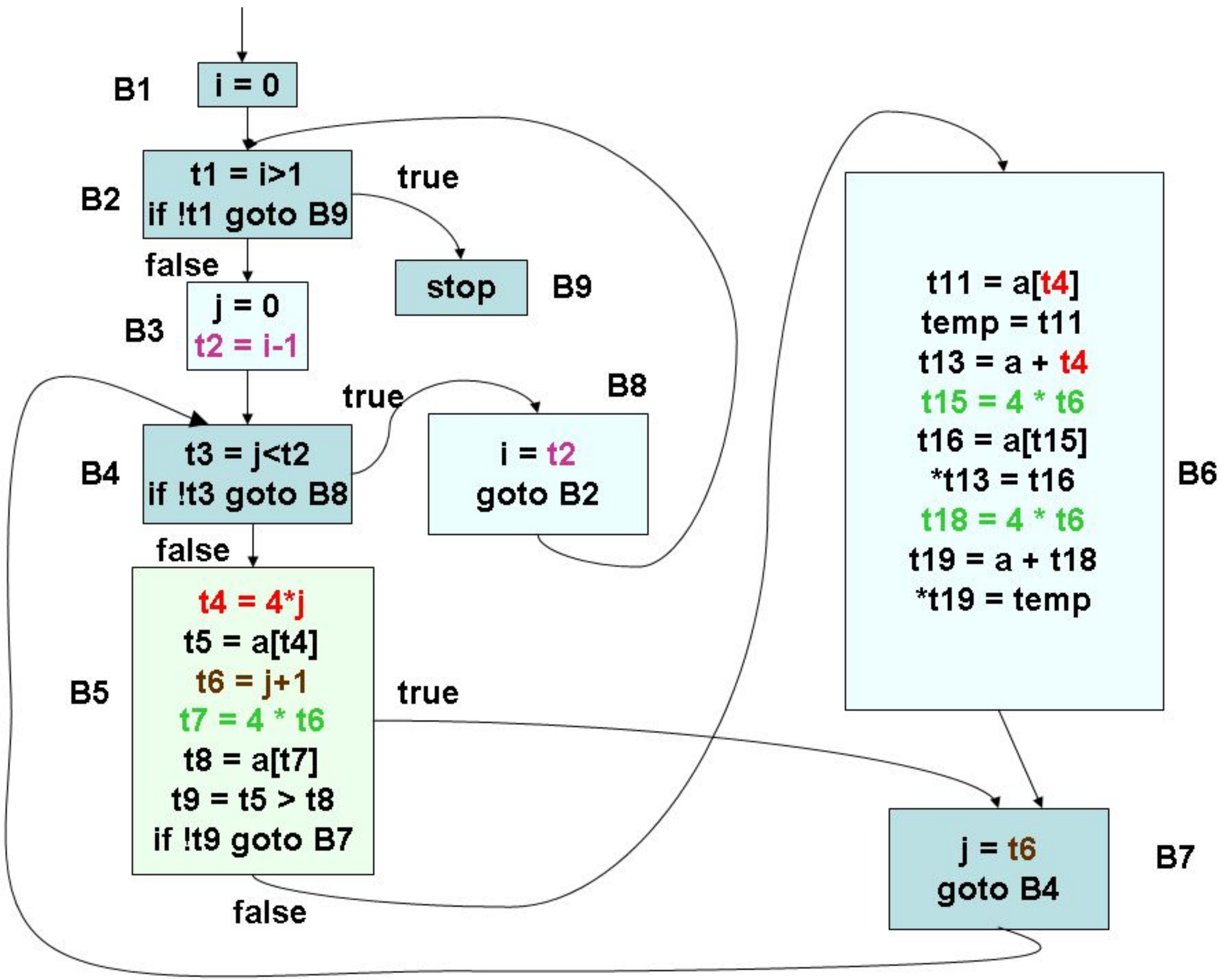


GCSE and Copy Propagation

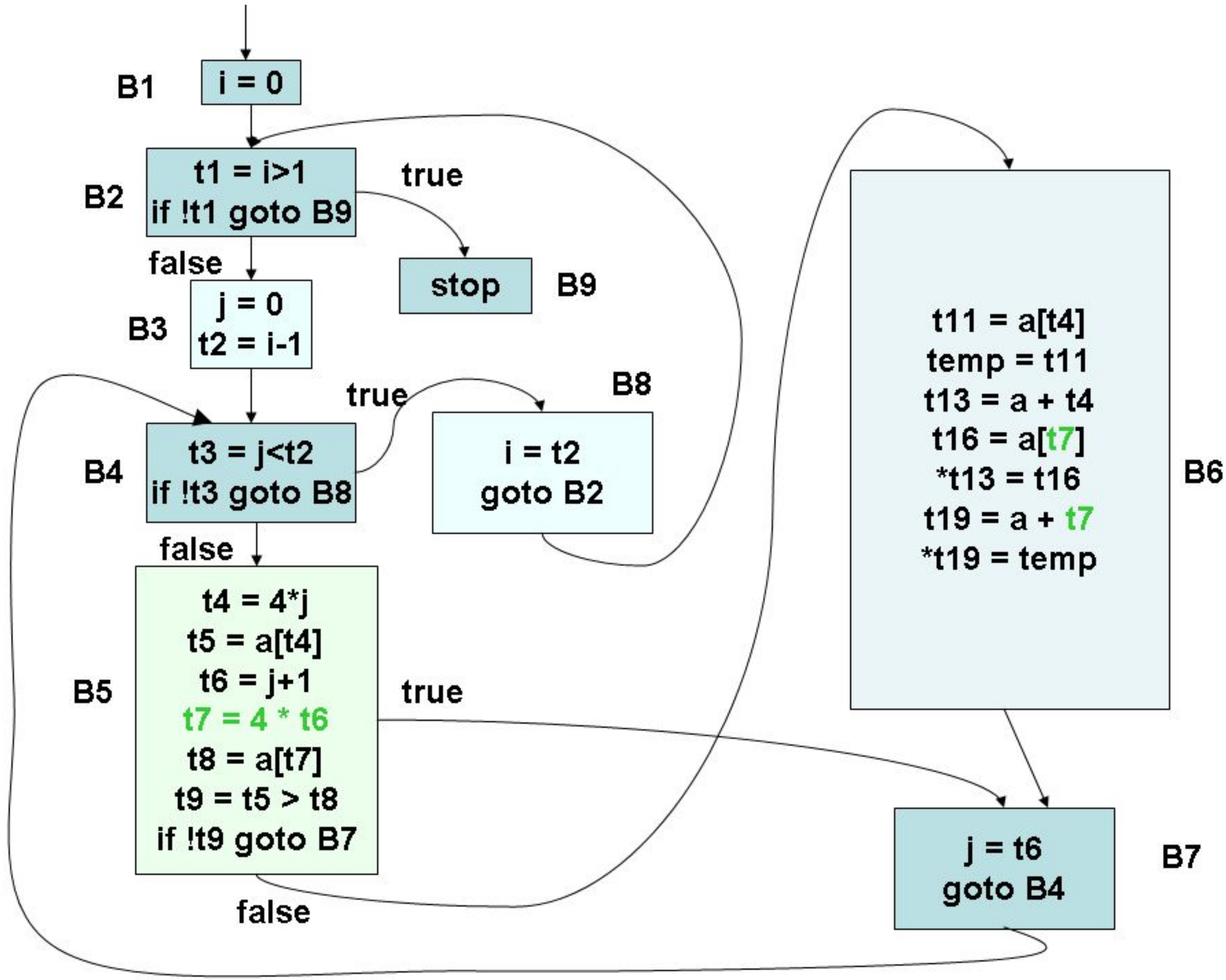


Demonstrating the need for repeated application of GCSE

GCSE and Copy Propagation



GCSE and Copy Propagation



Optimizations

- Global common subexpression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction

Copy Propagation and Constant Folding