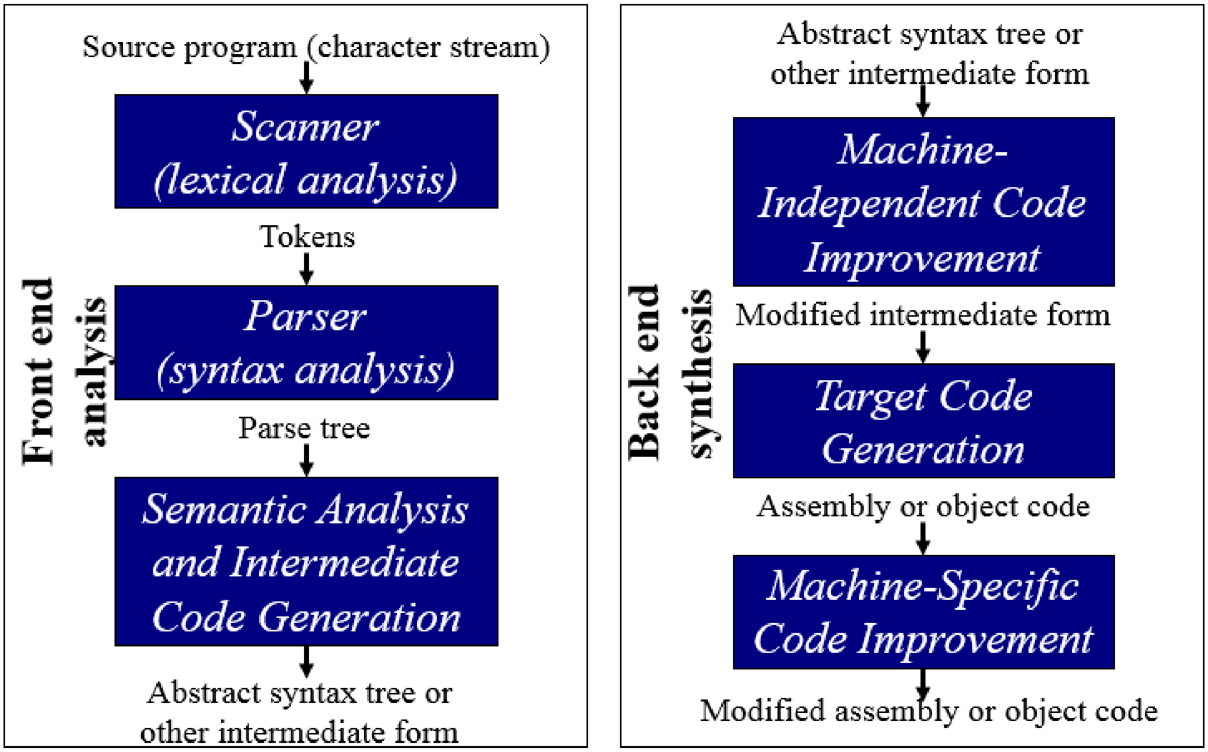
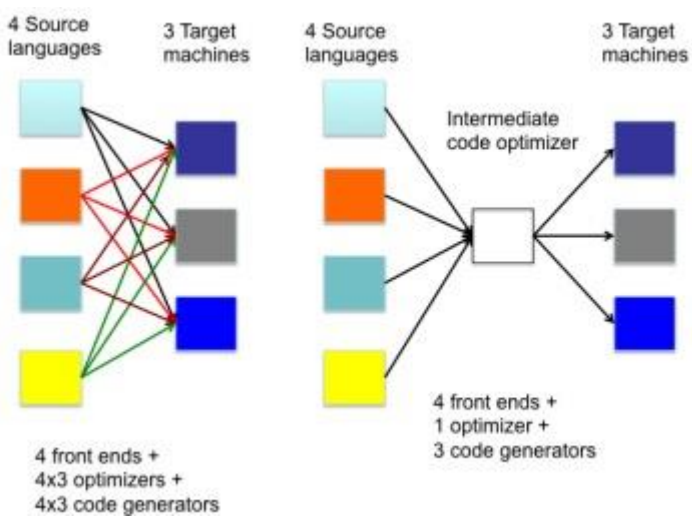
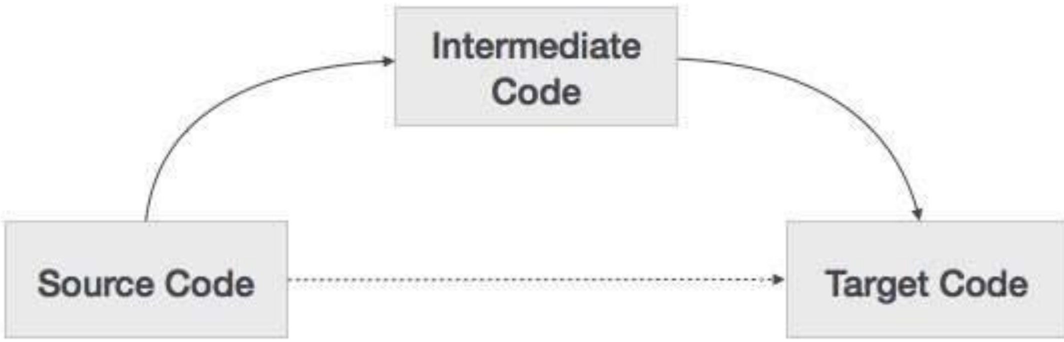
Compiler Front- and Back-end





A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portions same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Example-1

C-Program

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Intermediate code

```
    dot_prod = 0;          |    T6 = T4[T5]
    i = 0;                 |    T7 = T3*T6
L1: if(i >= 10)goto L2     |    T8 = dot_prod+T7
    T1 = addr(a)           |    dot_prod = T8
    T2 = i*4               |    T9 = i+1
    T3 = T1[T2]            |    i = T9
    T4 = addr(b)           |    goto L1
    T5 = i*4               |L2:
```

C-Program

```
int a[10], b[10], dot_prod, i; int* a1; int* b1;
dot_prod = 0; a1 = a; b1 = b;
for (i=0; i<10; i++) dot_prod += *a1++ * *b1++;
```

Intermediate code

```
     dot_prod = 0;          |     b1 = T6
     a1 = &a                |     T7 = T3*T5
     b1 = &b                |     T8 = dot_prod+T7
     i = 0                  |     dot_prod = T8
L1:  if(i>=10)goto L2       |     T9 = i+1
     T3 = *a1               |     i = T9
     T4 = a1+1              |     goto L1
     a1 = T4                |L2:
     T5 = *b1
     T6 = b1+1
```

BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

- A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

- The following sequence of three-address statements forms a basic block

- t1 : = a * a

  t2 : = a * b

  t3 : = 2 * t2

t4 : = t1 + t3

t5 : = b * b

t6 : = t4 + t5

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block Method:

1. We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.

   c. Any statement that immediately follows a goto or conditional goto statementis a leader.

2. For each leader, its basic block consists of the leader and all statements up to

but not including the next leader or the end of the program.

Consider the following source code for dot product of two vectors:

```
begin
        prod :=0;
        i:=1;
        do begin
        prod :=prod+ a[i] * b[i];
        i :=i+1;
        end
        while i <= 20
end
```

The three-address code for the above source program is given as :

(1) prod := 0

(2) i := 1

(3) t1 := 4* i

(4) t2 := a[t1] /*compute a[i] */

(5) t3 := 4* i

(6) t4 := b[t3] /*compute b[i] */

(7) t5 := t2*t4

(8) t6 := prod+t5

(9) prod := t6

(10) t7 := i+1

(11) i := t7

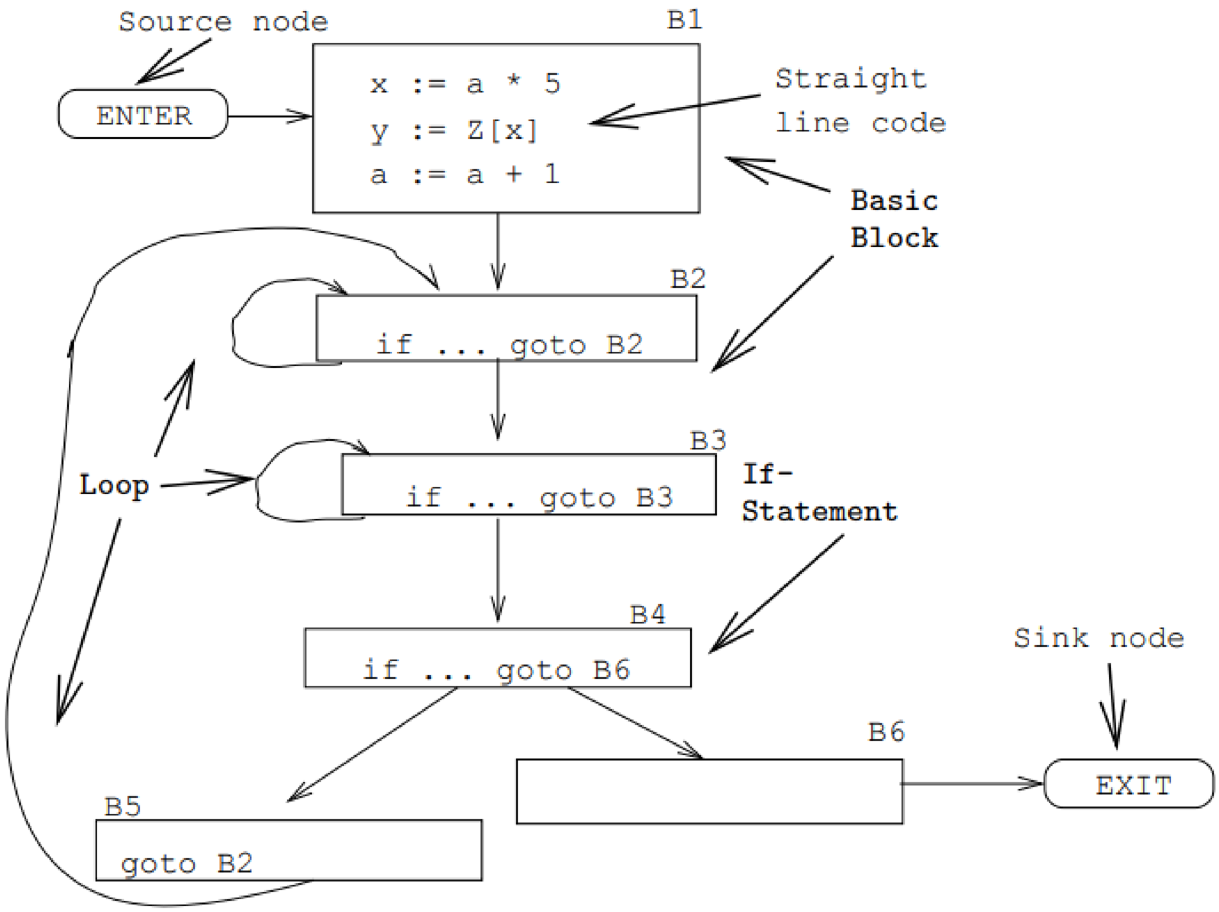
(12) **if i<=20 goto (3)**

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

Control Flow Graphs

- We divide the intermediate code of each procedure into basic blocks. A basic block is a piece of straight line code, i.e. there are no jumps in or out of the middle of a block.
- The basic blocks within one procedure are organized as a *(control) flow graph*, or *CFG*. A flow-graph has
  - basic blocks $B_1 \cdots B_n$ as nodes,
  - a directed edge $B_1 \longrightarrow B_2$ if control can flow from B_1 to B_2.
  - Special nodes $\boxed{\texttt{ENTER}}$ and $\boxed{\texttt{EXIT}}$ that are the *source* and *sink* of the graph.
- Inside each basic block can be any of the IRs we've seen: tuples, trees, DAGs, etc.



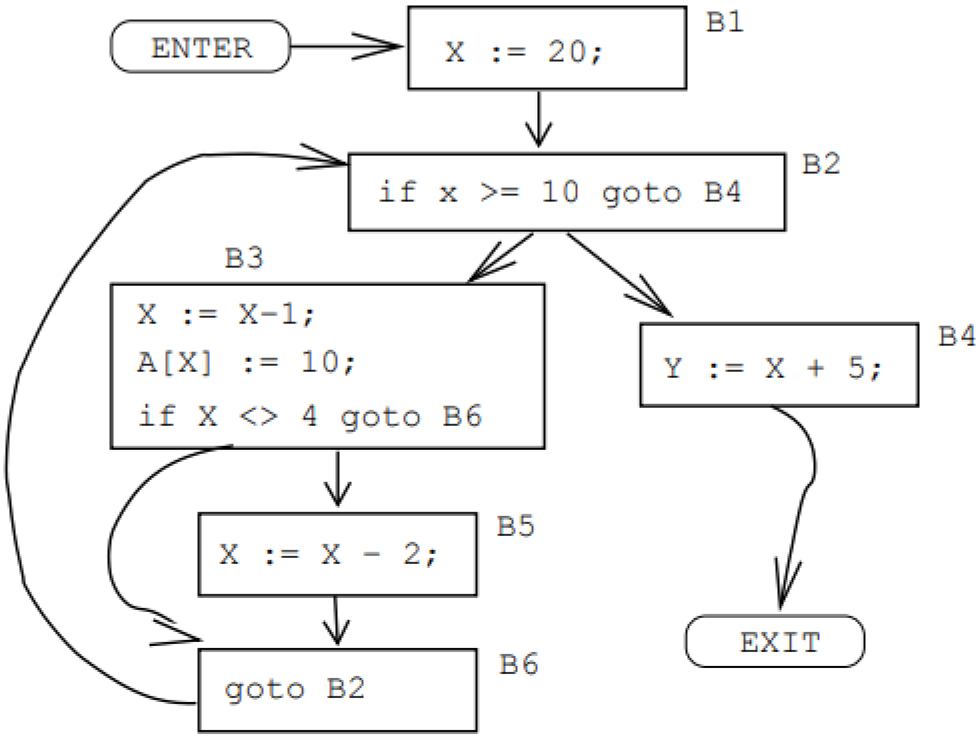
Source Code:

X := 20; **WHILE** X < 10 **DO**
  X := X-1; A[X] := 10;
  IF X = 4 **THEN** X := X - 2; **ENDIF**;
ENDDO; Y := X + 5;

Intermediate Code:

(1) X := 20	(5) if X<>4 goto (7)
(2) if X>=10 goto (8)	(6) X := X-2
(3) X := X-1	(7) goto (2)
(4) A[X] := 10	(8) Y := X+5

Flow Graph:



- Draw the control flow graph for the tuples.

```
int A[5],x,i,n;
for (i=1; i<=n; i++) {
  if (i<n) {
    x = A[i];
  } else {
    while (x>4) {
      x = x*2+A[i];
    };
  };
  x = x+5;
}
```

```
(1) i := 1
(2) IF i>n GOTO (14)
(3) IF i>=n GOTO (6)
(4) x := A[i]
(5) GOTO (11)
(6) IF x<=4 GOTO (11)
(7) T1 := x*2
(8) T2 := A[i]
(9) x := T1+T2
```

```
(10)  GOTO (6)
(11)  x := x+5
(12)  i := i+1
(13)  GOTO (2)
```

Code Optimization:

- What is code optimization?
- Types of code optimizations
- Illustrations of code optimizations

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)

Examples of Machine-Independant Optimizations

- Global common sub-expression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Partial redundancy elimination
- Loop unrolling
- Function inlining
- Tail recursion removal
- Vectorization and Concurrentization
- Loop interchange, and loop blocking

- Code optimization needs information about the program
  - which expressions are being recomputed in a function?
  - Which expressions are partially redundant?
  - which definitions reach a point?
  - Which copies and constants can be propagated? Etc.
- All such information is gathered through data-flow analysis

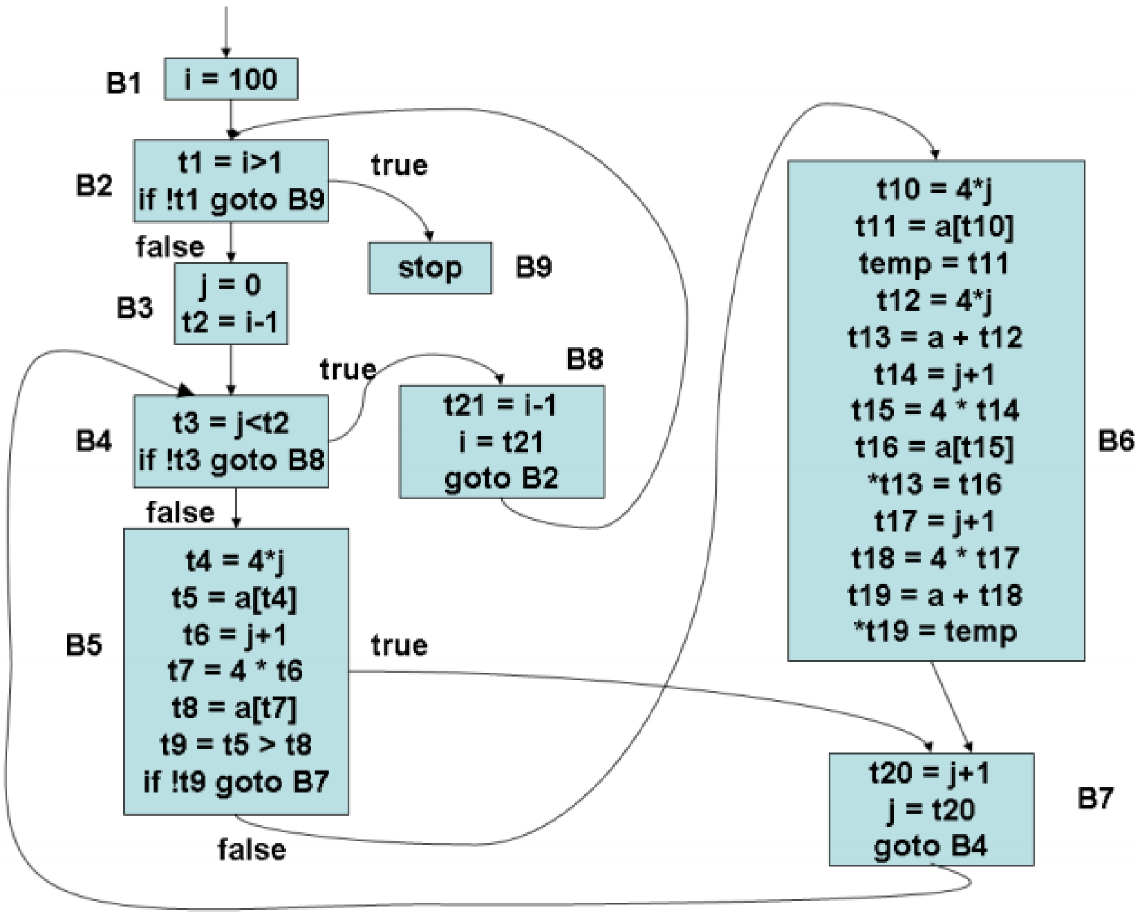
Bubble Sort Running Example

Bubble Sort

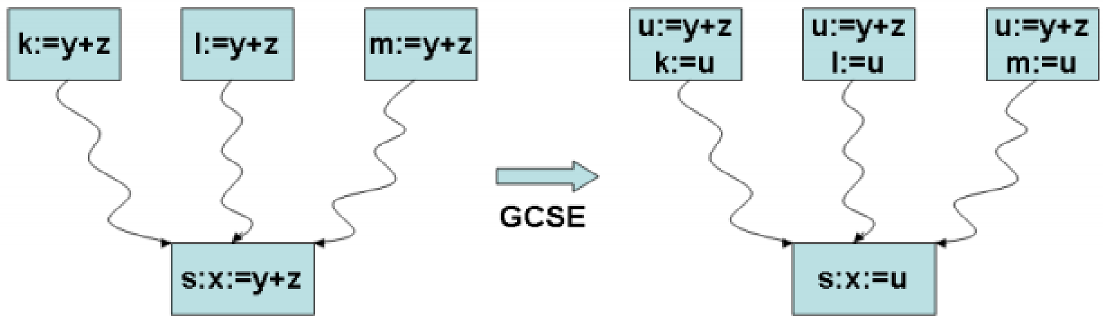
```
for (i=100; i>1; i--) {
    for (j=0; j<i-1; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j+1] = a[j];
            a[j] = temp;
        }
    }
}
```

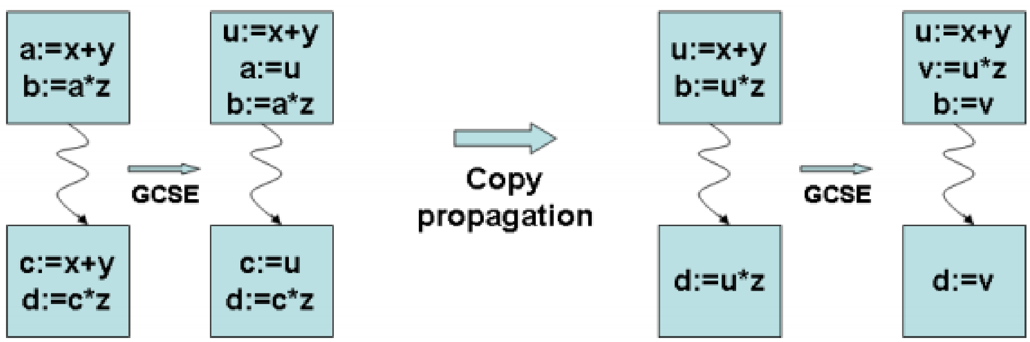
- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted

Control Flow Graph of Bubble Sort



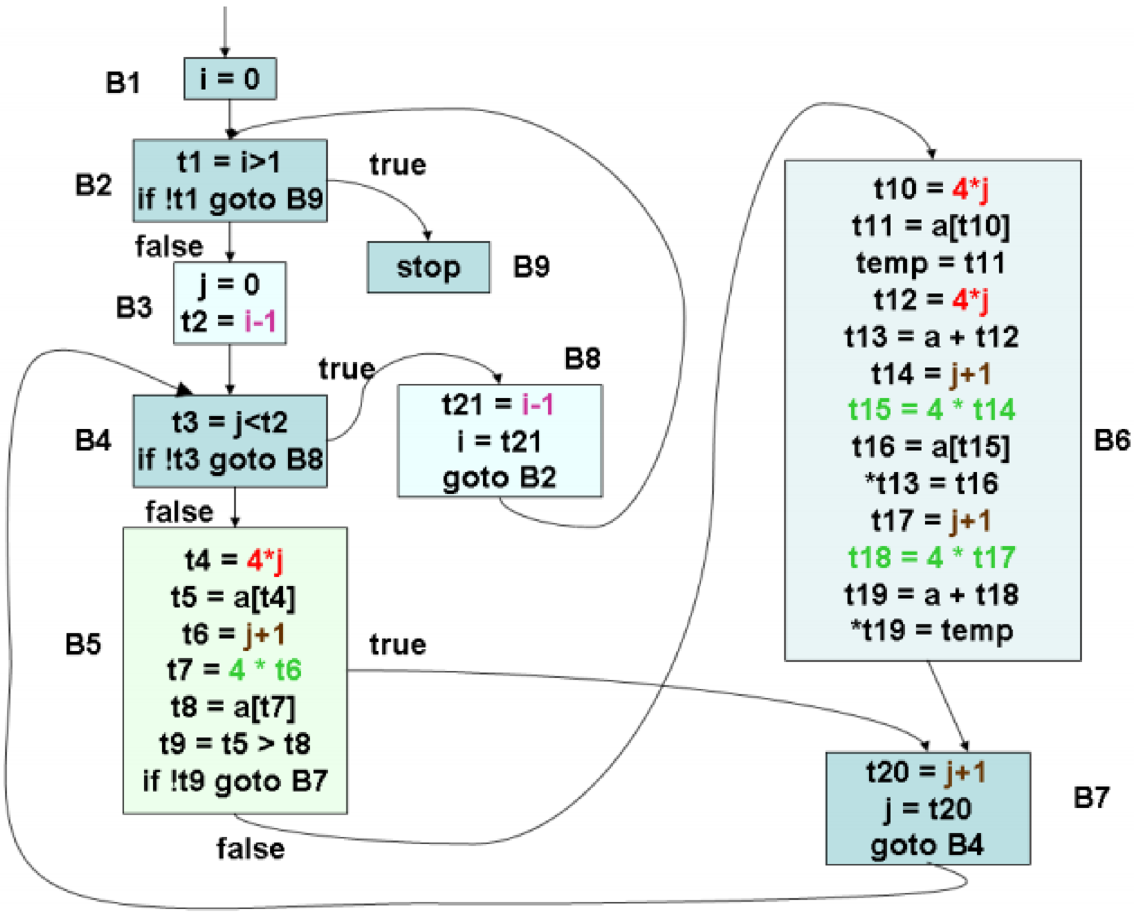
GCSE Conceptual Example



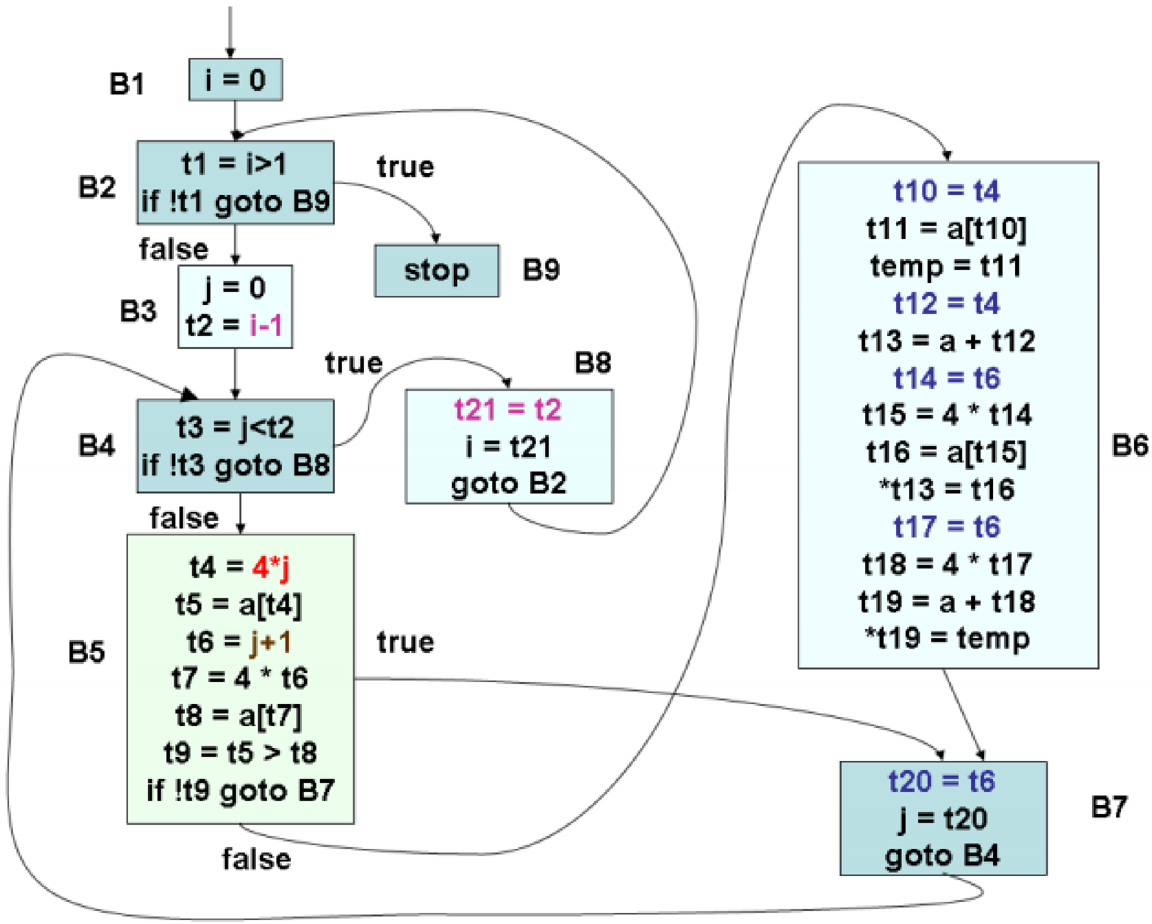


Demonstrating the need for repeated application of GCSE

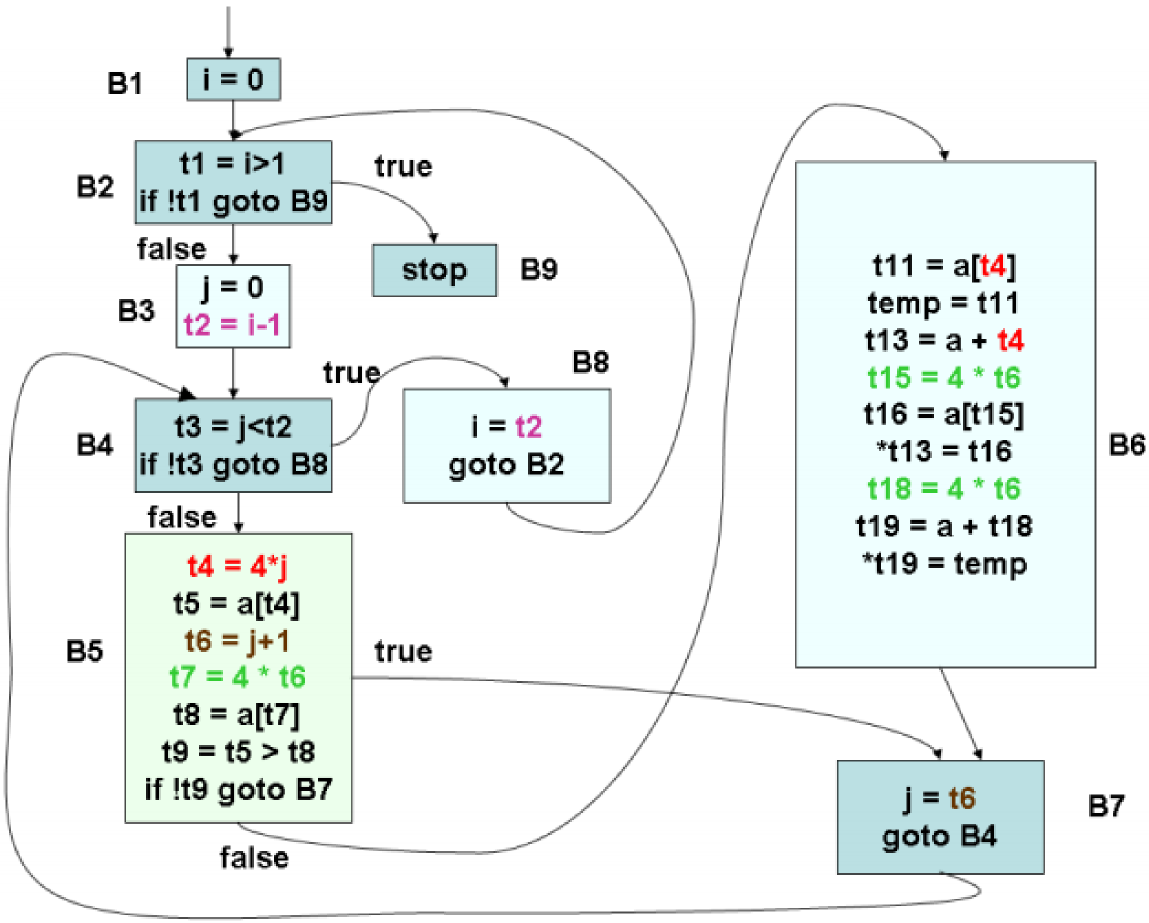
GCSE on Running Example - 1



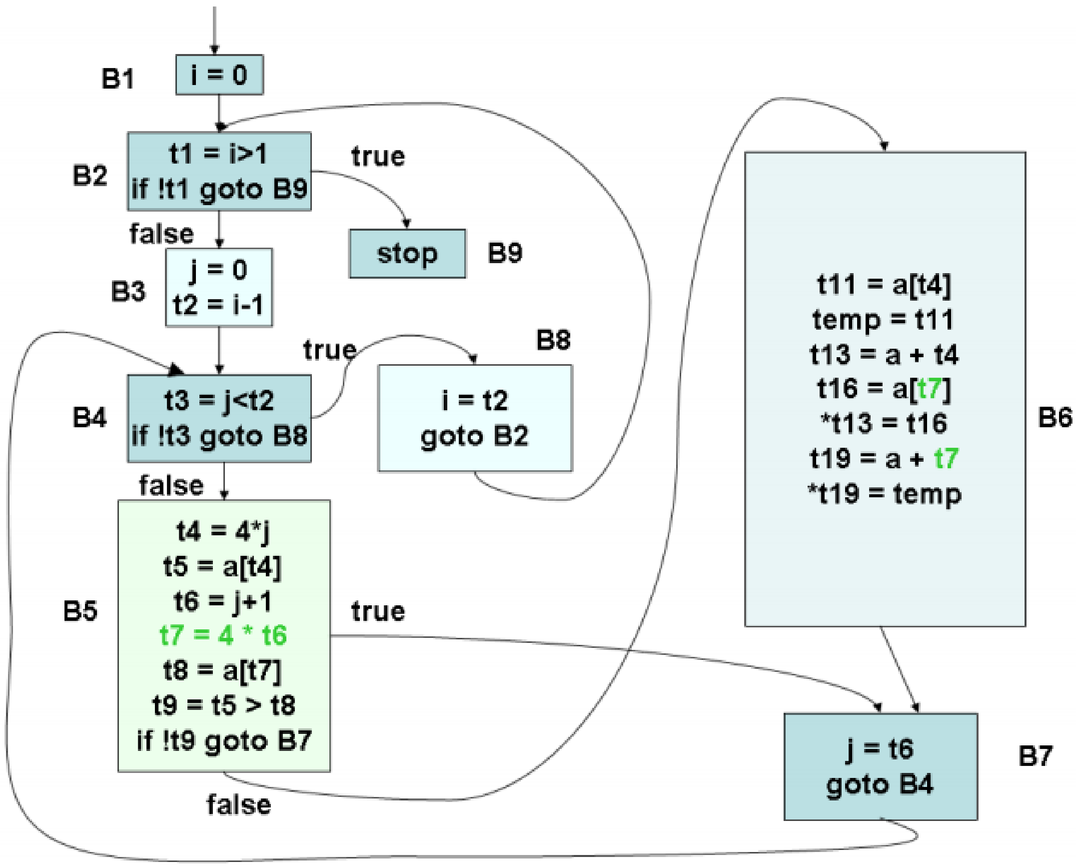
GCSE on Running Example - 2



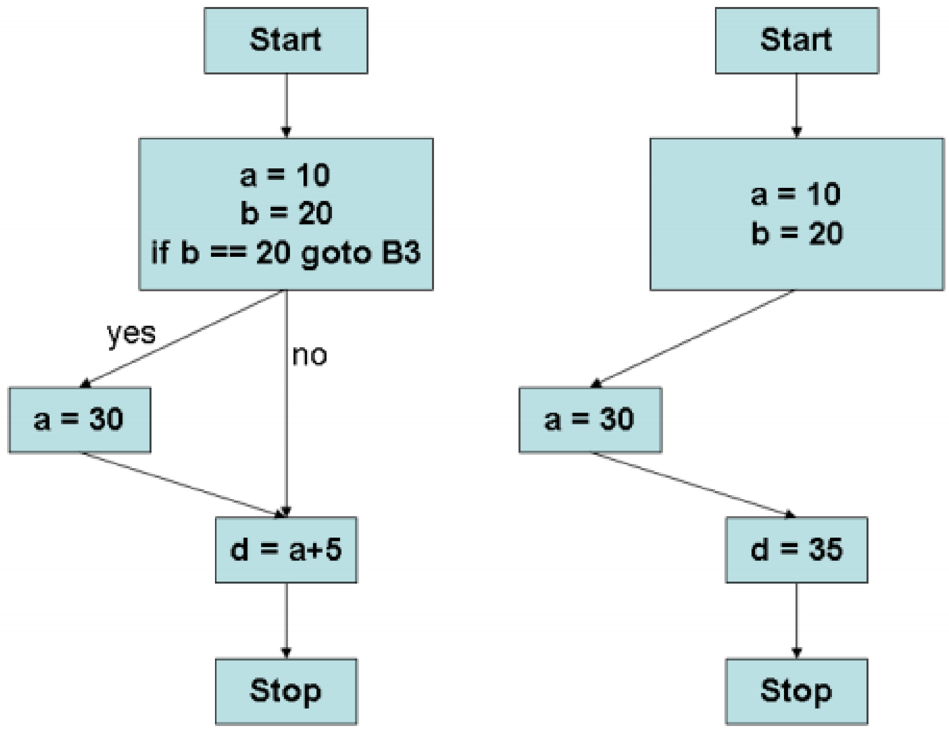
Copy Propagation on Running Example



GCSE and Copy Propagation on Running Example



Constant Propagation and Folding Example



Before constant propagation

After constant propagation and folding

Loop Invariant Code motion Example

```
     t1 = 202
     i = 1
L1:  t2 = i>100
     if t2 goto L2
     t1 = t1-2
     t3 = addr(a)
     t4 = t3 - 4
     t5 = 4*i
     t6 = t4+t5
     *t6 = t1
     i = i+1
     goto L1
L2:
```

Before LIV code motion

```
     t1 = 202
     i = 1
     t3 = addr(a)
     t4 = t3 - 4
L1:  t2 = i>100
     if t2 goto L2
     t1 = t1-2
     t5 = 4*i
     t6 = t4+t5
     *t6 = t1
     i = i+1
     goto L1
L2:
```

After LIV code motion

Strength Reduction

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before strength reduction for t5

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t6 = t4+t7
      *t6 = t1
      i = i+1
      t7 = t7 + 4
      goto L1
L2:
```

After strength reduction for t5 and copy propagation

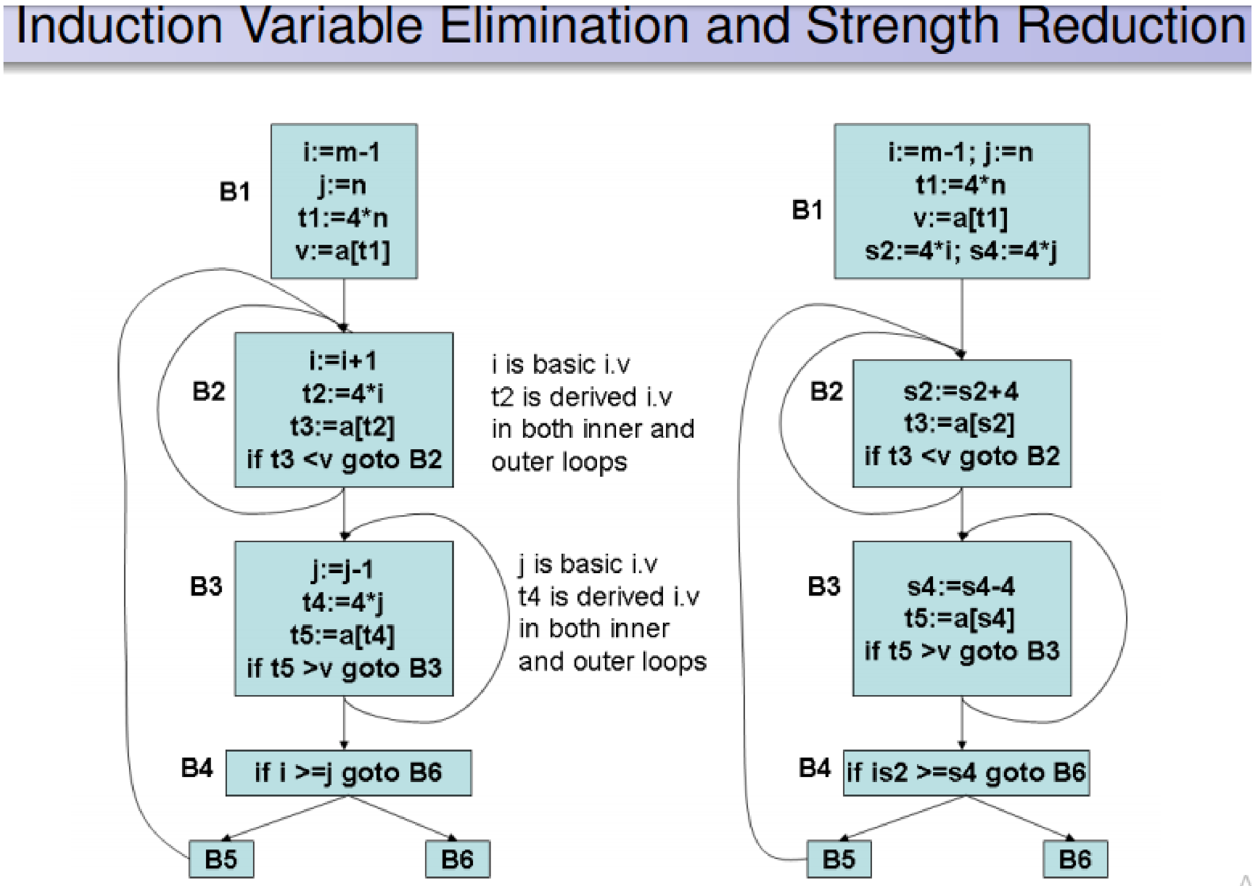
Induction Variable Elimination

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t6 = t4+t7
      *t6 = t1
      i = i+1
      t7 = t7 + 4
      goto L1
L2:
```

Before induction variable elimination (i)

```
      t1 = 202
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = t7 >400
      if t2 goto L2
      t1 = t1-2
      t6 = t4+t7
      *t6 = t1
      t7 = t7 + 4
      goto L1
L2:
```

After eliminating i and replacing it with t7



Loop Unrolling

Loop overhead can be reduced by reducing the number of iterations and replicating the body of the loop.

Example:

In the code fragment below, the body of the loop can be replicated once and the number of iterations can be reduced from 100 to 50.

```
for (i = 0; i < 100; i++)
  g ();
```

Below is the code fragment after loop unrolling.

```
for (i = 0; i < 100; i += 2)
{
  g ();
  g ();
}
```

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Program 1:

filter_none

edit

play_arrow

brightness_4

```
// This program does not uses loop unrolling.
#include<stdio.h>

int main(void)
{
    for (int i=0; i<5; i++)
        printf("Hello\n"); //print hello 5 times

    return 0;
}
```

Program 2:

filter_none

edit

play_arrow

brightness_4

```
// This program uses loop unrolling.
#include<stdio.h>

int main(void)
{
    // unrolled the for loop in program 1
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");

    return 0;
```

```
}
```

Output:

```
Hello
Hello
Hello
Hello
Hello
```

Illustration:
Program 2 is more efficient than program 1 because in program 1 there is a need to check the value of i and increment the value of i every time round the loop. So small loops like this or loops where there is fixed number of iterations are involved can be unrolled completely to reduce the loop overhead.

Advantages:

- Increases program efficiency.
- Reduces loop overhead.
- If statements in loop are not dependent on each other, they can be executed in parallel.

Disadvantages:

- Increased program code size, which can be undesirable.
- Possible increased usage of register in a single iteration to store temporary variables which may reduce performance.
- Apart from very small and simple codes, unrolled loops that contain branches are even slower than recursions.

- **Loop Jamming:**

Loop jamming is the combining the two or more loops in a single loop. It reduces the time taken to compile the many number of loops.

Example:

Initial Code:

```
for(int i=0; i<5; i++)
    a = i + 5;
for(int i=0; i<5; i++)
    b = i + 10;
```

Optimized code:

```
for(int i=0; i<5; i++)
{
 a = i + 5;
 b = i + 10;
```

```
}
```

Dead Code Elimination

The following example contains dead code:

```
while (true) {
  if (key === 'up') {
    shoot();
  } else if (false) {
    // this is unreachable code
    deleteShip();
  } else {
    key = 'exit'
    break;
    // this is also unreachable code
    key = 'down';
  }
}
```

The resulting code looks like this:

```
while (true) {
  if (key === 'up') {
    shoot();
  } else {
    key = 'exit';
    break;
  }
}
```

Function Inlining

The overhead associated with calling and returning from a function can be eliminated by expanding the body of the function inline, and additional opportunities for optimization may be exposed as well.

Example:

In the code fragment below, the function add() can be expanded inline at the call site in the function sub().

```
int add (int x, int y)
{
  return x + y;
}

int sub (int x, int y)
{
  return add (x, -y);
}
```

Expanding add() at the call site in sub() yields:

```
int sub (int x, int y)
{
  return x + -y;
}
```

which can be further optimized to:

```
int sub (int x, int y)
{
  return x - y;
}
```