Code Optimizations

Code Optimizations

- Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

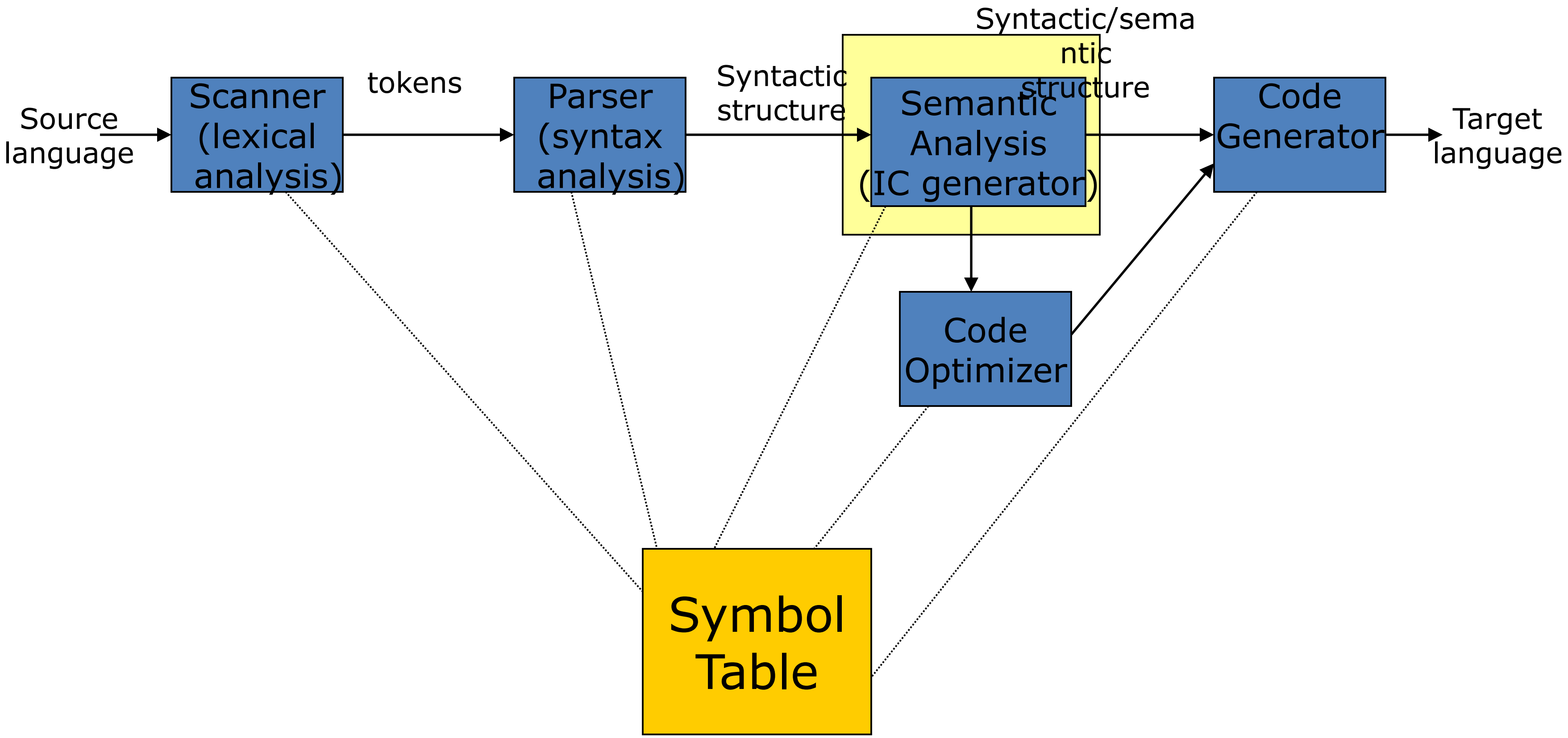
- Introduction
  - Optimized code
    - Executes faster
    - efficient memory usage
    - yielding better performance.

  - Compilers can be designed to provide code optimization.

- Criteria for Code-Improving Transformation:
  - Meaning must be preserved (correctness)
  - Speedup must occur on average.
  - Work done must be worth the effort.

- Opportunities:
  - Programmer
  - Intermediate code
  - Target code

Code Optimization



Levels

- Window – peephole optimization
- Basic block
- Procedural – global (control flow graph)
- Program level – intraprocedural (program dependence graph)

- In compiler theory, peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code.
- The set is called a "peephole" or a "window".
- It works by recognizing sets of instructions that can be replaced by shorter or faster sets of instructions.

Peephole Optimizations

- Constant Folding

EX1:

x := 32 becomes **x := 64**

x := x + 32

EX2:

```
int f (void) { return 3 + 5; }
     becomes
int f (void) { return 8; }
```

- Unreachable Code

```
EX1:
goto L2
x := x + 1
```

← unneeded

```
EX2:
Function add(X,Y)
{
  return X+Y;
  int ans= X*Y;
}
```

- Flow of control optimizations

```
goto L1          becomes    goto L2
 …
L1: goto L2
```

Peephole Optimizations

- Algebraic Simplification
- Algebraic simplifications use algebraic properties of operators or particular operator-operand combinations to simplify expressions .
- Simplifications for integers:

```
i + 0 = 0 + i = i − 0 = i
0 - i= -i
i * 1 = 1 * i = i / 1 = i
i * 0 = 0 * i = 0
```

- Dead code

```
EX1:
x := 32 ←  where x not used after statement
y := 32 + y

EX2:
Function add(X,Y)
{
  int ans= X*Y;
  return X+Y;
}
```

- Reduction in strength

  `x := x * 2` → `x := x + x`

Peephole Optimizations

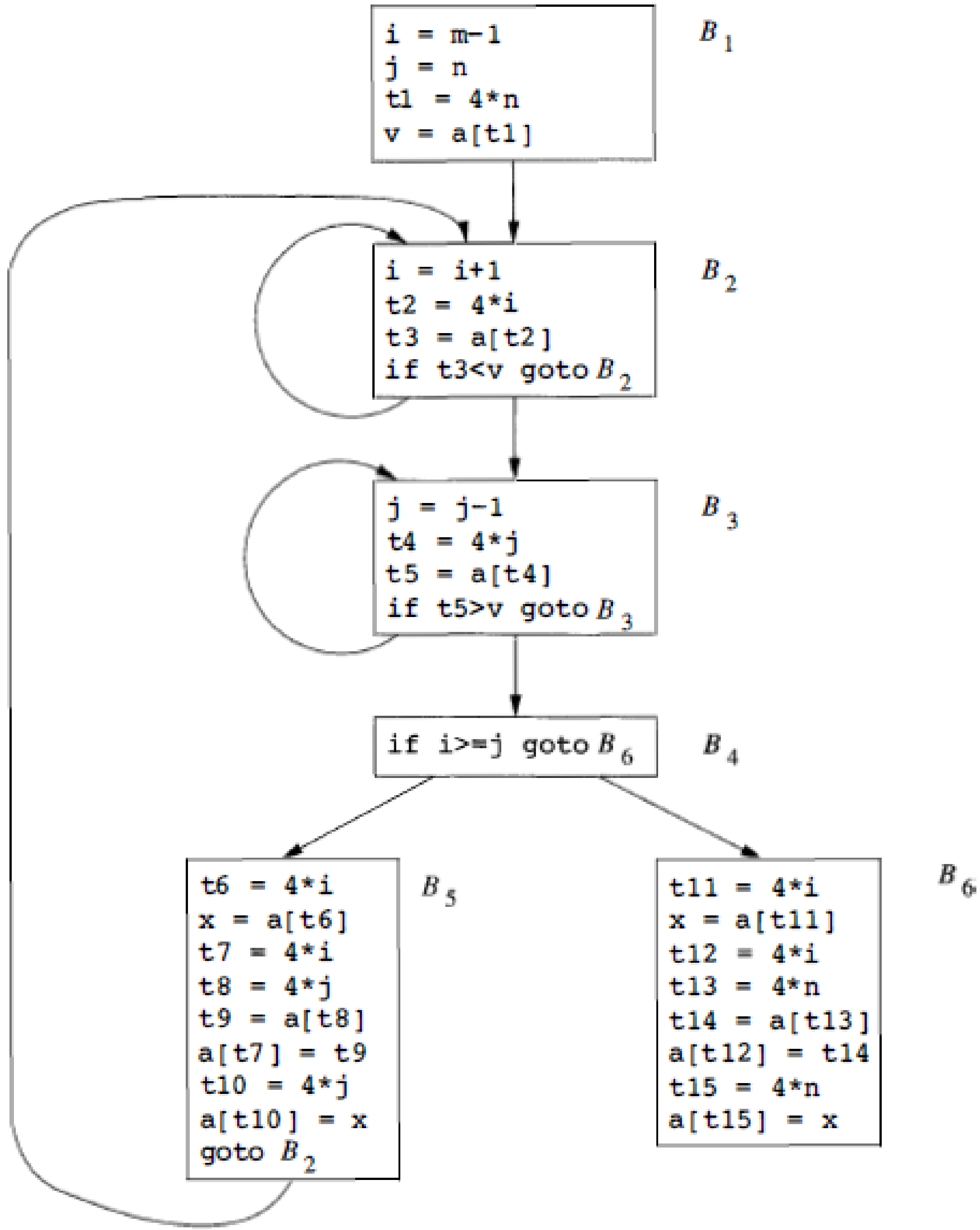
- Local in nature
- Pattern driven
- Limited by the size of the window

Basic Block Level

- Common Subexpression elimination
- Constant Propagation
- Dead code elimination
- Plus many others such as copy propagation, value numbering, partial redundancy elimination, …

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

```
(1)     i = m-1
(2)     j = n
(3)     t1 = 4*n
(4)     v = a[t1]
(5)     i = i+1
(6)     t2 = 4*i
(7)     t3 = a[t2]
(8)     if t3<v goto (5)
(9)     j = j-1
(10)    t4 = 4*j
(11)    t5 = a[t4]
(12)    if t5>v goto (9)
(13)    if i>=j goto (23)
(14)    t6 = 4*i
(15)    x = a[t6]
(16)    t7 = 4*i
(17)    t8 = 4*j
(18)    t9 = a[t8]
(19)    a[t7] = t9
(20)    t10 = 4*j
(21)    a[t10] = x
(22)    goto (5)
(23)    t11 = 4*i
(24)    x  = a[t11]
(25)    t12 = 4*i
(26)    t13 = 4*n
(27)    t14 = a[t13]
(28)    a[t12] = t14
(29)    t15 = 4*n
(30)    a[t15] = x
```


i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
B5
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
B6

Local common-sub expression elimination

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

B_5

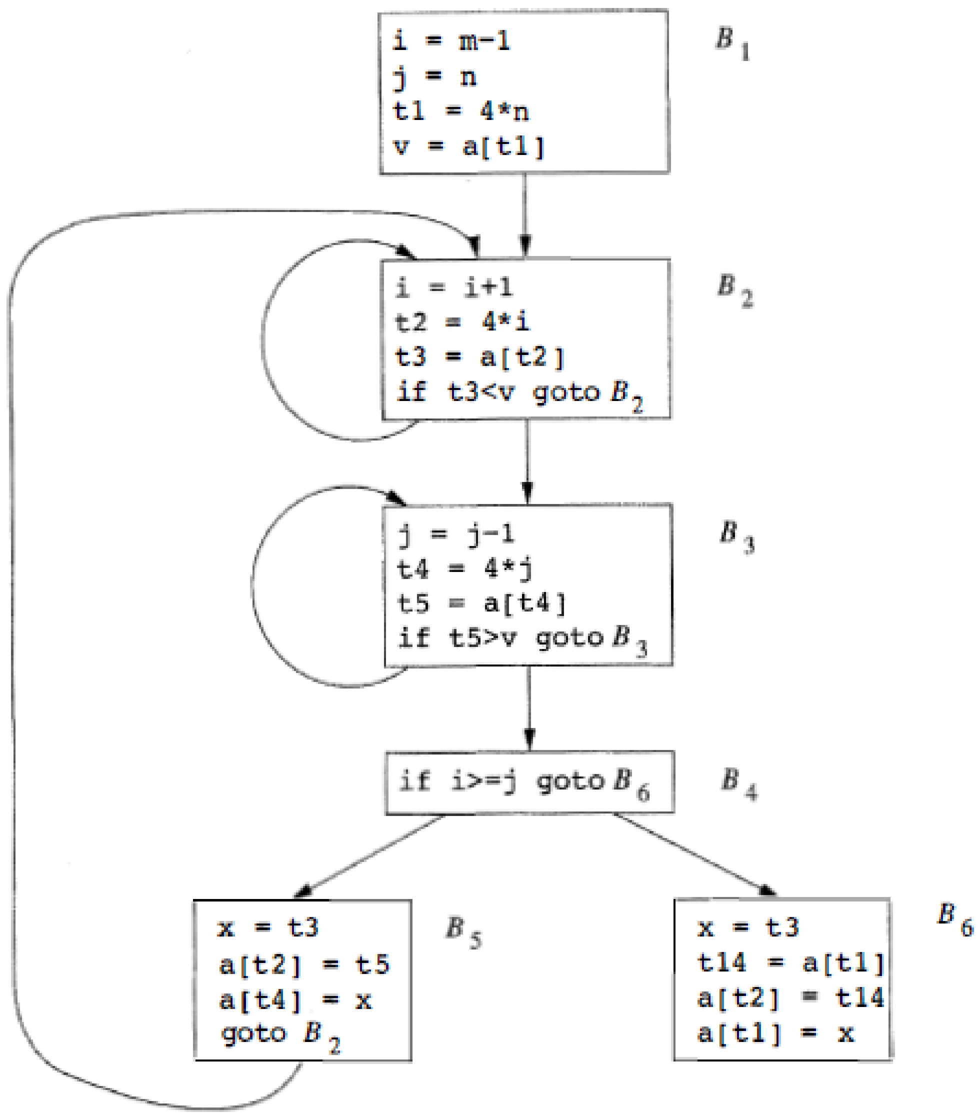
(a) Before.

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

B_5

(b) After.

- t8 = 4*j
- t9 = a[t8]
- a[t8] = x
- in B5 can be replaced by
- t9 = a[t4]
- a[t4] = x



i = m-1
j = n
t1 = 4*n
v = a[t1]
B1
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
B2
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
B3
if i>=j goto B6
B4
x = t3
a[t2] = t5
a[t4] = x
goto B2
B5
x = t3
t14 = a[t1]
a[t2] = t14
a[t1] = x
B6

Copy Propagation

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Dead- Code Elimination

```
a[t2] = t5
a[t4] = t3
```
goto B_2

CODE MOTION

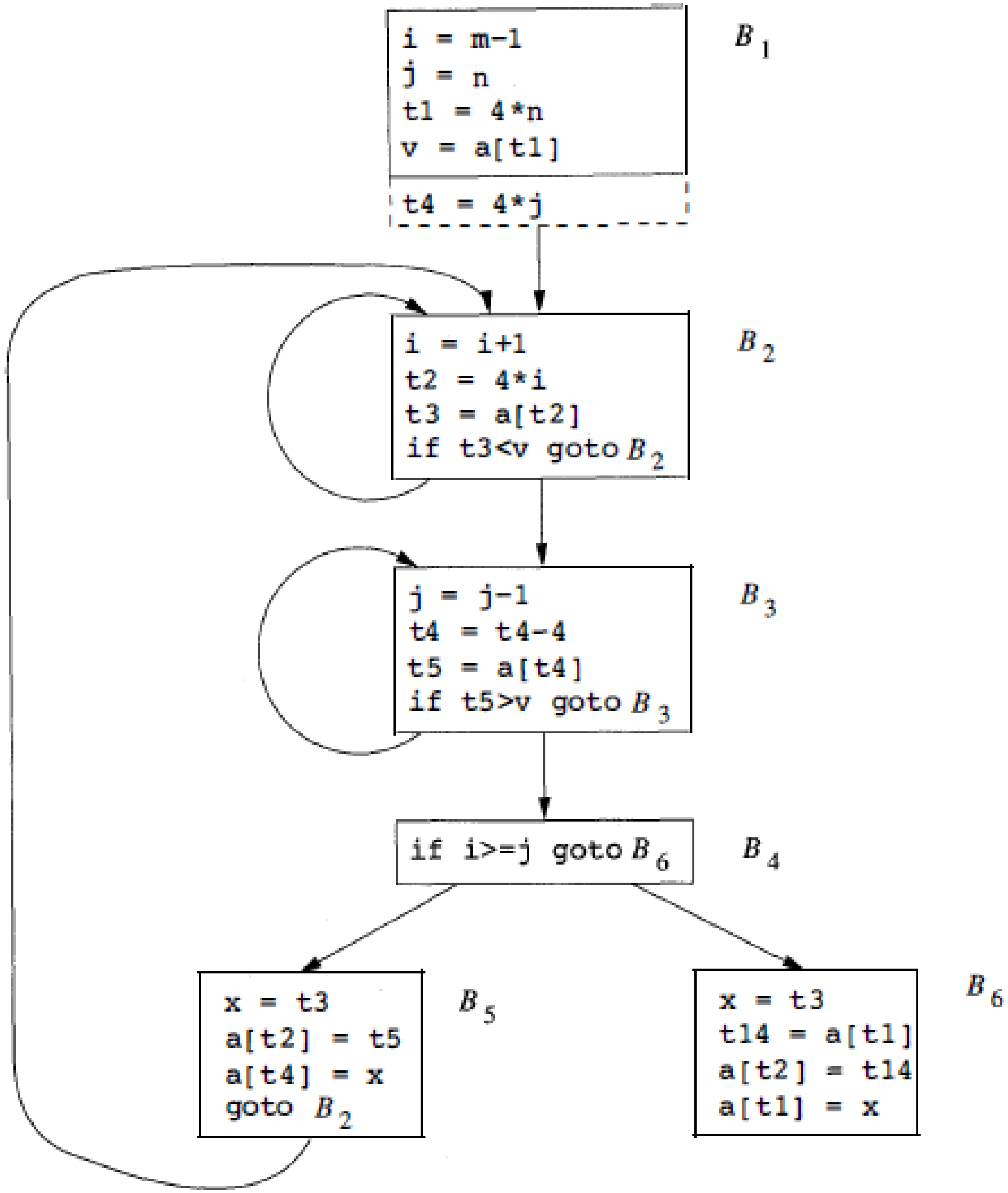
- Code Motion
  - Any code inside a loop that always computes the same value can be moved before the loop.
  - Example:

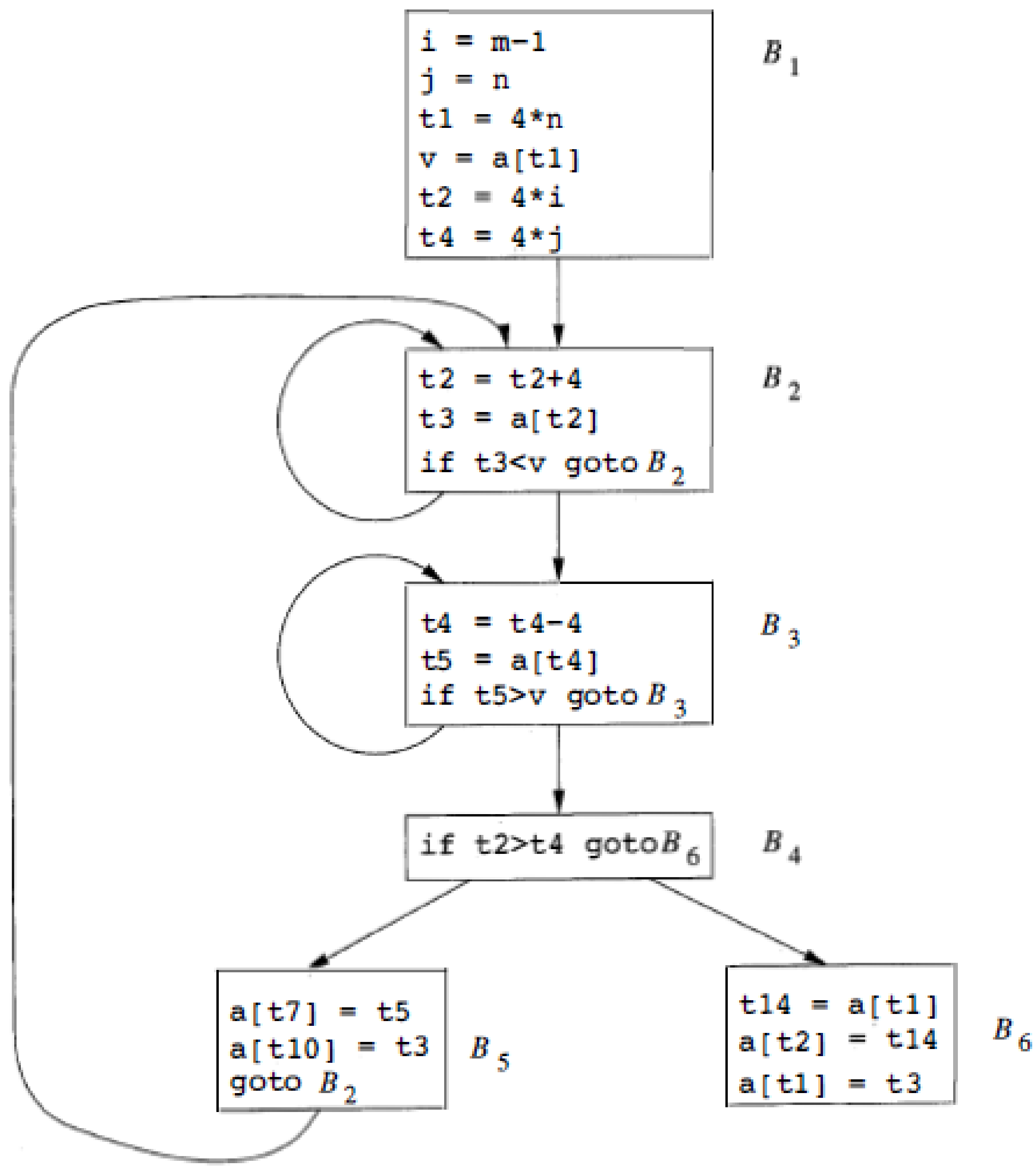
```
while (i <= limit-2)
do {loop code}
```

where the loop code doesn't change the limit variable. The subtraction, limit-2, will be inside the loop. Code motion would substitute:

```
t = limit-2;
while (i <= t)
do {loop code}
```

Induction Variables and Reduction in Strength




i = m-1
j = n
t1 = 4*n
v = a[t1]
t2 = 4*i
t4 = 4*j
B1
t2 = t2+4
t3 = a[t2]
if t3<v goto B2
B2
t4 = t4-4
t5 = a[t4]
if t5>v goto B3
B3
if t2>t4 goto B6
B4
a[t7] = t5
a[t10] = t3
goto B2
B5
t14 = a[t1]
a[t2] = t14
a[t1] = t3
B6

Simple example: a[i+1] = b[i+1]

- t1 = i+1
- t2 = b[t1]
- t3 = i + 1
- a[t3] = t2

- t1 = i + 1
- t2 = b[t1]
- t3 = i + 1 ← *no longer live*
- a[t1] = t2

Common expression can be eliminated

Now, suppose i is a constant:

- i = 4
- t1 = i+1
- t2 = b[t1]
- a[t1] = t2

- i = 4
- t1 = 5
- t2 = b[t1]
- a[t1] = t2

- i = 4
- t1 = 5
- t2 = b[5]
- a[5] = t2

Final Code:

- i = 4
- t2 = b[5]
- a[5] = t2

Compiler Code Optimizations

- A Code optimizer sits between the front end and the code generator.
  - Works with intermediate code.
  - Can do control flow analysis.
  - Can do data flow analysis.
  - Does transformations to improve the intermediate code.

Compiler Code Optimizations

- Optimizations provided by a compiler includes:
  - Inlining small functions
  - Code hoisting
  - Dead store elimination
  - Eliminating common sub-expressions
  - Loop unrolling
  - Loop optimizations: Code motion, Induction variable elimination, and Reduction in strength.

Compiler Code Optimizations

- Inlining small functions
  - Repeatedly inserting the function code instead of calling it, saves the calling overhead and enable further optimizations.
  - Inlining large functions will make the executable too large.

Compiler Code Optimizations

- Code hoisting
  - Moving computations outside loops
  - Saves computing time

Compiler Code Optimizations

- Code hoisting
  - In the following example (2.0 * PI) is an invariant expression there is no reason to recompute it 100 times.

```
DO I = 1, 100
        ARRAY(I) = 2.0 * PI * I
ENDDO
```

  - By introducing a temporary variable 't' it can be transformed to:

```
t = 2.0 * PI
DO I = 1, 100
        ARRAY(I) = t * I
END DO
```

Compiler Code Optimizations

- Dead store elimination
  - If the compiler detects variables that are never used, it may safely ignore many of the operations that compute their values.

Compiler Code Optimizations

- Eliminating common sub-expressions
  - Optimization compilers are able to perform quite well:

    X = A * LOG(Y) + (LOG(Y) ** 2)

  - Introduce an explicit temporary variable t:

    t = LOG(Y)

    X = A * t + (t ** 2)

  - Saves one 'heavy' function call, by an elimination of the common sub-expression LOG(Y), the exponentiation now is:

    X = (A + t) * t

Compiler Code Optimizations

- Loop unrolling
  - The loop exit checks cost CPU time.
  - Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.
  - If you know a loop is only performed a certain number of times, or if you know the number of times it will be repeated is a multiple of a constant you can unroll this loop.

Compiler Code Optimizations

- Loop unrolling
  - Example:

```
// old loop
for(int i=0; i<3; i++) {
        color_map[n+i] = i;
}
// unrolled version
int i = 0;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
i++;
colormap[n+i] = i;
```

Compiler Code Optimizations

- Code Motion
  - Any code inside a loop that always computes the same value can be moved before the loop.
  - Example:

```
while (i <= limit-2)
do {loop code}
```

where the loop code doesn't change the limit variable. The subtraction, limit-2, will be inside the loop. Code motion would substitute:

```
t = limit-2;
while (i <= t)
do {loop code}
```

Compiler Code Optimizations

- Conclusion
  - Compilers can provide some code optimization.
  - Programmers do have to worry about such optimizations.
  - Program definition must be preserved.