Code Improvement - I

Why?

- The intermediate and target Code generated by a simple compiler may be correct but it is suboptimal[a].
- There are many sources of redundant computations and corresponding improvements e.g. redundant load-store, common subexpression evaluation, constant propagation, constant folding, ...

[a] The optimal code generation is undecidable!

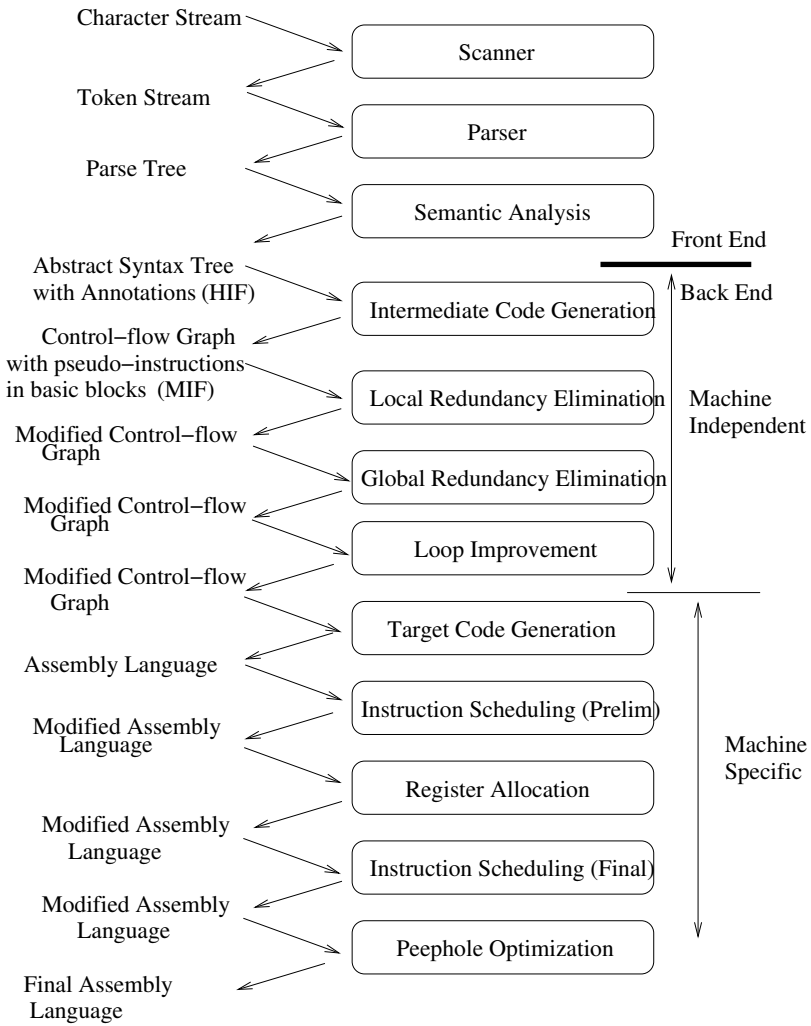
Why?

- $\cdots$ strength reduction, copy propagation, dead code removal, loop invariant computation, chain of goto, induction variable and strength reduction, different architecture dependent improvements etc.

Different Phases

- Peephole optimizations - a small window of code sequence is taken code improvement.
- Code improvement within a basic block, local optimization.
- Global improvement across basic blocks: within a region or within a procedure.
- Inter procedural improvements.
- Architecture dependent improvements.

Different Phases



Analysis

- Control-Flow Analysis: to find the flow of control within a procedure.
- Data-Flow Analysis: to gather information about the flow of data from one basic block to another.
- Call graph: to get information about procedure call sequence.

Optimization within a Basic Block: an Example

Matrix Multiplication program: `int a[20][20], b[20][20], c[20][20]`, size of `int` is 4.

```
for(i=0; i<n; ++i)
    for(j=0; j<n; ++j)
        c[i][j] = 0;
```

An Example

```
for(i=0; i<n; ++i)
   for(j=0; j<n; ++j)
      for(k=0; k<n; ++k)
         c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

3-Address Code

```
    i = 0
    goto L2 (B6)          ..... B1
L1:
    j = 0
    goto L4: (B4)         ..... B2
L3:
    $1 = 80*i
    $2 = 4*j
    $3 = $1+$2
```

```
    $4 = a + $3
    *$4 = 0
    j = j+1          ..... B3
L4:
    if j<n goto L3: (B3) .... B4
    i = i+1          ..... B5
L2:
    if i<n goto L1: (B2) .... B6
    i = 0
    goto L6 (B15)        .... B7
L5:
```

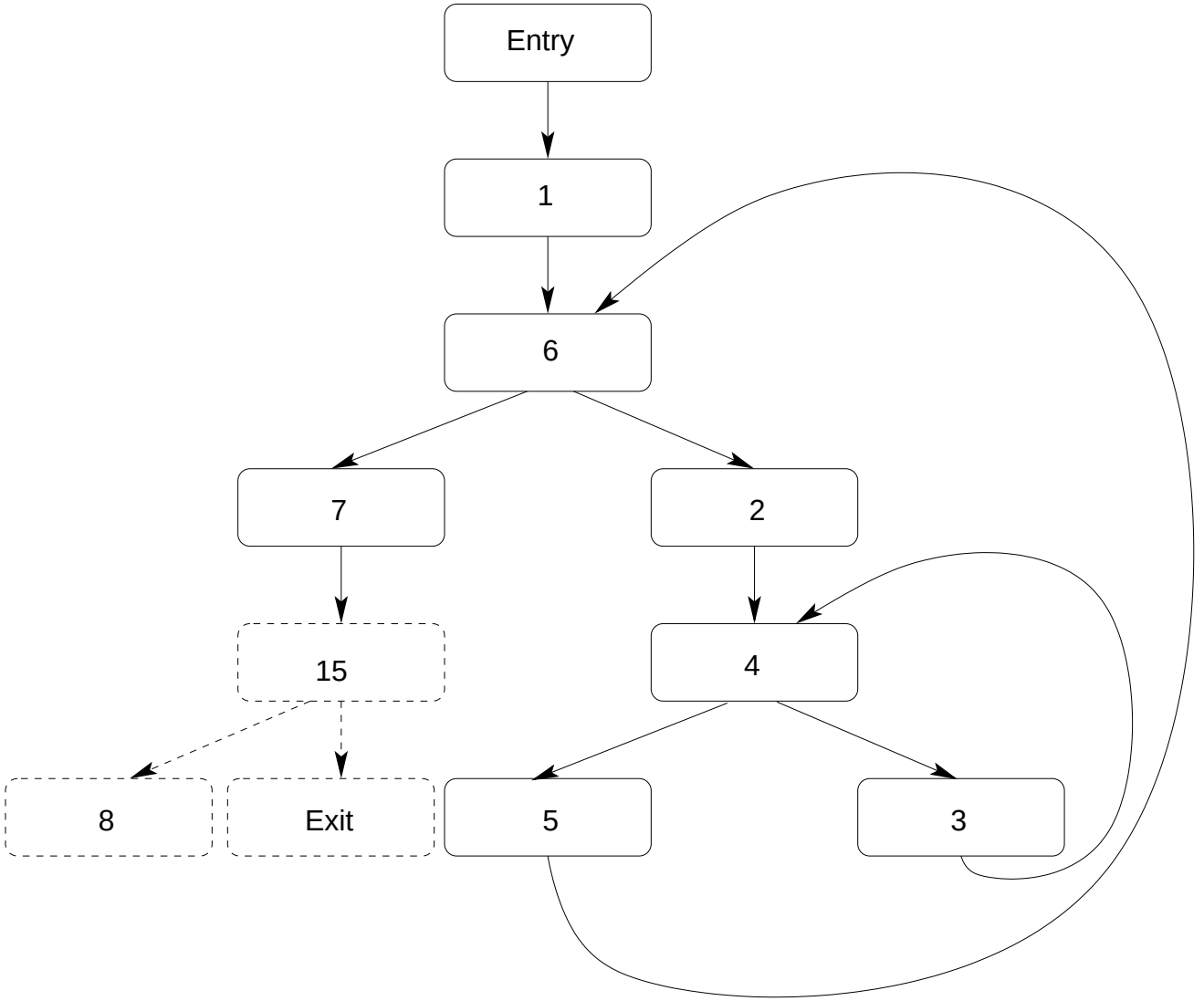
```
    j = 0
    goto L8: (B13)          .... B8
L7:
    k = 0
    goto L10: (B11)         .... B9
L9:
    $1 = 80*i
    $2 = 4*j
    $3 = $1 + $2
    $4 = c + $3
    $5 = 80*i
```

```
$6 = 4*j
$7 = $5 + $6
$8 = C + $7
$9 = *$8
$10 = 80*i
$11 = 4*k
$12 = $10 + $11
$13 = a + $12
$14 = *$13
$15 = 80*k
$16 = 4*j
```

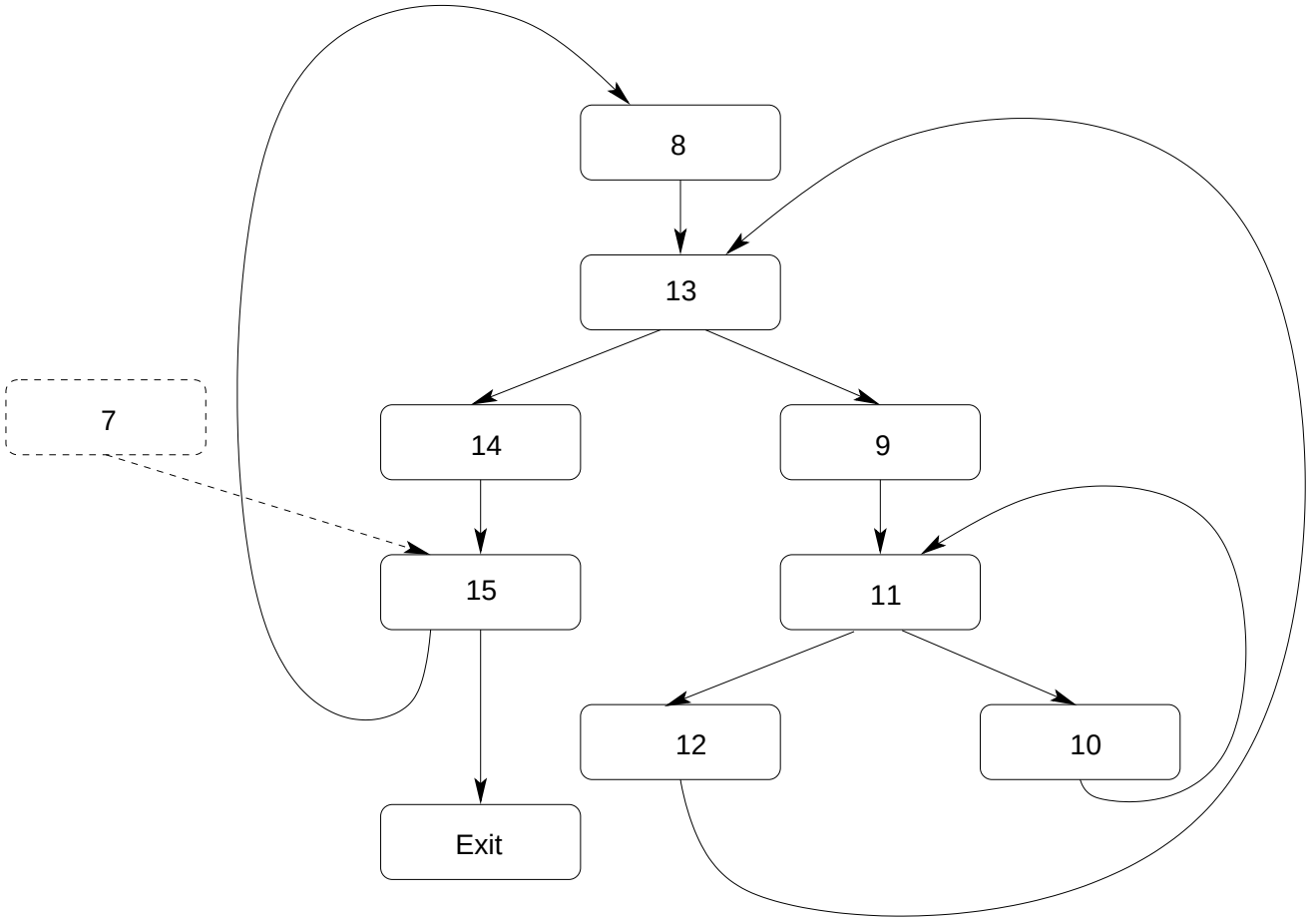
```
    $17 = $15 + $16
    $18 = b + $17
    $19 = *$18
    $20 = $14 * $19
    $21 = $9 + $20
    *$4 = $21
    k = k+1          ..... B10
L10:
    if k<n goto L9 (B10) .... B11
    j = j + 1        .... B12
L8:
```

```
    if j<n goto L7: (B9) ... B13
    i = i + 1      ..... B14
L6:
    if i<n goto L5: (B8) ... B15
```

Control Flow



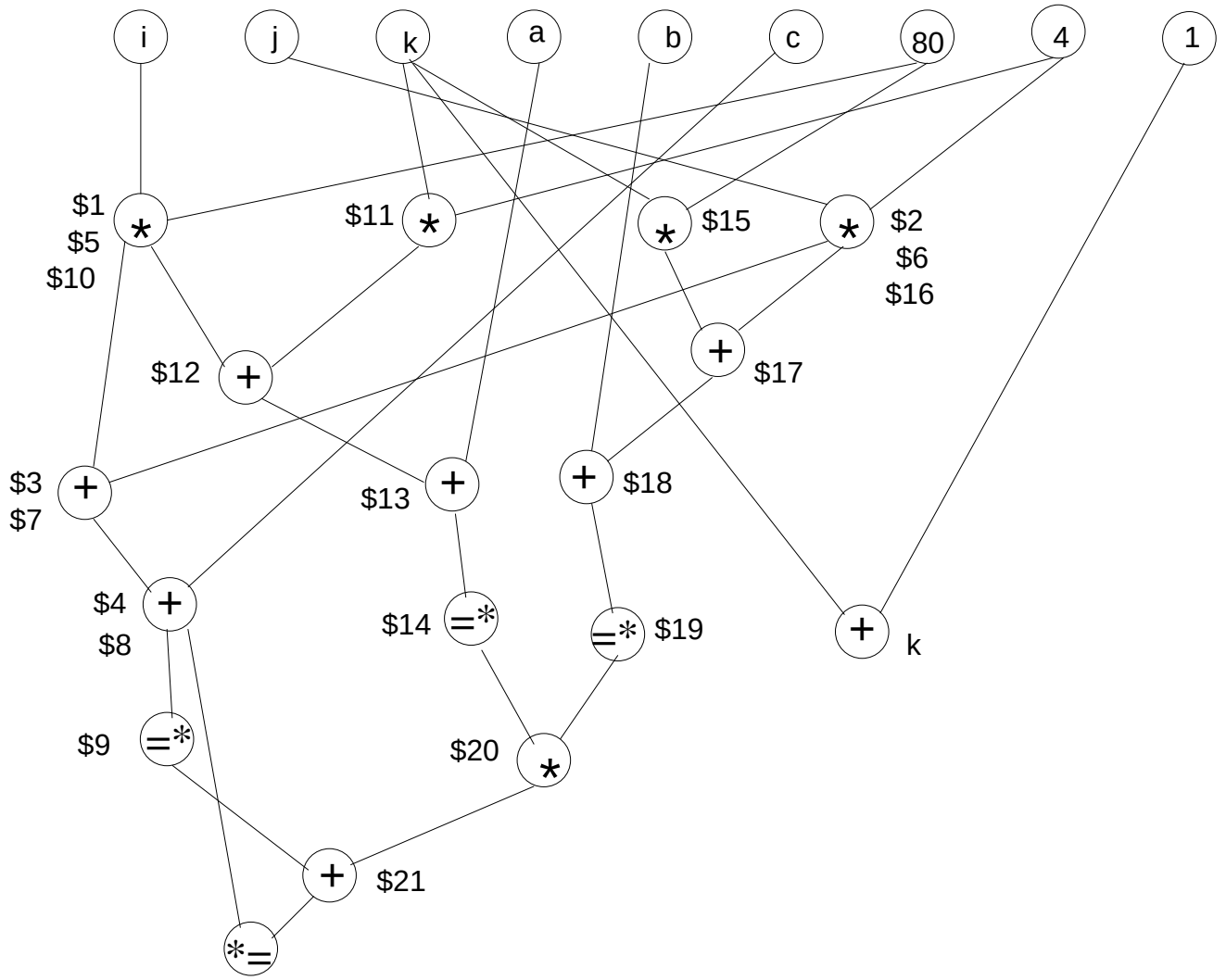
Control Flow



Basic Block B_{10}

```
1 $1 = 80*i      9 $9 = *$8       17 $17 = $15+$16
2 $2 = 4*j      10 $10 = 80*i     18 $18 = b+$17
3 $3 = $1+$2    11 $11 = 4*k      19 $19 = *$18
4 $4 = c+$3     12 $12=$10+$11    20 $20 = $14*$19
5 $5 = 80*i     13 $13 = a+$12    21 $21 = $9+$20
6 $6 = 4*j      14 $14 = *$13     22 *$4 = $21
7 $7 = $5+$6    15 $15 = 80*k     23 k = k+1
8 $8 = c+$7     16 $16 = 4*j
```

DAG of B_{10}



Common Sub-Expressions

1. `$1 = 80*i` in line: 1, 5, 10.
2. `$7 = 4*j` in line: 2, 6, 16.
3. We remove them and also perform copy propagation.

Basic Block B_{10}

```
1  $1 = 80*i        9  $9 = *$8       17 $17 = $15+$16
2  $2 = 4*j        10 $10 = $1        18 $18 = b+$17
3  $3 = $1+$2      11 $11 = 4*k       19 $19 = *$18
4  $4 = c+$3       12 $12=$10+$11     20 $20 = $14*$19
5  $5 = $1         13 $13 = a+$12     21 $21 = $9+$20
6  $6 = $2         14 $14 = *$13      22 *$4 = $21
7  $7 = $5+$6      15 $15 = 80*k      23 k = k+1
8  $8 = c+$7       16 $16 = $2
```

Basic Block B_{10}

```
1 $1 = 80*i
2 $2 = 4*j
3 $3 = $1+$2
4 $4 = c+$3
7 $7 = $1+$2
8 $8 = c+$7
9 $9 = *$8
11 $11 = 4*k
12 $12=$1+$11
13 $13 = a+$12
14 $14 = *$13
15 $15 = 80*k
17 $17 = $15+$2
18 $18 = b+$17
19 $19 = *$18
20 $20 = $14*$19
21 $21 = $9+$20
22 *$4 = $21
23 k = k+1
```

Dead Code

1. Line 5, 6, 10, 17 are dead code.
2. New common sub-expressions are in 3, 7.
   We also perform copy propagation.

Basic Block B_{10}

```
1 $1 = 80*i        9 $9 = *$8        17 $17 = $15+$2
2 $2 = 4*j        11 $11 = 4*k       18 $18 = b+$17
3 $3 = $1+$2      12 $12=$1+$11      19 $19 = *$18
4 $4 = c+$3       13 $13 = a+$12     20 $20 = $14*$19
7 $7 = $3         14 $14 = *$13      21 $21 = $9+$20
8 $8 = c+$3       15 $15 = 80*k      22 *$4 = $21
                                     23 k = k+1
```

Dead Code

1. Line 7 is dead code.
2. New common sub-expressions are in 4, 8.
   We also perform copy propagation and line 8 is a dead code.

Basic Block B_{10}

```
1 $1 = 80*i
2 $2 = 4*j
3 $3 = $1+$2
4 $4 = c+$3
7 $7 = $3
```

```
 9 $9 = *$4
11 $11 = 4*k
12 $12=$1+$11
13 $13 = a+$12
14 $14 = *$13
15 $15 = 80*k
```

```
17 $17 = $15+$2
18 $18 = b+$17
19 $19 = *$18
20 $20 = $14*$19
21 $21 = $9+$20
22 *$4 = $21
23 k = k+1
```

Strength Reduction

1. We can replace the operations of 1, 2, 11, 15 with less costly operations.
2. $80 \times i = 64 * i + 16 * i$ and this can be performed using less costly shift and add operations.

Beyond Basic Block B_{10}

- The value of `$1 = 80*i` changes only when the value of `i` changes in B_{14}.
- The sequence of its values is: $0, 80, 160, \cdots$.
- Such variables are called an induction variables.
- It is initialized at B_7 and updated at B_{14}.
- But the decision requires data-flow analysis.

B_7 and B_{14}

```
B7:
    i = 0
    $1 = 0
    goto L6 (B15)
B14:
    i = i + 1
    $1 = $1 + 80
    goto (B15)
```

Basic Block B_{10}

```
2 $2 = 4*j
3 $3 = $1+$2
4 $4 = c+$3
7 $7 = $3
```

```
 9 $9 = *$4
11 $11 = 4*k
12 $12=$1+$11
13 $13 = a+$12
14 $14 = *$13
15 $15 = 80*k
```

```
17 $17 = $15+$2
18 $18 = b+$17
19 $19 = *$18
20 $20 = $14*$19
21 $21 = $9+$20
22 *$4 = $21
23 k = k+1
```

Use of a Name

- Statement i assigns a value to a variable x.
- Control can flow from statement i to statement j.
- In the control path from $i \rightarrow j$, there is no other assignment to x.
- Statement j uses the value of x defines in i.
- x is live at statement i.

Live and Next Use within Basic Block

- We compute liveness and next use information in a basic block.
- It is computed backward from the end to the beginning of a block.
- For each 3-address code $i: \ x = y + z$ within the block we associate the liveness and next-use information.

Live and Next Use within Basic Block

- Information about the liveness and next use (yes/no) of a variable may be available from the symbol table.
- All program defined variables in the block are live on exit from the block.
- For each 3-address code $i: \; x = y + z$ we compute the following.

Live and Next Use within Basic Block

- Information about x, y, z from the symbol table are attached to i.
- The symbol table is updates for x, y, z:
  $x : (N, -)$ (not live, no next use), :
  $y, z : (L, i)$ (live, next use at i).

Live and Next Use: B_{10}

Srl. No.	Attach	Update Sym.Tab
Exit	All live	
23 : k = k + 1	K(L, -)	k(L,23)
22 : *$4 = $21	$4(L,-), $21(L, -)	k(L,23), $4(L,22), $21(L, 22)