Outline

Code Optimization

Jianguo Lu

November 27, 2019

Which program runs faster?

```
1   for j=1:n
2       for i=1:n
3           a(i,j)=a(i,j)^2;
4       end
5   end
```

or

```
1   for i=1:n
2       for j=1:n
3           a(i,j)=a(i,j)^2;
4       end
5   end
```

Which program runs faster?

```
1  for j=1:n
2      for i=1:n
3          a(i,j)=a(i,j)^2;
4      end
5  end
```

or

```
1  for i=1:n
2      for j=1:n
3          a(i,j)=a(i,j)^2;
4      end
5  end
```

- The language is Matlab. n=5000.

```
1  ans =
2      1.8
3  ans =
4      2.9
```

Which program runs faster?

```
1       int n=5000;
2       Double [][] A=new Double [m][n];
3
4
5       for (int j=0; j<n; j++){
6         for (int i=0; i<m; i++){
7            A[i][j]=A[i][j]*A[i][j];
8          }
9       }
10
11      for (int i=0; i<m; i++){
12         for (int j=0; j<n; j++){
13           A[i][j]=A[i][j]*A[i][j];
14         }
15      }
```

- Run 10 times
  - 4774 1825 2312 1662 4097 2554 1656 1688 4170 2618
  - 656 589 2982 982 604 605 603 3357 604 623

Outline

Code optimization

- Improve the code so that it is more efficient.
- Why called *code optimization*?
  - The code is not 'optimal'
  - Better to be called 'code improvement'?
- Different from the code improvement by designing a new algorithm as in 'data structures and algorithms'
  - There an 'improved' algorithm can often have a better time complexity
  - e.g.,MergeSort vs InsertionSort. $O(n^2) \rightarrow O(n \log n)$
- Code optimization makes 'minor' changes
- Result in smaller improvement, a constant factor
- Often the code becomes uglier
- Done after algorithm and data structure design. Often inside compiler.
  - Hence 'Compiler optimization'.

Compiler optimization is a big topic...

CSCD70

COMPILER OPTIMIZATION, WINTER 2018

- ANNOUNCEMENTS
- COURSE INFO
- COURSE CONTENTS
- ASSIGNMENTS
- REFERENCES

GENERAL COURSE INFORMATION

Instructor: Prof. Gennady Pekhimenko

Teaching Assistant: Bojian Zheng

Discussion Board: https://piazza.com/utoronto.ca/winter2018/cscd70/home

Syllabus: [PDF]

COURSE DESCRIPTION

The goal of this course is introduce students to the theoretical and practical aspects of building optimizing compilers that effectively exploit modern architectures. The course will begin with the fundamentals of compiler optimization, and will build upon these fundamentals to address issues in state-of-the-art commercial and research machines. Topics include: intermediate representations, basic blocks and flow graphs, data flow analysis, partial evaluation and redundancy elimination, loop optimizations, register allocation, instruction scheduling, interprocedural analysis, memory hierarchy optimizations, extracting parallelism, and dynamic optimizations. Students will implement significant optimizations within LLVM, a modern research compiler framework.

Types of code optimization

- Levels of optimization:
  - High-level: e.g., at the AST(Abstract Syntax Tree)-level (e.g., inlining)
  - Low-level: e.g., right before instruction selection (e.g., register allocation)
- Local vs. global optimization
  - They can be applied just looking locally at computation
  - No need to understand control flow
- Manual vs. automated
  - Differs greatly from compiler to compiler
  - We need to know what a compiler do to avoid ugly manual optimization.

Outline

Outline

Loop invariant Code motion

- Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of loop
  - Loop Invariant Code Motion
- Example

```
1  while (i<limit-2){
2    //some code does not change limit
3  }
```

can be transformed into

```
1  t=limit-2;
2  while (i<t){
3    //some code does not change limit
4  }
```

Loop invariant in condition

```
1   for (i=0; i<n; i+=s*10) {
2     ...
3   }
```

becomes

Loop invariant in condition

```
1   for (i=0; i<n; i+=s*10) {
2     ...
3   }
```

becomes

```
1   int t = s*10;
2   for (i=0; i<n; i+=t) {
3     ...
4     }
```

Loop invariant in loop body

```
1   for j = 1: big_number
2      for i = 1: large_number
3          a(i) = a(i) * b(j)
```

becomes

Loop invariant in loop body

```
1   for j = 1: big_number
2      for i = 1: large_number
3          a(i) = a(i) * b(j)
```

becomes

```
1   for j = 1: big_number
2       tmp = b(j)
3       for  i = 1: large_number
4           a(i) = a(i) * tmp
```

- Array references also should be minimized

Loop invariant in condition and body

```
1  void lower(char *s){
2    int i;
3    for (i = 0; i < strlen(s); i++)
4      if (s[i] >= 'A' && s[i] <= 'Z')
5        s[i] -= ('A' - 'a');
6  }
```

becomes

Loop invariant in condition and body

```
1  void lower(char *s){
2    int i;
3    for (i = 0; i < strlen(s); i++)
4      if (s[i] >= 'A' && s[i] <= 'Z')
5        s[i] -= ('A' - 'a');
6  }
```

becomes

```
1  void lower(char *s){
2    int i;
3    int len = strlen(s);
4    for (i = 0; i < len; i++)
5      if (s[i] >= 'A' && s[i] <= 'Z')
6        s[i] -= ('A' - 'a');
7  }
```

Loop invariant in condition and body

```
1  void lower(char *s){
2    int i;
3    for (i = 0; i < strlen(s); i++)
4      if (s[i] >= 'A' && s[i] <= 'Z')
5        s[i] -= ('A' - 'a');
6  }
```

becomes

```
1  void lower(char *s){
2    int i;
3    int len = strlen(s);
4    for (i = 0; i < len; i++)
5      if (s[i] >= 'A' && s[i] <= 'Z')
6        s[i] -= ('A' - 'a');
7  }
```

Further optimizations?

From Code Motion to Strength Reduction

```
1   for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3       a[n*i + j] = b[j];
```

becomes

From Code Motion to Strength Reduction

```
1   for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3       a[n*i + j] = b[j];
```

becomes

```
1   for (i = 0; i < n; i++)
2     int ni =n*i;
3     for (j = 0; j < n; j++)
4       a[ni + j] = b[j];
```

becomes

From Code Motion to Strength Reduction

```
1  for (i = 0; i < n; i++)
2    for (j = 0; j < n; j++)
3      a[n*i + j] = b[j];
```

becomes

```
1  for (i = 0; i < n; i++)
2    int ni =n*i;
3    for (j = 0; j < n; j++)
4      a[ni + j] = b[j];
```

becomes

```
1  int ni =0;
2  for (i = 0; i < n; i++){
3    for (j = 0; j < n; j++){
4      a[ni + j] = b[j];
5    }
6    ni=ni+n;
7  }
```

What is the method used here in the second step?

Outline

Strength reduction

- Reduction in strength: the transformation of replacing an expensive operation by an inexpensive one.
- e.g., Shift, add instead of multiply or divide

```
1   x*2
```

becomes

Strength reduction

- Reduction in strength: the transformation of replacing an expensive operation by an inexpensive one.
- e.g., Shift, add instead of multiply or divide

```
1  x*2
```

becomes

```
1   x+x
```

```
1   x div 8
```

becomes

Strength reduction

- Reduction in strength: the transformation of replacing an expensive operation by an inexpensive one.
- e.g., Shift, add instead of multiply or divide

```
1  x*2
```

becomes

```
1   x+x
```

```
1   x div 8
```

becomes

```
1   x >> 3
```

```
1  16*x
```

becomes

Strength reduction

- Reduction in strength: the transformation of replacing an expensive operation by an inexpensive one.
- e.g., Shift, add instead of multiply or divide

```
1  x*2
```

becomes

```
1   x+x
```

```
1   x div 8
```

becomes

```
1   x >> 3
```

```
1  16*x
```

becomes

```
1  x << 4
```

Reduction of Multiplications in a Loop

```
1  increment = xmax /large_number
2  do i = 1, large_number
3    x(i) = i * increment
4  enddo
```

becomes

```
1  increment = xmax /large_number
2  sum = increment
3  do i = 1, large_number
4    x(i) = sum
5    sum = sum + increment
6  enddo
```

Outline

Common subexpression

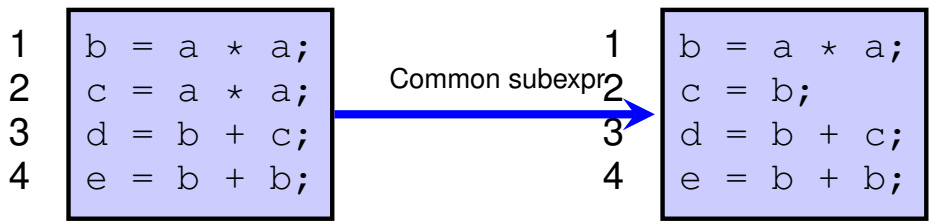
- Identify common subexpressions
- Replace the second with previous calculation
- Apply copy propagation may introduce more common subexpressions
  - Copy propagation: After assignment x = y, replace uses of x with y
  - replace *c* by *b* in this example

```
1  b = a * a;
2  c = a * a;
3  d = b + c;
4  e = b + b;
```

How to do it automatically?

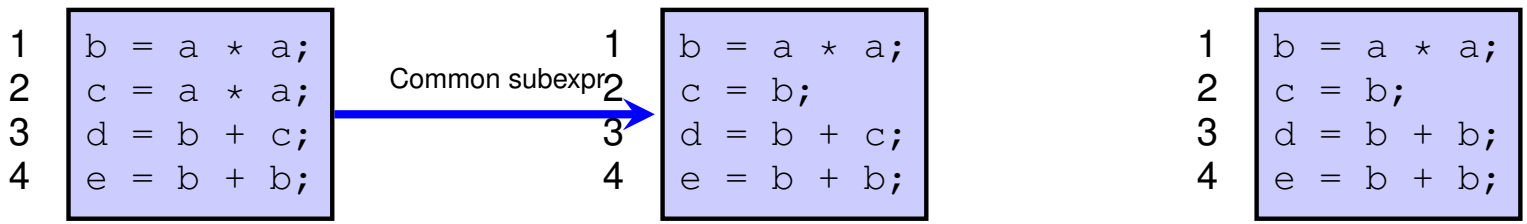
Common subexpression

- Identify common subexpressions
- Replace the second with previous calculation
- Apply copy propagation may introduce more common subexpressions
  - Copy propagation: After assignment x = y, replace uses of x with y
  - replace *c* by *b* in this example



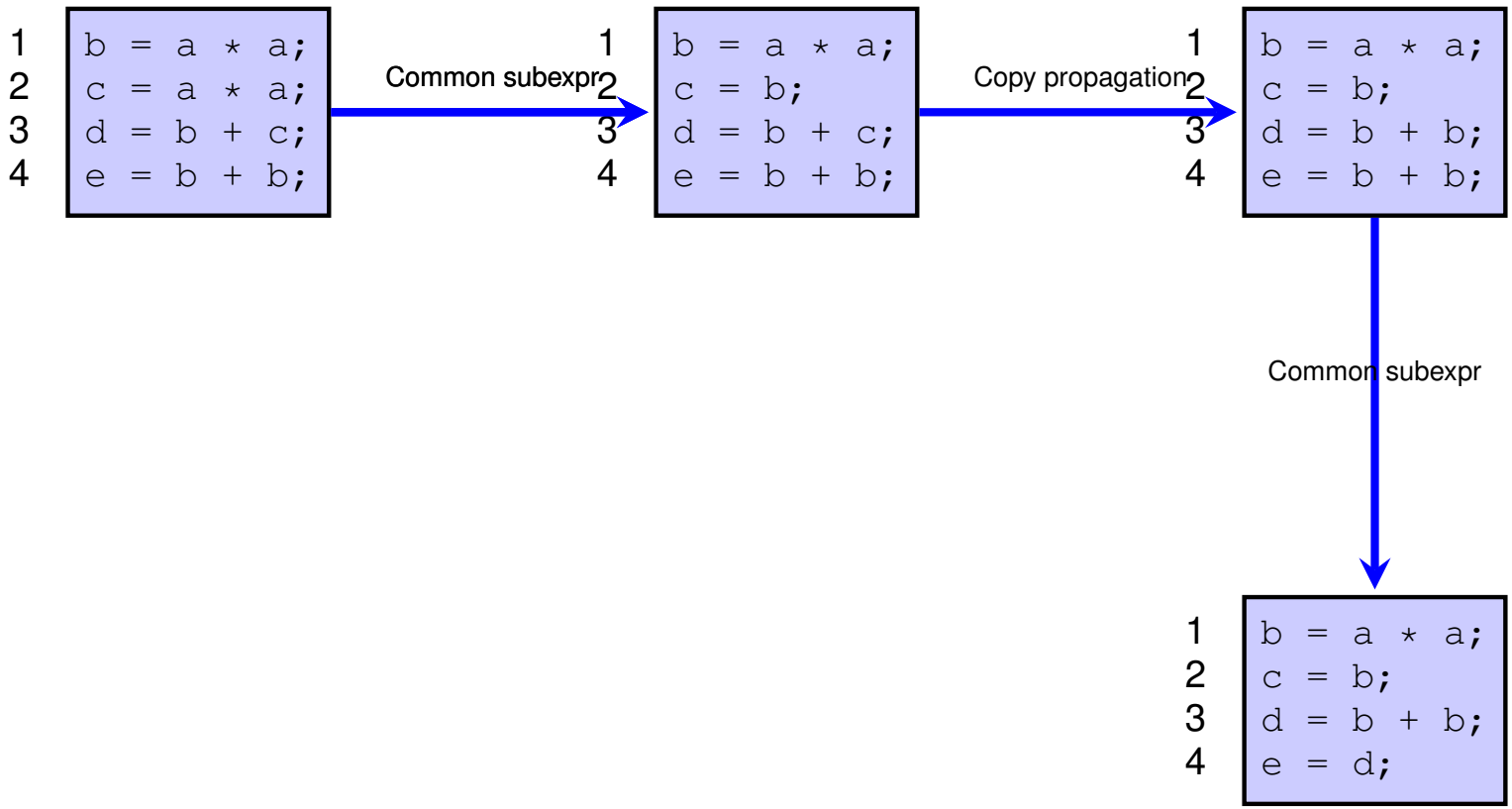
Common subexpression

- Identify common subexpressions
- Replace the second with previous calculation
- Apply copy propagation may introduce more common subexpressions
  - Copy propagation: After assignment x = y, replace uses of x with y
  - replace *c* by *b* in this example



Common subexpression

- Identify common subexpressions
- Replace the second with previous calculation
- Apply copy propagation may introduce more common subexpressions
  - Copy propagation: After assignment x = y, replace uses of x with y
  - replace *c* by *b* in this example

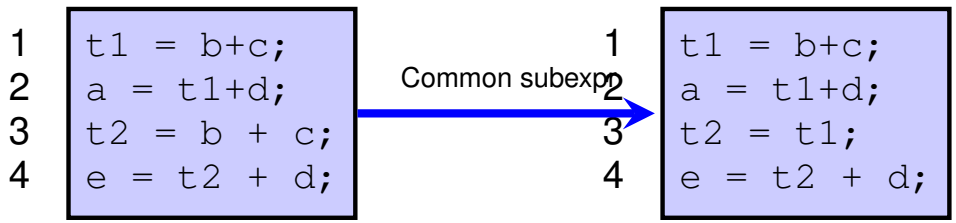


How to do it automatically?

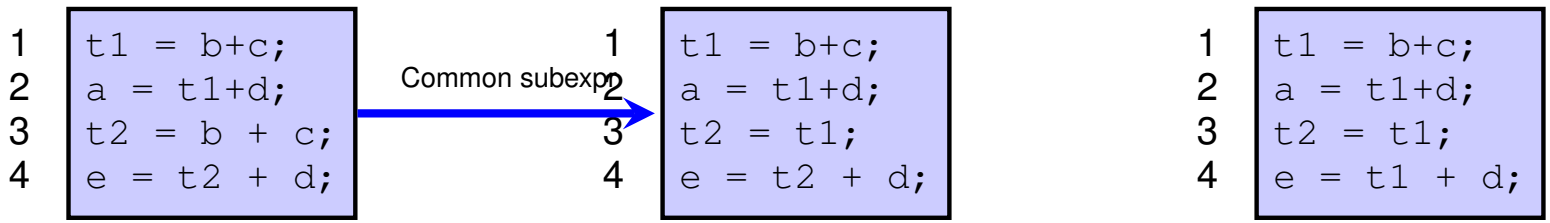
Common subexpression and copy propagation

```
1  t1 = b+c;
2  a = t1+d;
3  t2 = b + c;
4  e = t2 + d;
```

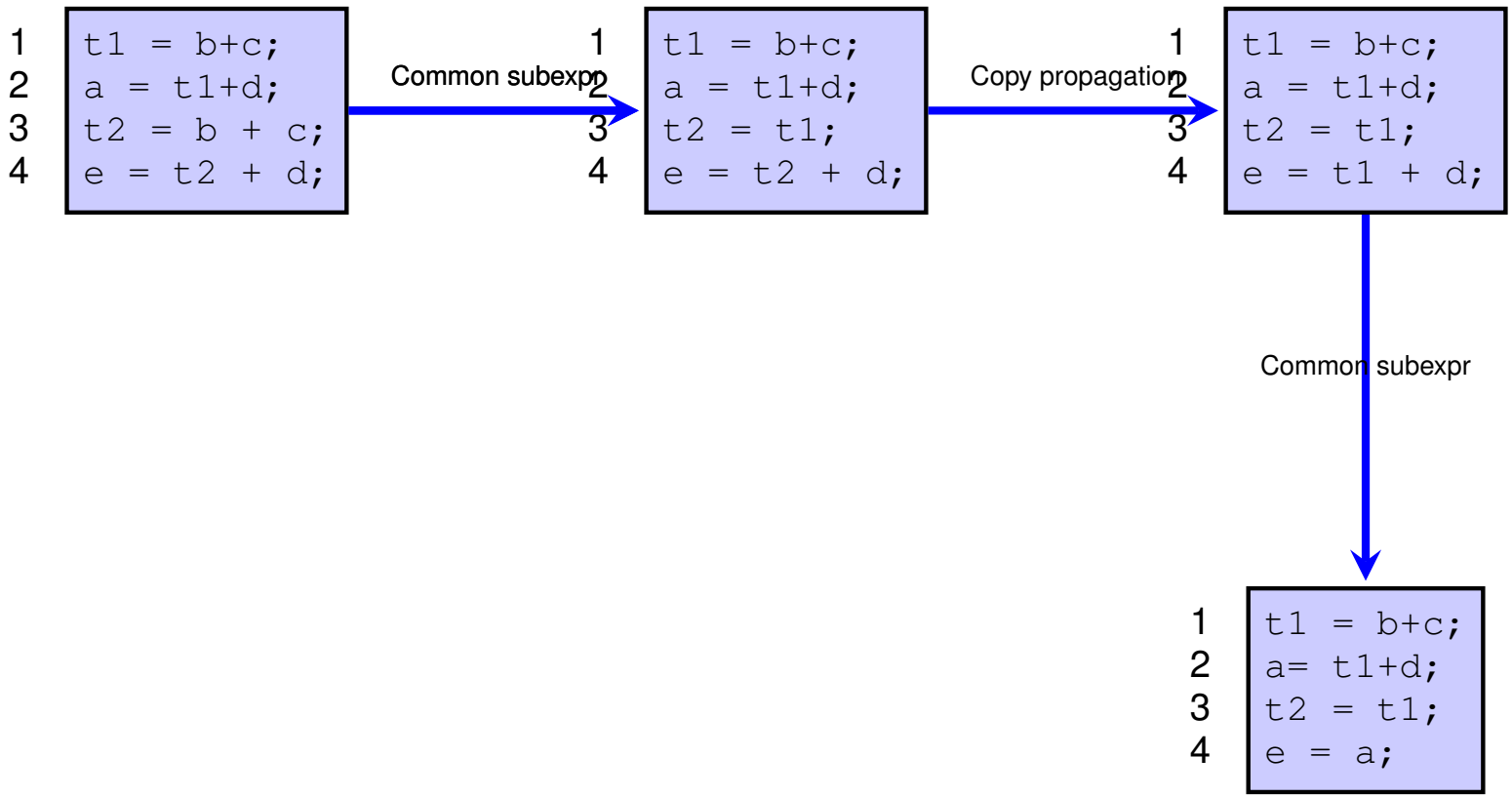
Common subexpression and copy propagation



Common subexpression and copy propagation



Common subexpression and copy propagation



Common subexpressions are more common in Intermediate Representation

Consider the swap operation:

```
1        x=a[i];
2        a[i]=a[j];
3        a[j]=x;
```

Intermediate code if each array element uses 4 bytes:

```
1        t6=4*i;
2        x=a[t6];
3        t7=4*i;
4        t8=4*j;
5        t9=a[t8]
6        a[t7]=t9;
7        t10=4*j;
8        a[t10]=x;
```

Outline

Exploit Algebraic Identities and properties

- More general form of constant folding

```
1  a * 1 ==>   a
2  a* 0  ==> 0
3  a/1  ==> a
4  a+0  ==> a
5  b || false ==> b
6  b && true     b
```

- Can be more advanced

```
1  if (sqrt(x1**2 + y1**2) < sqrt(x2**2 + y2**2))
```

Exploit Algebraic Identities and properties

- More general form of constant folding

```
1  a * 1 ==>   a
2  a* 0  ==> 0
3  a/1  ==> a
4  a+0  ==> a
5  b || false ==> b
6  b && true     b
```

- Can be more advanced

```
1  if (sqrt(x1**2 + y1**2) < sqrt(x2**2 + y2**2))
```

becomes

Exploit Algebraic Identities and properties

- More general form of constant folding

```
1  a * 1 ==>   a
2  a* 0  ==> 0
3  a/1  ==> a
4  a+0  ==> a
5  b || false ==> b
6  b && true     b
```

- Can be more advanced

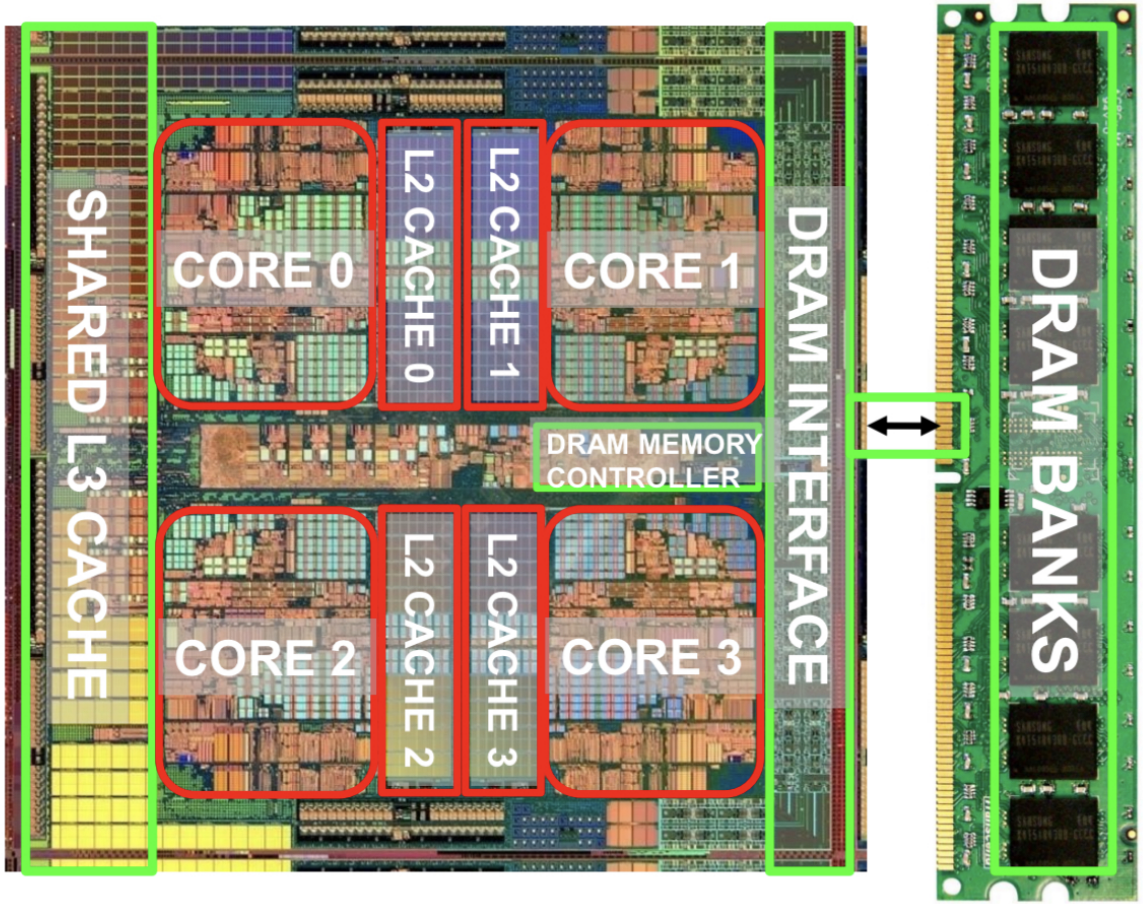
```
1  if (sqrt(x1**2 + y1**2) < sqrt(x2**2 + y2**2))
```

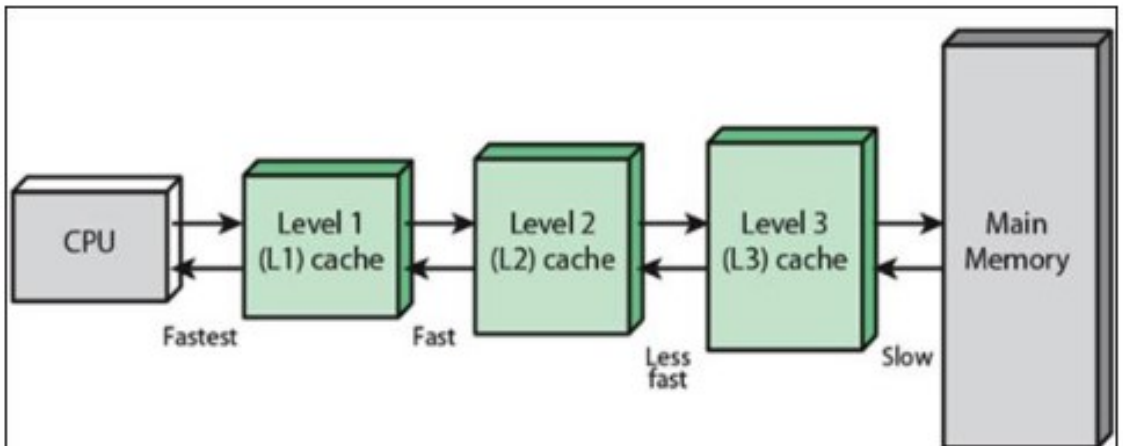
becomes

```
1  if (x1**2 + y1**2 <  x2**2 + y2**2)
```

Outline

Memory hierarchy

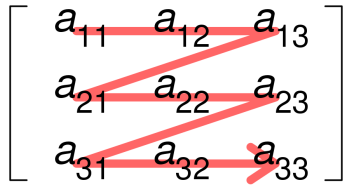




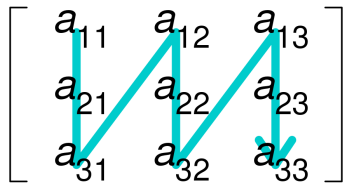
Column and Row Major

Memory Location	Matlab (Column Major)	C / C++ (Row Major)
1	a(1,1)	a[0][0]
2	a(2,1)	a[0][1]
3	a(3,1)	a[0][2]
⋮	⋮	⋮
n	a(n,1)	a[0][n-1]
n+1	a(1,2)	a[1][0]
n+2	a(2,2)	a[1][1]
⋮	⋮	⋮
2n	a(n,2)	a[1][n-1]
2n+1	a(1,3)	a[2][0]
2n+2	a(2,3)	a[2][1]
⋮	⋮	⋮

Row-major order



Column-major order



- Row-major: C, C++, Numpy in Python
- Column-major: Matlab, Fortran
- Neither: Java, Python

Rearranging Loop Order

```
1   for i = 1, 500
2     for  j = 1, 625
3         a(i,j) = i * exp(j)
```

will take a different amount of execution time than

```
1   for  j = 1, 625
2     for  i = 1, 500
3         a(i,j) = i * exp(j)
```

```
n=5000;
t = cputime;
for i=1:n
    for j=1:n
        a(i,j)=1.0;
    end
end
cputime-t

t = cputime;
for j=1:n
    for i=1:n
        a(i,j)=a(i,j)^2;
    end
end
cputime-t

t = cputime;
for i=1:n
    for j=1:n
        a(i,j)=a(i,j)^2;
    end
end
cputime-t
```

```
ans =

    1.8300


ans =

    2.9200
```

Outline

Testing by Order of Frequency

```
1  select case (number)
2    case (rarely true)
3      useful stuff done here
4    case (sometimes true)
5      something else useful done here
6    case (usually true)
7      normal thing done here
8  end select
```

replace with

```
1  select case (number)
2    case (usually true)
3      normal thing done here
4    case (sometimes true)
5      something else useful done here
6    case (rarely true)
7      useful stuff done here
8  end select
```

Stop Testing When You Know the Answer

```
1   if (x > 10) and (x < 20)
```

replace with

Stop Testing When You Know the Answer

```
1   if (x > 10) and (x < 20)
```

replace with

```
1   if (x > 10) then
2      if (x < 20) then
```

or

```
1   if (x < 20) then
2      if (x > 10) then
```

Branches in Loops

- Move conditionals outside of the loops.

```
1   for (i = 1, 1000) {
2     if (i < 100) then
3       a(i) = 10
4     else
5       a(i) = 20
6   }
```

Branches in Loops

- Move conditionals outside of the loops.

```
1  for (i = 1, 1000) {
2    if (i < 100) then
3      a(i) = 10
4    else
5      a(i) = 20
6  }
```

becomes

```
1  for (i = 1, 99){
2      a(i) = 10
3  }
4
5  for  (i = 100, 1000){
6      a(i) = 20
7  }
```

Loop Unrolling

- Loops have extra cost on testing conditions
- Unrolling the loops saves some testing executions

```
1  for (i = 1; i< 400000; i++){
2    a(i) = i * exp(i)
3  }
```

could be better written as

```
1  for (i = 1; i<400000; i=i+4){
2    a(i) = i * exp(i)
3    a(i+1) = (i+1) * exp(i+1)
4    a(i+2) = (i+2) * exp(i+2)
5    a(i+3) = (i+3) * exp(i+3)
6  }
```

Eliminate Loops with Low Trip Counts

```
1   do i = 1, 3
2     a(i) = i * exp(i)
3   enddo
```

could better be written as

```
1     a(1) =  exp(1)
2     a(2) =  2*exp(2)
3     a(3) =  3*exp(3)
```

Optimization of higher-order functions

```
(map f)(map g x)
```

becomes

```
map (f o g) x
```

Changing Loop Order

```
1   do j = 1, 100
2     do i = 1, 5
3       total = total + a(i,j)
4     enddo
5   enddo
```

- The inner loop has 6 tests. The outer loop repeats 100 times (101 tests). Total 601 total tests.

Changing Loop Order

```
1   do j = 1, 100
2     do i = 1, 5
3       total = total + a(i,j)
4     enddo
5   enddo
```

- The inner loop has 6 tests. The outer loop repeats 100 times (101 tests). Total 601 total tests.

becomes

```
1   do j = 1, 5
2     do i = 1, 100
3       total = total + a(i,j)
4     enddo
5   enddo
```

- The inner loop tests 101 times.
- The outer loop executes 5 times (6 tests) - thus 505+1 tests in the loop.

Procedure In-lining

- There is overhead each time a function or routine is called.
- You can eliminate this overhead by “in-lining” the function or subroutine into the code.
- This can usually be done in one of three ways
  - Specify the routines to in-line on the compiler line
  - Putting in-line directives into the code
  - Letting the compiler figure it out automatically

```
1  double pi(double r){ return pi*r*r;}
2  ...
3  for (int i=0; i<1000; i++){
4          x=pi(i);
5   }
```

Outline

Constant folding and simplification

- Deduce at compile time that a value of an expression is a constant
- Use the constant to simplify the program
- Folding: Constant variables are replaced by its definition

```
1   3+4
```

turns into

```
1   7
```

```
1   x*1
```

becomes

```
1   x
```

Constant folding

```
1   k = 23
2   tmp1 = 100
3   for i = 1:1000
4     j = i + tmp1 * k
```

becomes

Constant folding

```
1   k = 23
2   tmp1 = 100
3   for i = 1:1000
4     j = i + tmp1 * k
```

becomes

```
1   k = 23
2   tmp1 = 100
3   for i = 1:1000
4     j = i + 2300
```

Constant folding

```
1   for  (int i = 1; i<1000; i++){
2     j = i + 100 * 15 + sin(3.1) * exp(4)
3   }
```

becomes

Constant folding

```
1   for  (int i = 1; i<1000; i++){
2     j = i + 100 * 15 + sin(3.1) * exp(4)
3   }
```

becomes

```
1   for  (int i = 1; i<1000; i++){
2     j = i + 204.63973
3   }
```

Constant folding

```
1  for  (int i = 1; i<1000; i++){
2    j = i + 100 * 15 + sin(3.1) * exp(4)
3  }
```

becomes

```
1  for  (int i = 1; i<1000; i++){
2    j = i + 204.63973
3  }
```

```
1  if true then s else t
```

Constant folding

```
1  for  (int i = 1; i<1000; i++){
2    j = i + 100 * 15 + sin(3.1) * exp(4)
3  }
```

becomes

```
1  for  (int i = 1; i<1000; i++){
2    j = i + 204.63973
3  }
```

```
1  if true then s else t
```

```
1   s
```

Constant folding lead to unreachable code removal

```
1          debug=FALSE;
2          S1;
3          if (debug) {
4            print ...
5          }
6          S2;
```

```
1          S1;
2          S2;
```

Outline

Dead code removal

```
1          x=y+1;
2          y=2*z;
3          x=y+z;
4          z=1;
5          z=x;
```

- Which statements are dead and can be removed?

Dead code removal

```
1          x=y+1;
2          y=2*z;
3          x=y+z;
4          z=1;
5          z=x;
```

- Which statements are dead and can be removed?

```
1   x=y+1;
2   y=2*z;
3   x=y+z;
4   z=1;
5   z=x;
```

Dead code removal

```
1          x=y+1;
2          y=2*z;
3          x=y+z;
4          z=1;
5          z=x;
```

- Which statements are dead and can be removed?

```
1  x=y+1;
2  y=2*z;
3  x=y+z;
4  z=1;
5  z=x;
```

```
1  x=y+1;
2  y=2*z;
3  x=y+z;
4  z=1;
5  z=x;
```

Dead code removal

```
1   x=y+1;
2   y=2*z;
3   x=y+z;
4   z=1;
5   z=x;
```

- Need to know whether values assigned to x at (1) is never used later (i.e., x is dead at statement (1))
- Obvious for this simple example (with no control flow)
- Not obvious for complex flow of control

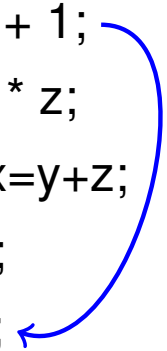
Add control flow to example

```
1   x = y + 1;
2   y = 2 * z;
3   if (d) x=y+z;
4   z = 1;
5   z = x;
```

- Is $x = y + 1$ deadcode?
- Is $z = 1$ deadcode?

add control flow to the example

1. x = y + 1;
2. y = 2 * z;
3. if(d) x=y+z;
4. z = 1;
5. z = x;



- $x = y + 1$ is not dead code
- on **some executions**, value of x is used later

Dead variable in while loop

```
1   while (b) {
2       x = y + 1;
3       y = 2 * z;
4       if(d) x=y+z;
5       z = 1;
6   }
7   z = x;
```

- is x=y+1 dead code?
- is $z = 1$ dead code?

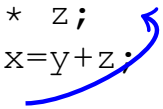
Dead code in while loop

```
1   while (b) {
2             x = y + 1;
3             y = 2 * z;
4             if(d) x=y+z;
5             z = 1;
6   }
7   z = x;
```

- $x = y + 1$ not dead (same as before)
- $z = 1$ no longer dead !

Dead code in while loop

```
1   while (b) {
2             x = y + 1;
3             y = 2 * z;
4             if(d) x=y+z;
5             z = 1;
6   }
7   z = x;
```



- $x = y + 1$ not dead (same as before)
- $z = 1$ no longer dead !

Low level code

Harder to eliminate dead code in low-level code:

```
1    label L1
2    fjump c L2
3    x = y + 1;
4    y = 2 * z;
5    fjump d L3
6    x = y+z;
7    label L3
8    z = 1;
9    jump L1
10   label L2
11   z = x;
```

is x=y+1 or z=1 dead statement?

low level code

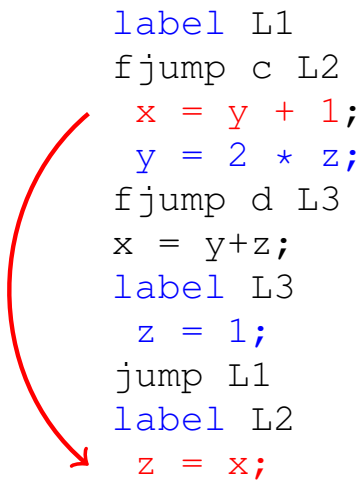
Harder to eliminate dead code in low-level code:

```
1     label L1
2     fjump c L2
3      x = y + 1;
4      y = 2 * z;
5     fjump d L3
6     x = y+z;
7     label L3
8      z = 1;
9     jump L1
10    label L2
11     z = x;
```

is x=y+1 or z=1 dead statement?

low level code

Harder to eliminate dead code in low-level code:

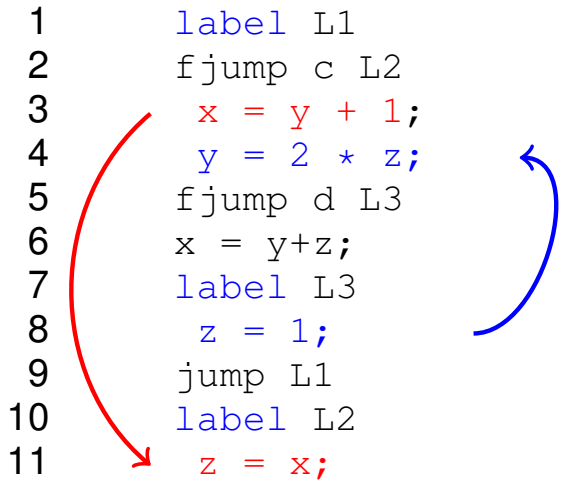


```
1     label L1
2     fjump c L2
3      x = y + 1;
4      y = 2 * z;
5     fjump d L3
6     x = y+z;
7     label L3
8      z = 1;
9     jump L1
10    label L2
11     z = x;
```

is x=y+1 or z=1 dead statement?

low level code

Harder to eliminate dead code in low-level code:



is x=y+1 or z=1 dead statement?

Outline

Outline

Three-address code

- Have at most three addresses (may have fewer)

```
1  a=b OP c
```

- Example

```
1  a=(b+c)*(-d)
```

becomes

```
1  t1=b+c
2  t2=-d
3  a=t1*t2
```

- Intermediate representation describes the Instruction Set of an abstract machine

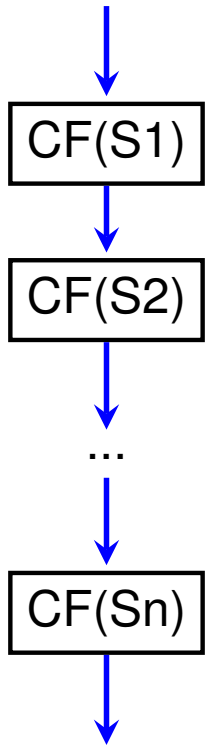
Outline

Control flow graph for high level code

- CF(S)= control flow graph of S
- CF(S) is a single-entry single-exit graph
- Define CF(S) recursively

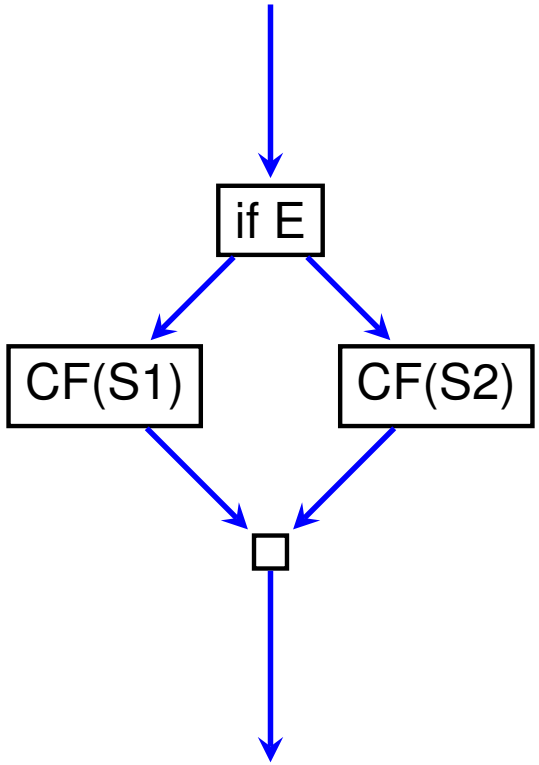
CF for sequential compositions

CF(S1; S2; ...; Sn)=



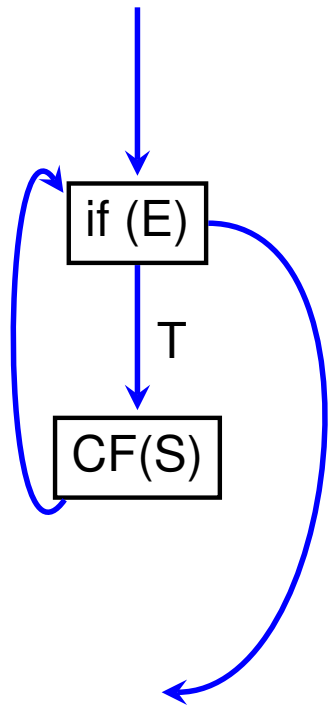
CF graph for IF statement

CF(if E then S1else S2)=



CF graph for WHILE statement

CF(while (E) S)=



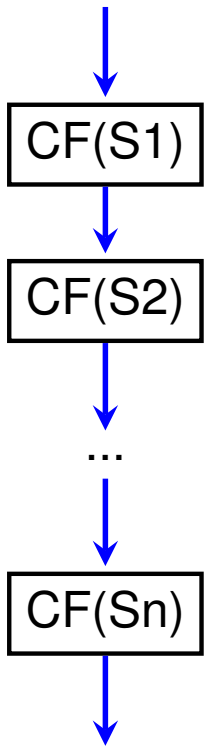
example

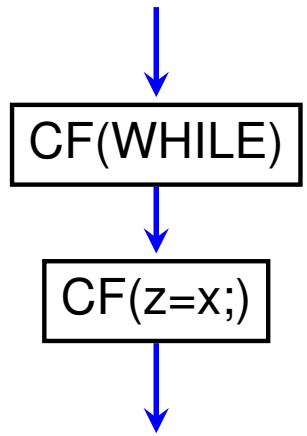
```
1   while (c) {
2     x = y + 1;
3     y = 2 * z;
4     if (d) x = y+z;
5     z = 1;
6   }
7   z = x;
```

example

```
1   while (c) {
2     x = y + 1;
3     y = 2 * z;
4     if (d) x = y+z;
5     z = 1;
6   }
7   z = x;
```

CF(S1; S2; ...; Sn)=

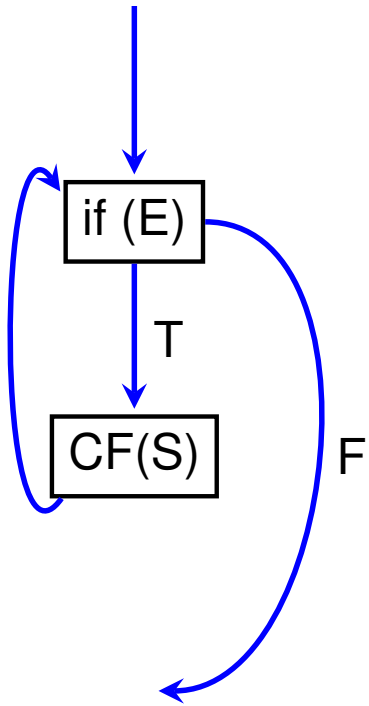


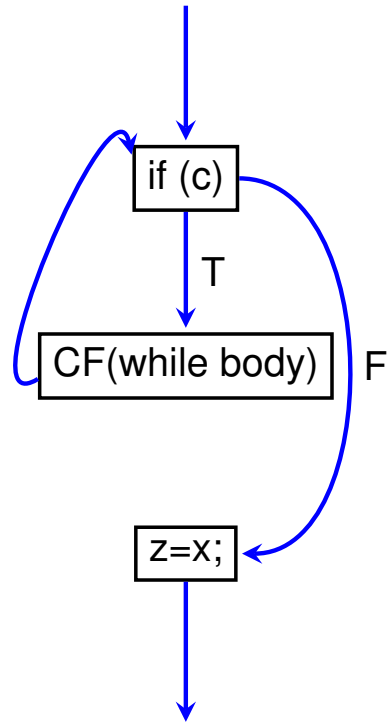


Example of translating into a CF graph

```
1   while (c) {
2     x = y + 1;
3     y = 2 * z;
4     if (d) x = y+z;
5     z = 1;
6   }
7   z = x;
```

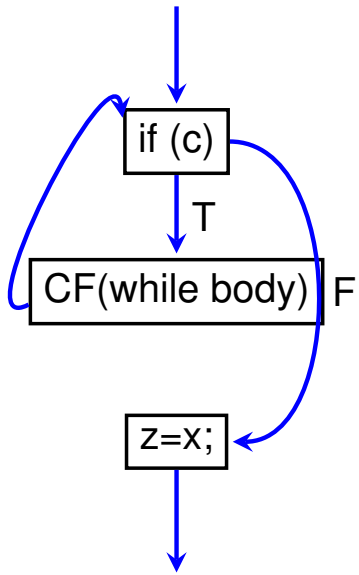
CF(while (E) S)=

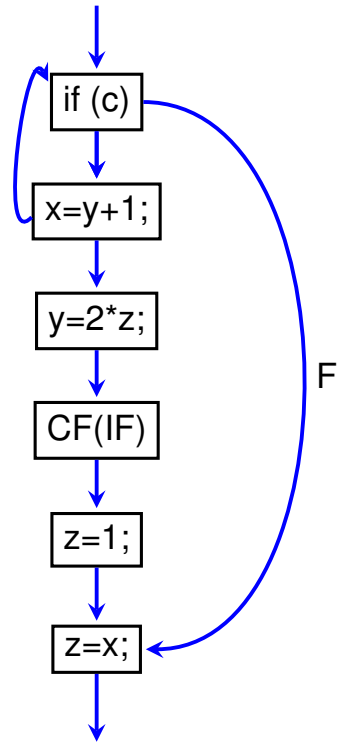




example

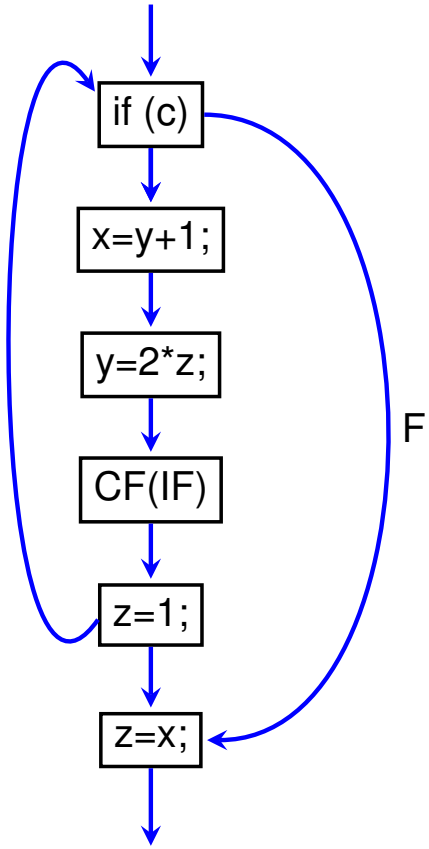
```
1   while (c) {
2     x = y + 1;
3     y = 2 * z;
4     if (d) x = y+z;
5     z = 1;
6   }
7   z = x;
```

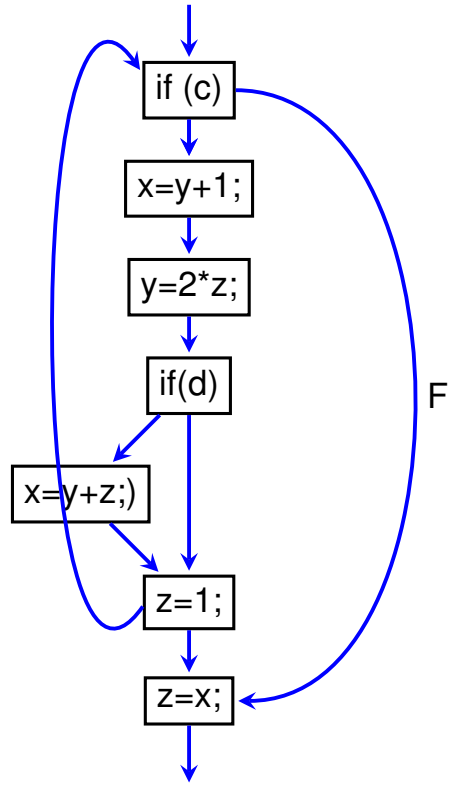




example

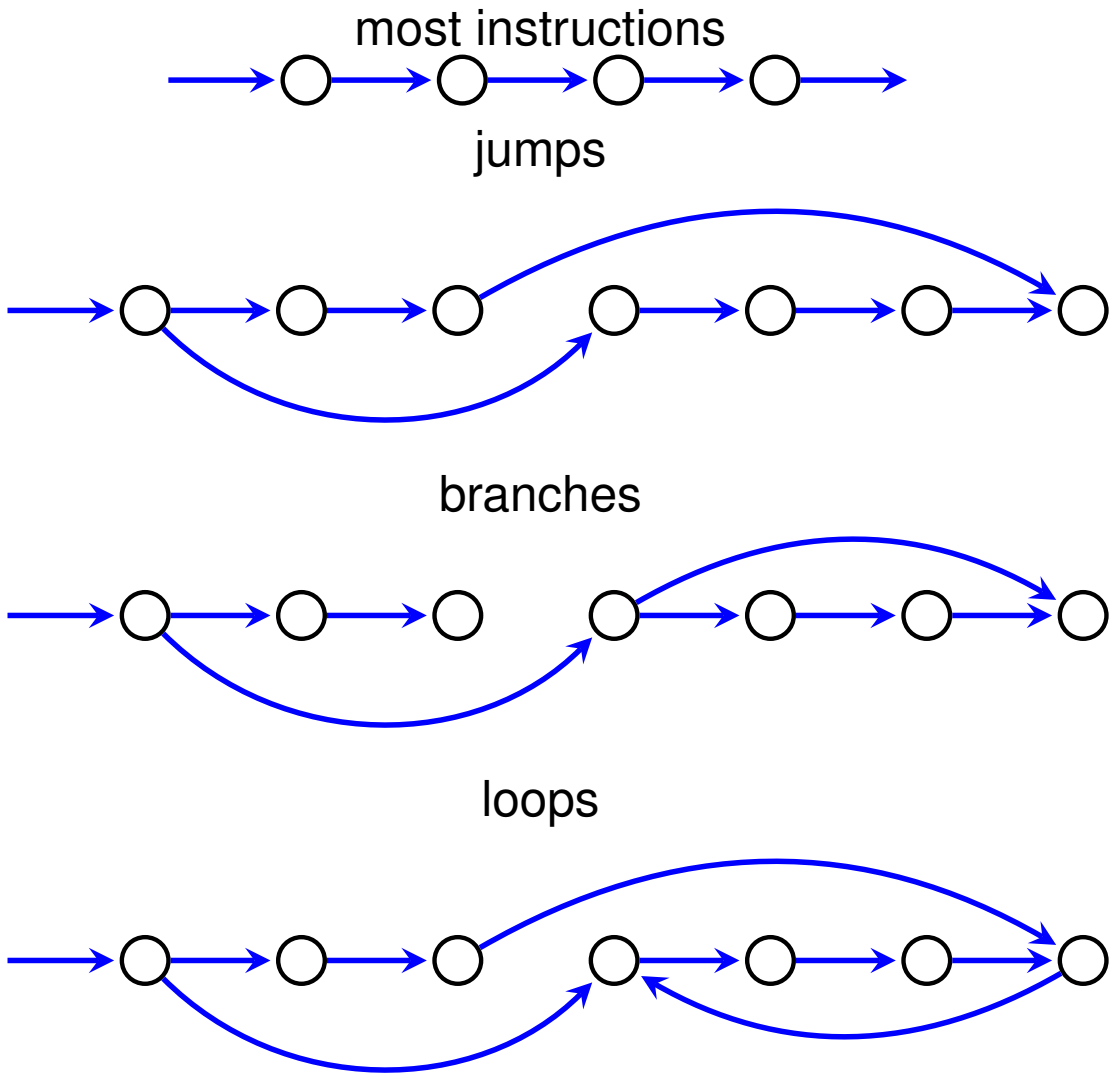
```
1   while (c) {
2     x = y + 1;
3     y = 2 * z;
4     if (d) x = y+z;
5     z = 1;
6   }
7   z = x;
```





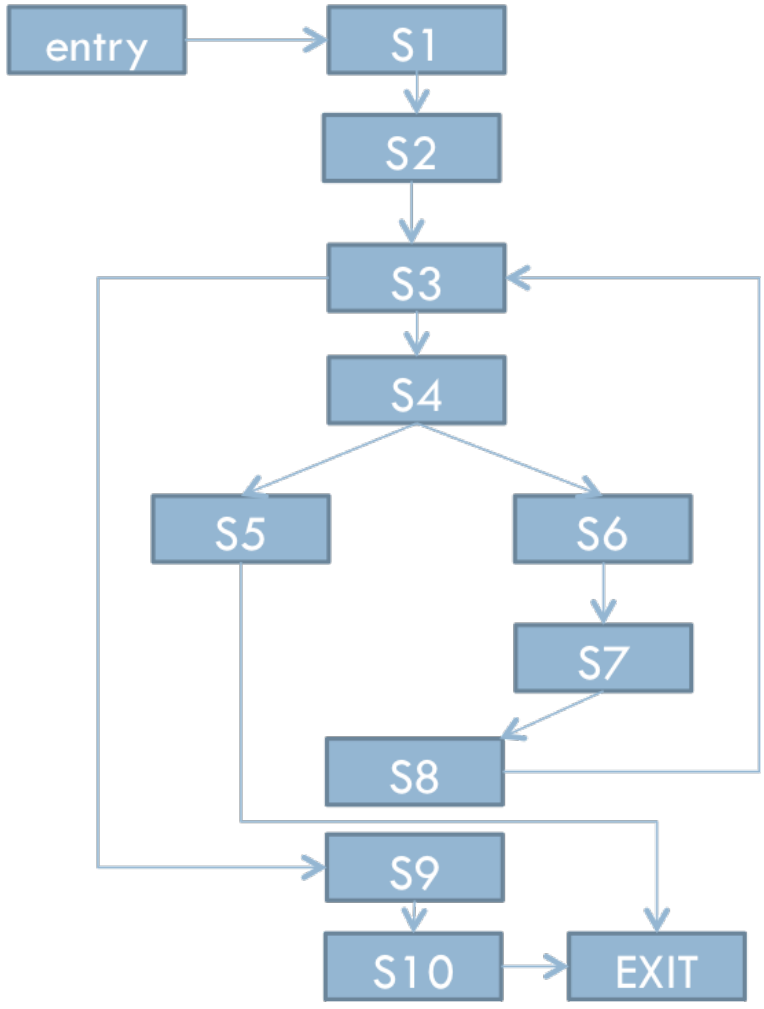
Outline

Control flow graph



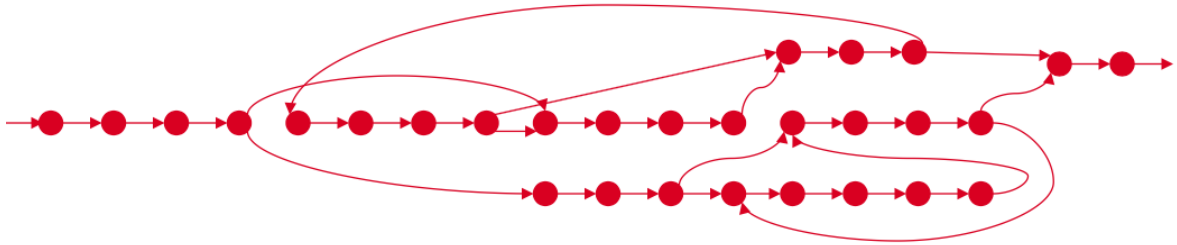
control flow graph

```
Procedure AVG
S1          count=0;
S2          fread(fptr , n)
S3          while(not EOF) do
S4            if(n<0)
S5              return(error)
              else
S6              nums[count]=n
S7             count++
              endif
S8          fread(fptr , n);
            endwhile
S9          avg= mean(nums , count)
S10         return (avg)
```

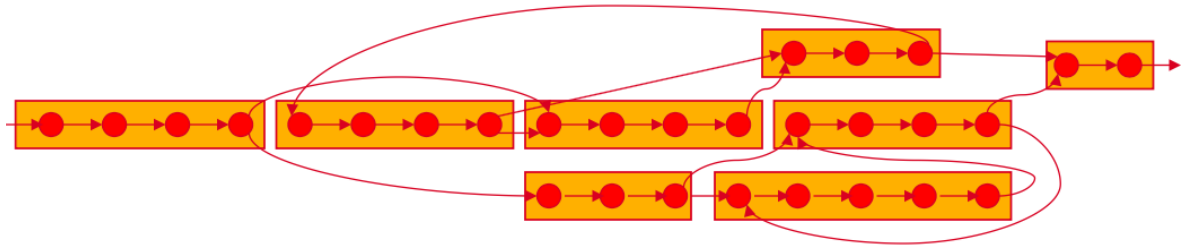


the graph can be very large, needs to be simplified

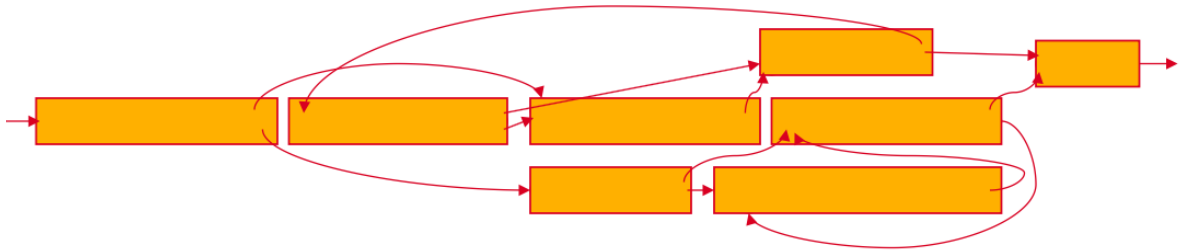
control flow graph



most instructions are executed sequentially



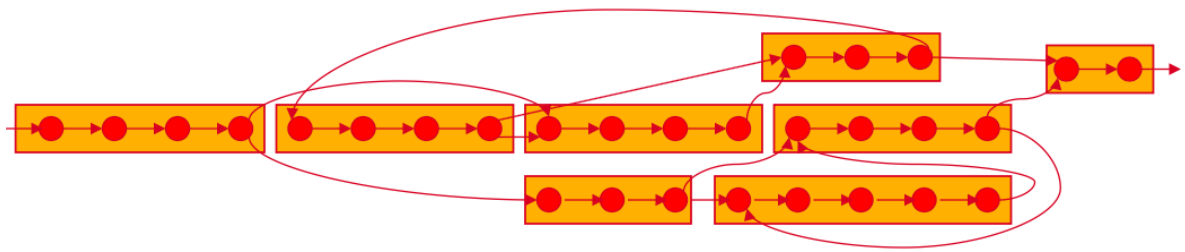
they do not change control flow, should be merged



the merged blocks are called basic blocks

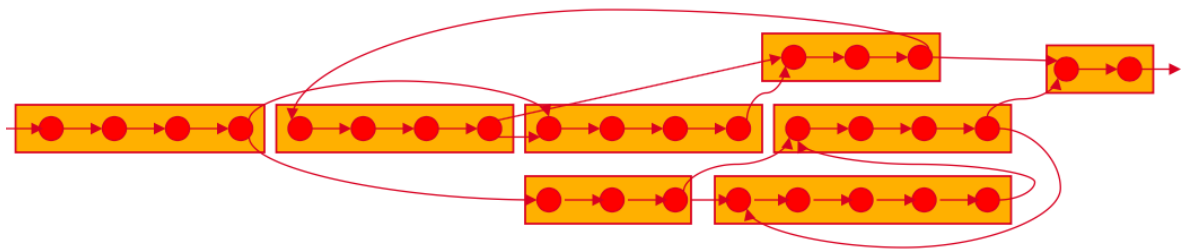
definition of basic blocks

- Basic block: a maximal sequence of instructions such that
  - only the first instruction can be reached from outside of the basic block
  - all the instructions are executed consecutively iff the first instruction is executed
  - no branch or jump instruction in the basic block (except the last instruction)
  - no labels within the basic block (except before the first instruction)



identifying basic block

- Determine the leaders, the first statements of basic blocks
  - S is a leader if
    - S is The first statement in the sequence (entry point) or
    - S is the target of a branch (conditional or unconditional) or
    - S immediately following a branch (conditional or unconditional) or a return
- For each leader, its basic block is the leader and all statements up to, but not including, the next leader or the end of the program



Basic block example for low level code

Can we find the leaders/blocks for the following?

```
1   (1) i:=m-1
2   (2) j:=n
3   (3) t1:=4*n
4   (4) v := a[t1]
5   (5) i := i + 1
6   (6) t2:=4*i
7   (7) t3 := a[t2]
8   (8) if t3<v goto(5)
9   (9) j:=j -1
10  (10) t4 := 4 * j
11  (11) t5 := a[t4]
12  (12) if t5 > v goto (9)
13  (13) if i = j goto (23)
14  (14) t6 := 4*i
15  (15) x := a[t6]
```

Basic block example for low level code

Can we find the leaders/blocks for the following?

```
1   (1) i:=m-1
2   (2) j:=n
3   (3) t1:=4*n
4   (4) v := a[t1]
5   (5) i := i + 1
6   (6) t2:=4*i
7   (7) t3 := a[t2]
8   (8) if t3<v goto(5)
9   (9) j:=j -1
10  (10) t4 := 4 * j
11  (11) t5 := a[t4]
12  (12) if t5 > v goto (9)
13  (13) if i = j goto (23)
14  (14) t6 := 4*i
15  (15) x := a[t6]
```

```
1   (1) i:=m-1
2   (2) j:=n
3   (3) t1:=4*n
4   (4) v := a[t1]
5   (5) i := i + 1
6   (6) t2:=4*i
7   (7) t3 := a[t2]
8   (8) if t3 < v goto (5)
9   (9) j:=j - 1
10  (10) t4 := 4 * j
11  (11) t5 := a[t4]
12  (12) if t5 > v goto (9)
13  (13) if i = j goto (23)
14  (14) t6 := 4*i
15  (15) x := a[t6]
```

The DAG representation of basic blocks

- DAGs(Directed Acyclic Graph) are useful for determining common subexpressions within a block,
- Labels on nodes of DAG:
  - Leaves are labeled by variable names or constants.
  - Interior nodes are labeled by operator symbols.
  - Nodes are also optionally given sequence of identifiers for labels (representing computed values).

DAG used for Common subexpression elimination

- Which subexpressions are common?
- Reflected in DAG (b and d are the same node)

```
1    a := b+c
2    b := a-d
3    c := b+c
4    d := a-d
```

becomes

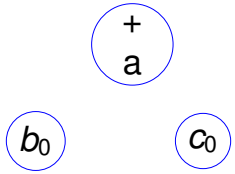
DAG used for Common subexpression elimination

- Which subexpressions are common?
- Reflected in DAG (b and d are the same node)

```
1       a := b+c
2       b := a-d
3       c := b+c
4       d := a-d
```

becomes

```
1       a := b+c
2       b := a-d
3       c := b+c    (different from first statement)
4       d := b
```



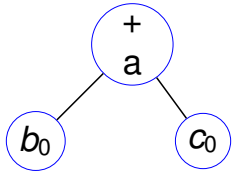
DAG used for Common subexpression elimination

- Which subexpressions are common?
- Reflected in DAG (b and d are the same node)

```
1       a := b+c
2       b := a-d
3       c := b+c
4       d := a-d
```

becomes

```
1       a := b+c
2       b := a-d
3       c := b+c     (different from first statement)
4       d := b
```



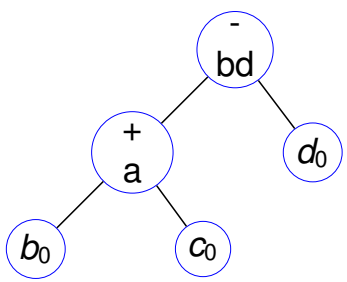
DAG used for Common subexpression elimination

- Which subexpressions are common?
- Reflected in DAG (b and d are the same node)

```
1       a := b+c
2       b := a-d
3       c := b+c
4       d := a-d
```

becomes

```
1       a := b+c
2       b := a-d
3       c := b+c     (different from first statement)
4       d := b
```



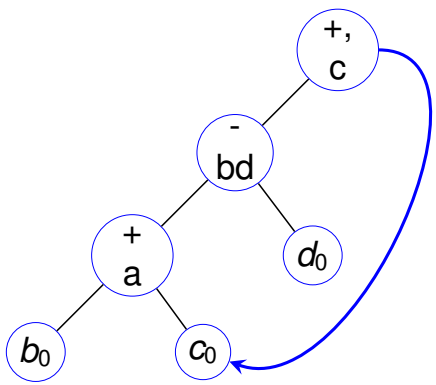
DAG used for Common subexpression elimination

- Which subexpressions are common?
- Reflected in DAG (b and d are the same node)

```
1       a := b+c
2       b := a-d
3       c := b+c
4       d := a-d
```

becomes

```
1       a := b+c
2       b := a-d
3       c := b+c     (different from first statement)
4       d := b
```



Outline

Dead code and Live variable

- **Dead-code elimination:** Suppose x is dead (never subsequently used), at the point after the statement $x := y + z$ appears in a basic block. Then, this statement may be removed.
- Live variable: variable holding values that may be used later
- Live variable set changes when a statement *I* is executed:
  - For a statement x:=y+z
  - x is not live before the statement
  - y, z are live before the statement

Live variable

- if a variable is defined in I, it is NOT live before I.
- if a variable is used in I, it is live before I.
- For other variables, their status remain the same
- Mathematically

$$before[I] = after[I] - def[I] \cup use[I] \tag{1}$$

- where
  - before[I]: live variables at the beginning of I
  - after[I]: live variables at the end of I
  - def[I]: variables defined by instruction I
  - use[I]: variables used by instruction I

calculate live variables

2. x=y+1

4. y=2*z

6. if(d)
7. *liveVariables*

calculate live variables

2. x=y+1

4. y=2*z
5. $liveVariables \cup \{d\}$
6. if(d)
7. $liveVariables$

calculate live variables

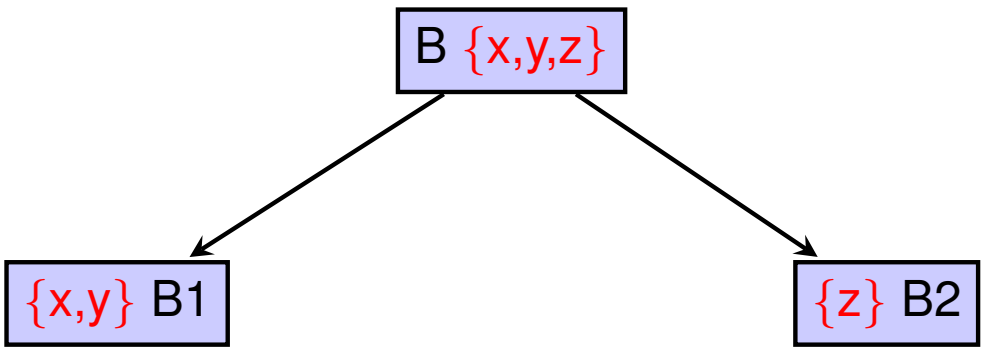
2. x=y+1
3. $liveVariables \cup \{d, z\} - \{y\}$
4. y=2*z
5. $liveVariables \cup \{d\}$
6. if(d)
7. $liveVariables$

calculate live variables

1. $liveVariables \cup \{d, z, y\} - \{x\}$
2. x=y+1
3. $liveVariables \cup \{d, z\} - \{y\}$
4. y=2*z
5. $liveVariables \cup \{d\}$
6. if(d)
7. $liveVariables$

rules for calculating live variables (for blocks)

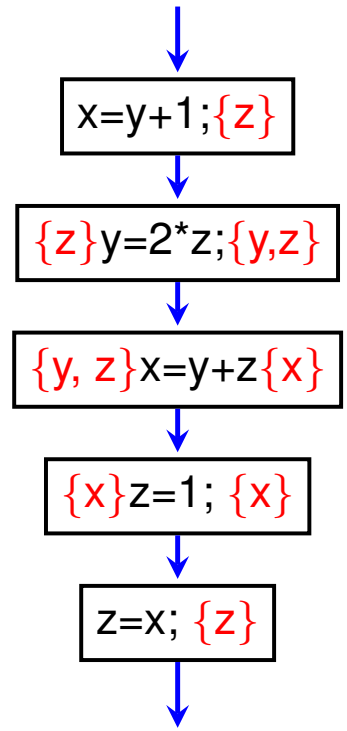
- if a variable is live before a successor of block B, it is also live after the B



Live variable tracing: within basic block

- works backwards from the bottom
- After ”x=y+1”, *x* is not a live variable, hence it can be deleted.
- After ”z=x”, *z* is not live, so this is a dead statement.

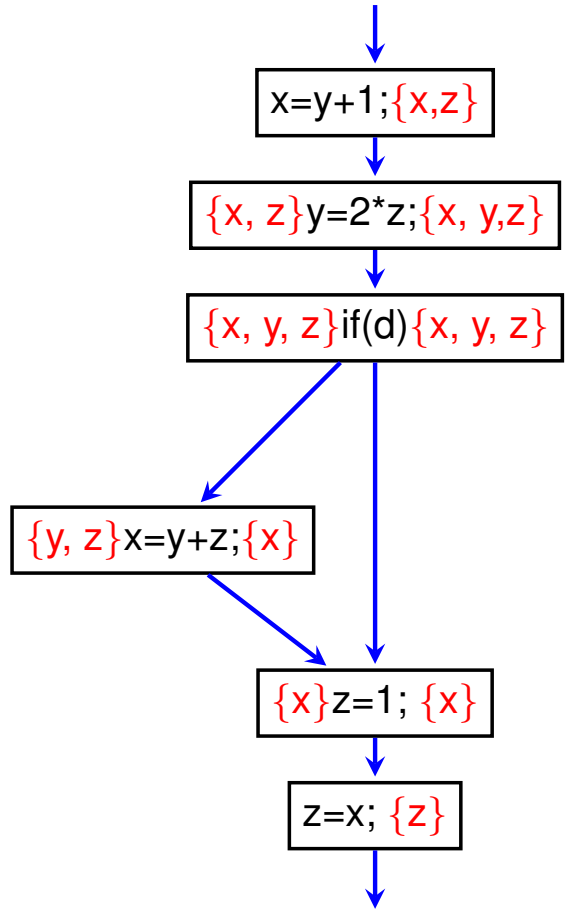
```
1    x = y + 1;
2    y = 2 * z;
3    x = y+z;
4    z = 1;
5    z = x;
6    return z;
```



Live variable tracing: branches

- The key difference is the branch part:
  - IF block collects live variables from both branches
  - Here ”x, y, z” is the union of ”y,z” and ”x” from two branches.

```
1    x = y + 1;
2    y = 2 * z;
3    if (d) x = y+z;
4    z = 1;
5    z = x;
6    return z;
```



Another example: which statement(s) can be removed?

1. a = b;

3. c = a;
4. d = a + b;

6. e = d;

8. d = a;

10. f = e;
11. {b, d}

Another example: which statement(s) can be removed?

1. a = b;

3. c = a;
4. d = a + b;

6. e = d;

8. d = a;
9. {b,d}
10. f = e;
11. {b, d}

Another example: which statement(s) can be removed?

1. a = b;

3. c = a;
4. d = a + b;

6. e = d;
7. {b,a}
8. d = a;
9. {b,d}

11. {b, d}

Another example: which statement(s) can be removed?

1. a = b;

3. c = a;
4. d = a + b;
5. {b,a}

7. {b,a}
8. d = a;
9. {b,d}

11. {b, d}

Another example: which statement(s) can be removed?

1. a = b;

3. c = a;

5. {b,a}

7. {b,a}
8. d = a;
9. {b,d}

11. {b, d}

Another example: which statement(s) can be removed?

1. a = b;
2. {b,a}

5. {b,a}

7. {b,a}
8. d = a;
9. {b,d}

11. {b, d}

gcc Optimization Options

```
1  -g: Include debug information, no optimization
2  -O0: Default, no optimization
3  -O1: Do optimizations that don t take too long
4  CP, CF, CSE, DCE, LICM, inlining small functions
5  -O2: Take longer optimizing, more aggressive scheduling
6  -O3: Make space/speed trade-offs: loop unrolling, more inlining
7  -Os: Optimize program size
```

Takeaways

- Optimization needs to be done only when performance requirement is critical.
- It can make code look ugly, and difficult to understand
- Many of them are done by compilers.
- Some optimizations need to be done manually
- Common optimization methods: Loop Invariant code motion; Constant folding; Algebraic simplification; common subexpression elimination; dead code removal;
- Implementation techniques: Control flow graph; Basic block; live variable.