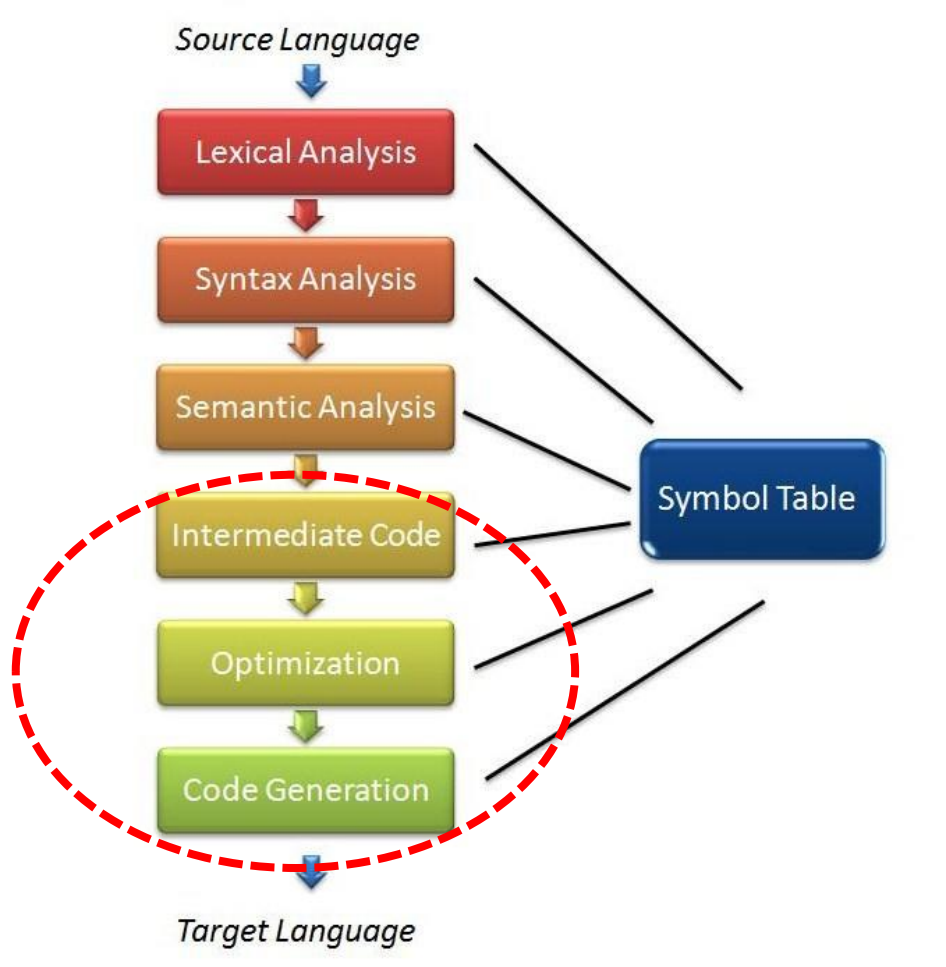
Compiler Construction

ARNOLD MEIJSTER
A.MEIJSTER@RUG.NL

Compiler Structure



Example translation (to pseudo assembly)

```
x = 0;
while ((x+1)*(x+1) <= N) {
  x++;
}
---------------------------
       M(x) = 0
lw:    R0 = M(x)
       R0 = R0 + 1
       R1 = R0
       R1 = R1*R0
       cc = R1 ? N
       jmpgtr lew
       M(x) = R0
       jmp lw
lew:
```

Example translation (to minimalistic C)

```
x = 0;
while ((x+1)*(x+1) <= N) {
  x++;
}
---------------------------
       x = 0;
lb1:   int t0 = x+1;
       int t1 = x+1;
       int t2 = t0*t1;
       if (t2 > N) goto lb2;
       x++;
       goto lb1;
lb2:
```

Example translation (optimized)

```
x = 0;
while ((x+1)*(x+1) <= N) {
  x++;
}
---------------------------
       x = 0;
lb1:   int t0 = x+1;
       int t1 = t0*t0;
       if (t1 > N) goto lb2;
       x++;
       goto lb1;
lb2:
```

Intermediate Representation (IR)

Consider the expression **`x - 2*y`**

Stack based (one address)

```
push x
push 2
push y
multiply
subtract
```

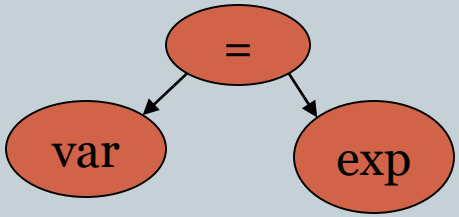
(three address) quadruples
≤ 3 operands, 1 operator

```
t1 = x
t2 = 2
t3 = y
t4 = t2 * t3
t5 = t1 - t4
```

Optimized Translation to quadruple IR instructions

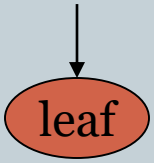
```
a = (a*b) + ((a-c) + (d*e))

int t1 = a*b;
int t2 = a-c;
int t3 = d*e;
int t4 = t2 + t3;
int t5 = t1 + t4;
a = t5;
```

AST Translation to IR instructions



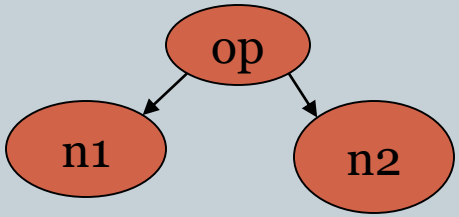

```
/* idx is current t index (temporary index) */
genIRcode(exp);  /* side effect: idx is changed! */
printf("%s = t%d;\n", nameStr(var), idx);
```

Translation to IR instructions



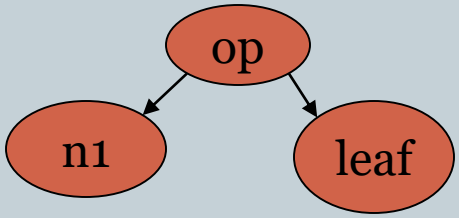

```
/* idx is current temporary index */
idx++;
printf("int t%d = %s;\n", idx, leafStr(leaf));
```

AST Translation to IR instructions



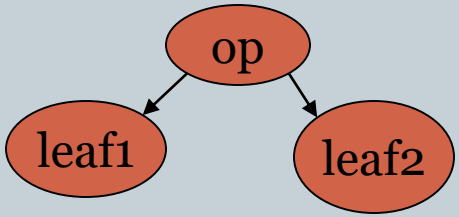

```
/* idx is current temporary index */
genIRcode(n1);
idx1 = idx;
genIRcode(n2);
printf("int t%d = t%d %s t%d;\n",
        idx+1, idx1, operatorStr(op), idx);
idx++;
```

Optimization: Translation to IR instructions




```
/* idx is current temporary index */
genIRcode(n1);
printf("t%d = t%d %s %s;\n",
        idx+1, idx, operatorStr(op), leafStr(leaf));
idx++;
```

Optimization: Translation to IR instructions




```
/* idx is current temporary index */
printf("t%d = %s %s %s;\n", idx+1, leafStr(leaf1),
       operatorStr(op), leafStr(leaf2));
idx++;
```

Translation to IR instructions: if

```
if (x != 100) {  /* equivalent: x-100 != 0 */
    x = x + 1;
}
```

```
            int t0 = x;
            int t1 = 100;
            int t2 = t0 - t1;
            if (t2 == 0) goto lb42:
            int t3 = x;
            int t4 = 1;
            int t5 = t3 + t4;
            x = t5;
      lb42:
```

Translation to IR instructions: if

```
if (B) {
    S
}
```

```
/* lab is current label index */
genIRcode(B);
lab++;
lab1 = lab;
printf("if (t%d==0) goto lb%d;\n", idx, lab);
genIRcode(S);
printf("lb%d: ", lab1);
```

Translation to IR instructions: if-else

```
if (x != 100) {
    S0;
} else {
    S1;
}
```

Note that an if-else produces a jump for both cases. Therefore, if-elses are a bit less efficient than ifs (without elses).

```
            int t0 = x;
            int t1 = 100;
            int t2 = t0 - t1;
            if (t2 == 0) goto lb42:
            …… /* code for S0 */
            goto lb43;
      lb42: …… /* code for S1 */
      lb43:
```

Translation to IR instructions: if-else

```
if (B) {
    S0
} else {
    T0
}
```

```
/* lab is current label index */
genIRcode(B);
lab1 = lab + 1;
lab += 2;
printf("if (!t%d) goto lb%d;\n", idx, lab1);
genIRcode(S0);
printf("goto lb%d;\n", lab1+1);
printf("lb%d: ", lab1);
genIRcode(T0);
printf("lb%d: ", lab1+1);
```

Translation to IR instructions: while

```
x = 0;
while (x != 100) do {
    x = x + 1;
}
```

```
        int t0 = 0;
        x = t0;
lb42:   int t1 = x;
        int t2 = 100;
        int t3 = t1 - t2;
        if (t3 == 0) goto lb43
        int t4 = x;
        int t5 = 1;
        int t6 = t4 + t5;
        x = t6;
        goto lb42;
lb43:
```

Translation to IR instructions: while

```
while (B) do {
    S
}
```

```
/* lab is current label index */
lab1 = lab + 1;
lab += 2;
printf("lb%d: ", lab1);
genIRcode(B);
printf("if (!t%d) goto lb%d;\n", idx, lab1 + 1);
genIRcode(S);
printf("goto lb%d;\n", lab1);
printf("lb%d: ", lab1 + 1);
```

Translation to IR instructions: for

```
for (init; test; update){
    S
}
```

```
/* lab is current label index */
genIRcode(init);
lab1 = lab + 1;
lab += 2;
printf("lb%d: ", lab1);
genIRcode(test);
printf("if (!t%d) goto lb%d;\n", idx, lab1 + 1);
genIRcode(S);
genIRcode(update);
printf("goto lb%d;\n", lab1);
printf("lb%d: ", lab1 + 1);
```

```
init;
while (test){
    S;
    update
}
```

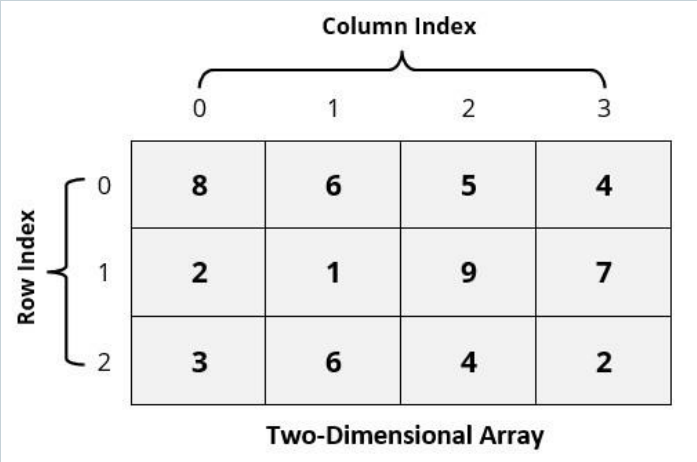
Translation to IR instructions: do-while

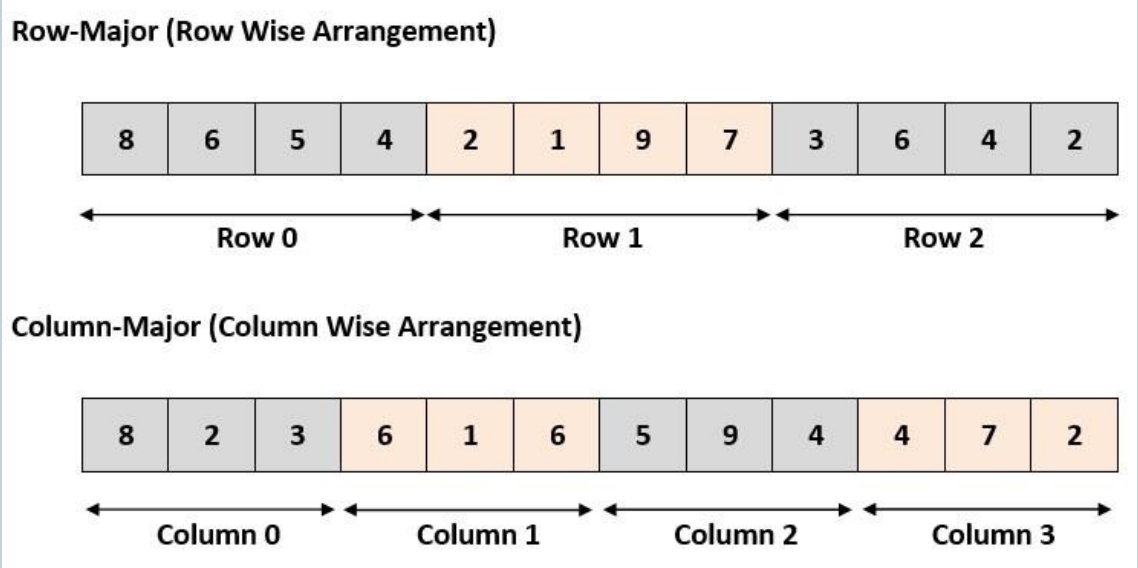
```
do {
    S
} while (B);
```

Note that a do-while is slightly more efficient than a while.
A do-while requires only one jump.

```
/* lab is current label index */
lab1 = lab + 1;
lab++;
printf("lb%d: ", lab1);
genIRcode(S);
genIRcode(B);
printf("if (t%d) goto lb%d;\n", idx, lab1);
```

Indexing 2D arrays: memory layout





C uses Row-Major storage. Fortran uses Column-Major storage.

Indexing C arrays: a[i][j]

- Declaration: **`int a[N][M];`**
- Address of **`a[i][j]`** is given by

```
32 bit int: address(a)+ 4*(i*M + j)
64 bit int: address(a)+ 8*(i*M + j)
```

`address(a)` is the address of `a[0][0]`

Indexing arrays: A[i,j]

- Pascal declaration:

```
var a : array [1..N, 0..M] of integer;
```

- Address of **`a[i,j]`** is given by

$$\texttt{address(a) + sizeof(type)*}$$
$$\texttt{((i-low}_1\texttt{(a))*(high}_2\texttt{(a)-low}_2\texttt{(a)+1) + (j-low}_2\texttt{(a)))}$$

  - $\texttt{low}_i$ and $\texttt{high}_i$ are subscript bounds in dimension `i`
  - `address(a)` is the address of the first element of `a`

Translation to IR instructions

```
x = a[i][j+2];   /* int a[100][20]; */

 int t0 = j;
 int t1 = 2;
 int t2 = t0 + t1;
 int t3 = i * 20;
 int t4 = t2 + t3;
 int t5 = 4 * t4;  /* 32 bit ints */
 char *t6 = (char*)a; /* byte ptr: address of a[0][0] */
 char *t7 = t6 + t5;
 int t8 = *((int *)t7);
 x = t8;
```

Some optimizations for a[i][j]

- In loops accessing arrays, we can often do better
  - we can move "loop-invariant" parts of the address calculation outside the loop

```
for (i=0; i < 100; i++) {
  for (j=0; j < 20; j++) {
    a[i][j]++;
  }
}
```

```
for (i=0; i < 100; i++) {
  int *t0 = a[i];
  for (j=0; j < 20; j++) {
    t0[j]++;
  }
}
```

```
int *t0 = a[0];
for (i=0; i < 2000; i++) {
  t0[i]++;
}
i = 100; j = 20;
```

Optimizations: What do we want to optimize?

- Code size/number of instructions?
- Efficient/minimal memory usage?
  - Cache friendly
- Execution time?
  - Usually this is the one we go for!
- Power consumption?
  - For devices using batteries

Code Optimizer

- A code optimizer sits between the front end and the (assembly) code generator.
  - Transformations to ‘improve’ the intermediate code.
  - Can do control flow analysis.
  - Can do data flow analysis.

How to optimize?

- Use the AST (Abstract Syntax Tree) to discover opportunities through program analysis
- Safety – transformation must not change meaning
  - Must generate correct results
  - Optimizations must be conservative
    - Changing `a+b` into `b+a` !?
    - Perfectly ok for ints, not for doubles!
    - Sometimes, it can pay off to speculate (be optimistic) but then you need to recover if reality turns out to be different

Some examples of code optimizations

- Moving invariant computations out of loops (hoisting)
- Inlining small functions
- Dead store elimination
- Common subexpression elimination
- Constant propagation
- Replace some multiplications by shifts and additions
  - E.g. 7*x equals 4*x + 2*x + x equals 2*(2*x + x) + x
  - Or 7*x equals 8*x – x (beware of overflow!)
- Register allocation
- Unroll loops
- Split loops
- Reorder instructions
- Tail recursion removal
- Many others…

Code Optimizations

- Code hoisting/motion
  - Moves invariant computations outside loops
  - Saves recomputation time

```
for (i=0; i<1000; i++) {
  a[i] = 2*pi*b[i];
}
```

```
t1 = 2*pi;
for (i=0; i<1000; i++) {
  a[i] = t1*b[i];
}
```

Code hoisting/Motion

```
for (int i = 0; i < n; i++) {
  x = y + z;
  a[i] = 6*i + x*x;
}
```

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; i++) {
  a[i] = 6*i + t1;
}
```

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

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

where the loop code does not change the limit variable.

Code Optimizations

- Inlining small (non recursive) functions
  - Inserting the body of a function instead of calling it, saves the calling overhead and enables further optimizations.
  - Inlining large functions will make the executable too large.
  - Requires complex renaming/substitution of variables/parameters!

```
a = square(b+1);
```

```
int square(int x) {
    return x*x
}
```

```
a = (b+1)*(b+1);
```

Code Optimizations

- Dead store elimination
  - If the compiler detects variables that do not influence the final outcome, it may safely ignore them.

```
t0=time();
for (i=0; i<1000; i++) {
  x = f();
}
t1=time();
printf("%f sec.\n", t1-t0);
/* 0 sec. !? */
```

```
t0=time();
for (i=0; i<1000; i++) {
  x = f();
}
t1=time();
printf("%f sec.\n", t1-t0);
printf("x=%d\n", x);
/* 0.1 sec. !? */
```

```
t0=time(); s = 0;
for (i=0; i<1000; i++) {
  x = f();
  s = s + x;
}
t1=time();
printf("%f sec.\n", t1-t0);
printf("x=%d, s=%d\n", x, s);
/* 100 sec. */
```

```
t0=time(); /* really smart compiler */
x = f();
s = 1000*x;
i = 1000;
t1=time();
printf("%f sec.\n", t1-t0);
printf("x=%d, s=%d\n", x, s);
/* 0.1 sec. */
```

Code Optimizations

- Loop unrolling
  - The loop termination check costs CPU time.
  - Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.

```
for (i=0; i<3; i++) {
  a[i]=b[i];
}
```

```
a[0] = b[0];
a[1] = b[1];
a[2] = b[2];
i=3;
```

```
for (i=0; i<8000; i++) {
  a[i]=b[i];
}
```

```
for (i=0; i<8000; i+=8){
  a[i]=b[i];
  a[i+1]=b[i+1];
  ..
  a[i+7]=b[i+7];
}
```

Code Optimizations

- Local optimizations – within basic blocks
  - Algebraic simplifications
  - Constant folding
  - Reordering instructions
  - Common subexpression elimination (i.e., redundancy elimination)
  - Dead store elimination
  - etc.,

Copy Propagation (IR quadruples)

- Given an assignment **`x = y`**, replace later uses of **`x`** with uses of **`y`**, provided there are no intervening assignments to **`x`** or **`y`**.

- <u>Local</u> copy propagation
  - Performed within basic blocks (BB)
  - Only for direct assignments and **`lhs=var op var`**
    - **`a=b`**
    - **`a=b+c  (but not a=(b*c)+e)`**
  - Algorithm sketch:
    - traverse BB from top to bottom
    - maintain table of copies encountered so far
    - modify applicable instructions on-the-fly

Local Copy Propagation

Example: Local copy propagation on basic block:

```
b = a;
c = b + 1;
d = b;
b = d + c;
b = d;
```

step	instruction	generated instr.	table contents
1	`b = a`	`b = a`	`{(b,a)}`
2	`c = b + 1`	`c = a + 1`	`{(b,a)}`
3	`d = b`	`d = a`	`{(b,a), (d,a)}`
4	`b = d + c`	`b = a + c`	`{(d,a)}`
5	`b = d`	`b = a`	`{(d,a), (b,a)}`

Note: if there was a definition of 'a' between 3 and 4, then we would have to remove (b,a) and (d,a) from the table. As a result, we wouldn't be able to perform local copy propagation at instructions 4 and 5.

Local Copy Propagation

Algorithm sketch for a basic block containing instructions $i_1, i_2, \ldots, i_n$

```
r = replace(opd, instr)
      if you find (opd, x) in table
           replace the use of opd in instr with x
           return x
      return opd
```

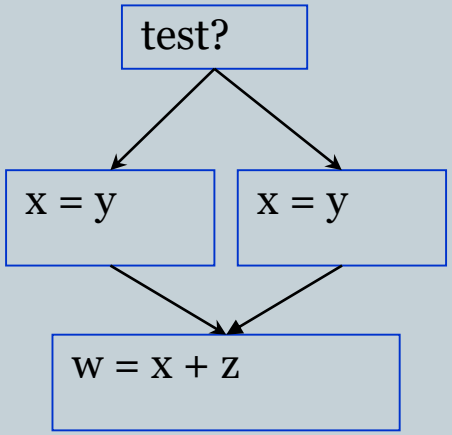
```
b = a;
c = b + 1;
d = b;
b = d + c;
b = d;
```

```
for instr = i1  to  in
      if  instr is of the form 'res = opd1  op  opd2'
           dontcare = replace(opd1, instr);
           dontcare = replace(opd2, instr);
           if the table contains any pairs involving res, remove them
      if instr is of the form 'res = var'
           r = replace(var, instr);   /* replaces var */
          if the table contains any pairs involving res, remove them
           insert (res, r) in the table
      emit(instr)
endfor
```

Copy Propagation

- Copy propagation may generate code that does not need to be evaluated any longer.
  - This will be handled by optimizations that perform redundancy elimination.

- Many implementations of global copy propagation will not detect the opportunity to replace x with y in the last basic block below:



Constant Propagation

- Given an assignment **`x = c`**, where **`c`** is a constant, replace later uses of **`x`** with uses of **`c`**, provided there are no intervening assignments to **`x`**.
  - Similar to copy propagation
  - Extra feature: It can analyze constant-value expressions and conditionals to determine whether a branch should be executed or not.

```
x=21;
y=2*x;
if (x < y) {
  printf("21<42\n");
}
```

```
x=21;
y=42;
printf("21<42\n");
```

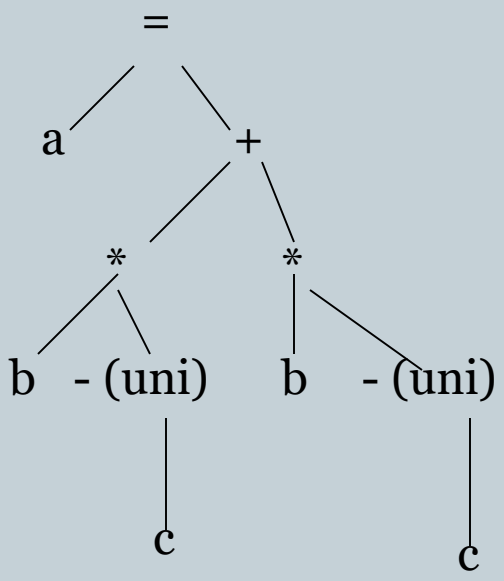
Redundancy Elimination

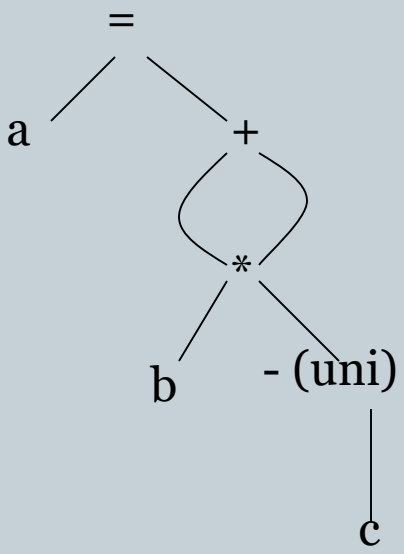
- Several optimizations deal with locating and appropriately eliminating redundant calculations.

- They include
  - common subexpression elimination
  - loop-invariant code motion/code hoisting
  - Dead code elimination

Common subexpression Eliminiation

a = b *(-c) + b*(-c)





Code Optimizations

- Common sub-expression elimination
  - Consider:

```
x = a*log(y) + log(y)*log(y);
```

  - The compiler introduces the temporary variable t:

```
int t = log(y);
x = a*t + t*t;
```

  - Saves two function calls, by elimination of the common sub-expression `log(y)`.
    - Even nicer is: `x = (a + t)*t;`

Common Subexpression Elimination

- <u>Local</u> common subexpression elimination
  - Performed within basic blocks
  - Algorithm sketch:
    - traverse BB from top to bottom
    - maintain table of expressions evaluated so far
      - Remove an expression from the table if any operand is changed
      - modify applicable instructions on-the-fly
      - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.

```
x = a + b
...
y = a + b
```

→

```
t = a + b
x = t
...
y = t
```

Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if (n * m) goto L
```

```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /* commutative */
a = j + a /* beware! */
k = t2
j = t3
a = -b
if (t2) goto L
```

table contains 4-tuples: (opd1, opr, opd2, tmp)

Local Analysis: Dead Store Elimination

- Some assignments to variables are redundant
  - a=1; followed by a=42; clearly we can omit a=1.
  - Other optimizations introduced unnecessary tmp variables
- (Local) Dead store elimination removes these redundant assignments.
- Technique: store for each variable the location of its most recent definition. At the time this location needs to be updated, determine whether the variable has been used between the previous definition and the current update.

Local Analysis: Dead Store Elimination

```
a = y + 2;
z = x + w;
x = a;
z = b + c;
b = a;
```

1. `a = y + 2;` `(a,1)`
2. ~~`z = x + w;`~~ `(a,1),(z,2)`
3. `x = a;` `(a,1),(z,2),(x,3)`
4. `z = b + c;` `(a,1),(x,3),(z,4)`

   `z` is redefined at 4, and was never used between 2 and 4;

   so **`2.z=x+w`** is "dead code"

5. `b = a;` `(a,1),(x,3),(z,4),(b,5)`

Lesson for today! And tomorrow.....

Programmers should only focus on optimizations not provided by the compiler such as choosing a faster and/or less memory intensive algorithm.