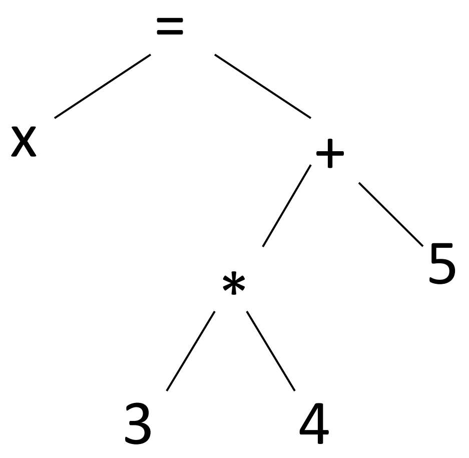
Simple Optimizations

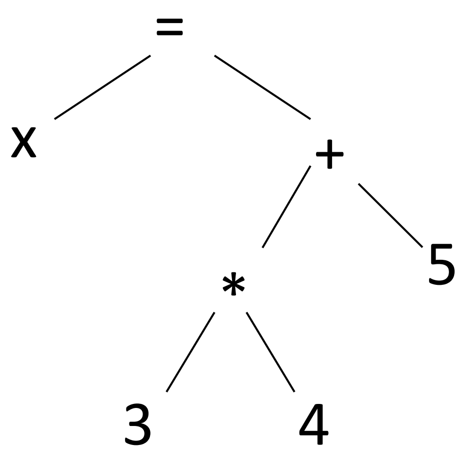
In this section we will discuss a number of *ad hoc* optimizations that can be applied to various stages of the compiler. None of these represent huge savings and they slow down the compilation time, but they will speed up your compiled code. Most programmers are willing to trade slower compilation (within reason) for faster execution: during the lifecycle of a significant program you compile infrequently and execute often.

Constant folding

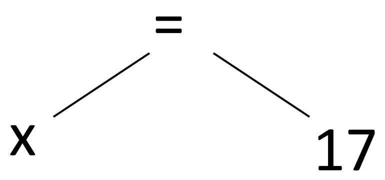
This is something you would apply to the intermediate representation of a program, such as our parse tree.

Suppose you have an expression x=3*4+5, which parses to

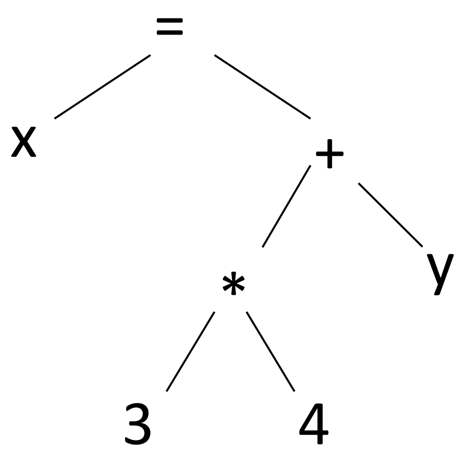




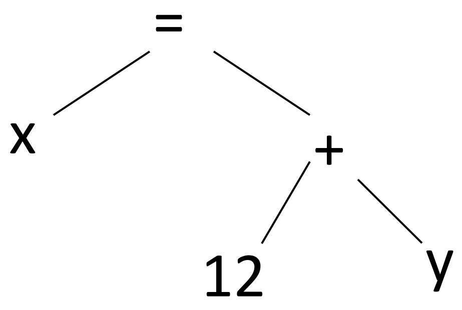
Obviously, this could, and should, be replaced by the tree



Similarly, the code x=3*4+y has parse tree



This could be replaced by



This is called *constant folding*.

There are many variations on constant folding.

Q: What is an algorithm for what we have discussed so far?

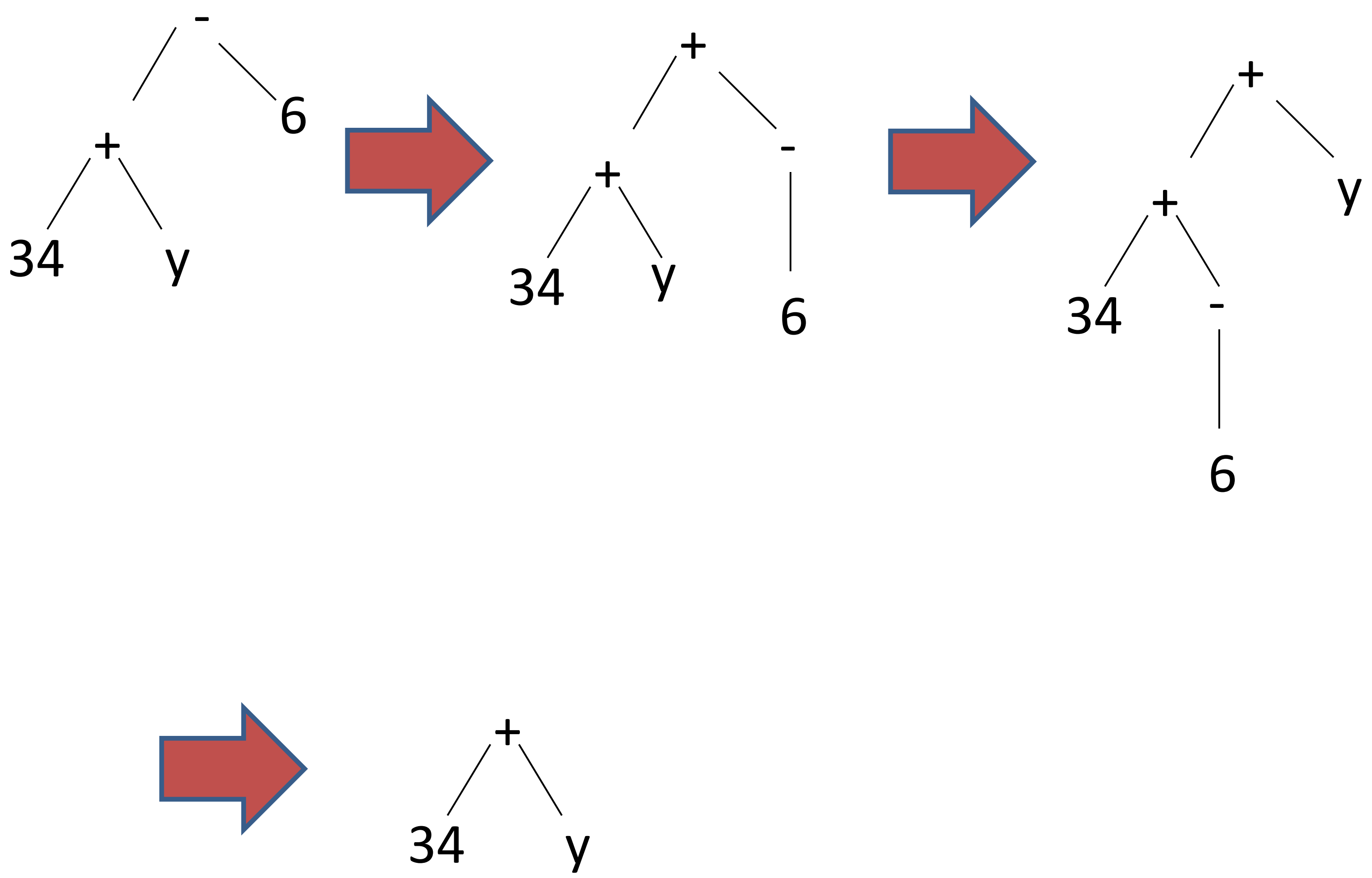
A: Make a bottom-up pass through the tree. At each node, if the node represents an arithmetic operator and both children of the node are values known at compile time (e.g., numbers or known constants) then replace the node with a constant that is the result of the expression.

Now, what about 34+y-6? We would like to replace this by 28+y

There are two issues here:

a) Addition is commutative (a+b=b+a) and subtraction is not. It is easier to rearrange an addition tree than a subtraction tree.
b) Since arithmetic operations are left-associative, it helps to move variables as far to the right as possible.

Here is a sequence of steps for turning 34+y-6 into 28+y:



Note that sometimes expressions such as these might not be the fault of the programmer. For example, if A is a 10X15 array and the programmer refers to A[3][5], the l-value for this expression is A+3*15+5. It would be nice if our compilers translated this to A+50.

Loop Jamming

A lot of programs spend most of their time running around loops. Anything you can do to speed up loops is worthwhile.

Loop jamming refers to joining together the bodies of two loops:

```
        for (int i=0; i < 300; i++)
                total = total+A[i]
        for (int i = 0; i < 300; i++)
                B[i]=A[i]/A[i+1]
```

could become

```
        for (int i=0; i < 300; i++)
                total = total+A[i]
                B[i]=A[i]/A[i+1]
```

We can jam two loops if their guards are the same and their bodies don't interact. For example, we could not jam

```
	for (int i=0; i < 300; i++)
		total = total+A[i]
	for (int i = 0; i < 300; i++)
		B[i]=A[i]/total
```

Loop Unrolling

Loop unrolling replaces a loop with straight code:

```
for (i=0; i < 4; i++)
        A[i][j]=0
```

is replaced by

```
A[0][j]=0
A[1][j]=0
A[2][j]=0
A[3][j]=0
```

Suppose A is a 4x4 array of 4-byte integers. Then

```
        A[0][j]=0
        A[1][j]=0
        A[2][j]=0
        A[3][j]=0
```

becomes

```
        A[0*4+j]=0
        A[1*4+j]=0
        A[2*4+j]=0
        A[3*4+j]=0
```

A smart compiler might evaluate this as follows:

- put A=4*j into the accumulator; this is the address of A[0][j]
- movl $0, 0(%rax)
- movl $0, 16(%rax) #address of A[1][j]
- movl $0, 32(%rax) #address of A[2][j]
- movl $0, 48(%rax) #address of A[3][j]

This would execute much more quickly than the original for-loop.

Loop unrolling requires you to know at compile time how many times the loop will be executed. It is really only practical when the number of loop iterations is small.

Loop Invariants

CS 150 students love to write code such as

```
A = 3
B = 4
x = 1
while (x<10) {
    C=A*B
    x = x+1
}
```

Should the compiler rescue them?

A *loop invariant* is an expression that doesn't change yet is evaluated every time around a loop. We can save cycles by evaluating the expression once, prior to the loop, and just referring to its value within the loop.

Here's another example:

```
x = 0
while (x<Width*Height) {
        ...
        x = x+1
}
```

If Width and Height are both on the order of 1000 and *if they don't change in the loop body* then the expression Width*Height is evaluated 1,000,000 and has the same value each time.

How would you find loop invariants?

Here is an algorithm for finding loop invariants:

- Keep a set CHANGED, initially empty. This will contain all of the variables that will be modified in the loop.
- Make a bottom-up pass through the loop body and guard, adding to CHANGED any variable that is on the left side of an assignment statement.
- Make another bottom-up pass through the loop body and guard, marking every variable in CHANGED and every internal node with at least one marked child.
- Any unmarked node is invariant and can be evaluated once, prior to the loop.
- Every assignment statement where the right side is unmarked can also be executed once, prior to the loop.

For example:

```
A = 10
B = 17
x = 1
while (x < A*B) {
        y = A+B;
        z = A+y;
        x=x+1;
}
```

CHANGED consists of {x, y, z}. The expressions A*B and y=A+B can be done prior to the loop. A more subtle algorithm can pull out z=A+y.

Basic Blocks

A *basic block* is a portion of a program with

a) Only 1 entry point
b) Control only flows from one statement to the next. No function calls, no conditional statements or loops.

Example:

```
A = 0;
x = 1
while (x< 100) {
    B = A+1;
    y=B+1;
    if (y < x) {
        B=B+1;
        C=A+1;
    }
    x=y;
}
```

The basic blocks are inside the rectangles.

Here is an algorithm by Jean-Paul Tremblay and Paul Sorenson for eliminating redundant expressions:

Divide the code into basic blocks and apply the following to each block.

First, assign two numbers to each node of the parse tree:

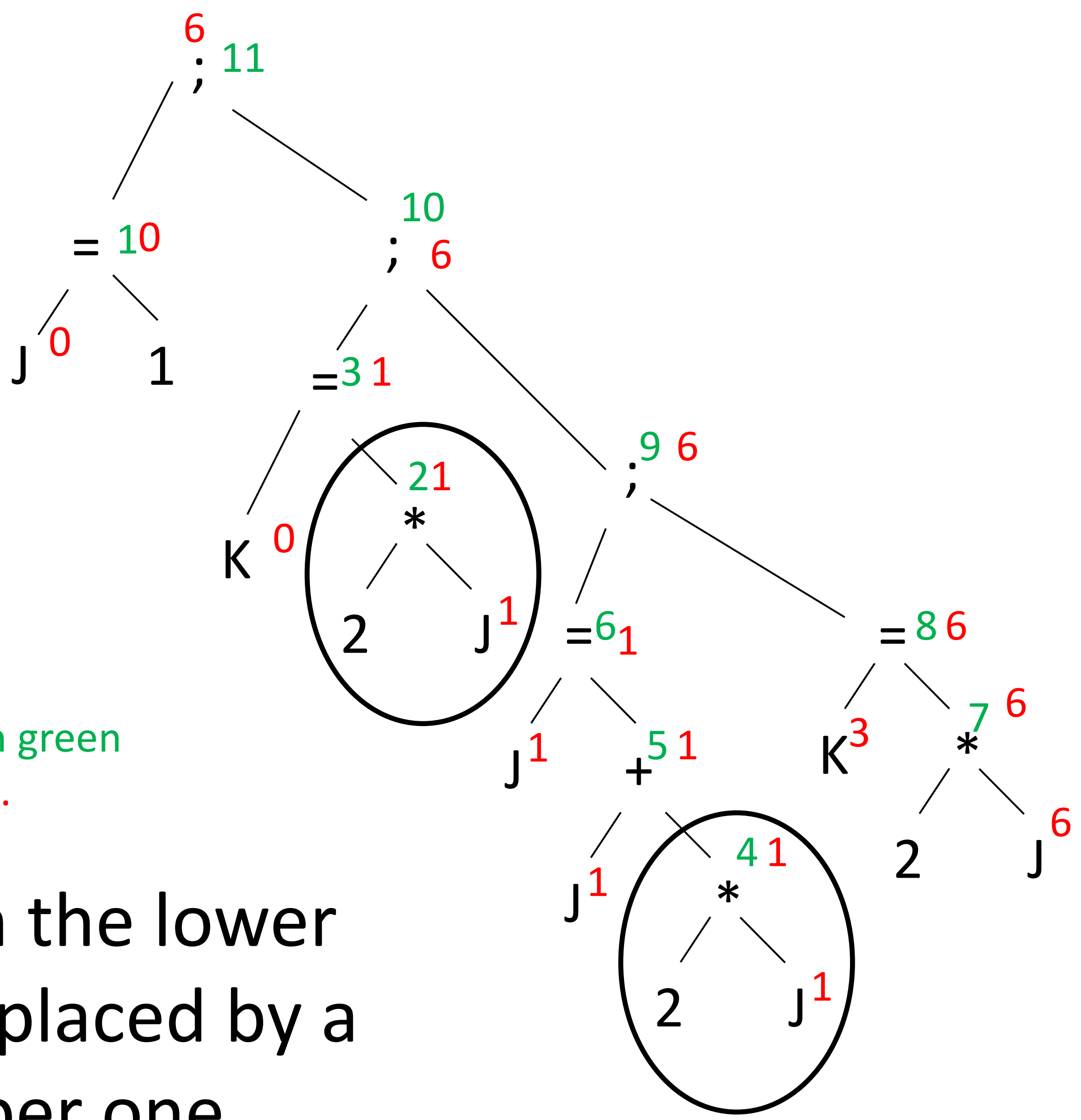
a) The *sequence number* of an internal node is its index (starting with 1) in a post-order traversal of the tree, not including the leaves.
b) The index number of a variable at any point is the sequence number of the node that last assigned a value to the variable, or 0 if the node is not assigned to in this block. The index number of an array is the sequence number of the last assignment to any element of the array.

c) The index number of an internal node is the largest index of its descendants.

THEN
If two subtrees are identical and have the same index number at their roots, the second can be replaced by a reference to the first.

Example:

```
J=1;
K=2*J;
J=J+2*J;
K=2*J;
```



Sequence numbers are in green

Index numbers are in red.

The subtree in the lower oval can be replaced by a link to the upper one.

Strength Reduction

Strength Reduction replaces "strong" expensive operations by cheaper ones. For example, multiplication or division by a power of 2 can be replaced by a shift.

Multiplication in general is a very expensive operation -- on the basic Pentium chip a multipication of two values in registers takes 11 clock cycles, while the addition of two registers takes 1. In many loops a multiplication of a constant times a loop counter can be replaced by an addition.

For example, the loop

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

can be replaced by

```
t=B
i=0
while (i<n) {
        *t=0
        t=t+4
        i=i+1
}
```

which is equivalent to

```
i=0
while (i < n) {
        *(B+4*i) = 0
        i = i+1
}
```

Similarly, the following loop, which assigns the identity matrix to nxn matrix A,

```
for(i=0; i<n; i++) {
        for(j=0; j<n;j++)
                A[i][j]=0
        A[i][i]=1
}
```

can be rewritten as

```
i=0
while (i<n) {
    j=0
    while (j<n) {
        A+4*(n*i+j) = 0
        j = j+1
    }
    A+4*(n+1)*i
    i=i+1
}
```

We can avoid the multiplications with

```
i=0
t1=4*n      # the row increment
t2= A               # starting row address
t3=A    # starting diagonal address
while (i<n) {
        j=0
        t=t2
        t2=t2+t1
        while (j<n) {
                *t = 0
                j = j+1
                t=t+4
        }
        *t3=1
        t3=t3+t1+4
        i=i+1
}
```

Peephole Optimizations

You can actually save 5% to 15% of the running time of a compiled program by looking through the generated code for stupid sequences. For example,

A. Jumps to jumps
B. Dumb sequences of operations, such as

```
        movl %eax, 8(%rbx)
        movl 8(%rbx), %eax
```

or

```
        addq $0, %rsp
```

or

or

```
        movl $1, %eax
        push %eax
        movl -8(%rbx), %eax
        addl 0(%rsp), %eax
        addq $8, %rsp
```

C. Unreachable code, such as

```
jmp .L5
movl $4, %eax
```

D. Things that can be simplified algebraically, such as

```
imul $1, %eax
```

or

```
addl $0, %eax
```