

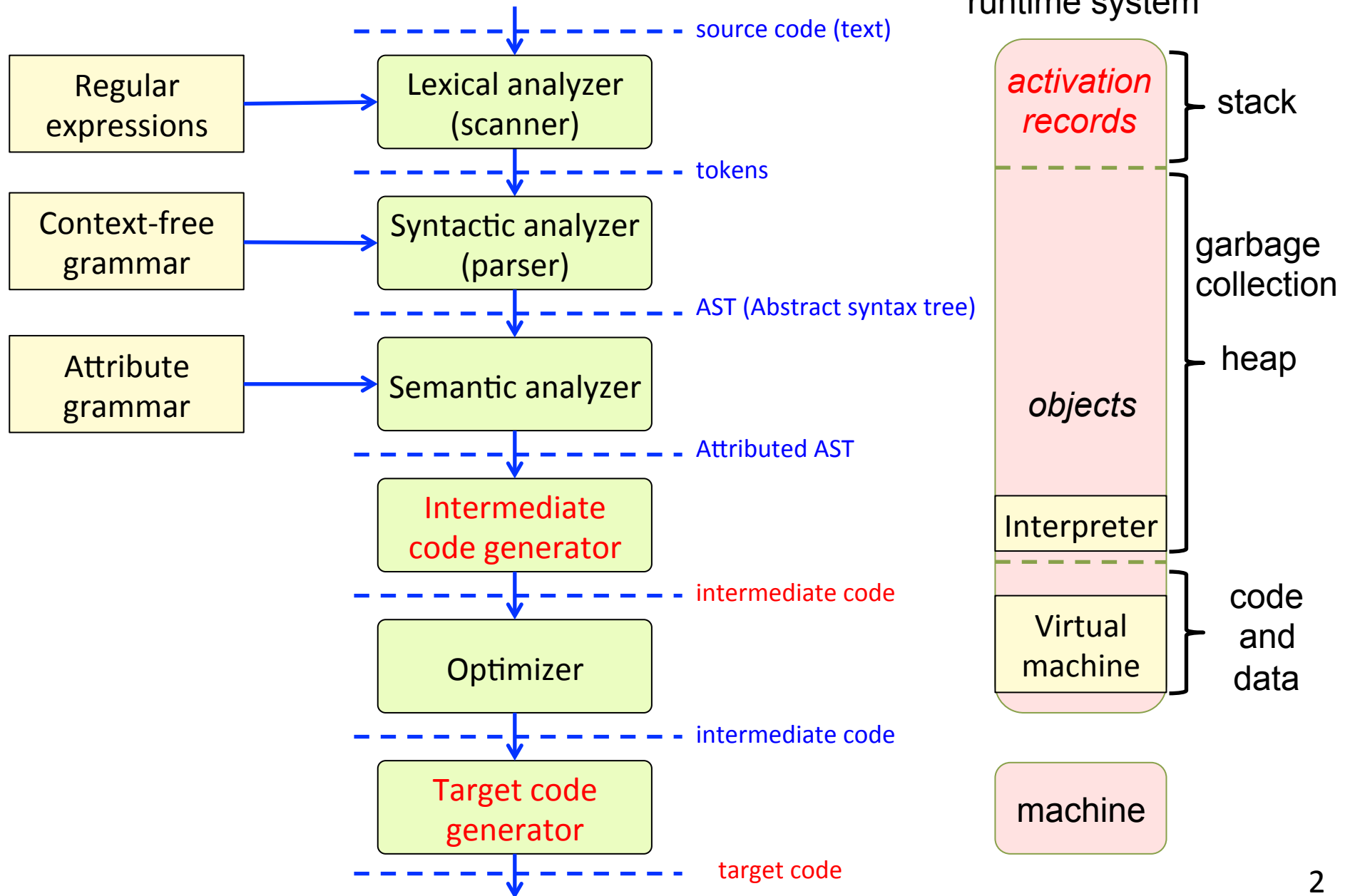
EDAN65: Compilers, Lecture 11

Code generation

Görel Hedin

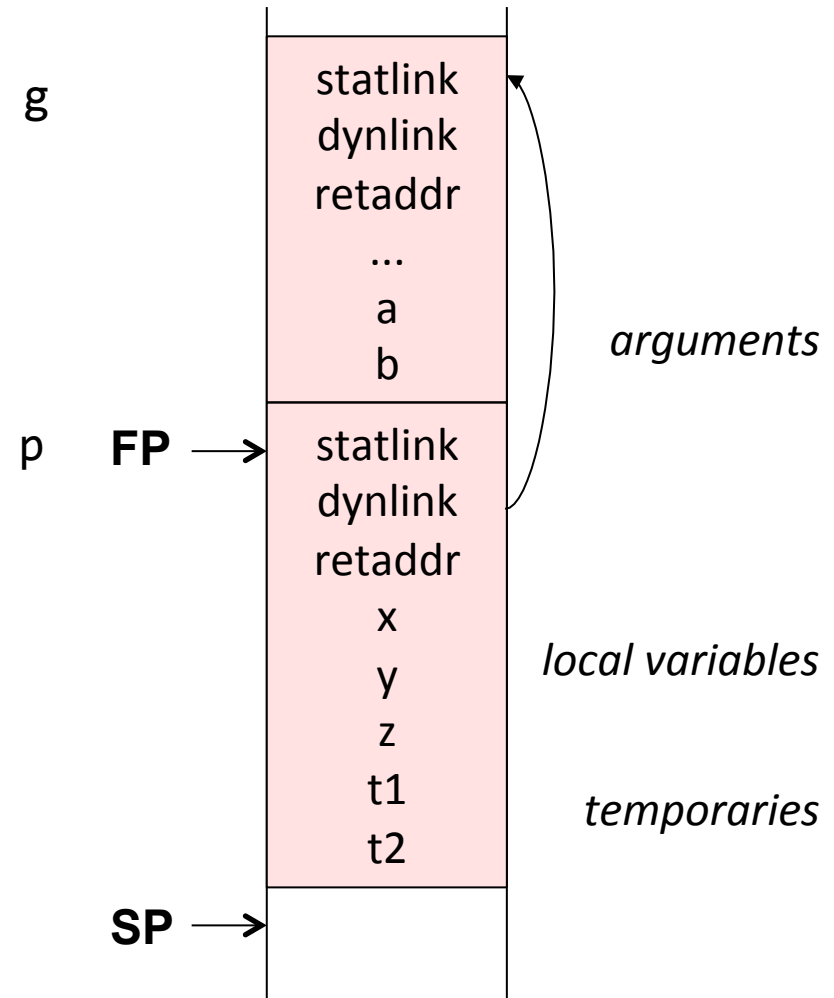
Revised: 2015-10-05

This lecture

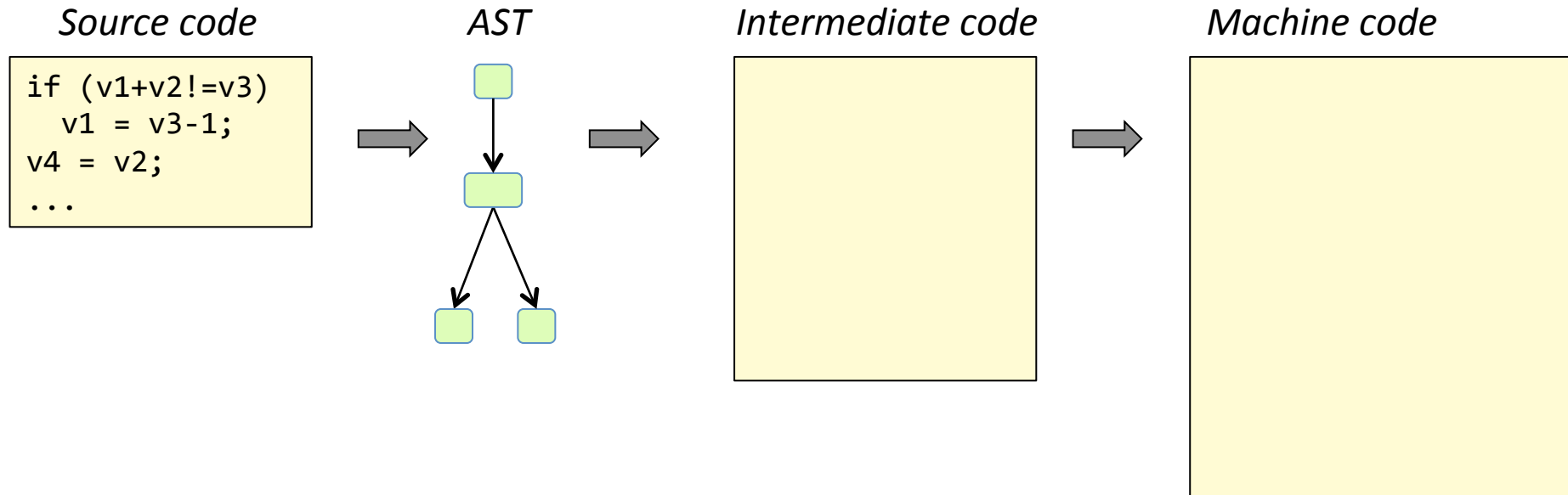


Recall: example activation layout

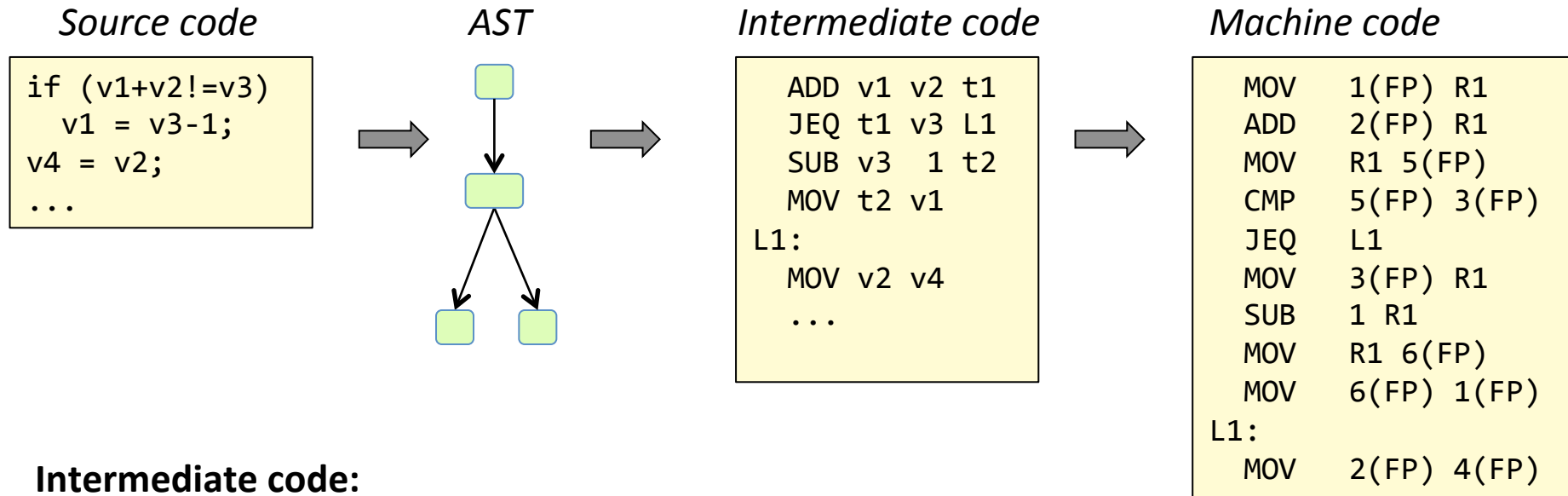
```
void g() {  
    ...  
    p(1,2);  
    ...  
}  
  
void p(int a, int b) {  
    int x = 1;  
    int y = 2;  
    int z = 3;  
    ...  
}
```



Generating code



Generating code



Intermediate code:

- Expressions are broken down to one operation per instruction, introducing temporary variables for each non-trivial expression.
- Variables have high-level symbolic names.
- Control structures are implemented using branch instructions that jump to labels.

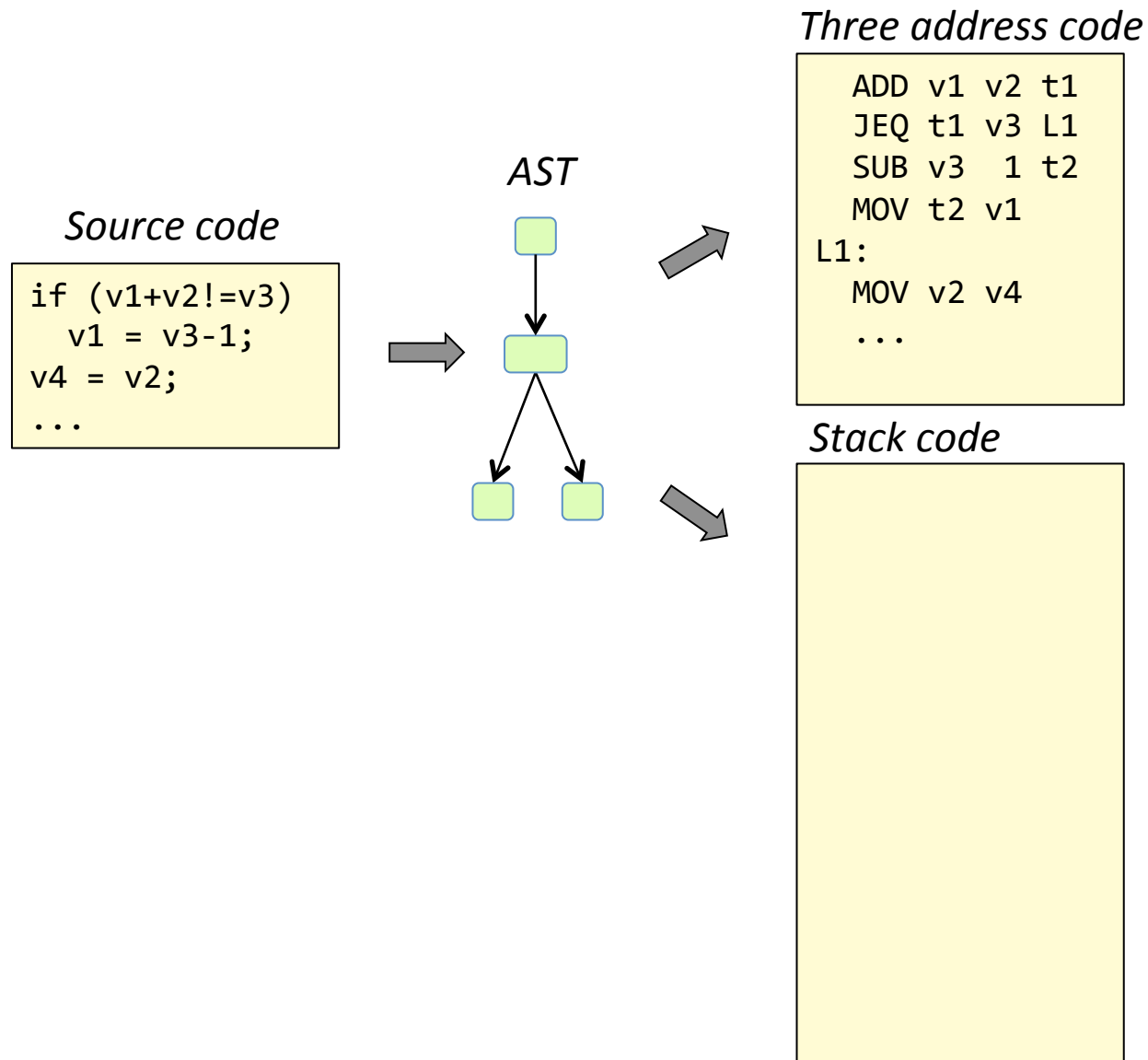
Machine code (assembly code):

- Many operations can only be done on registers.
- Values in memory need to be loaded to registers before performing the operation.
- Variable names are replaced by addresses, typically relative to the frame pointer.

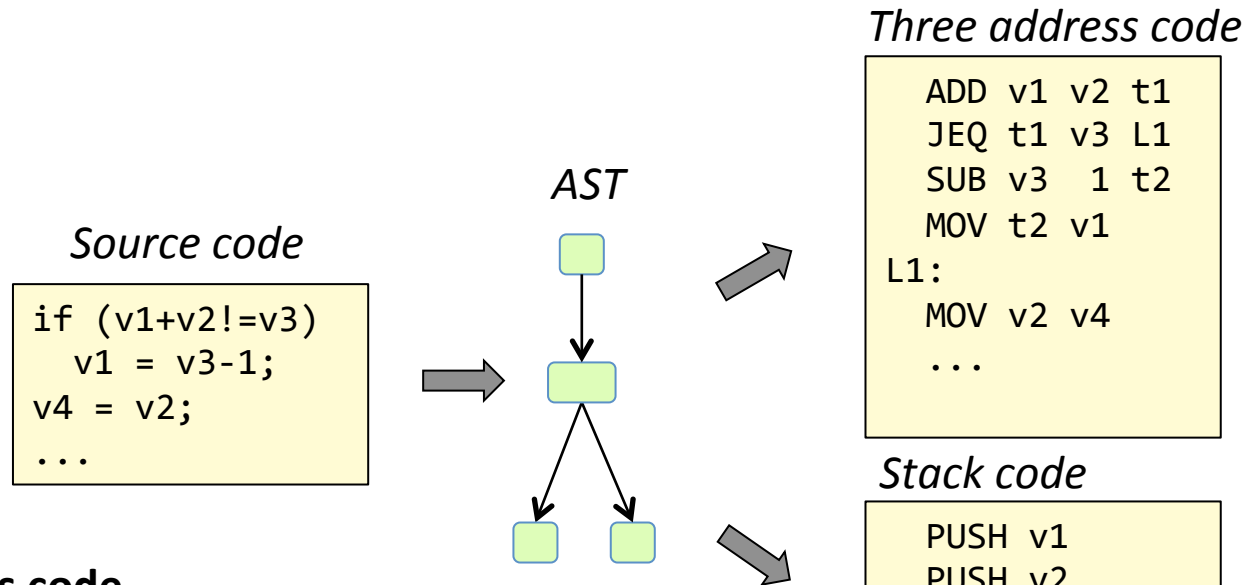
Variable addresses

v1	1(FP)
v2	2(FP)
v3	3(FP)
v4	4(FP)
t1	5(FP)
t2	6(FP)

Two kinds of intermediate code



Two kinds of intermediate code



Three address code

Each instruction typically has three operands:

op src1 src2 dest

Uses temporary variables.

Close to ordinary register-based machine.

Good for optimization.

Stack code

Uses a *value stack* instead of temporary variables.

Commonly used for interpreters and virtual machines.

Translate to three address code

Source code

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

Three address code

```
ADD b c t1  
ADD d e t2  
MUL t1 t2 t3  
MOV t3 a
```

One new temporary for each nontrivial value.

Why not try to reuse the temporaries?

And remove useless MOVs?

In principle, two temps would suffice here:

```
ADD b c t1  
ADD d e t2  
MUL t1 t2 a
```

Minimizing the number of temporaries (not meaningful).

Typically, the intermediate code is optimized at a later stage. The optimizations transform the code and introduce new temporaries. Temporaries are optimized as a final step, as part of register allocation. Trying to minimize the number of temporaries at the code generation stage is therefore meaningless.

Translate three address code to AT&T x86-64 assembly code

Source code

```
void m(int a, int b) {  
    int c, d;  
    ...  
    c = a + b  
    ...  
}
```

3 address code

```
...  
ADD a b t1  
MOV t1 c  
...
```

Variable addresses

```
a    16(%rbp)  
b    24(%rbp)  
c    -8(%rbp)  
d    -16(%rbp)  
t1   -24(%rbp)
```

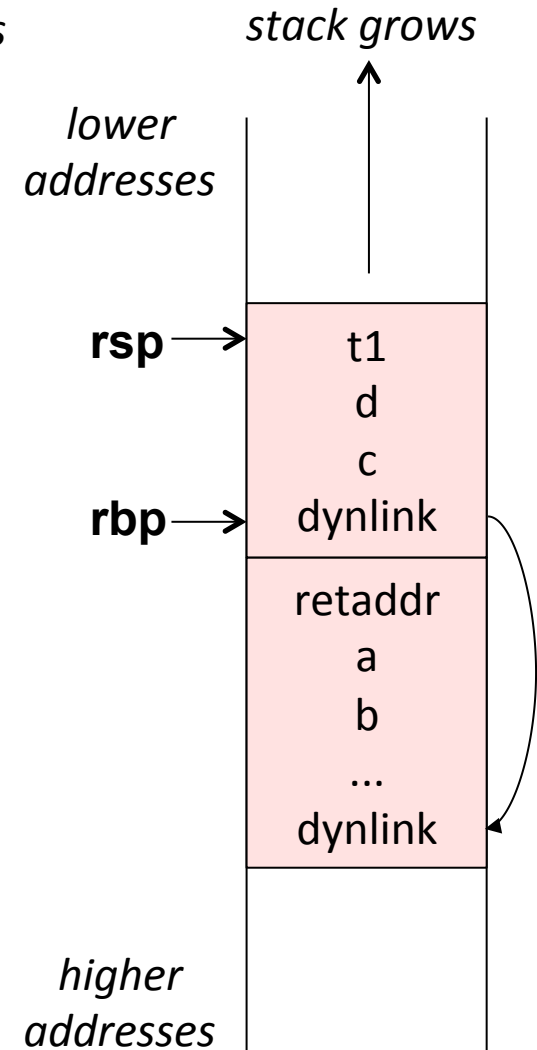
Registers and instructions

rsp: stack pointer (points to top of stack)
rbp: base pointer (frame pointer)
rip: instruction pointer (program counter)
rax, rbx: general registers

addq val, reg # reg + val -> reg (q: quad word 64 bits)

Assembly code

```
...  
subq $24, %rsp      # Make room on stack for c, d, t1  
...  
movq 16(%rbp), %rax    # a -> rax  
addq 24(%rbp), %rax    # b + rax -> rax  
movq %rax, -24(%rbp)   # rax -> t1  
movq -24(%rbp), -8(%rbp) # t1 -> c  
...
```



Translate to assembly code

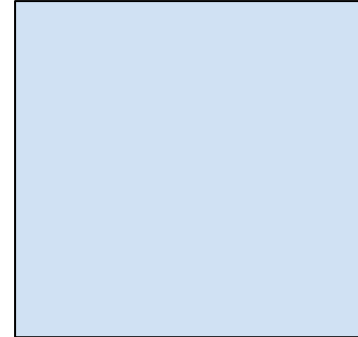
Source code

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

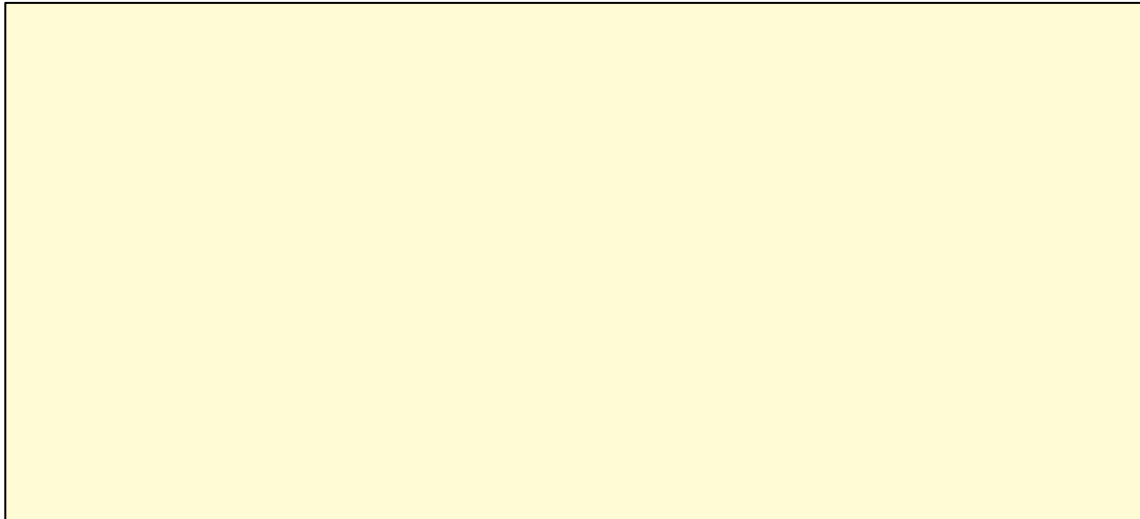
Three address code



Variable addresses



Unoptimized assembly code:



Translate to assembly code

Source code

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

Three address code

```
ADD a b t1  
ADD a c t2  
MUL t1 t2 t3  
MOV t3 d
```

Variable addresses

a	-8(%rbp)
b	-16(%rbp)
c	-24(%rbp)
d	-32(%rbp)
t1	-40(%rbp)
t2	-48(%rbp)
t3	-56(%rbp)

Unoptimized assembly code:

```
movq -8(%rbp), %rax    # a -> rax  
addq -16(%rbp), %rax   # b + rax -> rax  
movq %rax, -40(%rbp)   # rax -> t1  
movq -8(%rbp), %rax    # a -> rax  
addq -24(%rbp), %rax   # c + rax -> rax  
movq %rax, -48(%rbp)   # rax -> t2  
movq -40(%rbp), %rax   # t1 -> rax  
imulq -48(%rbp), %rax  # t2 * rax -> rax  
movq %rax, -56(%rbp)   # rax -> t3  
movq -56(%rbp), -32(%rbp) # t3 -> d
```

Can the use of registers be optimized?

Source code

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

Three address code

```
ADD a b t1  
ADD a c t2  
MUL t1 t2 t3  
MOV t3 d
```

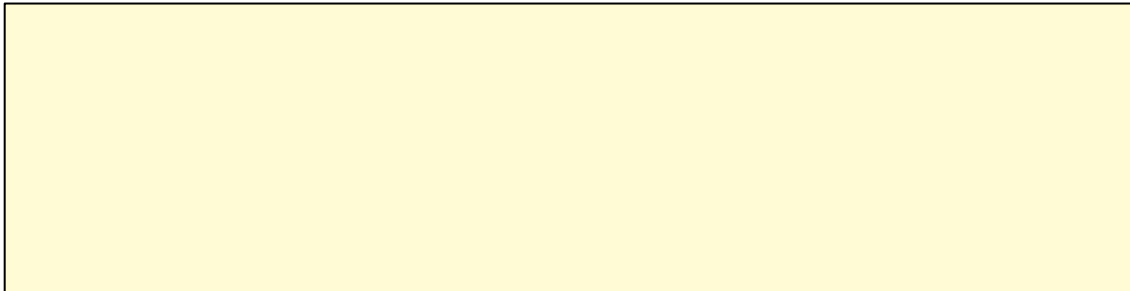
Variable addresses

a	-8(%rbp)
b	-16(%rbp)
c	-24(%rbp)
d	-32(%rbp)
t1	-40(%rbp)
t2	-48(%rbp)
t3	-56(%rbp)

Unoptimized assembly code: 11 memory accesses, 7 vars

```
movq -8(%rbp), %rax    # a -> rax  
addq -16(%rbp), %rax   # b + rax -> rax  
movq %rax, -40(%rbp)   # rax -> t1  
movq -8(%rbp), %rax    # a -> rax  
addq -24(%rbp), %rax   # c + rax -> rax  
movq %rax, -48(%rbp)   # rax -> t2  
movq -40(%rbp), %rax   # t1 -> rax  
imulq -48(%rbp), %rax  # t2 * rax -> rax  
movq %rax, -56(%rbp)   # rax -> t3  
movq -56(%rbp), -32(%rbp) # t3 -> d
```

Optimized assembly code:



Can the use of registers be optimized?

Source code

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

Three address code

```
ADD a b t1  
ADD a c t2  
MUL t1 t2 t3  
MOV t3 d
```

Variable addresses

a	-8(%rbp)
b	-16(%rbp)
c	-24(%rbp)
d	-32(%rbp)
t1	-40(%rbp)
t2	-48(%rbp)
t3	-56(%rbp)

Unoptimized assembly code: 11 memory accesses, 7 vars

```
movq -8(%rbp), %rax      # a -> rax  
addq -16(%rbp), %rax     # b + rax -> rax  
movq %rax, -40(%rbp)     # rax -> t1  
movq -8(%rbp), %rax      # a -> rax  
addq -24(%rbp), %rax     # c + rax -> rax  
movq %rax, -48(%rbp)     # rax -> t2  
movq -40(%rbp), %rax     # t1 -> rax  
imulq -48(%rbp), %rax    # t2 * rax -> rax  
movq %rax, -56(%rbp)     # rax -> t3  
movq -56(%rbp), -32(%rbp) # t3 -> d
```

Optimized assembly code: 4 memory accesses, 4 vars

```
movq -8(%rbp), %rax      # a -> rax  
movq %rax, %rbx          # rax -> rbx  
addq -16(%rbp), %rax     # b + rax -> rax  
addq -24(%rbp), %rbx     # c + rbx -> rbx  
imulq %rax, %rbx         # rax * rbx -> rbx  
movq %rbx, -32(%rbp)     # rbx -> d
```

Register allocation

Accessing memory is slow. Typical numbers:

Add instruction operating on registers: 1 ns

Access to L1 cache: 5 ns

Access to memory: 100 ns

Goal of register allocation:

Try to keep as many variables and temporaries as possible in registers, "spilling" as few of them as possible to memory.

Good algorithms exist, based on graph coloring.

See course on Optimizing Compilers, EDA230.

In assignment 6, we will **not** optimize the code.

Control structures

Source code

```
void m() {  
    int x, s;  
    ...  
    while (x > 1) {  
        s = s + x;  
    }  
    ...  
}
```

3 address code

```
m:  
    ...  
m_1:  
    JLE x 1 m_2      # if x <= 1 jump to m_2  
    ADD s x t1       # s + x -> t1  
    MOV t1 s         # t1 -> s  
    JMP m_1          # jump to label m_1  
m_2:  
    ...
```

Note:

Flip the condition to get simpler code

All labels must be unique in the program

Control structures

Source code

```
void m() {  
    int x, s;  
    ...  
    while (x > 1) {  
        s = s + x;  
    }  
    ...  
}
```

Variable addresses

x	-8(%rbp)
s	-16(%rbp)
t1	-24(%rbp)

3 address code

```
m:  
    ...  
m_1:  
    JLE x 1 m_2  
  
    ADD s x t1  
  
    MOV t1 s  
    JMP m_1  
m_2:  
    ...
```

x86 assembly code

```
m:  
    ...  
m_1:  
    cmpq -8(%rbp), $1      # Compare x and 1  
    jle m_2                # Jump if previous cmp was less-or-equal  
    movq -16(%rbp), %rax   # s -> rax  
    addq -8(%rbp), %rax    # x + rax -> rax  
    movq %rax, -24(%rbp)   # rax -> t1  
    movq -24(%rbp), -16(%rbp) # t1 -> s  
    jmp m_1  
m_2:  
    ...
```

New instructions used

```
cmpq a, b: compares a and b, sets condition codes  
jle lbl:   jumps to label lbl if le condition code is set  
jmp lbl:   jumps to label lbl
```


Method call

Source code

```
int x, y;  
...  
y = p(x+1, 2);  
...
```

3 address code



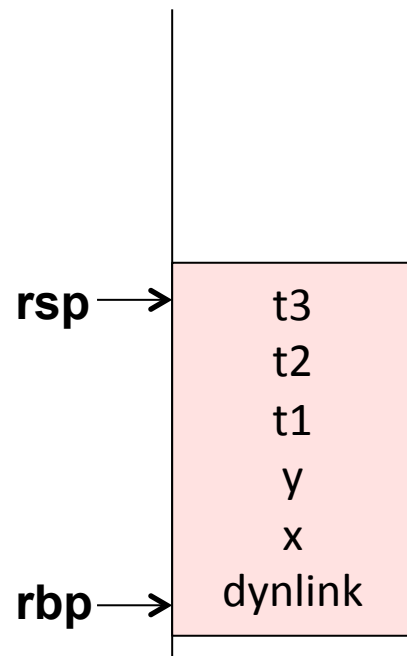
Method call

Source code

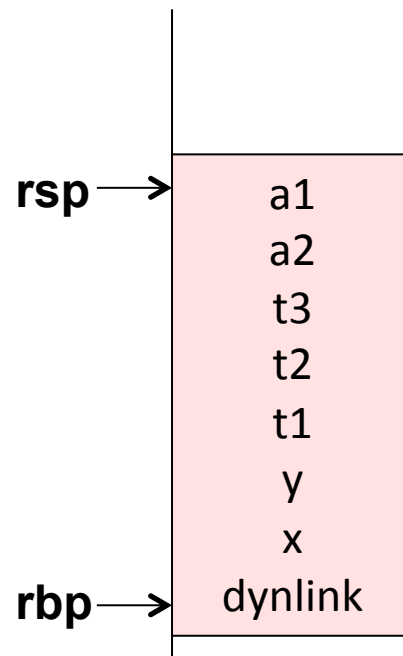
```
int x, y;  
...  
y = p(x+1, 2);  
...
```

3 address code

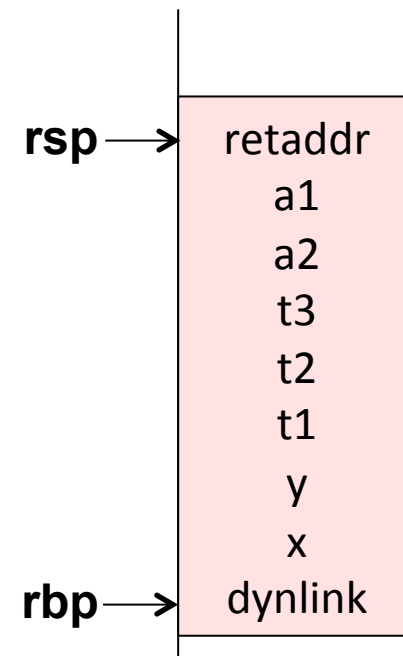
```
...  
ADD x 1 t1 # Eval arg 1  
MOV 2 t2   # Eval arg 2  
MOV t2 a2  # Pass arg 2  
MOV t1 a1  # Pass arg 1  
CALL p     # Do the call  
MOV rv t3  # Save the return value  
MOV t3 y  
...
```



Original situation



Passing the args



Calling p

Method call

Source code

```
int x, y;  
...  
y = p(x+1, 2);  
...
```

Calling conventions:

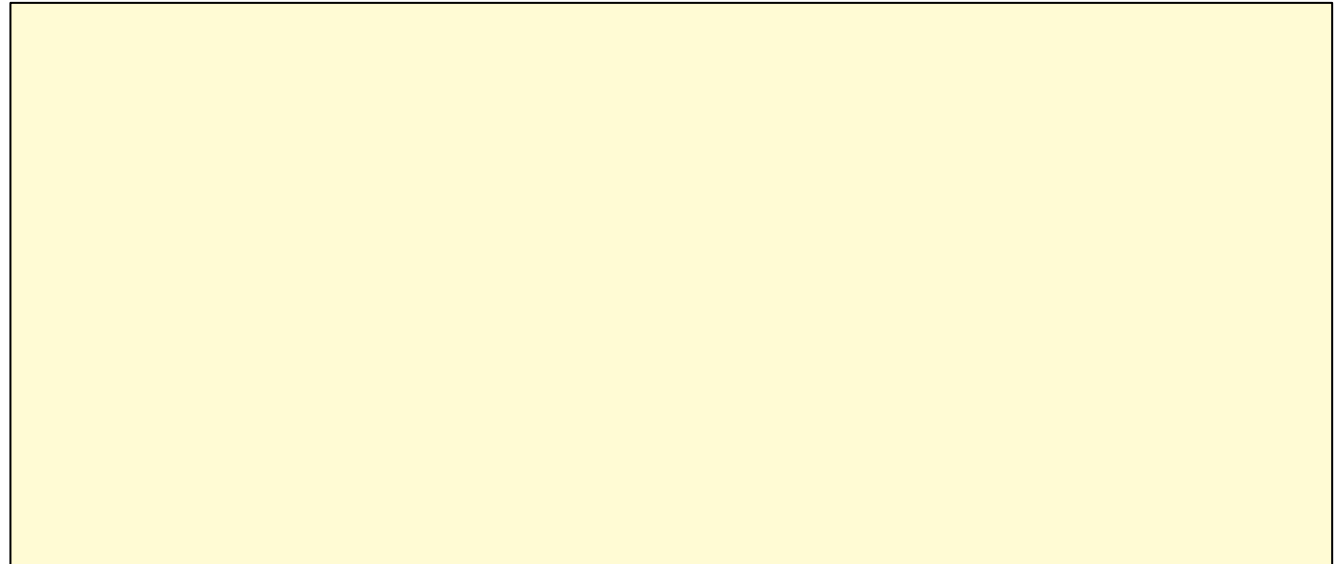
Args pushed in reverse order on stack

Return value stored in RAX register

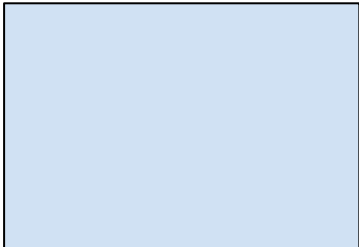
3 address code

```
...  
ADD x 1 t1  
  
MOV 2 t2  
MOV t2 a2  
MOV t1 a1  
CALL p  
  
MOV rv t3  
MOV t3 y  
...
```

Assembly code



Variable allocation



New instructions used

pushq v: pushes a value to the stack (moves rsp)
call m: pushes the return address and jumps to m

Method call

Source code

```
int x, y;  
...  
y = p(x+1, 2);  
...
```

Calling conventions:

Args pushed in reverse order on stack

Return value stored in RAX register

3 address code

```
...  
ADD x 1 t1  
  
MOV 2 t2  
MOV t2 a2  
MOV t1 a1  
CALL p  
  
MOV rv t3  
MOV t3 y  
...
```

Assembly code

```
...  
movq -8(%rbp), %rax      # x -> rax  
addq $1, %rax            # 1 + rax -> rax  
movq %rax, -24(%rbp)     # rax -> t1  
movq $2, -32(%rbp)       # 2 -> t2  
pushq -32(%rbp)          # push arg 2  
pushq -24(%rbp)          # push arg 1  
call p2                  # call p  
addq $16, %rsp           # pop arguments  
movq %rax, -40(%rbp)     # rax -> t3 (save return val)  
movq -40(%rbp), -16(%rbp) # t3 -> y  
...
```

Variable allocation

x	-8(%rbp)
y	-16(%rbp)
t1	-24(%rbp)
t2	-32(%rbp)
t3	-40(%rbp)

New instructions used

pushq v: pushes a value to the stack (moves rsp)
call m: pushes the return address and jumps to m

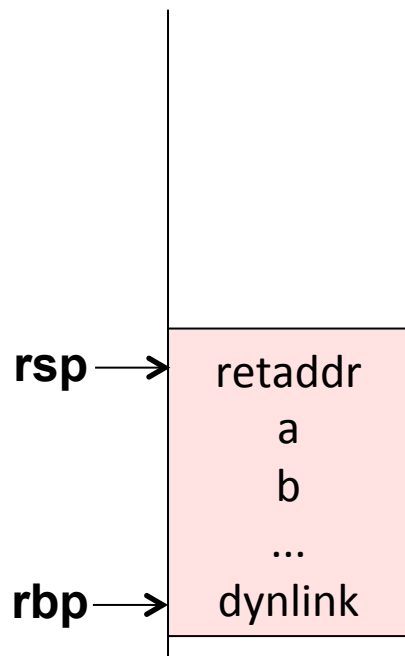
Method activation and return

Source code

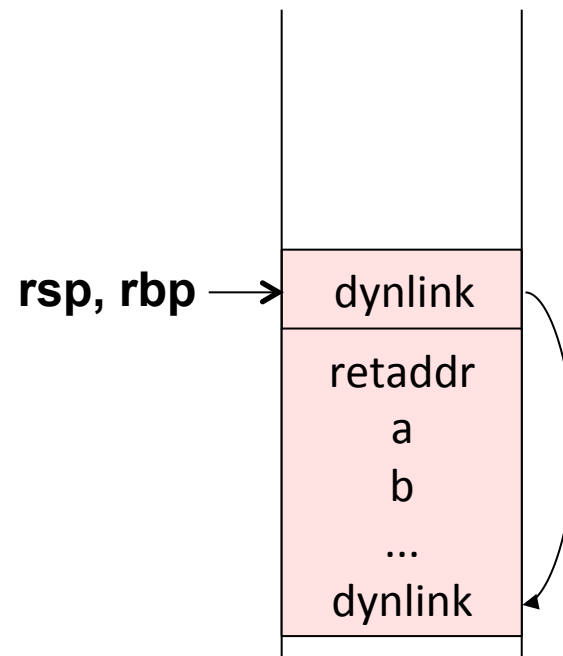
```
int p(int a, int b) {  
    int r;  
    ...  
    return r+1  
}
```

3 address code

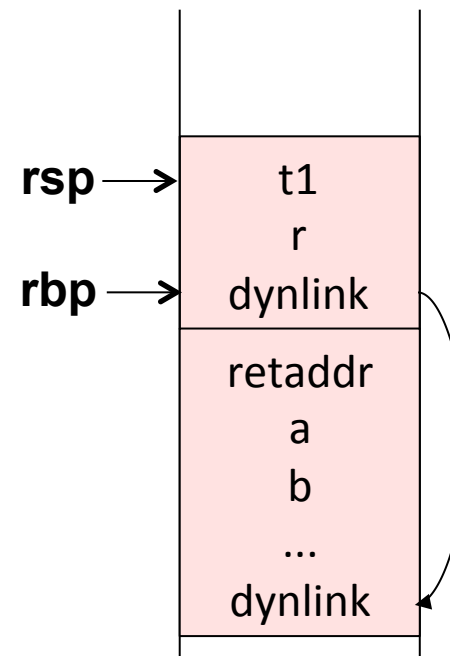
```
p:  
    START      # Start of activation  
    SPACE 2    # Make space for 2 vars and temps  
    ...  
    ADD r 1 t1  # Compute the value to return  
    MOV t1 rv   # Store the return value  
    RETURN     # Return to the caller
```



At label p



After START



After allocating space

Method activation and return

Source code

```
int p(int a, int b) {  
    int r;  
    ...  
    return r+1  
}
```

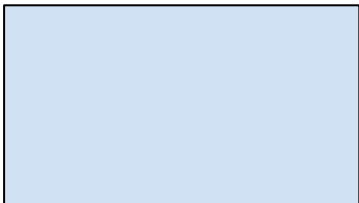
3 address code

```
p:  
    START  
  
    SPACE 2  
    ...  
    ADD r 1 t1  
  
    MOV t1 rv  
    RETURN
```

Assembly code



Variable addresses



New instructions used

```
popq r: pops top of stack, and stores it to reg r  
ret: pops the return address and jumps to it
```

Method activation and return

Source code

```
int p(int a, int b) {  
    int r;  
    ...  
    return r+1  
}
```

3 address code

```
p:  
    START  
  
    SPACE 2  
    ...  
    ADD r 1 t1  
  
    MOV t1 rv  
    RETURN
```

Assembly code

```
p:                # Label for p  
    pushq %rbp     # Push the dynamic link  
    movq %rsp, %rbp # Set the new frame pointer  
    subq $16, %rsp  # Make space for 2 vars and temps  
    ...  
    movq -8(%rbp), %rax # r -> rax  
    addq $1, %rax       # 1 + rax -> rax  
    movq %rax, -16(%rbp) # rax -> t1  
    movq -16(%rbp), %rax # t1 -> rax  
    movq %rbp, %rsp     # move back the stack pointer  
    popq %rbp          # restore the frame pointer  
    ret
```

Variable addresses

a	16(%rbp)
b	24(%rbp)
r	-8(%rbp)
t1	-16(%rbp)

New instructions used

popq r: pops top of stack, and stores it to reg r
ret: pops the return address and jumps to it

Summary questions

- What is the difference between intermediate code and assembly code?
- Mention two kinds of typical intermediate code. When are they useful?
- Why is it not meaningful to minimize the number of temporaries in intermediate code?
- What is register allocation?
- Given a source program, sketch intermediate three address code.
- Given a source program, sketch x86 assembly code.