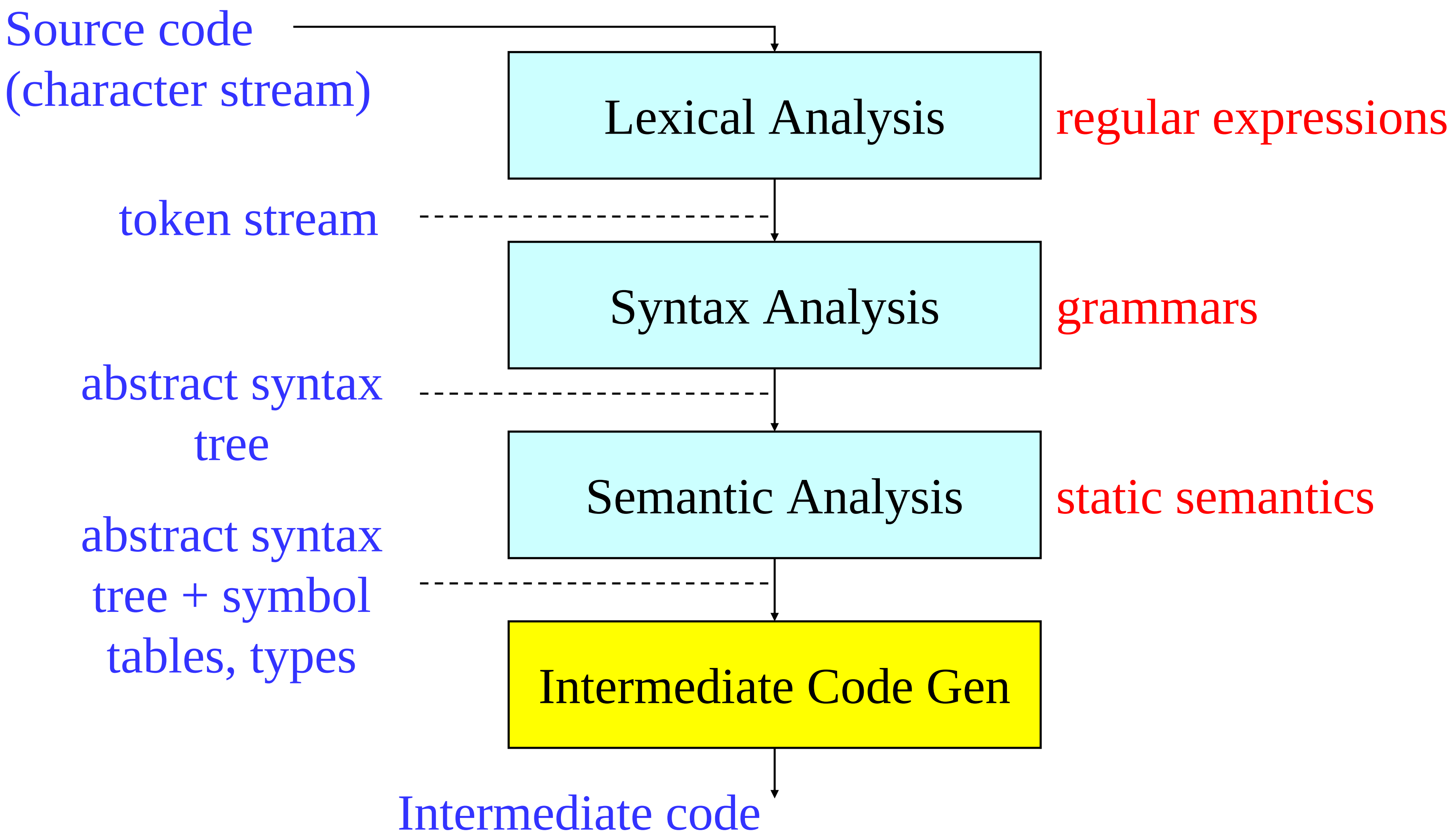
Intermediate Representation I High-Level to Low-Level IR Translation

EECS 483 – Lecture 17
University of Michigan
Monday, November 6, 2006

Where We Are...


Source code (character stream)

Lexical Analysis

regular expressions

token stream

Syntax Analysis

grammars

abstract syntax tree

Semantic Analysis

static semantics

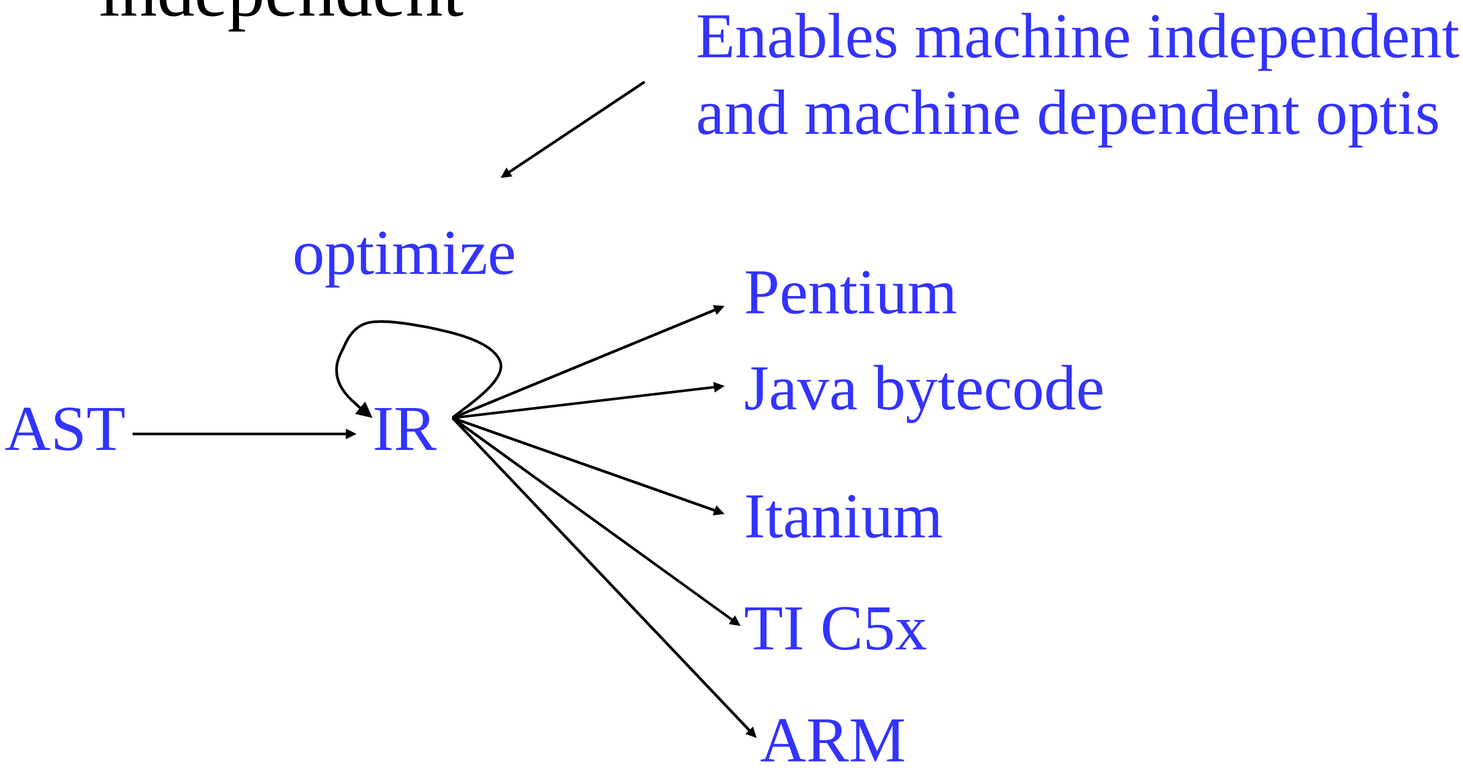
abstract syntax tree + symbol tables, types

Intermediate Code Gen

Intermediate code

Intermediate Representation (aka IR)

- The compilers internal representation
  - Is language-independent and machine-independent



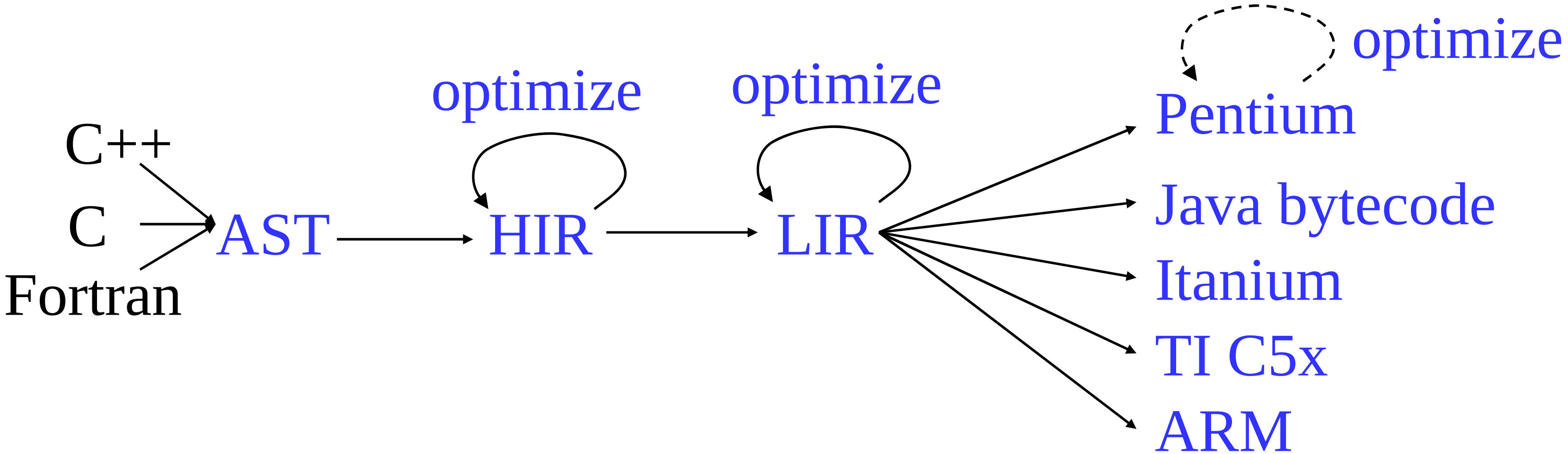
Enables machine independent and machine dependent optis

What Makes a Good IR?

- Captures high-level language constructs
  - Easy to translate from AST
  - Supports high-level optimizations
- Captures low-level machine features
  - Easy to translate to assembly
  - Supports machine-dependent optimizations
- Narrow interface: small number of node types (instructions)
  - Easy to optimize
  - Easy to retarget

Multiple IRs

- Most compilers use 2 IRs:
  - High-level IR (HIR): Language independent but closer to the language
  - Low-level IR (LIR): Machine independent but closer to the machine
  - A significant part of the compiler is both language and machine independent!



High-Level IR

- HIR is essentially the AST
  - Must be expressive for all input languages
- Preserves high-level language constructs
  - Structured control flow: if, while, for, switch
  - Variables, expressions, statements, functions
- Allows high-level optimizations based on properties of source language
  - Function inlining, memory dependence analysis, loop transformations

Low-Level IR

- A set of instructions which emulates an abstract machine (typically RISC)
- Has low-level constructs
  - Unstructured jumps, registers, memory locations
- Types of instructions
  - Arithmetic/logic (a = b OP c), unary operations, data movement (move, load, store), function call/return, branches

Alternatives for LIR

- 3 general alternatives
  - Three-address code or quadruples
    - a = b OP c
    - Advantage: Makes compiler analysis/opti easier
  - Tree representation
    - Was popular for CISC architectures
    - Advantage: Easier to generate machine code
  - Stack machine
    - Like Java bytecode
    - Advantage: Easier to generate from AST

Three-Address Code

- a = b OP c
  - » Originally, because instruction had at most 3 addresses or operands
    - This is not enforced today, ie MAC: a = b * c + d
  - » May have fewer operands
- Also called quadruples: (a,b,c,OP)
- Example

Compiler-generated temporary variable

```
a = (b+c) * (-e)        t1 = b + c
                        t2 = -e
                        a = t1 * t2
```

IR Instructions

- Assignment instructions
  - a = b OP C (binary op)
    - arithmetic: ADD, SUB, MUL, DIV, MOD
    - logic: AND, OR, XOR
    - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - a = OP b (unary op)
    - arithmetic MINUS, logical NEG
  - a = b : copy instruction
  - a = [b] : load instruction
  - [a] = b : store instruction
  - a = addr b: symbolic address
- Flow of control
  - label L: label instruction
  - jump L: unconditional jump
  - cjump a L : conditional jump
- Function call
  - call f(a1, ..., an)
  - a = call f(a1, ..., an)
- IR describes the instruction set of an abstract machine

IR Operands

- The operands in 3-address code can be:
  - Program variables
  - Constants or literals
  - Temporary variables
- Temporary variables = new locations
  - Used to store intermediate values
  - Needed because 3-address code not as expressive as high-level languages

Class Problem

Convert the following code segment to assembly code

```
n = 0;
while (n < 10) {
        n = n+1;
}
```

Translating High IR to Low IR

- May have nested language constructs
  - E.g., while nested within an if statement
- Need an algorithmic way to translate
  - Strategy for each high IR construct
  - High IR construct → sequence of low IR instructions
- Solution
  - Start from the high IR (AST like) representation
  - Define translation for each node in high IR
  - Recursively translate nodes

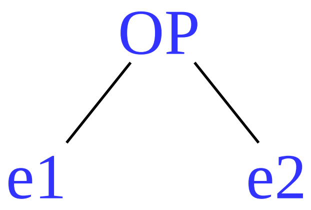
Notation

- Use the following notation:
  - [[e]] = the low IR representation of high IR construct e
- [[e]] is a sequence of low IR instructions
- If e is an expression (or statement expression), it represents a value
  - Denoted as: t = [[e]]
  - Low IR representation of e whose result value is stored in t
- For variable v: t = [[v]] is the copy instruction
  - t = v

Translating Expressions

- Binary operations: t = [[e1 OP e2]]
  - (arithmetic, logical operations and comparisons)

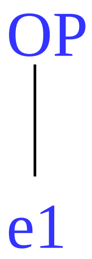
t1 = [[e1]]
t2 = [[e2]]
t1 = t1 OP t2



OP
e1
e2

- Unary operations: t = [[OP e]]

t1 = [[e1]]
t = OP t1

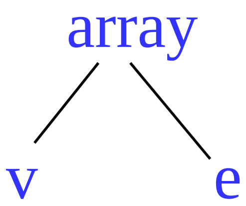


OP
e1

Translating Array Accesses

- Array access: t = [[v[e]]]
  - (type of e is array [T] and S = size of T)

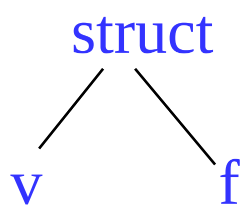
```
t1 = addr v
t2 = [[e]]
t3 = t2 * S
t4 = t1 + t3
t = [t4]    /* ie load */
```



Translating Structure Accesses

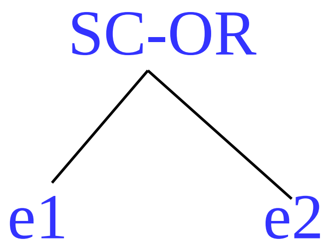
- Structure access: t = [[v.f]]
  - (v is of type T, S = offset of f in T)

```
t1 = addr v
t2 = t1 + S
t = [t2]    /* ie load */
```



Translating Short-Circuit OR

- Short-circuit OR: t = [[e1 SC-OR e2]]
  - e.g., || operator in C/C++



```
t = [[e1]]
cjump t Lend
t = [[e2]]
Lend:
```

semantics:

1. evaluate e1
2. if e1 is true, then done
3. else evaluate e2

Class Problem

- Short-circuit AND: t = [[e1 SC-AND e2]]
  - e.g., && operator in C/C++

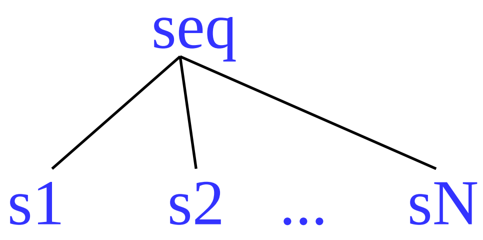
Semantics:

1. Evaluate e1
2. if e1 is true,
   then evaluate e2
3. else done

Translating Statements

- Statement sequence: [[s1; s2; ...; sN]]

[[s1]]
[[s2]]
...
[[sN]]



- IR instructions of a statement sequence = concatenation of IR instructions of statements

Assignment Statements

- Variable assignment: [[v = e]]

```
v = [[ e ]]
```

- Array assignment: [[v[e1] = e2]]

```
t1 = addr v
t2 = [[e1]]
t3 = t2 * S
t4 = t1 + t3
t5 = [[e2]
[t4] = t5    /* ie store */
```

recall S = sizeof(T)
where v is array(T)

Translating If-Then [-Else]

- [[if (e) then s]]

```
t1 = [[ e ]]
t2 = not t1
cjump t2 Lend
[[ s ]]
Lend:
```

- [[if (e) then s1 else s2]]

```
t1 = [[ e ]]
t2 = not t1
cjump t2 Lelse
Lthen: [[ s1 ]]
jump Lend
Lelse: [[ s2 ]]
Lend:
```

How could I do this more efficiently??

While Statements

- [[while (e) s]]

while-do translation

```
Lloop: t1 = [[ e ]]
t2 = NOT t1
cjump t2 Lend
[[ s ]]
jump Lloop
Lend:
```

or

do-while translation

```
t1 = [[ e ]]
t2 = NOT t1
cjump t2 Lend
Lloop:  [[ s ]]
t3 = [[ e ]]
cjump t3 Lloop
Lend:
```

Which is better and why?

Switch Statements

- [[switch (e) case v1:s1, ..., case vN:sN]]

```
t = [[ e ]]
L1: c =  t != v1
cjump c L2
[[ s1 ]]
jump Lend   /* if there is a break */
L2: c =  t != v2
cjump c L3
[[ s2 ]]
jump Lend    /* if there is a break */
...
Lend:
```

Can also implement switch as table lookup. Table contains target labels, ie L1, L2, L3. ‘t’ is used to index table.

Benefit: k branches reduced to 1.
Negative: target of branch hard to figure out in hardware

Call and Return Statements

- [[call f(e1, e2, ..., eN)]]

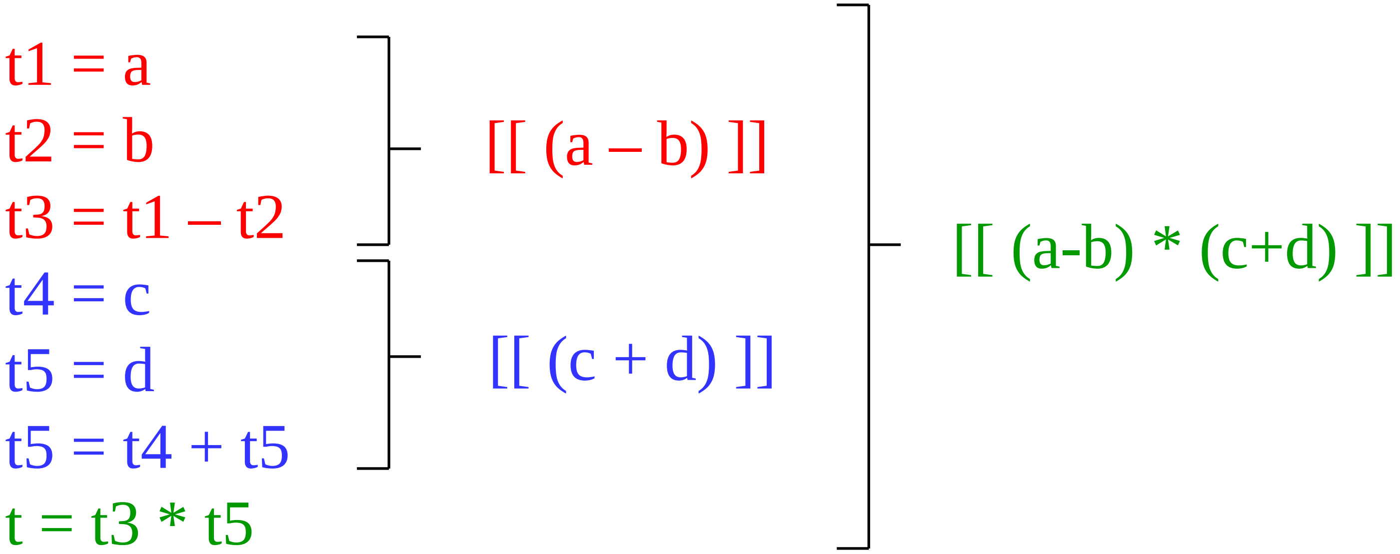
```
t1 = [[ e1 ]]
t2 = [[ e2 ]]
...
tN = [[ eN ]]
call f(t1, t2, ..., tN)
```

- [[return e]]

```
t = [[ e ]]
return t
```

Nested Expressions

- Translation recurses on the expression structure
- Example: t = [[(a – b) * (c + d)]]



t1 = a
t2 = b
t3 = t1 – t2
t4 = c
t5 = d
t5 = t4 + t5
t = t3 * t5

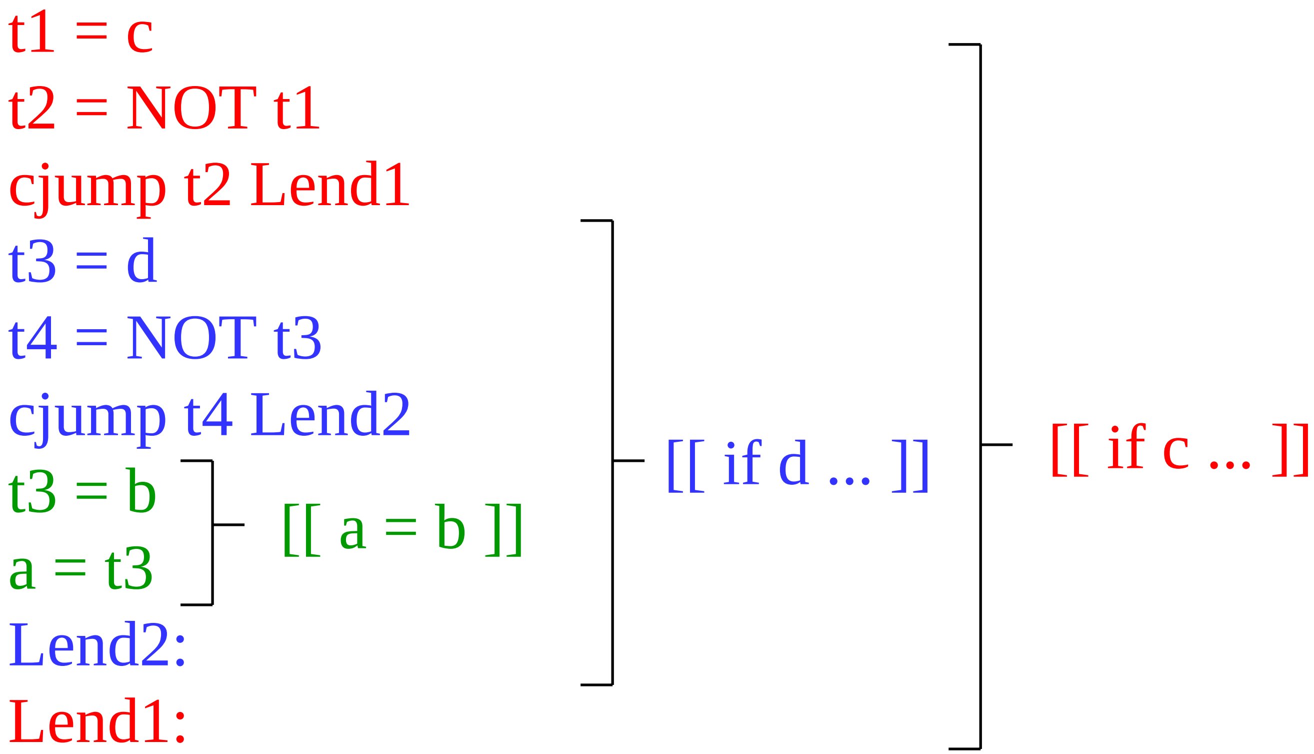
[[(a – b)]]

[[(c + d)]]

[[(a-b) * (c+d)]]

Nested Statements

- Same for statements: recursive translation
- Example: t = [[if c then if d then a = b]]



```
t1 = c
t2 = NOT t1
cjump t2 Lend1
t3 = d
t4 = NOT t3
cjump t4 Lend2
t3 = b
a = t3
Lend2:
Lend1:
```

[[a = b]]

[[if d ...]]

[[if c ...]]

Class Problem

Translate the following to the generic assembly code discussed

```
for (i=0; i<100; i++) {
    A[i] = 0;
}
```

```
if ((a > 0) && (b > 0))
    c = 2;
else
    c = 3;
```

Issues

- These translations are straightforward
- But, inefficient:
  - Lots of temporaries
  - Lots of labels
  - Lots of instructions
- Can we do this more intelligently?
  - Should we worry about it?