

# Checkin 21

**Translate the following A code into 3AC**

```
fn : () int -> f{  
  a:int;  
  a = 1 + 2;  
  return a;  
}
```

*ECS 665*

# COMPILER

## CONSTRUCTION

3AC Translation

# Last Time

## Intermediate Representations

## Intermediate Representations

### 3AC

#### What you should know:

- Rational of intermediate representations
- The basic idea of 3AC
  - The instruction set
  - What each instruction more-or-less does

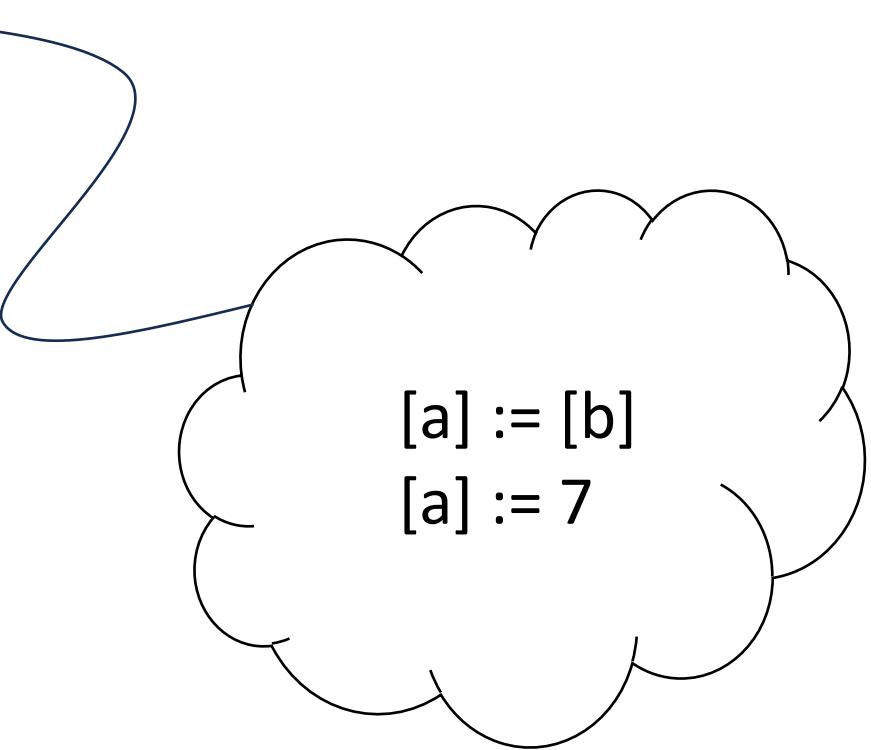


**Intermediate  
Representations**

# The List of Instruction Templates

Review: Our 3AC Instructions

<opd> := <opd> ←  
<opd> := <opr> <opd>  
<opd> := <opd> <opr> <opd>  
<lbl>: <INSTR>  
goto <lbl>  
ifz <opr> goto <lbl>  
nop  
call <name>  
enter <proc>  
leave <proc>  
setarg <int> <opd>  
getarg <int> <opd>  
setret <opd>  
getret <opd>



[a] := [b]  
[a] := 7

# 3AC: Exercise

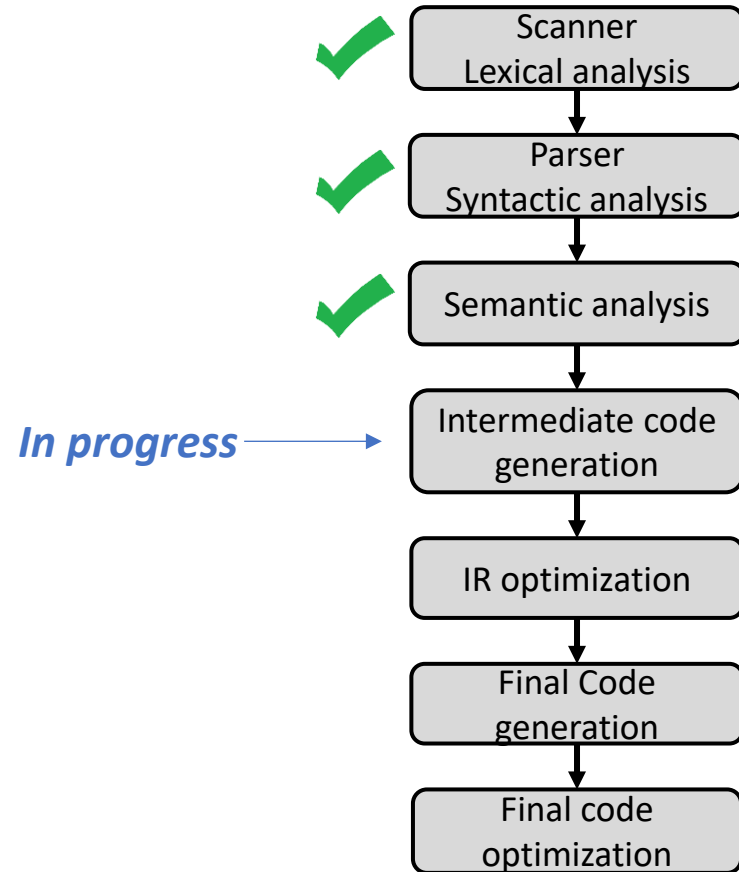
## Another Example

```
int x;  
int y;  
while (x < y) {  
    x = x * 2;  
}  
y = x;
```

```
#tmp 1 is an 8-byte local  
#x is an 8-byte local  
#y is an 8-byte local  
  
LBL1: [tmp1] := [x] LT64 [y]  
      ifz [tmp1] goto LBL2  
      [x] := [x] MULT64 2  
      goto LBL1  
  
LBL2: [y] := [x]
```

# Compiler Construction

## Progress Pics



### Done

- We've captured the semantics of the input
- We've checked the program for correctness

### Next Steps

- Prepare the program for output

# Today's Outline

## 3AC Translation

### The basic idea:

- Traversing the AST

### Some example nodes

- Node to quad translations

### Implementation details:

- From nodes to Operations/Operands



**Intermediate  
Representations**

# Flattening the Tree

AST Translation to 3AC





# Flattening the Tree

AST Translation to 3AC



# Flattening the Tree

AST Translation to 3AC

Consider two major task categories:

What we...

## **Generate**

- (i.e. the 3AC operations for the current node)

## **Propagate**

- (i.e. the 3AC operands used in parent nodes)



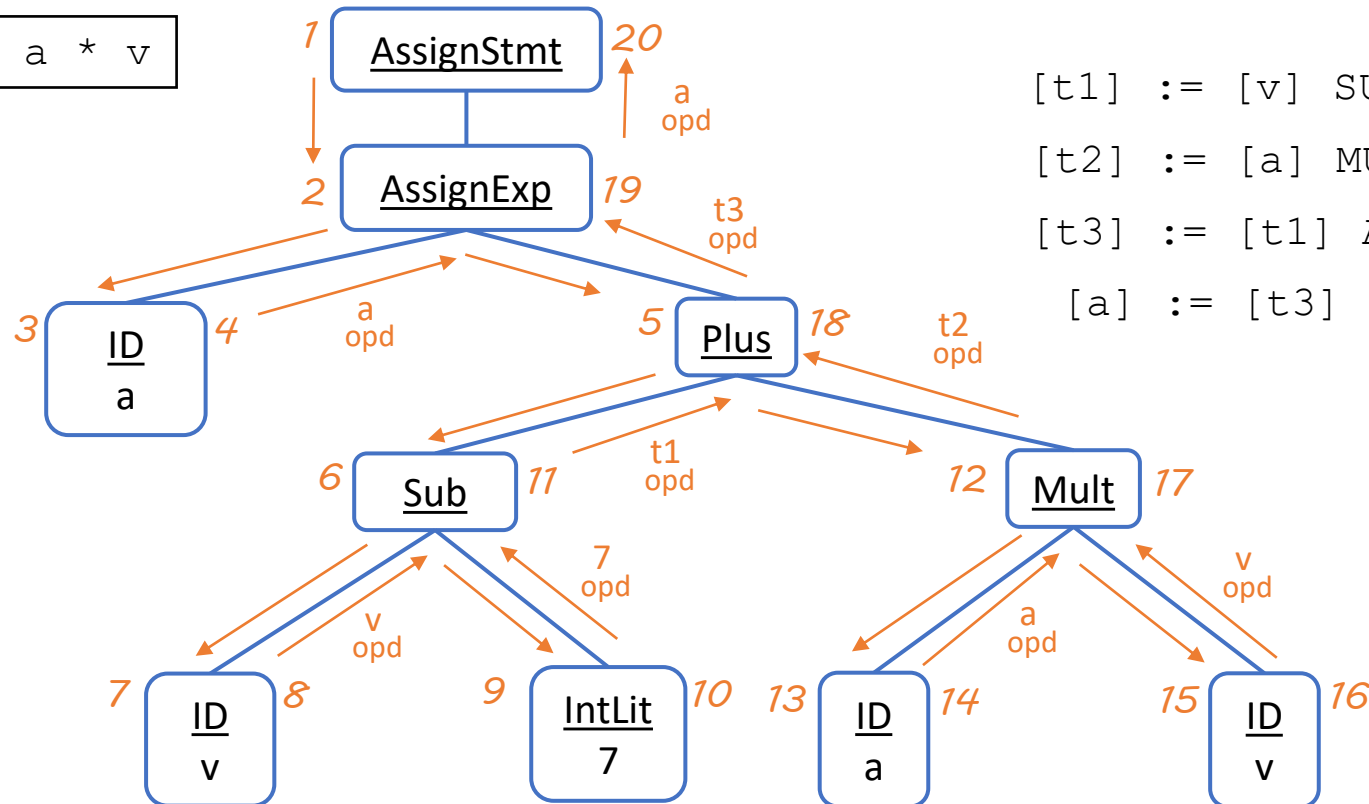
# Flattening The Tree: Example

Traversing the AST

## Traverse AST, performing two tasks

- Generate 3AC operations
- Propagate 3AC operands

a = (v - 7) + a \* v



```
[t1] := [v] SUB64 7
[t2] := [a] MULT64 [v]
[t3] := [t1] ADD64 [t2]
[a] := [t3]
```

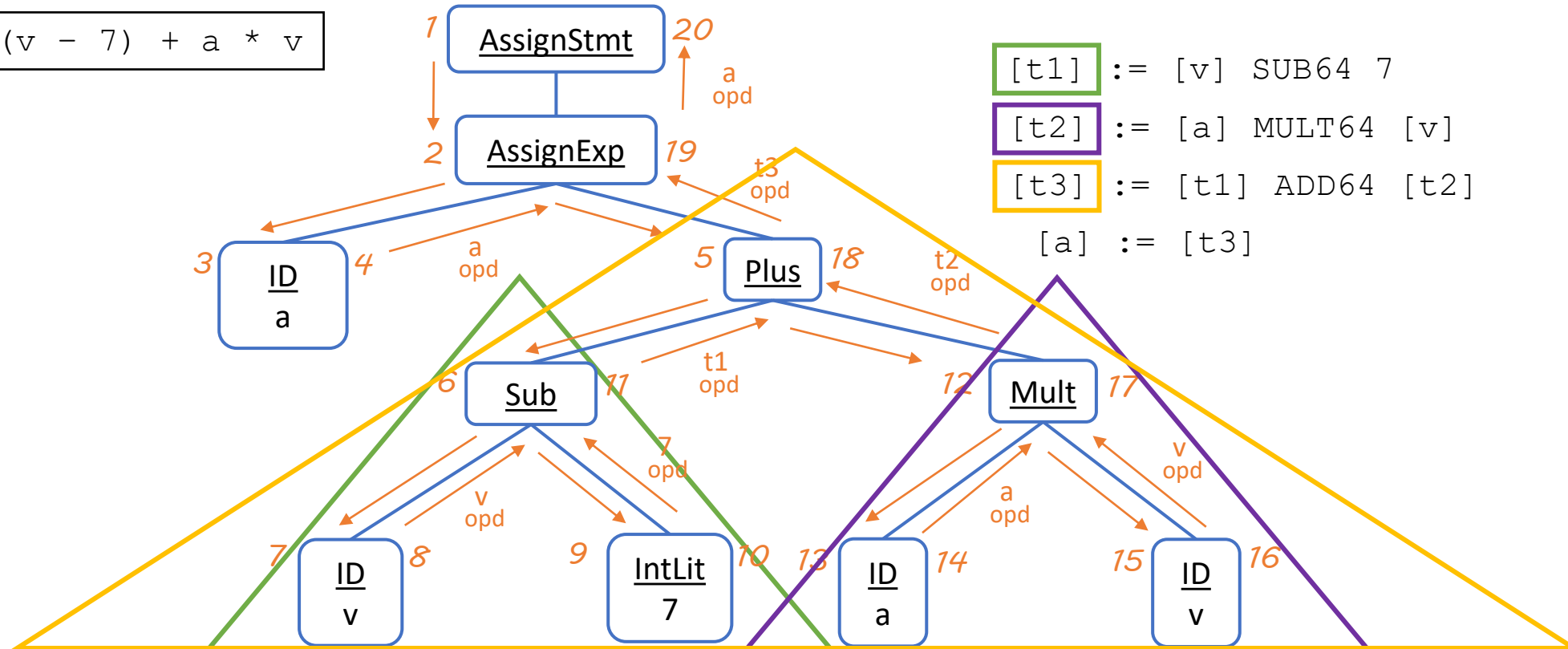
# Flattening The Tree: Example

Traversing the AST

## Traverse AST, performing two tasks

- Generate 3AC operations
- Propagate 3AC operands

a = (v - 7) + a \* v



# A Brief Aside on Sequencing

## Traversing the AST

### What if we walked the tree in a different order?

- Take the RHS of the Plus before the LHS

```
[t2] := [a] MULT64 [v]
```

```
[t1] := [v] SUB64 7
```

```
[t3] := [t1] ADD64 [t2]
```

```
[a] := [t3]
```

**versus**

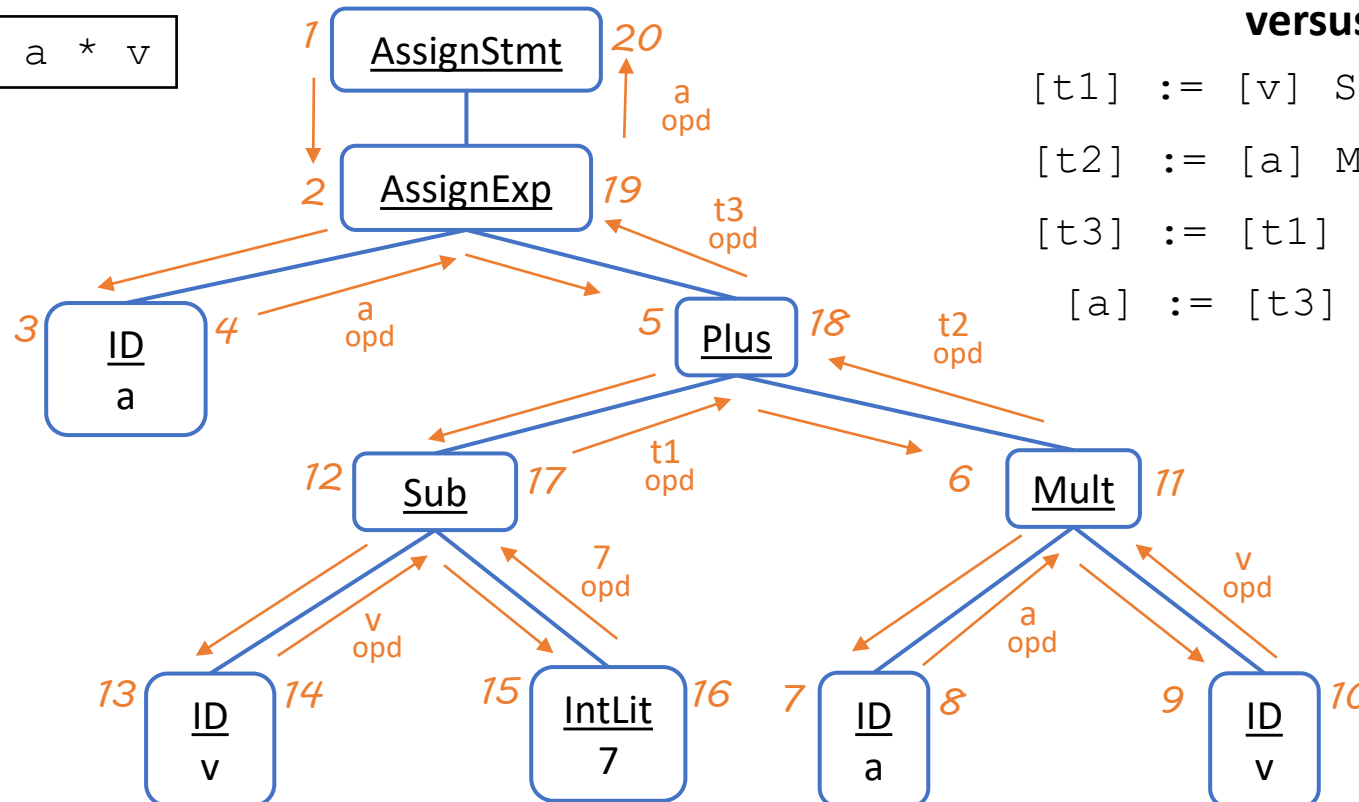
```
[t1] := [v] SUB64 7
```

```
[t2] := [a] MULT64 [v]
```

```
[t3] := [t1] ADD64 [t2]
```

```
[a] := [t3]
```

```
a = (v - 7) + a * v
```



# A Brief Aside on Sequencing

## Traversing the AST

### What if we walked the tree in a different order?

- Take the RHS of the Plus before the LHS
- C and C++ leave this choice to the compiler!

#### Participation

#### Does traversal order matter?

- In this AST?
- For all ASTs?

#### Example code

```
int foo(){ cout << "hi"; return 0; }
int bar(){ cout << "class"; return 0; }
int main(){
    cout << foo() + bar();
}
```

```
[t2] := [a] MULT64 [v]
```

```
[t1] := [v] SUB64 7
```

```
[t3] := [t1] ADD64 [t2]
```

```
[a] := [t3]
```

#### versus

```
[t1] := [v] SUB64 7
```

```
[t2] := [a] MULT64 [v]
```

```
[t3] := [t1] ADD64 [t2]
```

```
[a] := [t3]
```

# A Brief Aside on Sequencing

Traversing the AST

## What if we walked the tree in a different order?

- Take the RHS of the Plus before the LHS
- C and C++ leave this choice to the compiler!

## Order DOES matter

- Can change the program's semantics!

```
int g;
int foo() { return g; }
int bar() { g++; return g; }
int main(){ g = 0; return foo() * bar(); }
```

**For our language, always go left to right**  
(when possible)

```
[t2] := [a] * [v]
```

```
[t1] := [v] - 7
```

```
[t3] := [t1] + [t2]
```

```
[a] := [t3]
```

```
[t1] := [v] - 7
```

```
[t2] := [a] * [v]
```

```
[t3] := [t1] + [t2]
```

```
[a] := [t3]
```

# Today's Outline

## 3AC Translation

### **The basic idea:**

- Traversing the AST

### **Example Nodes:**

- Node to quad translations

### **Implementation details:**

- Operations and operators



**Intermediate  
Representations**



# Example Nodes

Node to Quad Translations

**This generate + propagate idea is powerful!**

- Basically worked for previous traversals:
  - Name analysis
  - Type analysis
  - Syntax-directed translation
- Let's see how it works for some various node types

# Translating AST Leaves (IDs and Lits)

AST Translation to 3AC

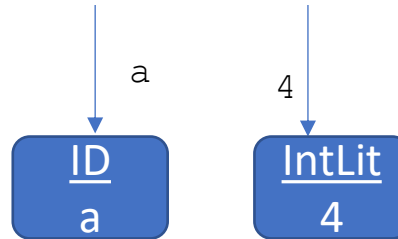
## Generate:

- Nothing!

## Propagate:

- The value for use in parent

### AST Snippet



# Translating AssignExp

AST Translation to 3AC

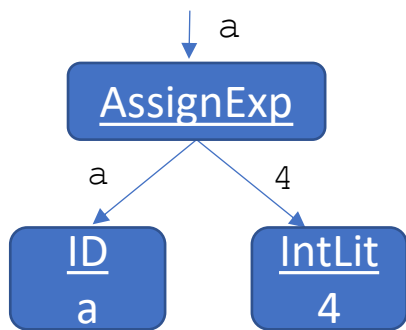
## Generate:

- Code for the LHS (recurse)
- Code for the RHS (recurse)
- The actual assignment instruction

## Propagate:

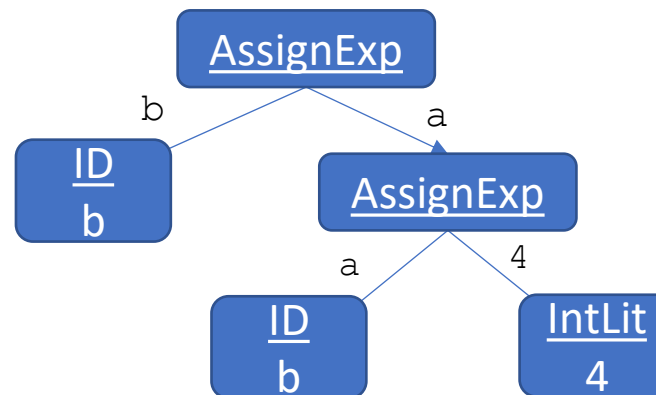
- The LHS of the assignment

a = 4



[a] := 4

b = (a = 4)



[a] := 4

[b] := [a]

# Translating BinaryOp Nodes

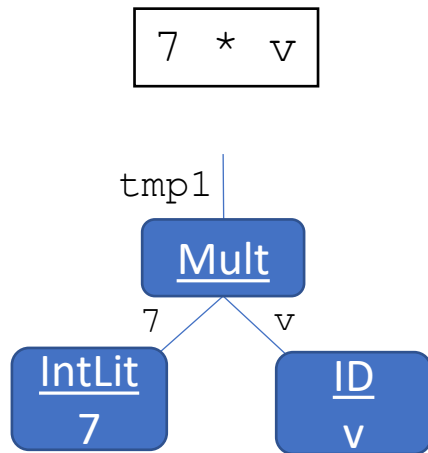
AST Translation to 3AC

## Generate:

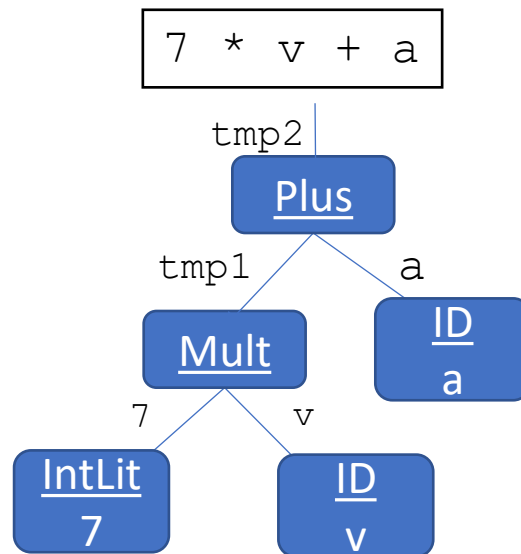
- Code for LHS, RHS (recurse in order)
- Node's operation kind, assigning to new temp

## Propagate:

- The new temp value (for use in parent)



```
[tmp1] := 7 MULT64 [v]
```



```
[tmp1] := 7 MULT64 [v]
```

```
[tmp2] := [tmp1] ADD64 [a]
```

# Translating CallExpNodes

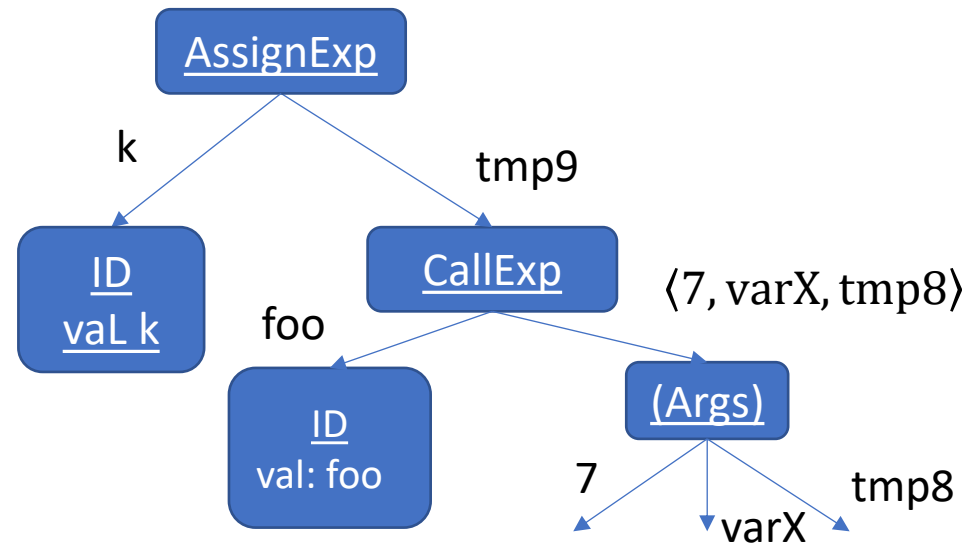
AST Translation to 3AC

## Generate:

- (Recurse over args, left to right)
- setarg instrs for each argument
- call instr for function
- getret instr for the result

## Propagate:

- The getret symbol



### src code snippet

```
k = foo(7, varX, a+b)
```

*(Arg evaluation)*

```
setarg 1, 7  
setarg 2, [varX]  
setarg 3, [tmp8]  
call fn_foo  
getret [tmp9]  
[k] := [tmp9]
```

# Translating FnDeclNodes

AST Translation to 3AC

## Generate:

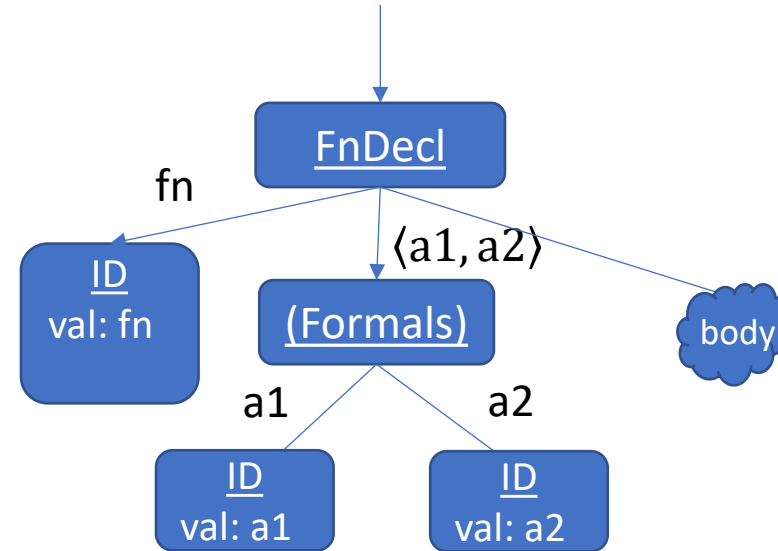
- `enter quad` to begin scope
- A label for function's end
- `getarg quads` for each argument
- (recurse into body)
- `leave quad` to end scope

## Propagate:

- Nothing

### src code snippet

```
void fn(int a1, int a2){  
    ...  
}
```



```
enter fn  
getarg 1, [a1]  
getarg 2, [a2]  
(body code)
```

```
L_fn_end: leave fn
```

# Translating ReturnStmtNodes

AST Translation to 3AC

## Generate:

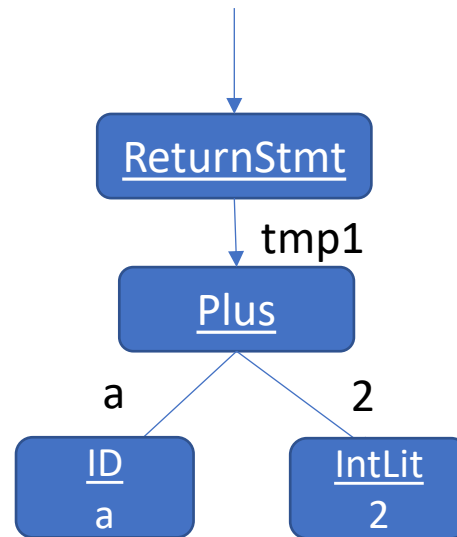
- (recurse into expression)
- `setret` quad for expression tmp
- `goto` for the function end

## Propagate:

- Nothing

### src code snippet

```
return a+2;
```



```
[tmp1] := [a] ADD64 2  
setret [tmp1]  
goto L_fn_end
```

# Translating IfStmtNodes

AST Translation to 3AC

## Generate:

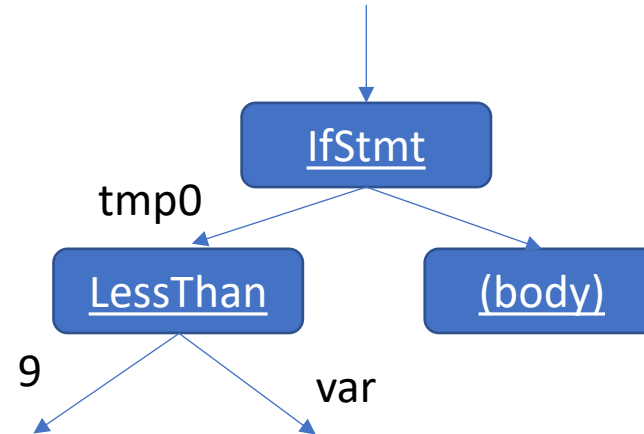
- (recurse into conditional)
- An “after the body” label
- ifz to after the body label
- (recurse into body)
- nop with the new label

## Propagate:

- Nothing

### src code snippet

```
if (9 < var) {  
    (body code)  
}
```



```
[tmp0] := 9 LT64 [var]
```

```
ifz [tmp0] goto L_a
```

```
(body code)
```

```
L_a: nop
```



# Translating While Loops

AST Translation to 3AC

## Generate:

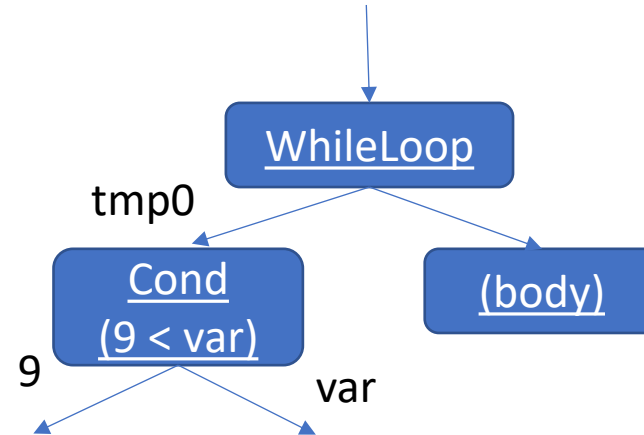
- Label for loop head
- nop for loop head label
- (recurse into conditional)
- ifz to “after the body”
- (recurse into body)
- Jump back to head

## Propagate:

- Nothing

### src code snippet

```
while (9 < var) {  
    (body code)  
}
```



```
L_head: nop  
[tmp0] := 9 LT64 [var]  
ifz[tmp0] goto L_a  
(body code)  
goto L_head  
L_a: nop
```

(repeat)

skip body

# Translating Index

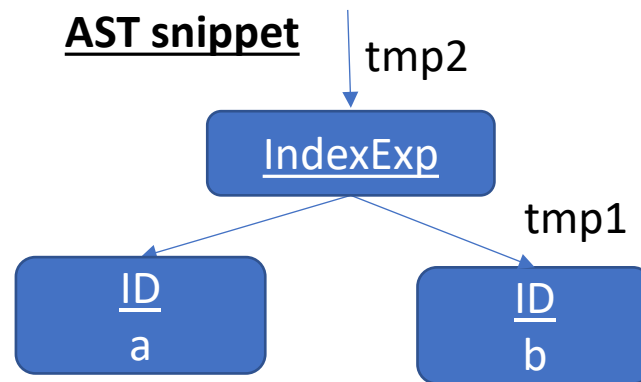
AST Translation to 3AC

## Generate:

- Assign *address* of expression to a new temp

## Propagate:

- New temp



## src code snippet

```
record R{  
  int a;  
}  
...  
R r;  
r.a = 1;
```

## 3AC snippet

```
[tmp1] := [b]  
tmp2 := r @ a
```

# Today's Outline

## 3AC Translation

### **The basic idea:**

- Traversing the AST

### **Example Nodes:**

- Node to quad translations

### **Implementation details:**

- Operations and operators



**Intermediate  
Representations**

# 3AC Data Structures

## AST Translation to 3AC: Implementation

- One class per 3AC node type
  - Often referred to as “Quads” – has at most 4 fields (+ label)
  - Each procedure maintains a list of its quads

L1: [tmp1] := [a] SUB64 2

lbl	dst	src1	opr	src2
L1	tmp1 val	a val	-	2 literal

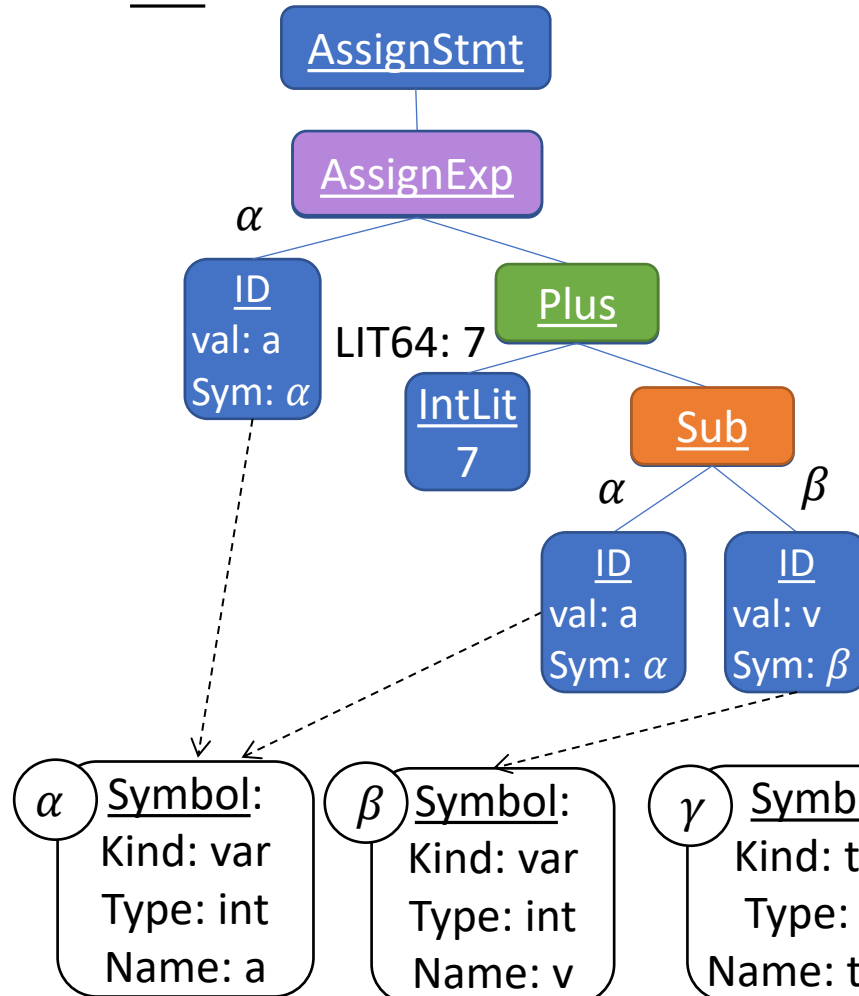
# Translation Implementation

## AST Translation to 3AC

### src code snippet

```
a = 7 + (a - v)
```

### AST



### Quads

lbl	dst	src <sub>1</sub>	opr	src <sub>2</sub>
	tmp1 ( $\gamma$ )	a ( $\alpha$ )	SUB 64	v ( $\beta$ )
	tmp2 ( $\delta$ )	7	ADD 64	tmp1 ( $\gamma$ )
	a ( $\alpha$ )	tmp2 ( $\delta$ )	ASG 64	

# Translation Implementation

## AST Translation to 3AC

### At this point, we can discard the AST

- New data structures for the 3AC representation:
  - Quad class (with subclasses for each quad type)
  - Procedure class
    - Contains list of quads
  - Operand abstraction (symbols)

### Quads

lbl	dst	src <sub>1</sub>	opr	src <sub>2</sub>
	tmp1 ( $\gamma$ )	a ( $\alpha$ )	SUB 64	v ( $\beta$ )
	tmp2 ( $\delta$ )	7	ADD 64	tmp1 ( $\gamma$ )
	a ( $\alpha$ )	tmp2 ( $\delta$ )	ASG 64	

# Lecture End

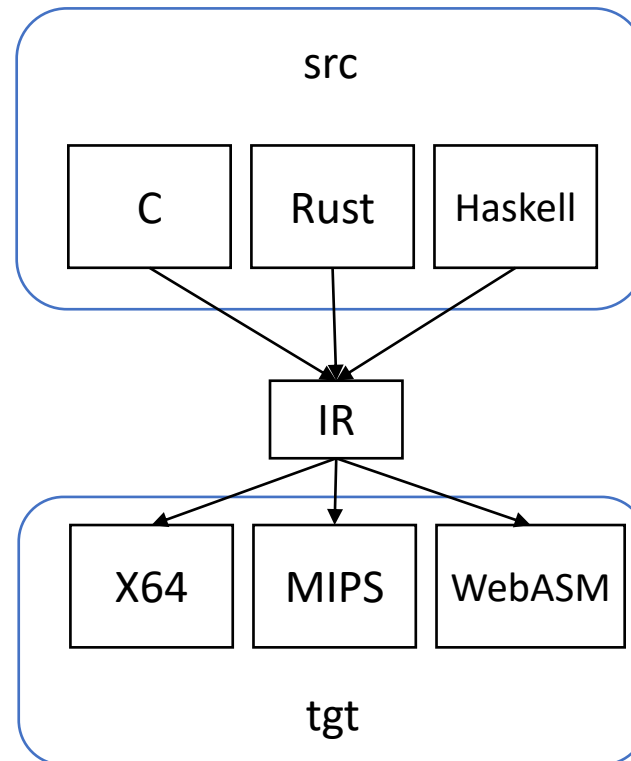
3AC Translation

## We've successfully flattened the AST

- Got a nice target for final code generation
- Removed the nesting
- Make execution order explicit

## Next time

- Start exploring the compiler targets



*The multicompile concept  
One IR for many sources, many targets*

# 3AC in Summary

AST Translation to 3AC

## **A Nice Linear IR**

- Gets us close to real hardware
- Abstract enough to be used in a variety of backends