

# Overview

- Goal: Generate a Machine Independent Intermediate Form that is Suitable for Optimization and Portability
- We'll Focus on
  - Revisit AST and DAGs with Attribute Grammars
  - Introduce and Elaborate on Three Address Coding
  - Review 3AC for:
    - Declarations
    - Assignments and Boolean Expressions
    - Case Statements and Procedure Calls
  - Review Prof. Michel's Approach to Intermediate Code Generation
  - Concluding Remarks/Looking Ahead

# Motivation

## ○ What we have so far...

### □ A Parser tree

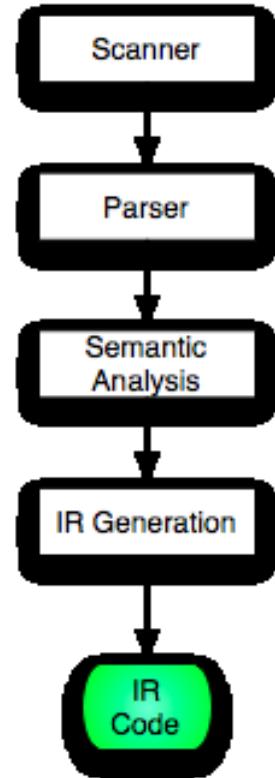
- With all the program information
- Known to be correct
  - Well-typed
  - Nothing missing
  - No ambiguities

## ○ What we need...

### □ Something “Executable”

### □ Closer to

- An operations schedule
- Actual machine level of abstraction



# What We Want

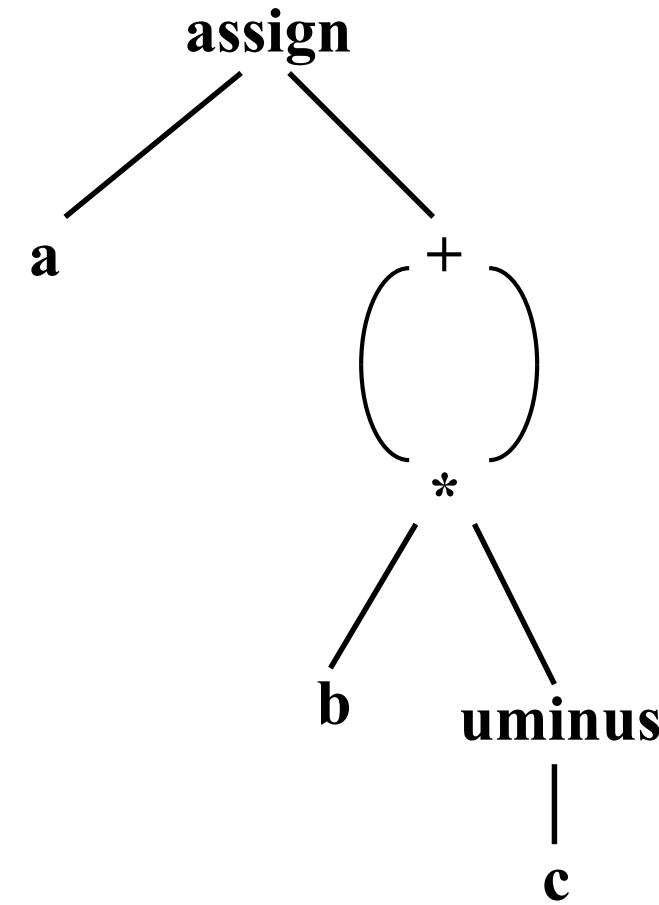
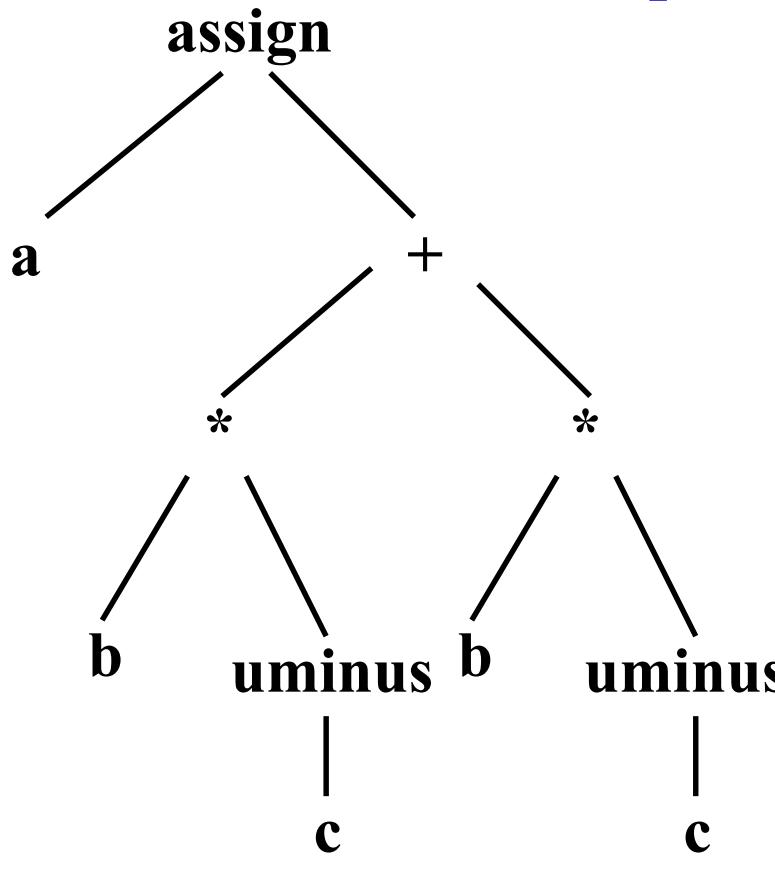
- A Representation that
  - Is closer to actual machine
  - Is easy to manipulate
  - Is target neutral (hardware independent)
  - Can be interpreted

# Recall ASTs and DAGs

CSE  
4100

## ○ Intermediate Forms of a Program:

- ASTs: Abstract Syntax Trees (below left)
- DAGs: Directed Acyclic Graphs (below right)
- What is the Expression?



# Attribute Grammar

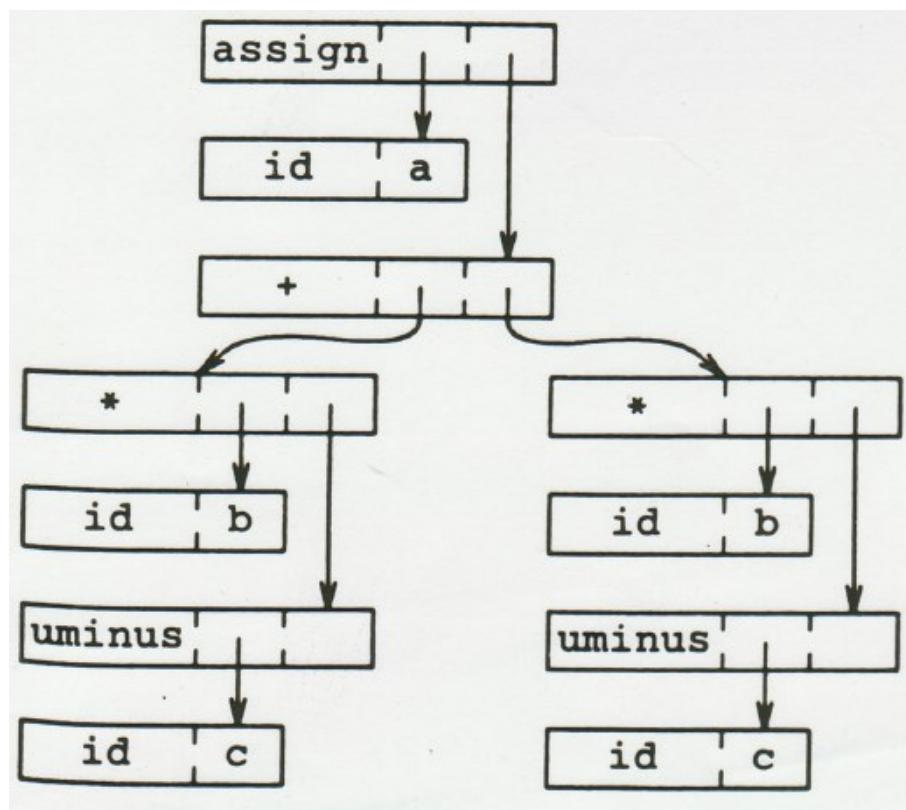
- Attribute Grammar for Generating an AST as a side Effect of the Translation Process:

PRODUCTION	SEMANTIC RULE
$S \rightarrow \text{id} := E$	$S.\text{nptr} := \text{mknode}(\text{'assign'}, \text{mkleaf}(\text{id}, \text{id.place}), E.\text{nptr})$
$E \rightarrow E_1 + E_2$	$E.\text{nptr} := \text{mknode}(\text{'+'}, E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow E_1 * E_2$	$E.\text{nptr} := \text{mknode}(\text{'*'}, E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow - E_1$	$E.\text{nptr} := \text{mkunode}(\text{'uminus'}, E_1.\text{nptr})$
$E \rightarrow ( E_1 )$	$E.\text{nptr} := E_1.\text{nptr}$
$E \rightarrow \text{id}$	$E.\text{nptr} := \text{mkleaf}(\text{id}, \text{id.place})$

# Representing Attribute Grammars

CSE  
4100

- Two Different Forms:
  - Linked Data Structure
  - Multi-Dimensional Array



0	<code>id</code>	<code>b</code>
1	<code>id</code>	<code>c</code>
2	<code>uminus</code>	1
3	*	0 2
4	<code>id</code>	<code>b</code>
5	<code>id</code>	<code>c</code>
6	<code>uminus</code>	5
7	*	4 6
8	+	3 7
9	<code>id</code>	<code>a</code>
10	<code>assign</code>	9 8
11	...	

# Objective

- Directly Generate Code From AST or DAG as a Side Effect of Parsing Process (Attribute Grammar)
- Consider Code Below:

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

(a) Code for the syntax trcc.

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a := t5
```

(b) Code for the dag.

- Each is Referred to as “3 Address Coding (3AC)” since there are at Most 3 Addresses per Statement
  - One for Result and At Most 2 for Operands

# The Intermediate Code Generation Machine

CSE  
4100

- A Machine with
  - Infinite number of temporaries (think registers)
  - Simple instructions
    - 3-operands
    - Branching
    - Calls with simple calling convention
  - Simple code structure
    - Array of instructions
  - Labels to define targets of branches.



# Temporaries

- The machine has an infinite number of temporaries
  - Call them  $t_0, t_1, t_2, \dots$
  - Temporaries can hold values of any type
  - The type of the temporary is derived from the generation
  - Temporaries go out of scope with the function they are in

# What is Three-Address Coding?

- A simple type of instruction

- 3 / 2 Operands x,y,z

 $x := y \text{ op } z$  $x := \text{op } z$ 

- Each operand could be

- A literal
    - A variable
    - A temporary

- Example

 $x + y * z$  $t_0 := y * z$   
 $t_1 := x + t_0$

# Types of Three Address Statements

- Assignment Statements of Form:
  - $X := Y \text{ op } Z$
  - $\text{op}$  is a Binary Arithmetic or Logical Operation
- Assignment Instructions of Form:
  - $X := \text{op } Y$
  - $\text{op}$  is Unary Operation such as Unary Minus, Logical Negative, Shift/Conversion Operations
- Copy Statements of Form:
  - $X := Y$  where value of  $Y$  assigned to  $X$
- Unconditional Jump of Form:
  - $\text{goto } L$  which goes to a three address statement labeled with  $L$

# Types of Three Address Statements

- Conditional Jumps of Form:
  - if x relop y goto L
  - with relop as relational operators and the goto executed if the x relop y is true
- Parameter Operations of Form:
  - param a (a parameter of function)
  - call p, n (call function p with n parameters)
  - return y (return value y from function – optional)
    - param a
    - param b
    - param c
    - call p, 3

# Types of Three Address Statements

- Indexed Assignments of Form:
  - $X := Y[i]$  (Set X to i-th memory location of Y)
  - $X[i] := Y$  (Set i-th memory location of X to Y)
  - Note the limit of 3 Addresses (X, Y, i)
  - Cannot do:  $x[i] := y[j]$ ; (4 addresses!)
- Address and Pointer Assignments of Form:
  - $X := \& Y$  (X set to the Address of Y)
  - $X := * Y$  (X set to the contents pointed to by Y)
  - $* X := Y$  (Contents of X set to Value of Y)

# Attribute Grammar for Assignments

- Concepts:
  - Need to Introduce Temporary Variables as Necessary do Decompose Assignment Statement
  - Every Generated Line of Code Must have at Most 3 Addresses!
- Three Attributes
  - S.Code : Three Address Code for Non-Term S
  - E.Place : The Name that Holds Value of E
  - E.Code : Sequence of 3AC for Expression E
- Function:
  - newtemp: Generate a New temp Variable

# Attribute Grammar

CSE  
4100

- How would Grammar work for:

- $A := -B$
- $X := Y + Z$
- $W := B^*C + D$

PRODUCTION	SEMANTIC RULES
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place ':= E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp; \text{ RETURNS TEMP VAR } t_1, t_2, \dots$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place ':= E_1.place '+' E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp; \text{ ~~~~ CONCATENATION }$ $E.code := E_1.code \parallel E_2.code \parallel$ $gen(E.place ':= E_1.place '*' E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp;$ $E.code := E_1.code \parallel gen(E.place ':= 'uminus' E_1.place)$
$E \rightarrow ( E_1 )$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code := ''$

# Three Address Code for While Loops

- Grammar Rule:
  - $S \rightarrow \text{while } E \text{ do } S_1$
- Conceptually:
  - Check Expression E
  - If Not True, Skip to First Statement after  $S_1$
  - Execute  $S_1$
  - Goto: Check Expression E
- Function:
  - newlabel : Generates a new label for goto/jump
  - Generate labels for Check Expression and 1<sup>st</sup> Statement after  $S_1$

# Attribute Grammar for While

CSE  
4100

*S.begin :*

*E.code*

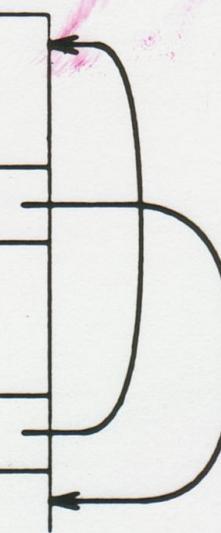
**if *E.place* = 0 goto *S.after***

*S<sub>1</sub>.code*

**goto *S.begin***

*S.after :*

...



PRODUCTION

$S \rightarrow \text{while } E \text{ do } S_1$

SEMANTIC RULES

*S.begin := newlabel;* *GENERATE A LABEL*

*S.after := newlabel;*

*S.code := gen(S.begin ':') ||*

*E.code ||*

*gen('if' *E.place* '=' '0' 'goto' *S.after*) ||*

*S<sub>1</sub>.code ||*

*gen('goto' *S.begin*) ||*

*gen(*S.after* ':')*

# Declarations

- Recall Runtime Environment in Chapter 7
- One Critical Issue was Memory Layout (Static, Stack, Heap)
- Stack Utilized during Procedure/Function Calls to
  - Allocate Space for P/F Variables
  - This now Includes Temporaries for 3AC
- We need to Track
  - Name
  - Type (Int, real, boolean, etc.)
  - Offset (with respect to some relative address)
- Function
  - enter (name, type, offset) creates symbol table entry
  - offset global initially 0

# Attribute Grammar for Memory Layout

CSE  
4100

- Note: Does not include Temporaries (Need to also Allocate Memory for them as well)
- How could this be supported?

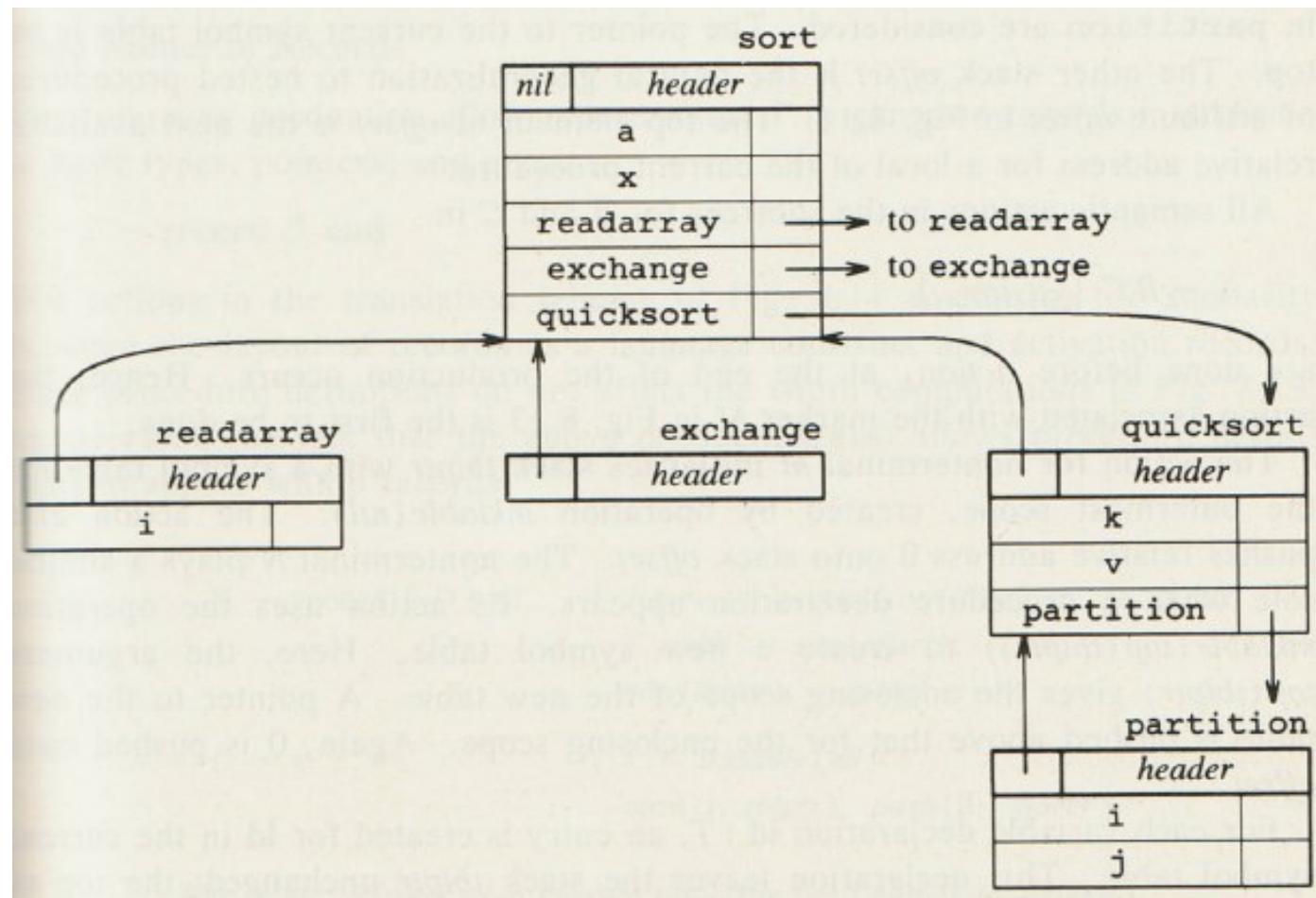
$P \rightarrow$	{ $offset := 0$ }
$D \rightarrow$	$D ; D$
$D \rightarrow id : T$	{ $enter(id.name, T.type, offset);$ $offset := offset + T.width$ }
$T \rightarrow integer$	{ $T.type := integer;$ $T.width := 4$ }
$T \rightarrow real$	{ $T.type := real;$ $T.width := 8$ }
$T \rightarrow array [ num ] of T_1$	{ $T.type := array(num.val, T_1.type);$ $T.width := num.val \times T_1.width$ }
$T \rightarrow \uparrow T_1$	{ $T.type := pointer(T_1.type);$ $T.width := 4$ }

# Handling Nested Procedure/Functions

- Recall that in Pascal, Procedures and Functions can be Defined within Other Procedures/Functions
- Grammar:
  - $P \rightarrow D ; S$
  - $D \rightarrow D ; D \mid id : T \mid proc\ id ; D ; S$
- In this Situation, Symbol Table Can't be Monolithic Structure but Must Support Nested Declarations
- Notations:
  - M, N: Marker Non Terminals so that All S. Actions Occur at End of RHS of Rule
  - mkttable (create ST), enter (ST entry)
  - addwidth – size of symbol table
  - enterproc – Add a Procedure Name to ST

# Nested Symbol Table

- Recall Quicksort from Chapter 7



# Corresponding Attribute Grammar

CSE  
4100

$P \rightarrow M \ D$     { addwidth (top (tblpti), top (offset));  
                  pop (tblpti); pop (offset) }

$M \rightarrow E$     { t := mktable (nil);  
                  push (t, tblpti); push (0, offset) }

$D \rightarrow D_1 ; D_2$

$D \rightarrow \text{proc id; ND;} ; S$     { t := top (tblpti);  
                  addwidth (t, top (offset));  
                  pop (tblpti); pop (offset);  
                  enterproc (top (tblpti), id.name, t) }

$D \rightarrow id : T$     { enter (top (tblpti), id.name, T.type, top (offset));  
                  top (offset) := top (offset) + T.width }

$ND \rightarrow E$     { t := mktable (top (tblpti));  
                  push (t, tblpti); push (0, offset) }

# From Concatenation to Generation

- Up to Now, Attribute Grammars have focused on:
  - Concatenation to Put Together a Long String
  - Resulting String Output at “End”
- However, this is not as Feasible from an Implementation Perspective
- Instead, we Would Like to Generate 3AC Statements as they Occur
- Introduce emit Function
  - At Relevant Stages of Attribute Grammar, output the entire 3AC Statement into a File (Stream)
  - Assumes Names of Symbols (IDs) have been Inserted into Symbol Table with Types by Combination of Lexical Analysis and Declarations

# Revised Attribute Grammar with Emit

$S \rightarrow \text{id} := E \quad \{ p := \text{lookup}(\text{id.name});$   
 $\quad \text{if } p \neq \text{nil} \text{ then}$   
 $\quad \quad \text{emit}(p' := E.\text{place})$   
 $\quad \text{else error} \}$

$E \rightarrow E_1 + E_2 \quad \{ E.\text{place} := \text{newtemp};$   
 $\quad \text{emit}(E.\text{place}' := E_1.\text{place}' + E_2.\text{place}) \}$

$E \rightarrow E_1 * E_2 \quad \{ E.\text{place} := \text{newtemp};$   
 $\quad \text{emit}(E.\text{place}' := E_1.\text{place}' * E_2.\text{place}) \}$

$E \rightarrow - E_1 \quad \{ E.\text{place} := \text{newtemp};$   
 $\quad \text{emit}(E.\text{place}' := '\text{uminus}' E_1.\text{place}) \}$

$E \rightarrow ( E_1 ) \quad \{ E.\text{place} := E_1.\text{place} \}$

$E \rightarrow \text{id} \quad \{ p := \text{lookup}(\text{id.name});$   
 $\quad \text{if } p \neq \text{nil} \text{ then}$   
 $\quad \quad E.\text{place} := p$   
 $\quad \text{else error} \}$

# Attribute Grammar for Boolean Expressions

$E \rightarrow E_1 \text{ or } E_2$	{ <i>E.place</i> := newtemp; emit( <i>E.place</i> ' := ' <i>E<sub>1</sub>.place</i> 'or' <i>E<sub>2</sub>.place</i> ) }
$E \rightarrow E_1 \text{ and } E_2$	{ <i>E.place</i> := newtemp; emit( <i>E.place</i> ' := ' <i>E<sub>1</sub>.place</i> 'and' <i>E<sub>2</sub>.place</i> ) }
$E \rightarrow \text{not } E_1$	{ <i>E.place</i> := newtemp; emit( <i>E.place</i> ' := ' 'not' <i>E<sub>1</sub>.place</i> ) }
$E \rightarrow ( E_1 )$	{ <i>E.place</i> := <i>E<sub>1</sub>.place</i> }
$E \rightarrow \text{id}_1 \text{ relop id}_2$	{ <i>E.place</i> := newtemp; emit('if' <i>id<sub>1</sub>.place</i> <i>relop.op</i> <i>id<sub>2</sub>.place</i> 'goto' <i>nextstat + 3</i> ); emit( <i>E.place</i> ' := ' '0'); emit('goto' <i>nextstat + 2</i> ); emit( <i>E.place</i> ' := ' '1') }
$E \rightarrow \text{true}$	{ <i>E.place</i> := newtemp; emit( <i>E.place</i> ' := ' '1' ) }
$E \rightarrow \text{false}$	{ <i>E.place</i> := newtemp; emit( <i>E.place</i> ' := ' '0' ) }

# Generated Code

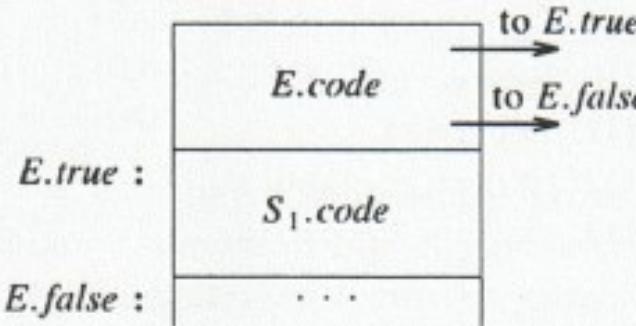
- Consider:  $a < b$  or  $c < d$  and  $e < f$
- How Does Attribute Grammar Generate Code?
- Do RM Derivation and Create Parse Tree

```
100: if a < b goto 103
101: t1 := 0
102: goto 104
103: t1 := 1
104: if c < d goto 107
105: t2 := 0
106: goto 108
107: t2 := 1
108: if e < f goto 111
109: t3 := 0
110: goto 112
111: t3 := 1
112: t4 := t2 and t3
113: t5 := t1 or t4
```

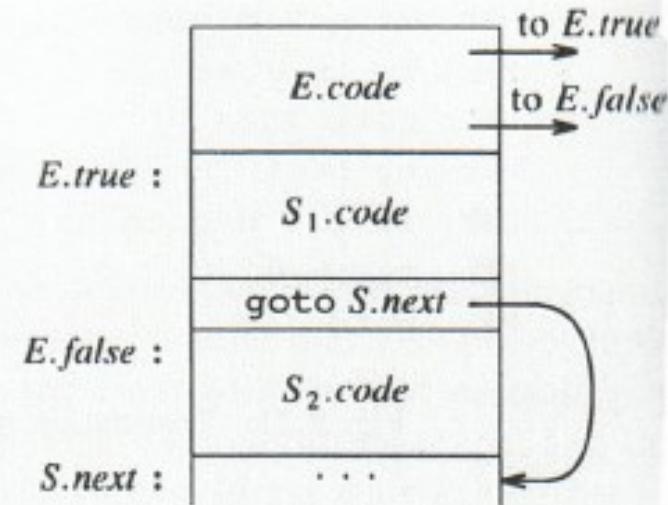
# Code Generation - Conceptual

CSE  
4100

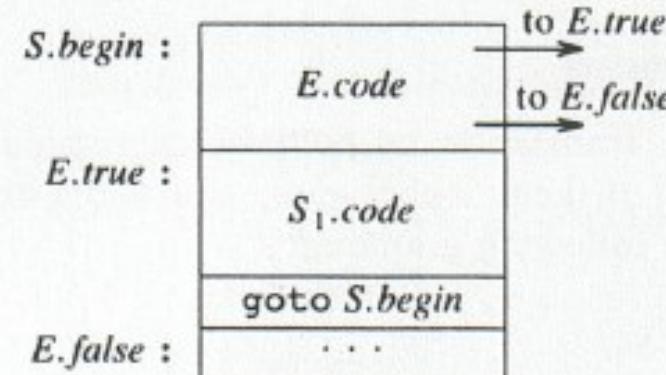
- For If-Then, If-Then-Else, and While-Do



(a) if-then



(b) if-then-else



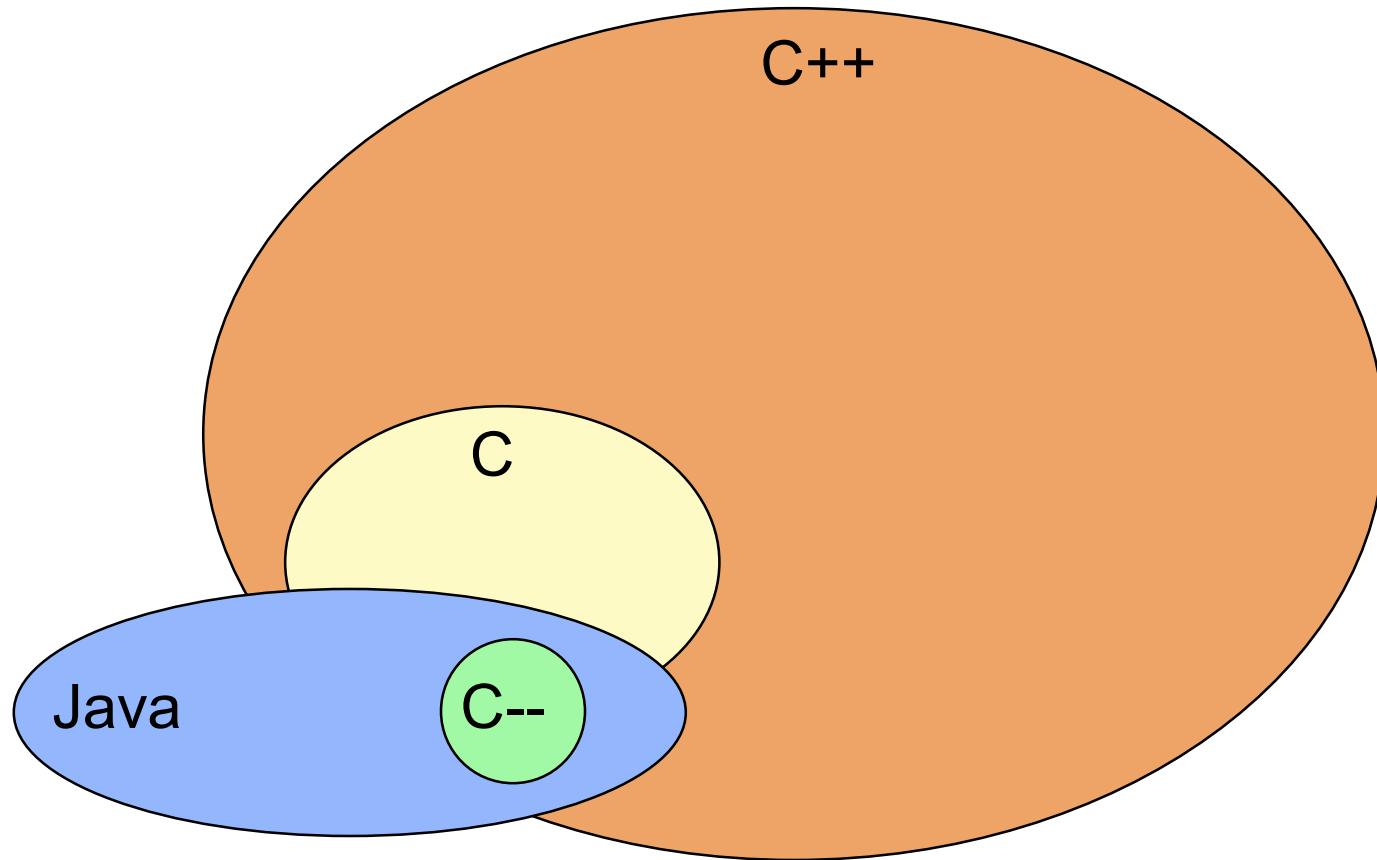
# Corresponding Attribute Grammar

CSE  
4100

PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{if } E \text{ then } S_1$	$E.\text{true} := \text{newlabel};$ $E.\text{false} := S.\text{next};$ $S_1.\text{next} := S.\text{next};$ $S.\text{code} := E.\text{code} \parallel$ $\quad \text{gen}(E.\text{true} ':\') \parallel S_1.\text{code}$
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$E.\text{true} := \text{newlabel};$ $E.\text{false} := \text{newlabel};$ $S_1.\text{next} := S.\text{next};$ $S_2.\text{next} := S.\text{next};$ $S.\text{code} := E.\text{code} \parallel$ $\quad \text{gen}(E.\text{true} ':\') \parallel S_1.\text{code} \parallel$ $\quad \text{gen}(\text{'goto'} S_2.\text{next}) \parallel$ $\quad \text{gen}(E.\text{false} ':\') \parallel S_2.\text{code}$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.\text{begin} := \text{newlabel};$ $E.\text{true} := \text{newlabel};$ $E.\text{false} := S.\text{next};$ $S_1.\text{next} := S.\text{begin};$ $S.\text{code} := \text{gen}(S.\text{begin} ':\') \parallel E.\text{code} \parallel$ $\quad \text{gen}(E.\text{true} ':\') \parallel S_1.\text{code} \parallel$ $\quad \text{gen}(\text{'goto'} S.\text{begin})$

# Review Prof. Michel's Approach

- Utilize AST Directly
- More Intuitive than Attribute Grammar
- Aligns with “emit” Example on Prior Slides
- Utilizes C- (see below)



## ○ The Object Oriented Language For Dummies

### □ C-- supports

- Classes
- Inheritance
- Polymorphism
- 2 basic types
- Arrays
- Simple control primitives
  - if-then-else
  - while



# C-- Example

CSE  
4100

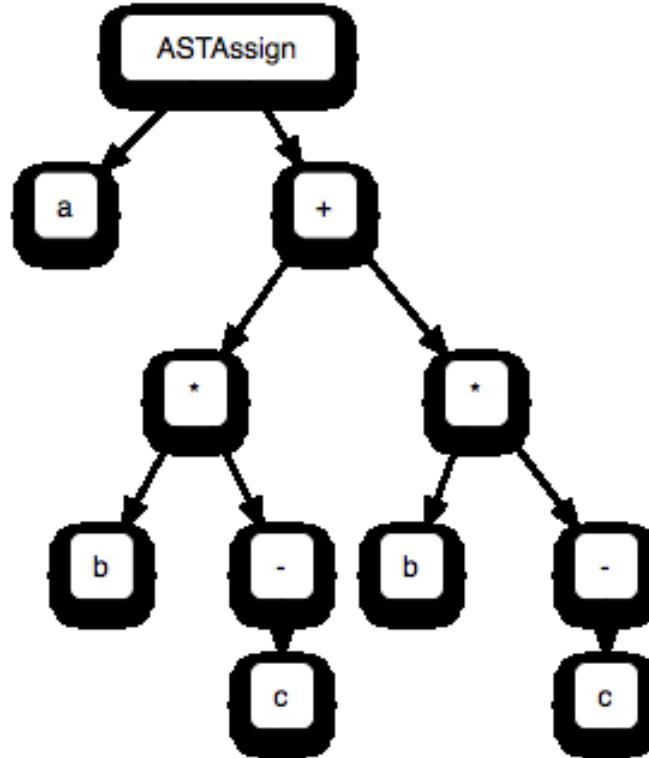
```
class Foo {  
  
    int fact(int n) {  
        return 0;  
    }  
  
    int fib(int x) {  
        return 1;  
    }  
};  
  
class Main extends Foo {  
  
    Main() {  
        int x;  
        x = fact(5);  
    }  
  
    int fact(int n) {  
        if (n==0) return 1;  
        else return n * fact(n-1);  
    }  
};
```

# Bigger Example

CSE  
4100

- Consider the AST

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



```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

# Special 3-address Statements

- We could have

- $x := y \text{ op } z$  This is an arithmetic / logical binary op
- $x := \text{op } z$  This is an arithmetic / logical unary op
- $x := z$  This is a copy operation
- Also called `mov x,z` in C--

# Branching Statements

- Where to branch ?
  - Add Labels to instruction
    - Label = offset of instruction from beginning of array
- How to branch ?
  - Unconditionally
    - JUMP L
  - Conditionally
    - JUMPZ x, L
    - JUMPNZ x,L
  - Variant
    - JUMP x relop y , L
      - {<,>,<=,>=,==,!=} with relop in

# Function Call Sequence

- Fairly abstract here
- Calling Conventions
  - Arguments passed on a stack
  - Returned value get/set with a special instruction
- At Call Site
  - Push the parameters
  - call the function
    - Pass function LABEL
    - Pass function ARITY
  - Get result & write in temporary

```
param nk
param nk-1
...
param n1
call f,k
get return ti
```

# Method Call Sequence

- Fairly abstract too!
- Calling Conventions
  - Arguments passed on a stack
  - Pass a reference to object as well
  - Returned value get/set with a special instruction
- At Call Site
  - Push the parameters
  - call the method
    - Pass Object SELF reference
    - Pass method ID
    - Pass method ARITY
  - Get result & write in temporary

```
param nk
param nk-1
...
param n1
param obj
invoke obj,m,k
get return ti
```

# In Callee

CSE  
4100

- Fairly simple too
  - Two instruction to enter/leave

prologue  
epilogue

# Indexed Assignments

## ○ Two Operations

 $x := y[i]$  $x[i] := y$ 

Indexed Read

Indexed Write

## ○ In Both cases

- Index can be
  - a constant
  - a name
  - a temporary
- Same for x,y

# Indexed Assignments in C-- IR

- In C-- these two are named
  - mov

$x := y[i]$

$x[i] := y$

$\text{mov } x, y(i)$

$\text{mov } x(i), y$

$\text{mov } x, i(y)$

$\text{mov } i(x), y$

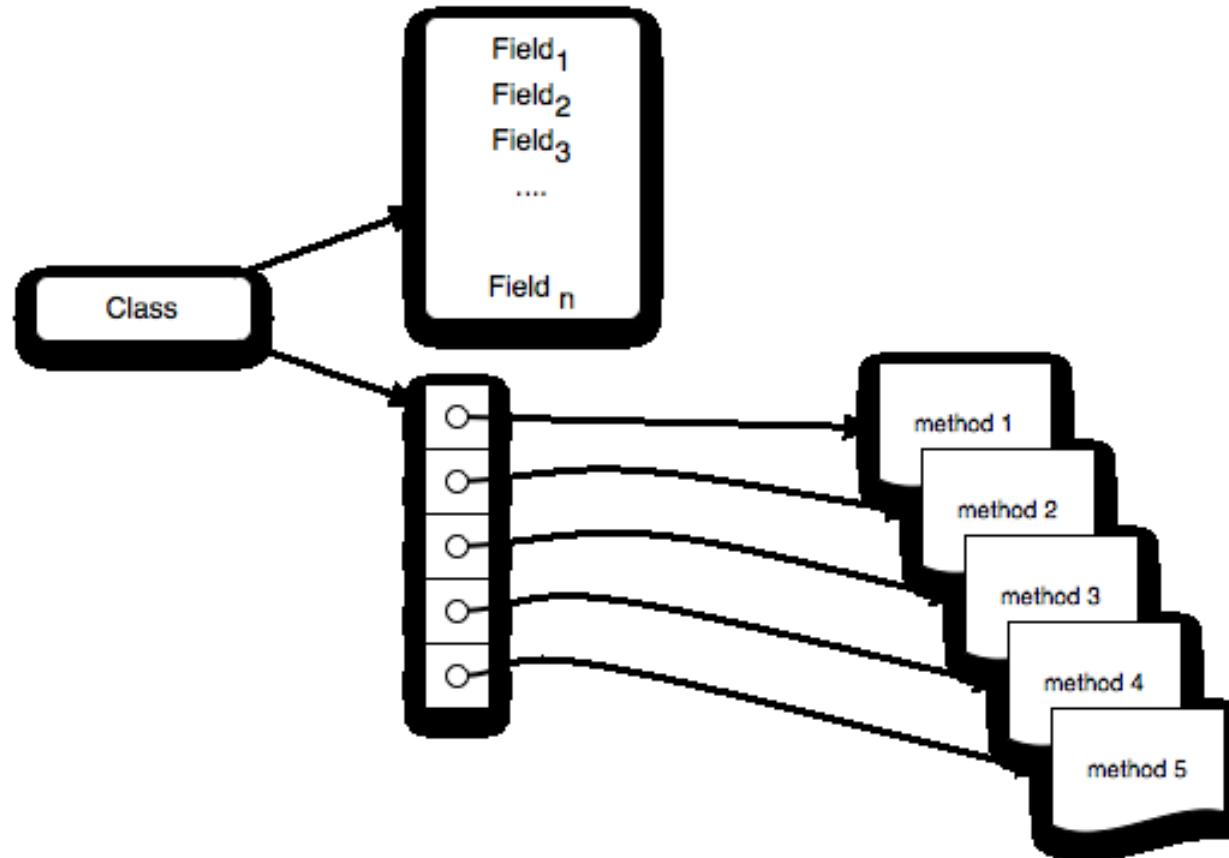
# Classes ?

CSE  
4100

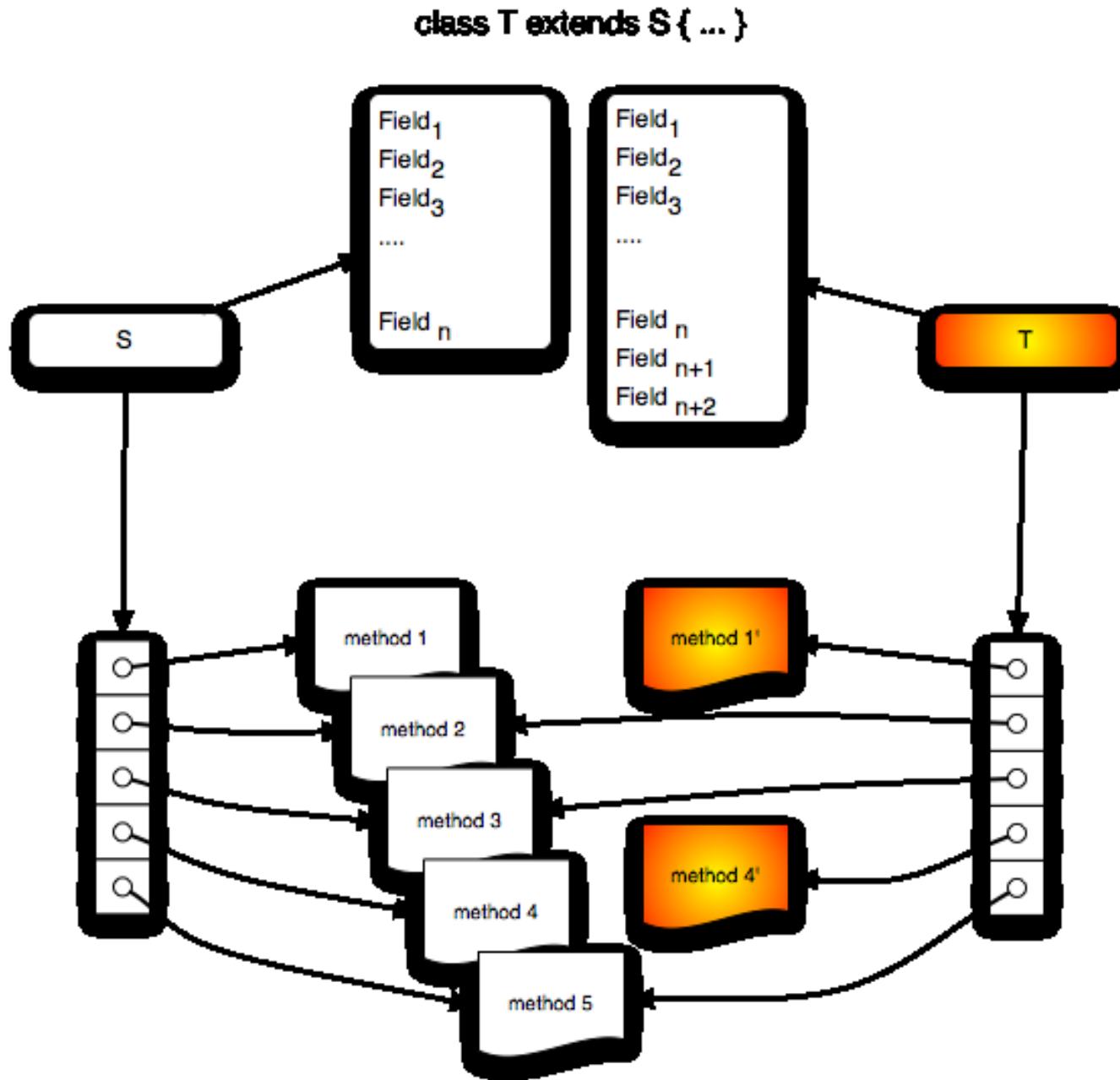
- What is a good representation

- A class is

- A bunch of attributes like a record/struct
    - A bunch of methods like an array

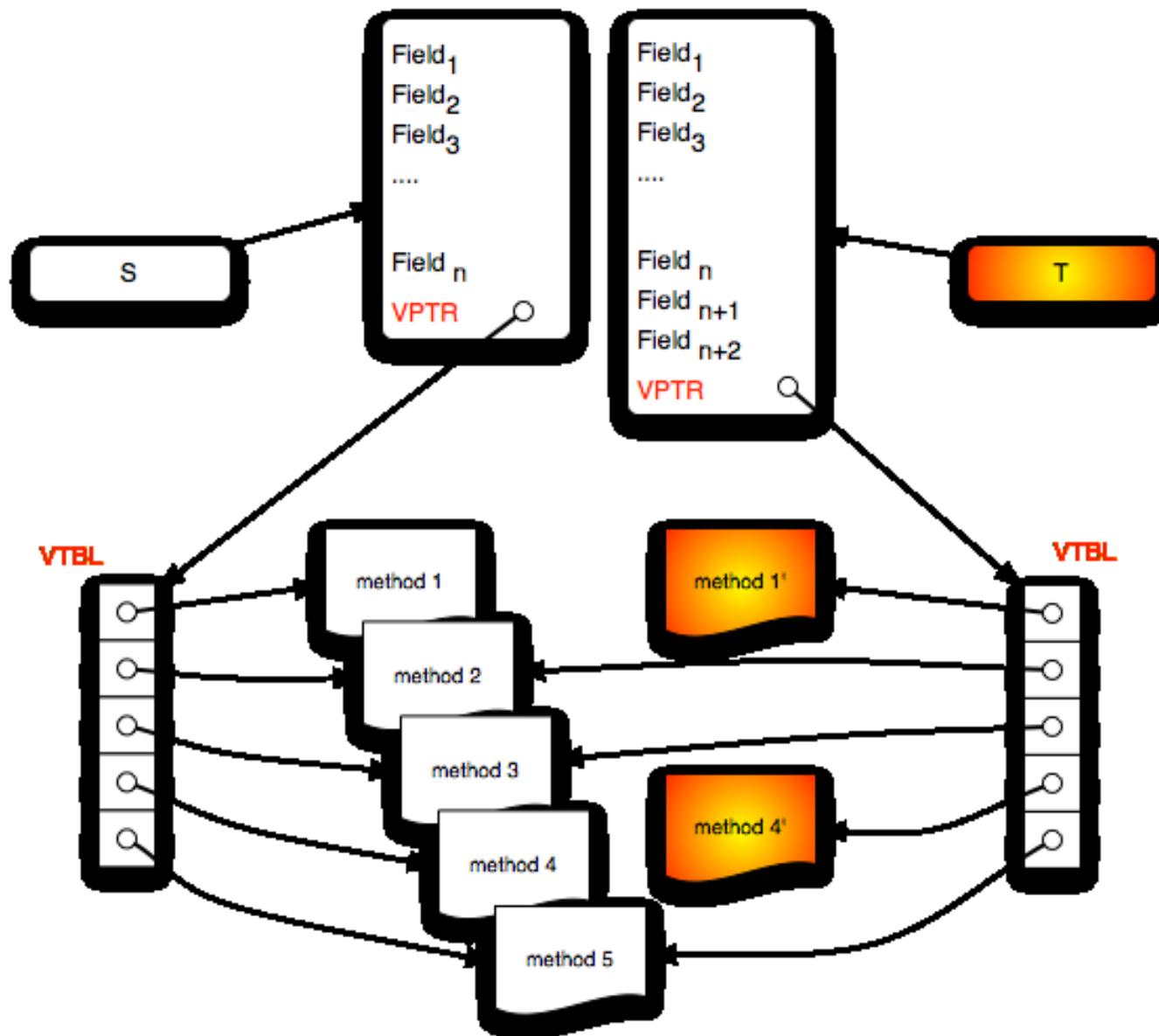


# Class Inheritance

CSE  
4100

# In practice...

**class T extends S { ... }**

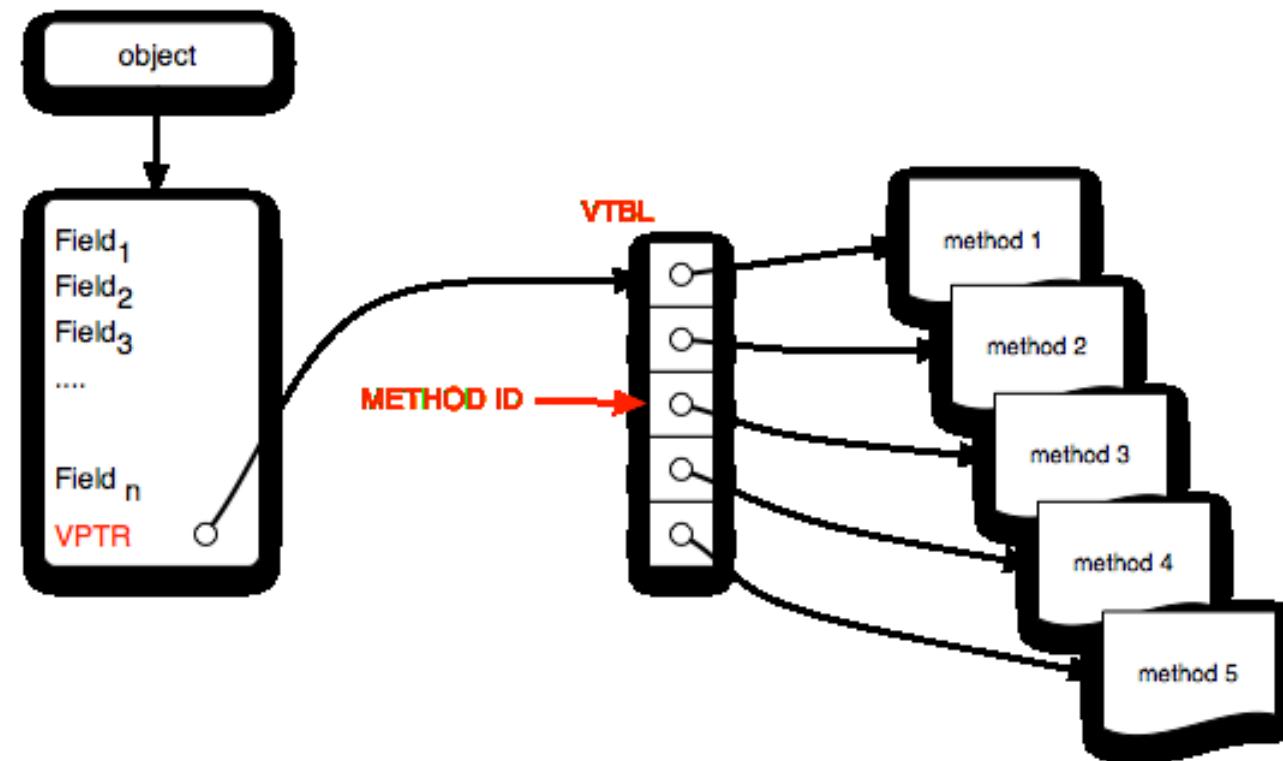


# Method Call

CSE  
4100

## Simple Mechanism

- Retrieve the VPTR for the object
- Index the VTBL with the method ID
- Pick up that “function” and call it



# Complete Example

CSE  
4100

## On C-- ! (test7.cmm)

```
class Foo {  
    int fact(int n) {  
        return 0;  
    }  
    int fib(int x) {  
        return 1;  
    }  
};  
  
class Main extends Foo {  
    Main() {  
        int x;  
        x = fact(5);  
    }  
    int fact(int n) {  
        if (n==0)  
            return 1;  
        else return n * fact(n-1);  
    }  
};
```

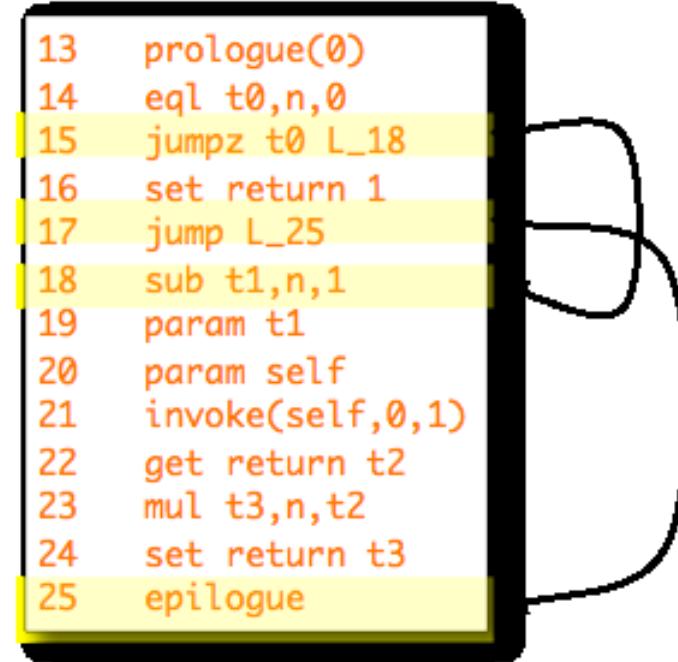
```
D_0=Foo.vtbl[0] = L_0  
        Foo.vtbl[1] = L_3  
  
D_1=Main.vtbl[0] = L_13  
        Main.vtbl[1] = L_3  
        Main.vtbl[2] = L_6  
  
0      prologue(0)  
1      set return 0  
2      epilogue  
3      prologue(0)  
4      set return 1  
5      epilogue  
6      prologue(0)  
7      param 5  
8      param self  
9      invoke(self,0,1)  
10     get return t0  
11     mov x,t0  
12     epilogue  
  
13     prologue(0)  
14     eql t0,n,0  
15     jumpz t0  
L_18  
16     set return 1  
17     jump L_25  
18     sub t1,n,1  
19     param t1  
20     param self  
21  
22     invoke(self,0,1)  
23     get return  
t2  
24     mul t3,n,t2  
25     set return  
t3  
26     epilogue  
param 4  
27     call malloc  
get return  
t0  
29     mov  
0(t0),D_1  
30     param t0  
31  
32     invoke(t0,2,0)  
33     get return
```

# Jumps

CSE  
4100

- Consider the factorial code
  - One conditional jump ( $n==0$ )
  - One unconditional jump (to the exit)
  - Both are forward

```
13  prologue(0)
14  eql t0,n,0
15  jumpz t0 L_18
16  set return 1
17  jump L_25
18  sub t1,n,1
19  param t1
20  param self
21  invoke(self,0,1)
22  get return t2
23  mul t3,n,t2
24  set return t3
25  epilogue
```

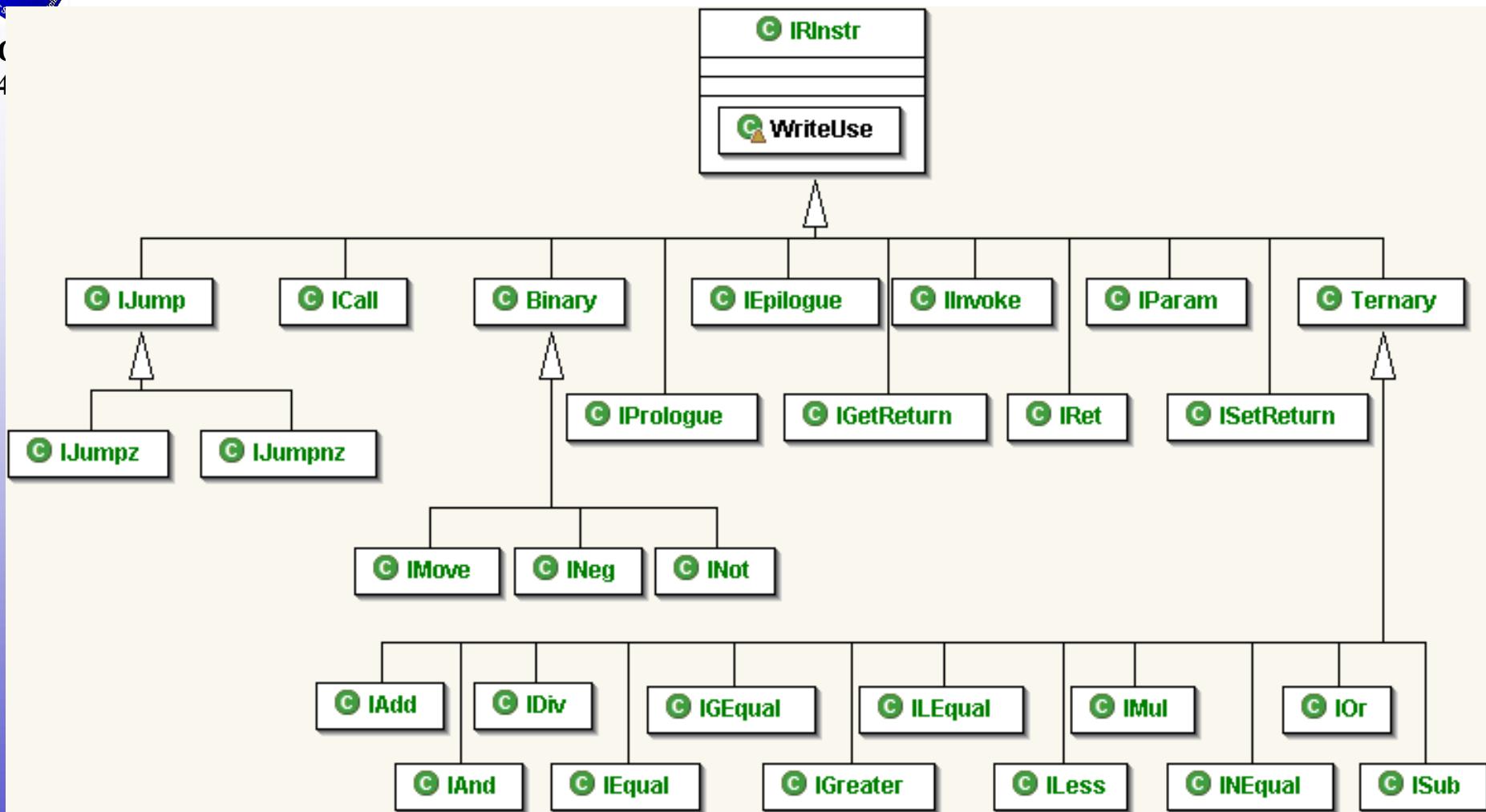


# Implementation

CSE  
4100

- Straightforward
  - One Array of “instructions”
  - Each instruction is polymorphic

# Instruction Hierarchy

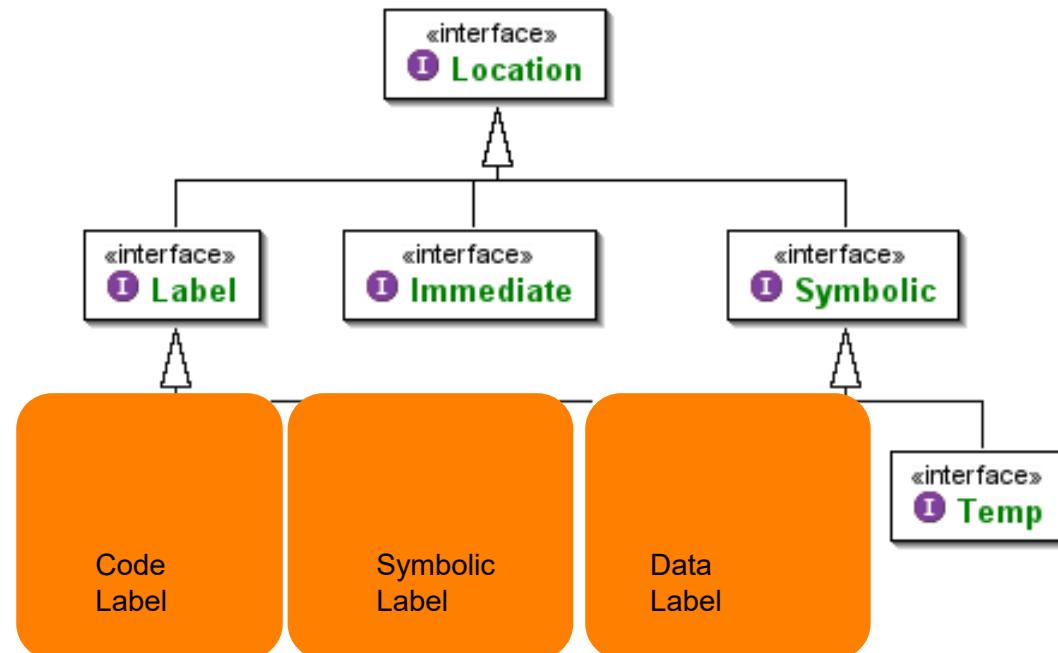


# Location ADT

CSE  
4100

## ○ Purpose

- Represent the various operands
  - For 3-address instructions
  - For branches
- Immediate, Temporaries, Labels, Symbolic





# Location Examples

CSE  
4100

Data Label

```
= Foo.vtbl[0] = L_13          Code Label  
Foo.vtbl[1] = L_3
```

```
D_1= Main.vtbl[0] = L_13  
Main.vtbl[1] = L_3  
Main.vtbl[2] = L_6
```

```
0 prologue(0)  
1 set return 0  
2 epilogue  
...
```

```
13 prologue(0)  
14 eql t0,n,0  
15 jumpz +0 +19          Code Label  
16 set re...  
17 jump L_25  
18 su...          Code Label  
19 param t1  
20 param self  
21 invokeSelf(0, 1)  
22 symbolic  
23 get return t2  
24 mul t3,n,t2  
25 set return t3  
epilog...          Temporary
```



# Dealing With Declarations

- It all depends on the nature of the declaration
  - Class
  - Subclass
  - Instance variable
  - Method
  - Local Variable

# Class Declaration

CSE  
4100

- Actually...
  - Nothing much to do
  - Except
    - Setup the VTBL data structure to hold the code labels
    - Add the VTBL to the data section of the generated code
- Sub-class declaration
  - Same deal!

# Instance Variable [aka Attributes]

CSE  
4100

- Nothing to do
- The only important step will come later on:
  - Reserve space in the class Descriptor based on the type
  - Done during actual code generation
  - Done during IR generation if we need an interpreter

# Method Declaration

- Two simple ideas
  - First: Generate code according to a skeleton
    - Generate a fresh label Lk
    - Generate prologue instruction
    - Generate body
    - Generate epilogue instruction
  - Second, update some data-structures
    - Track label Lk in VTBL for the method ID generated
    - Track the set of temporaries needed by the method
      - Save them in the method descriptor
- What about constructors ?

Lk : prologue  
<body>  
epilogue



CSE  
4100

# Local Variables

- Nothing to do either!
- We will just have to reserve space in the frame for the actual code generation

# All Other Generation

- Always based on a template mechanism
  - (Hence the Grammar style of presentation in the book)



# Assignments

CSE  
4100

## ○ Easy

`id := E`

```
Location l = lookup(id);
Location r = generate(E);
emit(mov l,r);
```



# generate(E)

E1 + E2

```
Location a = generate(E1);
Location b = generate(E2);
Location c = newTEMP();
emit(c := a + b);
return c;
```

E1 - E2

```
Location a = generate(E1);
Location b = generate(E2);
Location c = newTEMP();
emit(c := a - b);
return c;
```

E1 \* E2

```
Location a = generate(E1);
Location b = generate(E2);
Location c = newTEMP();
emit(c := a * b);
return c;
```

E1 / E2

```
Location a = generate(E1);
Location b = generate(E2);
Location c = newTEMP();
emit(c := a / b);
return c;
```



# Literal

CSE  
4100

literal

```
Location a = newIMMEDIATE(literal);  
return a;
```

Very easy! The literal is just a special kind of operand... an immediate one!



# Identifier

CSE  
4100

**id**

```
Location a = newSYMBOLIC(lookup(id));  
return c;
```

# Method Invocation

- Nothing much
  - Generate code to evaluate the arguments & push Them
  - Generate code to evaluate the self reference and push it
  - Generate code to invoke the method
    - Retrieve #arguments
    - Retrieve method identifier
    - Retrieve self reference

# Method Invocation

## O Template

E0.id(E1, ..., En)

```
Location r = newTEMP();
Location obj = generate(E0);
Location an = generate(En);
emit(param an);
...
Location a2 = generate(E2);
emit(param a2);
Location a1 = generate(E1);
emit(param a1);
emit(param obj);
emit(invoker obj, lookupMethod(id), n);
emit(get return r);
return r;
```

# Class Instantiation

CSE  
4100

## ○ Template

- Generate code to allocate space to hold the instance vars.
  - call to malloc
  - setup the VPTR special field in the allocated space (e.g. field[0])
- Generate code for a “normal” method invocation
  - Return value of constructor is pointer to object.

# Array Access

CSE  
4100

- Template depends on array representation
- C-- arrays have this form

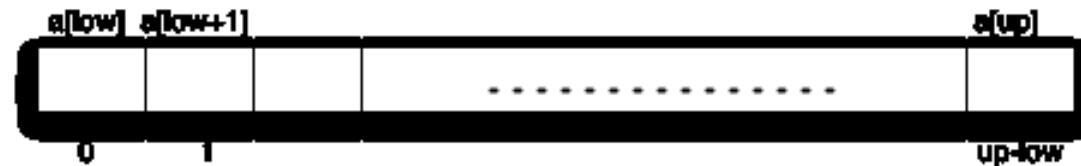
```
TYPE []myArray = new TYPE[low..up];
```

- How does it work?
- 

# Array access

## Basic Idea

- Check bounds [optional step but good practice!]
- Remap to 0-based range
- Scale offset to match a byte displacement
- Do a indexed load based on
  - Based address
  - Computed displacement



$$\text{offset} = (\text{E} - \text{low}) * \text{sizeof}(\text{T})$$



# Array Access Template

CSE  
4100

E0[E1]

```
Location a = generate(E0);
Location e = generate(E1);
Location t0 = makeTEMP();
Location t1 = makeTEMP();
emit(t0 := E - low);
emit(t0 := t0 * sizeof(T));
emit(mov t1, a(t0));
return t1;
```

# Multi-Dimensional Array?

CSE  
4100

## ○ Same Idea

- With a twist...
- Do an induction on the number of dimensions of the array
- Use Horner's rule to evaluate the index.
- Choose a striding for the array
  - 2D array Row major vs. Column major
  - In general (row major)

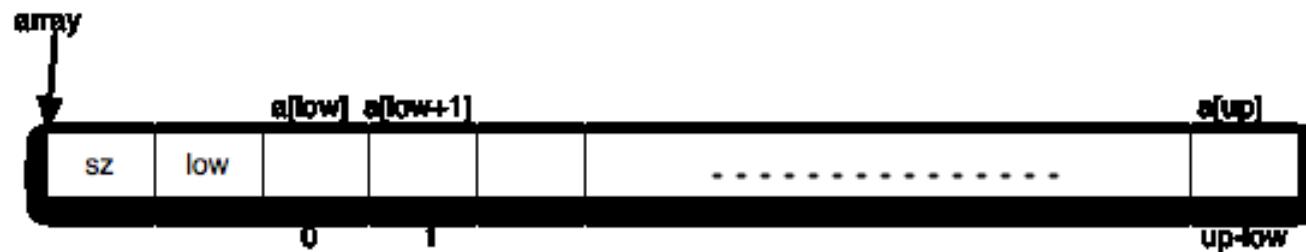
```
TYPE []myArray = new TYPE[low0..up0, low1..up1, ..., lown..upn];  
...  
x = myArray[E0, E1, ... En];
```

$$\begin{aligned} N_0 &= 1 \\ N_{n-1} &= up_{n-1} - low_{n-1} + 1 \\ N_n &= up_n - low_n + 1 \end{aligned}$$

$$\text{offset} = (((E_0 - low_0) * N_0 + E_1 - low_1) * N_1 + E_2 \dots + E_N) * \text{sizeof}(T)$$

# Array Instantiation

- Depend on language complexity
- Assume C--
  - 1 dimension
  - Dynamic (evaluated at runtime) lower and upper bounds



- Questions
  - Why do we store
    - `sz` ?
    - `low` ?

# Revisit Array Access

- Displacement changed!
  - It now sport a new 8 byte increase
    - To skip the sz/low pair

E0[E1]

```
Location a = generate(E0);
Location e = generate(E1);
Location t0 = makeTEMP();
Location t1 = makeTEMP();
emit(t0 := E - low);
emit(t0 := t0 * sizeof(T));
emit(t0 := t0 + 8);
emit(mov t1, a(t0));
return t1;
```

# Array Instantiation

- Similar to class instantiation
  - Generate code to evaluate the bounds for the dimension(s)
    - Low / Up
  - Generate code to compute the size for each dimension
  - Generate code to scale offset
    - From type-based
    - To byte-based
  - Increase the space to account for
    - Size storage
    - Low storage
  - Generate code to call malloc
  - Generate code to store size/low values

# Array Instantiation

CSE  
4100

## Template

```
new TYPE[E1..E2];
```

```
Location l = generate(E1);
Location u = generate(E2);
Location t0 = makeTEMP();
Location t1 = makeTEMP();
Location t2 = makeTEMP();
emit(t0 := l - u);
emit(t0 := t0 + 1);
emit(t0 := t0 * sizeof(T));
emit(t1 := t0 + 8);           // t1 holds the size in bytes
emit(param t0);
emit(call malloc);
emit(get result t2);
emit(mov t2(0),t1);          // save the size
emit(mov t2(4),l);           // save the low (value of E1)
return t2;
```

# Field Access

CSE  
4100

## Template

- ❑ Generate code to get object
- ❑ Generate code to access field (based on its offset)

E0.id

```
Location a = generate(E0);
Location t0 = makeTEMP();
int f0fs = lookupField(Type(E0),id);
emit(mov t0, a(f0fs));
return t0;
```

# Boolean Expressions

- For relational expressions
  - Nothing much about them
    - Generate the code as for arithmetic expression
    - Store result in a temporary
- For boolean connectives
  - Two possibilities
    - Either generate like arithmetic expressions
    - Or generate short-circuit evaluation code

# Short-Circuit Evaluation

CSE  
4100

- Purpose
  - Evaluate the strict minimum
- Example

```
boolean b = . . . && . . .
```

Evaluate First Conjunct

Evaluate Second Conjunct



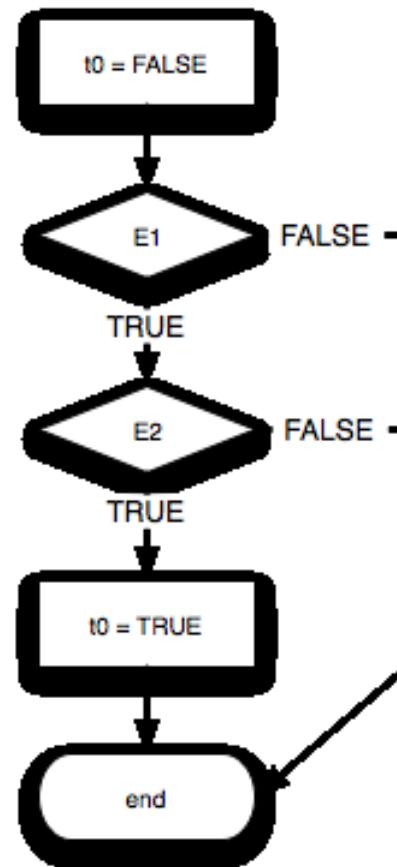
- Motivation
  - Second conjunct only makes sense if first is true.  
IF AND ONLY IF  
The first conjunct is TRUE
- Applies to
  - Conjunction, Disjunction (and any other logical connective)

# Boolean Conjunction

CSE  
4100

## Template

E1 &amp;&amp; E2



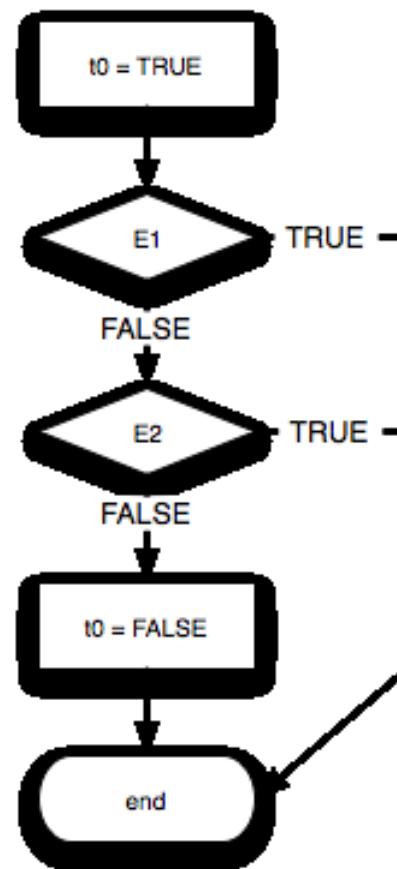
```
Location t0 = makeTEMP();  
emit(mov t0, FALSE);  
Location a = generate(E1);  
emit(jumpz a, exitLabel);  
Location b = generate(E2);  
emit(jumpz b, exitLabel);  
emit(mov t0, TRUE);  
emit(exitLabel: NOOP);  
return t0;
```

# Boolean Disjunction

CSE  
4100

## Template

E1 || E2



```
Location t0 = makeTEMP();  
emit(mov t0, TRUE);  
Location a = generate(E1);  
emit(jumpnz a, exitLabel);  
Location b = generate(E2);  
emit(jumpnz b, exitLabel);  
emit(mov t0, FALSE);  
emit(exitLabel: NOOP);  
return t0;
```

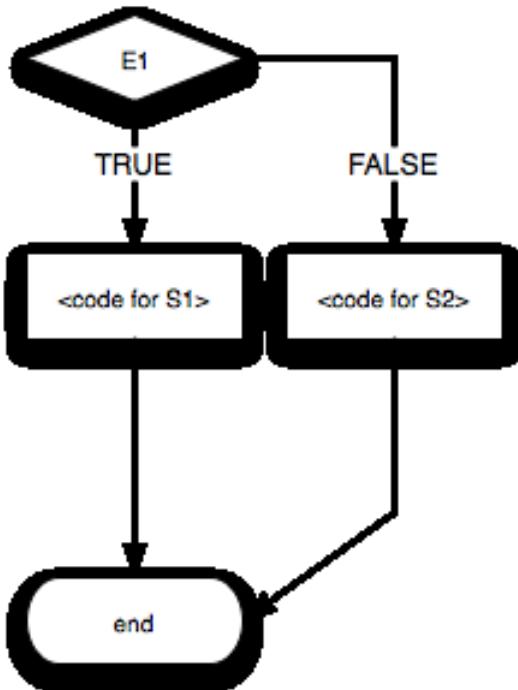
# If-Then-Else

CSE  
4100

## Simple Template

```
if E1  
then S1  
else S2
```

```
Location a = generate(E1);  
emit(jumpz a,elseLabel);  
generate(S1);  
emit(jump endLabel);  
emit(elseLabel:);  
generate(S2);  
emit(endLabel:);
```

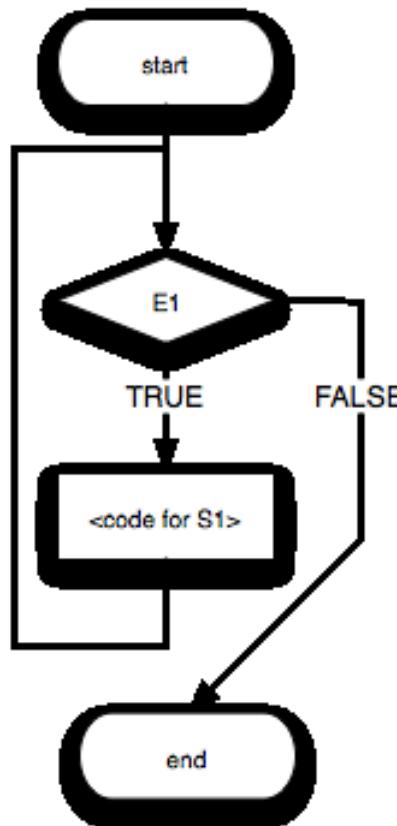


# While

## ○ Same Idea!

```
while (E1)
    S1
```

```
emit(startLabel:);
Location a = generate(E1);
emit(jumpz a,endLabel);
generate(S1);
emit(jump startLabel);
emit(endLabel:);
```





# Switch-case

CSE  
4100

- How can we handle this ?

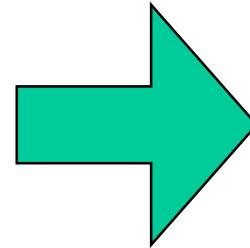
```
switch(E1) {  
    case c1: S1;  
    case c2: S2;  
    ...  
    case cn: Sn;  
    default: Sd;  
}
```

???

# Switch Statement

- The old-fashioned way
  - As a syntactic rewrite!

```
switch(E1) {  
    case c1: S1;  
    case c2: S2;  
    ...  
    case cn: Sn;  
    default: Sd;  
}
```



```
Temporary t = E1;  
if (t == c1) S1;  
else if (t == c2) S2;  
else if (t == c3) S3;  
....  
else if (t == cn) Sn;  
else Sd;
```

# Switch Statement

## Table Driven

```
switch(E1) {  
    case c1: S1;  
    case c2: S2;  
    ...  
    case cn: Sn;  
    default: Sd;  
}
```

```
Labels[] switchTable= {L1,L2,...,Ln};  
Temporary t = E1;  
jump switchTable[t];  
L1: S1;  
    jump end;  
L2: S2;  
    jump end;  
...  
Ln: Sn;  
    jump end;  
Ld: Sd  
end:
```



## What have we gained ?

# Back-patching vs symbolic Labels

CSE  
4100

## ○ Motivation

- Sometimes we emit
  - Backward jumps
  - Forward jumps
- Why are forward jumps “difficult” ?

## ○ Example. A while loop

```
...  
125  S1  
126  S2  
127  JUMP  222  
128  ...  
129  S4  
...  
187  JUMP 125
```

We don't know the target  
of the jump yet!

This jump is backward, we  
already know the target

# Concluding Remarks/Looking Ahead

- Intermediate Code Generation can be Achieved by
  - Attribute Grammars that Generate “String” of Code
  - Attribute Grammars that Emit 3AC
  - Directly from AST Following 3AC Rules and Templates that Match Desired “Logic”
- Process must also Consider:
  - Generation of Variables/Storage
  - Handling of Temporaries
- Looking Ahead:
  - Optimization – Since ICG Does Lousy Job!
  - Final Code Generation (Relocatable to Target)
  - Final Exam Review