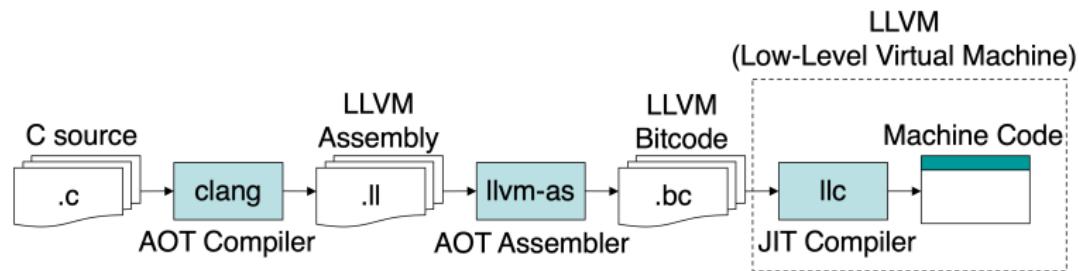


Code Generation

Multiple IRs in the Compiler

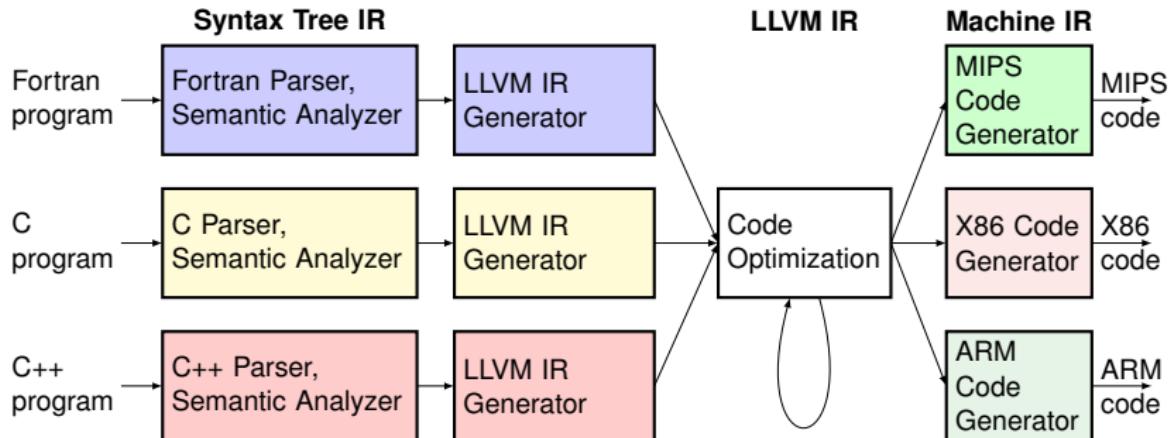
Modern Compiler Framework (Clang/LLVM)

- ❑ Remember this diagram from our first day?



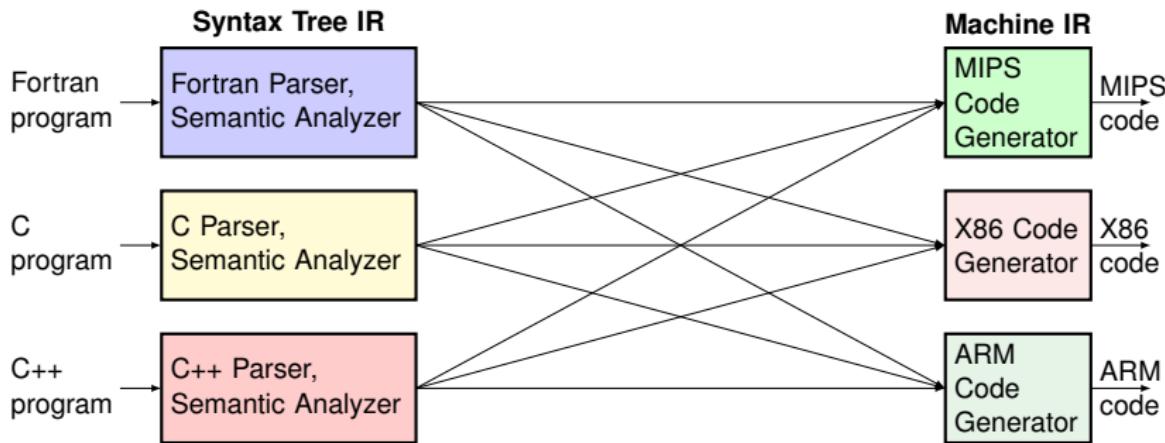
- ❑ LLVM Bitcode is in LLVM IR (Intermediate Representation)

Modern Compiler Framework (Clang/LLVM)



- ❑ Common LLVM IR for all languages and backends means:
 - Code optimizations need to be written only once
 - Implementation complexity of $O(M + N)$ instead of $O(M * N)$ (where $M = \text{number of frontends}$, $N = \text{number of backends}$)

Why $O(M * N)$ when no common IR?



- ❑ Must translate M languages to N machine codes
 - Must also do optimizations during each of these translations

High-Level IRs

- ❑ Goal: Express the syntax and semantics of source code
- ❑ Examples: Abstract Syntax Tree, Parse Tree
- ❑ Differs on: Source code programming language
- ❑ Uses:
 - Generated by syntax analysis
 - Used by semantic analysis for binding and type checking
 - Language-specific optimizations

High-Level IRs: Language-specific Optimizations

- ❑ IR expresses language constructs specific to language
→ Suitable for language-specific optimizations

High-Level IRs: Language-specific Optimizations

- ❑ IR expresses language constructs specific to language
 - Suitable for language-specific optimizations
- ❑ E.g. Devirtualization for languages with polymorphism
 - Polymorphic method calls are indirect jumps using a vtable (Vtable: a table of function pointers specific to each class)
 - **Devirtualization:** changing polymorphic calls to direct calls
 - If runtime object type can be proven (using type inference)
 - Indirect call can be changed to direct call
 - Direct call can sometimes be inlined into caller method

High-Level IRs: Language-specific Optimizations

- ❑ IR expresses language constructs specific to language
 - Suitable for language-specific optimizations
- ❑ E.g. Devirtualization for languages with polymorphism
 - Polymorphic method calls are indirect jumps using a vtable (Vtable: a table of function pointers specific to each class)
 - **Devirtualization:** changing polymorphic calls to direct calls
 - If runtime object type can be proven (using type inference)
 - Indirect call can be changed to direct call
 - Direct call can sometimes be inlined into caller method
- ❑ E.g. Type specialization for languages with dynamic typing
 - Naively, all ops on variables needs dynamic type checking (A big switch/case statement with code for each type)
 - **Type specialization:** generating code for just one type
 - By proving runtime type of variable (using type inference)
 - By profiling type at runtime and generating code for that type

Low-Level IRs

- ❑ Goal: Express code in the ISA of an abstract machine
- ❑ Examples:
 - Three address code
 - Static Single Assignment (SSA) code
- ❑ Differs on: Language and back-end machine agnostic
- ❑ Uses:
 - A common IR that connects front-ends and back-ends
 - Language / machine independent optimizations

Low-Level IRs: Universal Optimizations

- ❑ IR is language and machine agnostic
 - Do optimizations common to all languages/machines

Low-Level IRs: Universal Optimizations

- ❑ IR is language and machine agnostic
 - Do optimizations common to all languages/machines

- ❑ **Dead code elimination:** deletes unused code
 - $A=2; A=y;$ \equiv $A=y;$

Low-Level IRs: Universal Optimizations

- ❑ IR is language and machine agnostic
 - Do optimizations common to all languages/machines

- ❑ **Dead code elimination:** deletes unused code
 $A=2; A=y;$ \equiv $A=y;$
- ❑ **Redundancy elimination:** deletes repeated computation
 $A=x+y; B=x+y+z;$ \equiv $A=x+y; B=A+z$

Low-Level IRs: Universal Optimizations

- ❑ IR is language and machine agnostic
 - Do optimizations common to all languages/machines

- ❑ **Dead code elimination:** deletes unused code
 - $A=2; A=y;$ \equiv $A=y;$

- ❑ **Redundancy elimination:** deletes repeated computation
 - $A=x+y; B=x+y+z;$ \equiv $A=x+y; B=A+z$

- ❑ **Loop invariant code motion:** moves code out of loops
 - $\text{for } (i = 0; i < 100; i++) \{ \text{sum} += i + \textcolor{blue}{x * y}; \}$
 \equiv
 $\textcolor{blue}{t = x * y;}$
 $\text{for } (i = 0; i < 100; i++) \{ \text{sum} += i + \textcolor{blue}{t}; \}$

Machine IRs

- ❑ Goal: Generate code in the ISA of back-end machine
- ❑ Examples: x86 IR, ARM IR, MIPS IR
- ❑ Differs on: Back-end machine ISA
- ❑ Uses:
 - Register allocation / machine code generation
 - Machine-specific optimizations

Machine IRs: Machine-specific Optimizations

- ❑ IR expresses instructions specific to machine
 - Suitable for machine-specific optimizations

Machine IRs: Machine-specific Optimizations

- ❑ IR expresses instructions specific to machine
→ Suitable for machine-specific optimizations

- ❑ **Strength reduction:** replacing op with cheaper op
 $A=2^*a;$ \equiv $A=a\ll 1;$

Machine IRs: Machine-specific Optimizations

- ❑ IR expresses instructions specific to machine
→ Suitable for machine-specific optimizations

- ❑ **Strength reduction:** replacing op with cheaper op
 $A=2*a;$ \equiv $A=a\ll 1;$

- ❑ **Vectorization:** using CPU vector units to parallelize

```
for (i = 0; i < 64; i++) { C[i] = A[i] + B[i]; }
```


 \equiv

```
vec_add_64    vec_register, A, B
vec_store_64  C, vec_register
```

Low-Level IRs

Three Address Code

❑ In the form of $X = Y \text{ op } Z$ where X, Y, Z can be:

- variables, constants, temporaries
- temporaries: compiler-generated variables
(holds intermediate values for long expressions)

❑ An example:

$x * y + z / w$

is translated to

$t1 = x * y$; t1, t2, t3 are temporary variables

$t2 = z / w$

$t3 = t1 + t2$

- Internal nodes in AST are translated to temporary variables
- Can be generated through a depth-first traversal of AST

Three Address Code: Ops for an abstract machine

❑ Characteristics

- Long expressions are broken down to three address ops
- Control flow statements are converted to jumps
- Designed to be machine independent
 - Operators limited to those available on all machines
 - Function calls represented as generic call ops
 - Uses **variables** rather than **registers** to store values
(Variables are assigned to registers in machine code)

❑ Why this form?

- Boils language-specific ASTs down to actual computation
- Optimizations done on abstract machine are pertinent
(abstract machine ops are sufficiently similar to real ISAs)

Common Three-Address Statements (I)

- ❑ Binary operation statement:

$x = y \text{ op } z$

where op is an arithmetic or logical operation (binary operation)

- ❑ Unary operation statement:

$x = \text{op } y$

where op is an unary operation such as -, not, shift)

- ❑ Copy statement:

$x = y$

- ❑ Unconditional jump statement:

$\text{goto } L$

where L is label

Common Three-Address Statements (II)

- ❑ Conditional jump statement:

if (x relop y) goto L

where relop is a relational operator such as =, \neq , $>$, $<$

- ❑ Procedural call statement:

param x₁, ..., param x_n, call F_y, n

As an example, foo(x1, x2, x3) is translated to

param x₁

param x₂

param x₃

call foo, 3

- ❑ Procedural call return statement:

return y

where y is the return value (if applicable)

Common Three-Address Statements (III)

- ❑ Indexed assignment statement:

x = y[i]

or

y[i] = x

where x is an int and y is an array variable

- ❑ Address and pointer operation statement:

x = & y ; a pointer x is set to the address of y

y = * x ; y is set to value contained in the location
; pointed to by x

***y = x** ; location addressed by y gets value x

Static Single Assignment (SSA)

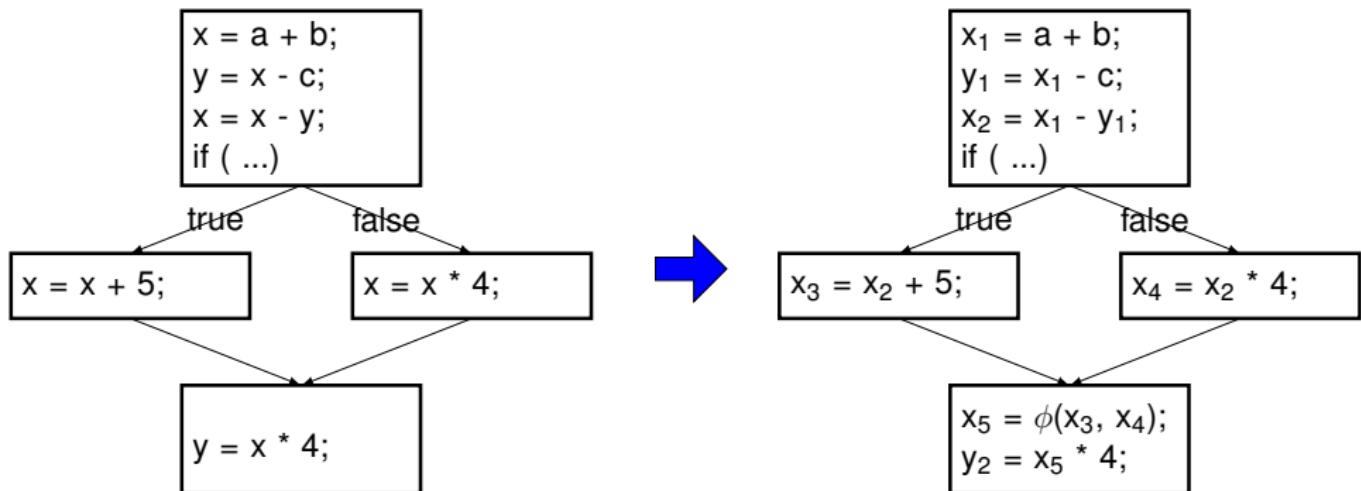
- ❑ Most modern IRs are both three address and SSA
 - Clang C/C++ Compiler LLVM IR
 - GCC C/C++ Compiler GIMPLE IR:
 - OpenJDK Java JIT-Compiler Ideal IR
 - Chrome V8 JavaScript JIT-Compiler TurboShaft IR

- ❑ Coined by Rosen, Wegman, Zadeck in "Global value numbers and redundant computations", 1988
 - Every variable is assigned to exactly once
 - Variable gains a version number whenever it is assigned to:

```
int x, y;  
x1 = 1;           // Current version of x is 1  
y1 = x1 * 2;    // Uses version 1 of x  
x2 = x1 + 1;    // Current version of x is 2  
y2 = x2 * 2;    // Uses version 2 of x
```

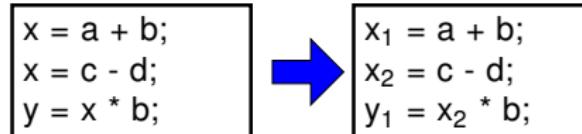
SSA example with control flow

- ❑ What if versions are different on control flow merge?
Φ-function combines two versions into one new version



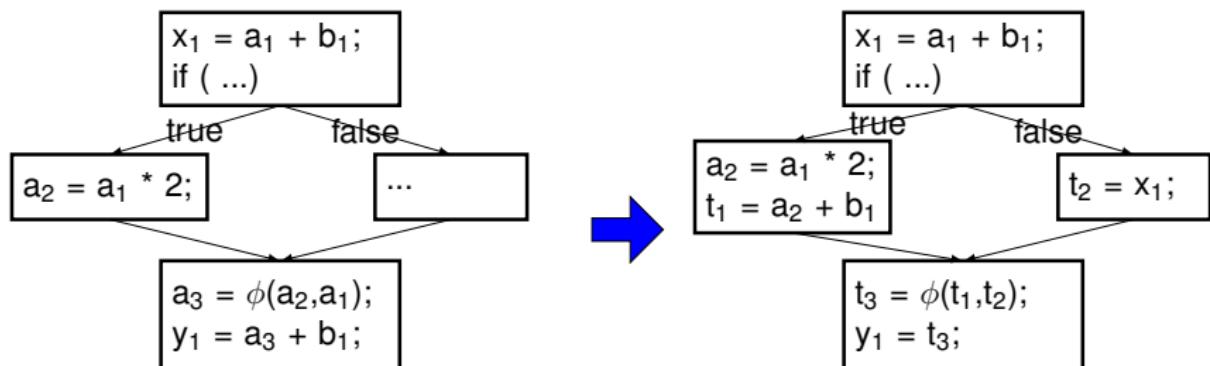
SSA helps compiler optimizations

- Dead Code Elimination (DCE):



.... x_1 is defined but never used, it is safe to remove

- Partial Redundancy Elimination (PRE):



Redundant $a + b$ computation on false branch is removed

Why does SSA help optimizations?

- ❑ Data dependencies between instructions are made explicit
 - Variables with same name guaranteed to have same value
- ❑ Without SSA, same name does not mean same value
 - Must maintain data dependence graph to express this info
- ❑ We will discuss more in **compiler optimization** phase

Laying Out Memory

Layout of Variables in Stack Memory

- ❑ Local variables of a function are allocated on stack frame
 - Maintain **offset** from base of frame to allocate next variable
 - $\text{address}(x) \leftarrow \text{offset}$
 - $\text{offset} += \text{sizeof}(x.\text{type})$

```
void foo() {  
    int a;  
    int b;  
    long long c;  
    int d;  
}
```

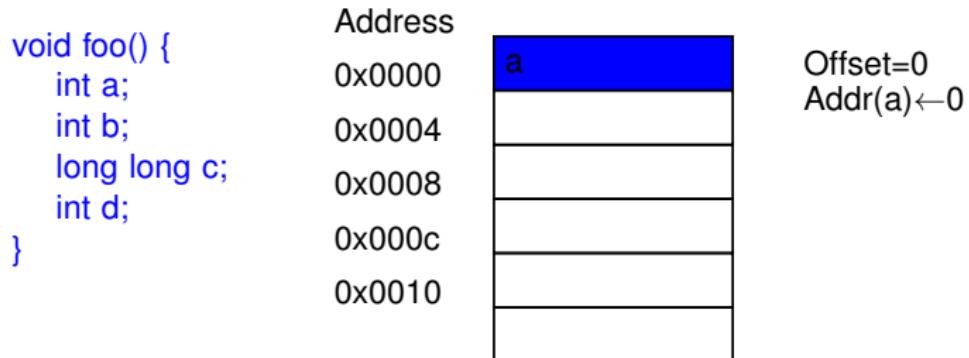
Layout of Variables in Stack Memory

- ❑ Local variables of a function are allocated on stack frame
 - Maintain **offset** from base of frame to allocate next variable
 - $\text{address}(x) \leftarrow \text{offset}$
 - $\text{offset} += \text{sizeof}(x.\text{type})$

	Address	
void foo() {	0x0000	
int a;	0x0004	
int b;	0x0008	
long long c;	0x000c	
int d;	0x0010	
}		Offset=0

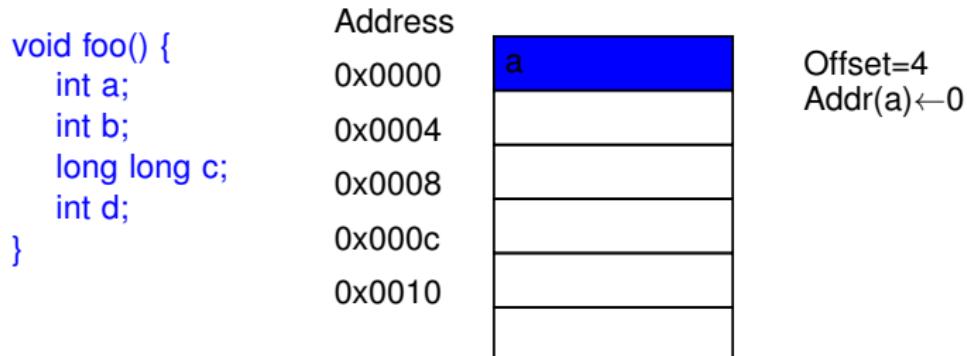
Layout of Variables in Stack Memory

- ❑ Local variables of a function are allocated on stack frame
 - Maintain **offset** from base of frame to allocate next variable
 - $\text{address}(x) \leftarrow \text{offset}$
 - $\text{offset} += \text{sizeof}(x.\text{type})$



Layout of Variables in Stack Memory

- ❑ Local variables of a function are allocated on stack frame
 - Maintain **offset** from base of frame to allocate next variable
 - $\text{address}(x) \leftarrow \text{offset}$
 - $\text{offset} += \text{sizeof}(x.\text{type})$



Layout of Variables in Stack Memory

- ❑ Local variables of a function are allocated on stack frame
 - Maintain **offset** from base of frame to allocate next variable
 - $\text{address}(x) \leftarrow \text{offset}$
 - $\text{offset} += \text{sizeof}(x.\text{type})$

	Address	
void foo() {	0x0000	a
int a;	0x0004	b
int b;	0x0008	
long long c;	0x000c	
int d;	0x0010	
}		

Offset=8
Addr(a)←0
Addr(b)←4

Layout of Variables in Stack Memory

- ❑ Local variables of a function are allocated on stack frame
 - Maintain **offset** from base of frame to allocate next variable
 - $\text{address}(x) \leftarrow \text{offset}$
 - $\text{offset} += \text{sizeof}(x.\text{type})$

	Address	
void foo() {	0x0000	a
int a;	0x0004	b
int b;	0x0008	c
long long c;	0x000c	c
int d;	0x0010	
}		

Offset=16
Addr(a)←0
Addr(b)←4
Addr(c)←8

Layout of Variables in Stack Memory

- ❑ Local variables of a function are allocated on stack frame
 - Maintain **offset** from base of frame to allocate next variable
 - $\text{address}(x) \leftarrow \text{offset}$
 - $\text{offset} += \text{sizeof}(x.\text{type})$

	Address	
void foo() {	0x0000	a
int a;	0x0004	b
int b;	0x0008	c
long long c;	0x000c	c
int d;	0x0010	d
}		

Offset=20
Addr(a)←0
Addr(b)←4
Addr(c)←8
Addr(d)←16

What if function has nested scopes?

- ❑ Let's take the below example code:

```
void foo() {  
    int a;  
    int b;  
    {  
        int i;  
    }  
    {  
        {  
            int j;  
        }  
        int k;  
    }  
}
```

- ❑ What is address(k)? 16?

Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
    }  
  
    {  
        {  
            int j;  
        }  
        int k;  
    }  
}
```

Symbol
Table Stack Offset
Stack

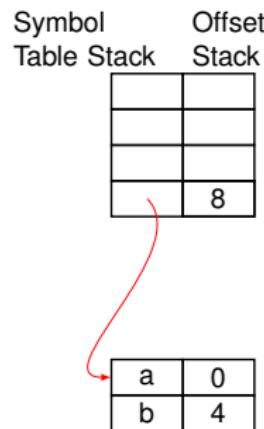
	8



a	0
b	4

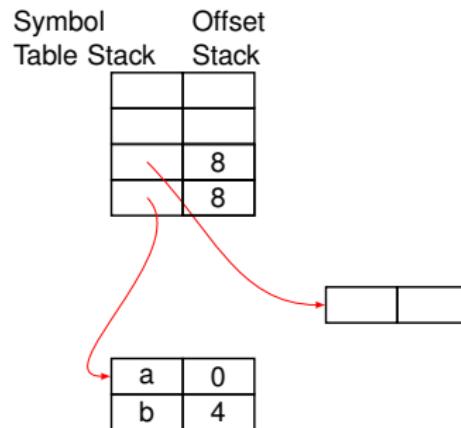
Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    {  
        {  
            int j;  
        }  
        int k;  
    }  
}
```



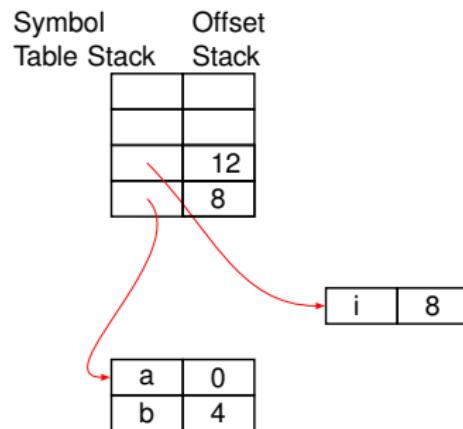
Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    {  
        {  
            int j;  
        }  
        int k;  
    }  
}
```



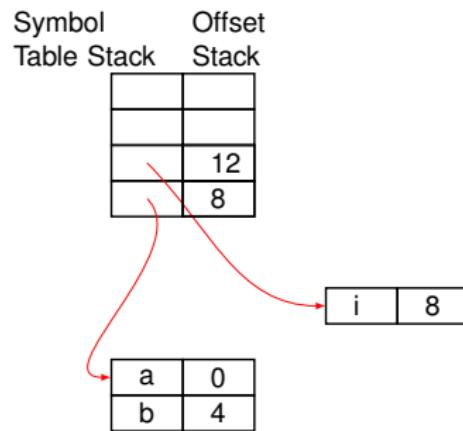
Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    {  
        {  
            int j;  
        }  
        int k;  
    }  
}
```



Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    { check point #3  
        {  
            int j;  
        }  
        int k;  
    }  
}
```



Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    { check point #3  
        {  
            int j;  
        }  
        int k;  
    }  
}
```

Symbol Table Stack Offset Stack

	8



a	0
b	4

Nested Scopes Example

```
void foo() {  
    int a;  
    int b;
```

check point #1

```
{  
    int i;  
    check point #2  
}
```

check point #3

```
{  
    int j;  
}
```

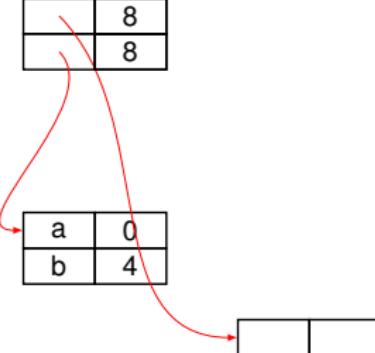
```
int k;
```

```
}
```

Symbol Table Stack Offset Stack

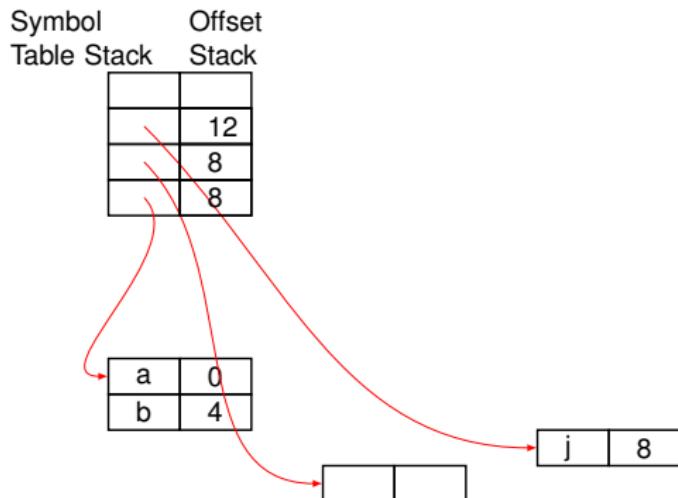
	8
	8

a	0
b	4



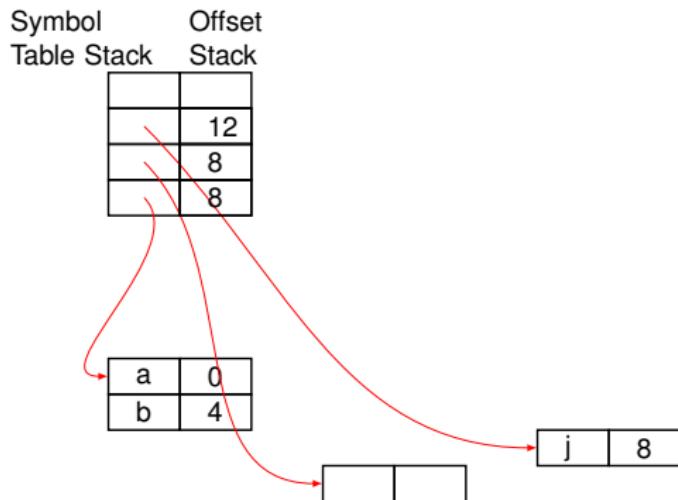
Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    { check point #3  
        {  
            int j;  
            check point #4  
        }  
        int k;  
    }  
}
```



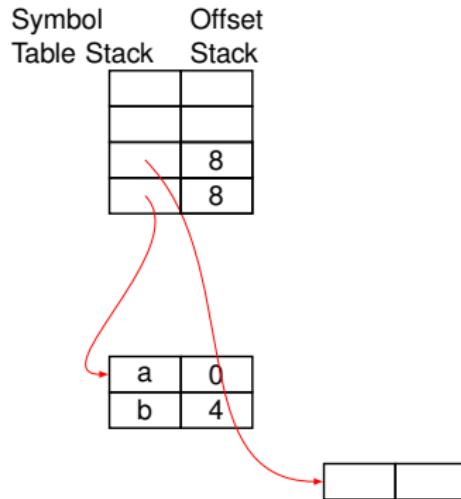
Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    { check point #3  
        {  
            int j;  
            check point #4  
        }  
        int k;  
        check point #5  
    }  
}
```



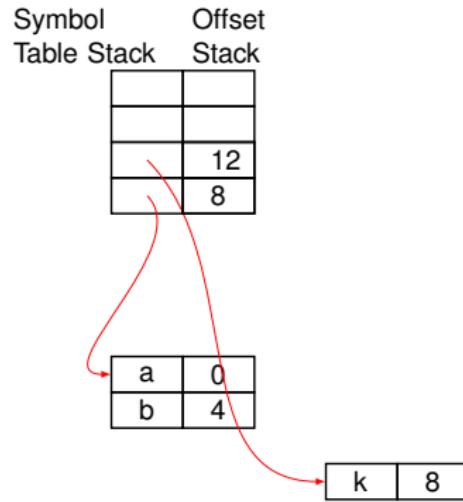
Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    { check point #3  
        {  
            int j;  
            check point #4  
        }  
        int k;  
        check point #5  
    }  
}
```



Nested Scopes Example

```
void foo() {  
    int a;  
    int b;  
    check point #1  
    {  
        int i;  
        check point #2  
    }  
  
    { check point #3  
        {  
            int j;  
            check point #4  
        }  
        int k;  
        check point #5  
    }  
}
```



Consideration 1: Allocation Alignment

- ❑ Enforce **addr(var) mod sizeof(memory word) == 0**
 - Memory word: unit of memory access in given CPU
 - If not, need to load two words and shift & concatenate

```
void foo() {  
    char a;          // addr(a) = 0;  
    int b;           // addr(b) = 4; /* instead of 1 */  
    int c;           // addr(c) = 8;  
    long long d;   // addr(d) = 16; /* instead of 12 */  
}
```

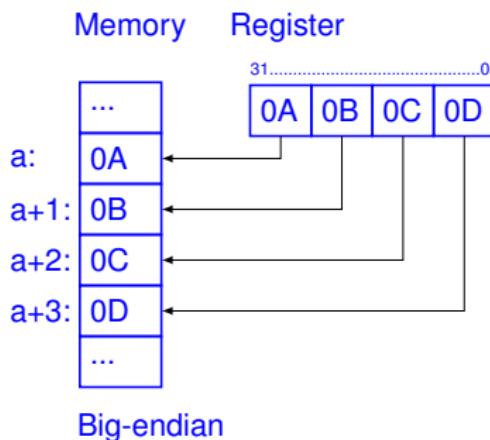
- ❑ This makes memory layout backend machine dependent
 - Memory layout made explicit only in Machine IR
 - Low-level IR needs to refer to locations in an abstract way

Consideration II: Endianness



Endianness

- **Big endian:** MSB (most significant byte) in lowest address
- **Little endian:** LSB (least significant byte) in lowest address

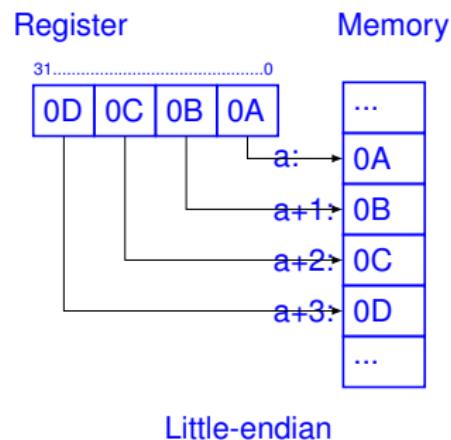
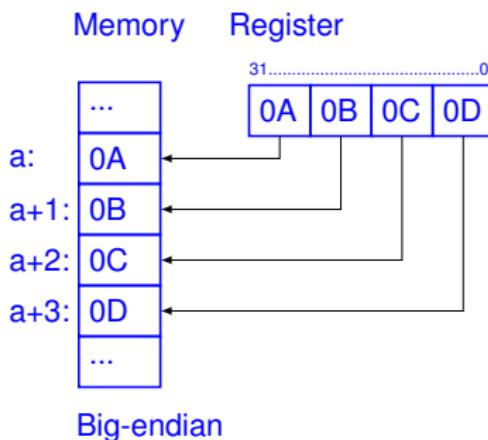


Consideration II: Endianness



Endianness

- **Big endian:** MSB (most significant byte) in lowest address
- **Little endian:** LSB (least significant byte) in lowest address



How about other memory besides stack memory?

❑ Static Memory

- Where global variables and other static variables reside
- Layout variables from base address in the same way

❑ Heap Memory

- Where dynamically allocated memory using malloc reside
- Handled by runtime memory management library
- Compiler not much to do with how this memory is laid out

Generating IR

Generating IR from Language Constructs

- ❑ Goal: translate **language constructs** in syntax tree to IR
- ❑ Two ways to implement semantic rules for translation
 - By depth-first traversal of syntax tree built by parser
 - This is what you are doing for Project 4
 - By a **syntax directed translation scheme**
 - This is what we will discuss now (based on LR parser)
 - But most concepts apply to both implementations

Generating IR from Language Constructs

- ❑ Goal: translate **language constructs** in syntax tree to IR
- ❑ Two ways to implement semantic rules for translation
 - By depth-first traversal of syntax tree built by parser
 - This is what you are doing for Project 4
 - By a **syntax directed translation scheme**
 - This is what we will discuss now (based on LR parser)
 - But most concepts apply to both implementations
- ❑ What language structures do we need to translate?
 - Declarations
 - Variables, functions (parameters and return types), ...
 - Statements
 - Assignment statements
 - Function call statements
 - Control flow statements (if-then-else, for/while loops)
 - Expressions
 - $x + y$, $x - y$, $x < y$, $x > y$, $x == y$, ...

Attributes to Evaluate in Translation

❑ Variable declaration:

T V e.g. int a,b,c;

- Type information **T.type T.width**
- Variable information **V.type, V.offset**

❑ Statement **S**

- **S.code**: synthesized attribute that holds IR code of S

❑ Expression **E**

- **E.code**: synthesized attribute that holds IR code for E
- **E.place**: temporary variable name to store result of E

Processing Declarations

- ❑ Translating declarations in a single scope
 - **enter(name, type, offset)**: insert variable into symbol table

$S \rightarrow M\ D$

$M \rightarrow \epsilon$

{ offset=0; } /* reset offset before layout */

$D \rightarrow D\ D$

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

$T \rightarrow \text{integer}$ { T.type=integer; T.width=4; }

$T \rightarrow \text{real}$ { T.type=real; T.width=8; }

$T \rightarrow T1[\text{num}]$ { T.type=array(num.val, T1.type);
T.width=num.val * T1.width; }

$T \rightarrow * T1$ { T.type=ptr(T1.type); T.width=4; }

Processing Declarations in Nested Scopes

❑ Translating declarations in nested scopes

- **push(item, stack)**: Pushes item on to stack
- **pop(stack)**: Pops item at the top of stack
- **top(stack)**: Returns item at the top of stack

$S \rightarrow M\ D$

{ pop(tblptr); pop(offset); }

$M \rightarrow \epsilon$

{ t=mktable(nil); push(t, tblptr); push(0,offset); }

$D \rightarrow D\ D$

$D \rightarrow \{ N\ D \}$

{ pop(tblptr); pop(offset); }

$N \rightarrow \epsilon$

{ t=mktable(nil); push(t, tblptr); push(top(offset), offset); }

$D \rightarrow T\ id;$

{ enter(id.name, T.type, top(offset));
top(offset) = top(offset)+ T.width; }

Processing Statements

- ❑ Statements rely on symbol table populated by declarations
 - **lookup(id)**: search id in symbol table, return nil if none
 - **emit(code)**: print three address IR for code
 - **newtemp()**: get a new temporary variable (or register)
 - E.g., in SSA form, it returns the next virtual register number

```
int newtemp() { return virtual_register++; }
```

S → id = E { P=lookup(id); if (P==nil) perror(...); else emit(P '=' E.place);}

E → E₁ + E₂ { E.place = newtemp(); emit(E.place '=' E₁.place '+' E₂.place); }

E → E₁ * E₂ { E.place = newtemp(); emit(E.place '=' E₁.place '*' E₂.place); }

E → - E₁ { E.place = newtemp(); emit(E.place '=' '-' E₁.place); }

E → (E₁) { E.place = E₁.place; }

E → id { P=lookup(id); E.place=P; }

Processing Array References

❑ Recall generalized row/column major addressing

❑ For example:

1-dimension: int $x[100]$; $x[i_1]$

2-dimension: int $x[100][200]$; $x[i_1][i_2]$

3-dimension: int $x[100][200][300]$; $x[i_1][i_2][i_3]$

❑ Row major: address of a k-dimension array item

1-dimension: $A_1 = \text{base} + a_1 * \text{width}$ $a_1 = i_1$

2-dimension: $A_2 = \text{base} + a_2 * \text{width}$ $a_2 = a_1 * N_2 + i_2$

3-dimension: $A_3 = \text{base} + a_3 * \text{width}$ $a_3 = a_2 * N_3 + i_3$

...

k-dimension: $A_k = \text{base} + a_k * \text{width}$ $a_k = a_{k-1} * N_k + i_k$

Processing Array References

❑ Processing an array reference (e.g. $A[i]$, $A[i][j]$, ...)

- **L.place**: temporary variable name to store a_k
- **L.base, L.width, L.bounds**:
base address, element width, upper bounds of array
- **L.dim**: current dimension (the k in a_k)

$E \rightarrow L$ { E.place = newtemp(); t= newtemp();
 emit(t '=' L.place '*' L.width);
 emit (E.place '=' '*' (L.base +' t)); }

$L \rightarrow id [E]$ { L.base = lookup(id).base; L.width = lookup(id).width;
 L.bounds = lookup(id).bounds; L.dim=1;
 L.place = E.place; }

$L \rightarrow L_1 [E]$ { L.base = L_1 .base; L.width = L_1 .width;
 L.bounds = L_1 .bounds; L.dim = L_1 .dim + 1;
 L.place = newtemp();
 emit(L.place '=' L_1 .place '*' L.bounds[L.dim]);
 emit(L.place '=' L.place +' E.place); }

Processing Boolean Expressions

❑ Boolean expression: **a op b**

- where op can be <, >, >=, &&, ||, ...

1. Without *short circuiting*

- Short circuiting:

- In expression A && B, not evaluating B when A is false
- In expression A || B, not evaluating B when A is true

- Without short circuiting, entire expression is evaluated:

$$S \rightarrow id = E$$
$$\equiv \text{lookup}(id) = E.place$$
$$E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f)$$
$$\equiv t1 = a < b$$
$$t2 = c < d$$
$$t3 = e < f$$
$$t4 = t2 \& t3$$
$$E.place = t1 || t4$$

Processing Boolean Expressions

2. With short circuiting (e.g. C/C++/Java)

➤ Processing simple boolean expressions:

- **E.true**: address of code to execute on 'true'
- **E.false**: address of code to execute on 'false'
- **S.next**: address of code after S

$S \rightarrow \text{if } E \text{ then } S_1$

$E \rightarrow a < b \quad \equiv \quad \begin{aligned} E.\text{true} &= \text{code for } S_1; \\ E.\text{false} &= S.\text{next}; \\ \text{emit(if } E \text{ goto } E.\text{true);} \\ \text{emit(goto } E.\text{false);} \end{aligned}$

➤ Processing compound boolean expressions:

- Chain together multiple of above by updating E.true/E.false
- $E \rightarrow E_1 \&& E_2$: $E_1.\text{true} = \text{code for } E_2$, $E_1.\text{false} = S.\text{next}$
- $E \rightarrow E_1 || E_2$: $E_1.\text{false} = \text{code for } E_2$, $E_1.\text{true} = \text{code for } S_1$

Processing Boolean Expressions

2. With short circuiting (cont'd)

- A short circuited compound boolean expression

$S \rightarrow \text{if } E \text{ then } S_1$

$E \rightarrow (a < b) \text{ or } (c < d \text{ and } e < f) \equiv$

E.true = code for S_1 ;

E.false = $S.\text{next}$;

emit(

if ($a < b$) goto E.true

goto L1

L1: if ($c < d$) goto L2

goto E.false

L2: if ($e < f$) goto E.true

goto E.false

);

- **Problem: E.true, E.false, S.next are non-L-attributes**

- Depend on code that has not been generated yet

E.true: Only available when S_1 is generated

E.false: Only available when code after S is generated

- Emitting any **forward jump** poses this problem

Syntax Directed Translation (SDT)

- ❑ Non-L-attributes complicates syntax directed translation
 - Even using parse tree, preclude simple left-to-right traversal
- ❑ Solutions: two methods
 - Two pass approach — process the code twice
 - Generate labels in the first pass
 - Replace labels with addresses in the second pass
 - One pass approach
 - Generate holes when address is needed but unknown
 - Backpatch holes when address is known later on
- ❑ We will discuss the more efficient one pass approach
 - It is also the method you will use in project 4.

One-Pass Based Syntax Directed Translation

- ❑ Non-L-attributes during code generation are unavoidable
 - Due to forward jumps to code on the right hand side
 - Example: E.true, E.false, S.next for boolean expressions
 - Is there a one-pass solution to the problem?

Idea:

1. Leave holes for non-L-attribute values we don't know
2. Fill the holes in when we know the values later on
 - *holelist*: synthesized attribute of 'holes' for one value
 - Holes are filled in by traversing list when value is known
 - All holes will be patched by the end of code generation
(Since all forward jumps would be resolved by then)

One-Pass Based Syntax Directed Translation

- ❑ Attributes for two pass based approach
 - Expression **E**
 - Synthesized attributes: **E.code**,
E.holes_truelist, and **E.holes_falselist**
 - Statement **S**
 - Synthesized attributes: **S.code** and **S.holes_nextlist**

- ❑ 3 functions for implementing backpatching
 - **makelist(i)**: create a holelist with statement index i
 - **merge(p1, p2)**: concatenate list p1 and list p2
 - **backpatch(p, i)**: insert index i in every hole in holelist p

Backpatching for if-then

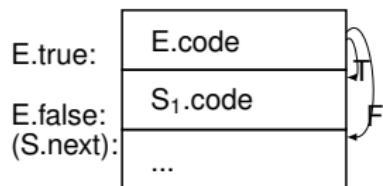
- ❑ Given rule $S \rightarrow \text{if } E \text{ then } S_1$, below is done in one-pass:
 - (1). Gen $E.\text{code}$, making $E.\text{holes_truelist}$, $E.\text{holes_falsealist}$
 - (2). Gen $S_1.\text{code}$, filling in $E.\text{holes_truelist}$ and merging $S_1.\text{holes_nextlist}$ with $E.\text{holes_falsealist}$
 - (3). Pass on merged list to $S.\text{holes_nextlist}$

$S \rightarrow \text{if } E \text{ then } M S_1$

```
{
  backpatch(E.holes_truelist, M.index);
  S.holes_nextlist = merge(S1.holes_nextlist, E.holes_falselist);
}
```

$M \rightarrow \epsilon$

```
{ M.index = curIndex; }
```



Backpatching for if-then-else



Given rule $S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$:

- (1). Gen $E.\text{code}$, making $E.\text{holes_truelist}$, $E.\text{holes_falsealist}$
- (2). Gen $S_1.\text{code}$, filling in $E.\text{holes_truelist}$
- (3). Emit goto to $S.\text{next}$ before S_2 , to skip over S_2
- (4). Gen $S_2.\text{code}$, filling in $E.\text{holes_falsealist}$
- (5). Merge relevant holes into $S.\text{holes_nextlist}$

```

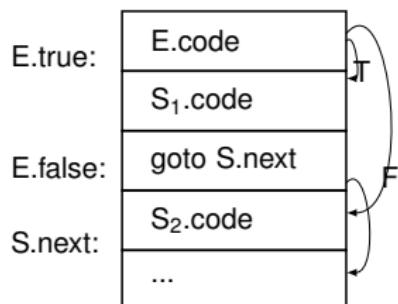
S → if E then M1 S1 N else M2 S2
{
backpatch(E.holes_truelist, M1.index);
backpatch(E.holes_falsealist, M2.index);
templist = merge(S1.holes_nextlist, N.holes_nextlist);
S.holes_nextlist = merge(templist, S2.holes_nextlist);
}

```

```

N → ε
{
N.holes_nextlist = makelist(curlIndex);
emit('goto ____');
}

```



Backpatching for S.holes_nextlist

- ❑ When does the holes in S.holes_nextlist get patched?
 - When the instruction after S is generated of course!

- ❑ Given rule $S \rightarrow S_1 \ S_2$:
 - (1). Gen $S_1.\text{code}$ making $S_1.\text{holes_nextlist}$
 - (2). Gen $S_2.\text{code}$ making $S_2.\text{holes_nextlist}$
 - (3). Fill in $S_1.\text{holes_nextlist}$ with index of $S_2.\text{code}$
 - (4). Pass on $S_2.\text{holes_nextlist}$ to $S.\text{holes_nextlist}$

```
S → S1 M S2
{
backpatch(S1.holes_nextlist, M.index);
S.holes_nextlist = S2.holes_nextlist;
}
```

```
M → ε
{ M.index = curlIndex; }
```

Backpatching for Boolean Expressions

$E \rightarrow E_1 \text{ or } M\ E_2$ { backpatch($E_1.\text{holes_falseList}$, $M.\text{index}$);
 $E.\text{holes_trueList} = \text{merge}(E_1.\text{holes_trueList}, E_2.\text{holes_trueList});$
 $E.\text{holes_falseList} = E_2.\text{holes_falseList};$ }

$E \rightarrow E_1 \text{ and } M\ E_2$ { backpatch($E_1.\text{holes_trueList}$, $M.\text{index}$);
 $E.\text{holes_falseList} = \text{merge}(E_1.\text{holes_falseList}, E_2.\text{holes_falseList});$
 $E.\text{holes_trueList} = E_2.\text{holes_trueList};$ }

$M \rightarrow \varepsilon$ { $M.\text{index} = \text{curIndex};$ }

Backpatching for Boolean Expressions

$E \rightarrow \text{not } E_1$ { E.holes_truelist = E₁.holes_falselist;
E.holes_falselist = E₁.holes_truelist; }

$E \rightarrow (E_1)$ { E.holes_truelist = E₁.holes_truelist;
E.holes_falselist = E₁.holes_falselist; }

$E \rightarrow \text{id1 relop id2}$ { E.holes_truelist = makelist(curlIndex);
E.holes_falselist = makelist(curlIndex+1);
emit('if' id1.place 'relop' id2.place 'goto ____');
emit('goto ____'); }

$E \rightarrow \text{true}$ { E.holes_truelist = makelist(curlIndex);
emit('goto ____'); }

$E \rightarrow \text{false}$ { E.holes_falselist = makelist(curlIndex);
emit('goto ____'); }

Backpatching Example

- $E \rightarrow (a < b) \text{ or } M_1 (c < d \text{ and } M_2 e < f)$
- When reducing $(a < b)$ to E_1 , we have
 - 100: if($a < b$) goto ____
 - 101: goto ____
- When reducing ε to M_1 , we have
- When reducing $(c < d)$ to E_2 , we have
 - 102: if($c < d$) goto ____
 - 103: goto ____
- When reducing ε to M_2 , we have
- When reducing $(e < f)$ to E_3 , we have
 - 104: if($e < f$) goto ____
 - 105: goto ____

$E_1.\text{hole_truelist}=(100)$
 $E_1.\text{hole_falsealist}=(101)$

$M_1.\text{index} = 102$

$E_2.\text{hole_truelist}=(102)$
 $E_2.\text{hole_falsealist}=(103)$

$M_2.\text{index} = 104$

$E_3.\text{hole_truelist}=(104)$
 $E_3.\text{hole_falsealist}=(105)$

Backpatching Example (cont.)

- When reducing (E_2 and $M2 E3$) to $E4$, we **backpatch((102), 104);**

102: if($c < d$) goto **104**

$E4.hole_truelist=(104)$

103: goto ____

$E4.hole_falseList=(103,105)$

104: if($e < f$) goto ____

105: goto ____

- When reducing (E_1 or $M1 E4$) to $E5$, we **backpatch((101), 102);**

100: if($a < b$) goto ____

$E5.hole_truelist=(100, 104)$

101: goto **102**

$E5.hole_falseList=(103,105)$

102: if($c < d$) goto 104

103: goto ____

104: if($e < f$) goto ____

105: goto ____

Backpatching Example (cont.)

- When reducing (E_2 and $M2 E3$) to $E4$, we **backpatch((102), 104);**

102: if($c < d$) goto **104**

$E4.hole_truelist=(104)$

103: goto ____

$E4.hole_falselist=(103,105)$

104: if($e < f$) goto ____

105: goto ____

- When reducing (E_1 or $M1 E4$) to $E5$, we **backpatch((101), 102);**

100: if($a < b$) goto ____

$E5.hole_truelist=(100, 104)$

101: goto **102**

$E5.hole_falselist=(103,105)$

102: if($c < d$) goto 104

103: goto ____

104: if($e < f$) goto ____

105: goto ____

- Are we done?

Backpatching Example (cont.)

- When reducing (E_2 and $M2 E3$) to $E4$, we **backpatch((102), 104);**

102: if($c < d$) goto **104**

$E4.hole_truelist=(104)$

103: goto ____

$E4.hole_falselist=(103,105)$

104: if($e < f$) goto ____

105: goto ____

- When reducing (E_1 or $M1 E4$) to $E5$, we **backpatch((101), 102);**

100: if($a < b$) goto ____

$E5.hole_truelist=(100, 104)$

101: goto **102**

$E5.hole_falselist=(103,105)$

102: if($c < d$) goto 104

103: goto ____

104: if($e < f$) goto ____

105: goto ____

- Are we done?

➤ Yes for this expression

