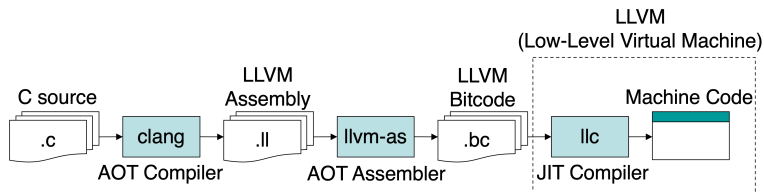


# Code Generation

# Multiple IRs in the Compiler

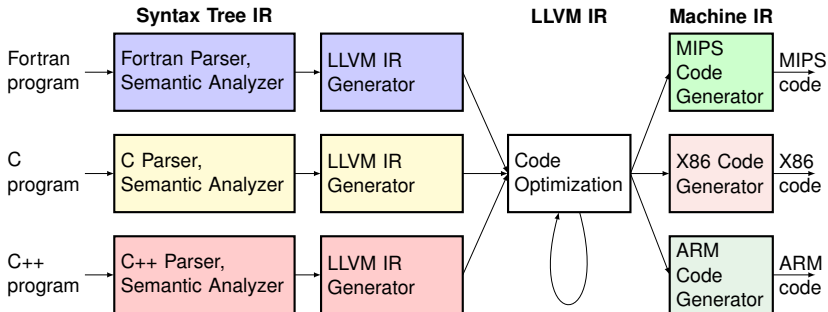
# Modern Compiler Framework (Clang/LLVM)

Remember this diagram from our first day?



LLVM Bitcode is in LLVM IR (Intermediate Representation)

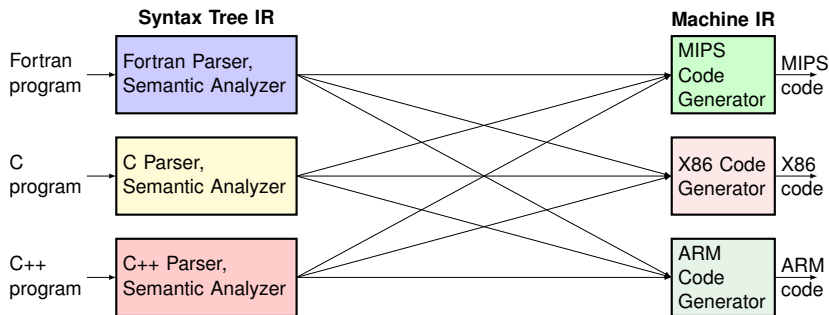
# Modern Compiler Framework (Clang/LLVM)



Common LLVM IR for all languages and backends means:

- Code optimizations need to be written only once
- Implementation complexity is  $O(M + N)$  instead of  $O(M * N)$  (where  $M$  = number of frontends,  $N$  = 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 (e.g. devirtualization)
  - Devirtualization: changing polymorphic calls to direct calls
    - Polymorphic method calls are indirect jumps using a vtable (A vtable is a table of function pointers for each class)
    - Sometimes the direct call is inlined into caller method

# Low-Level IRs

- ❑ Goal: Express code in the ISA of an abstract machine
- ❑ Examples: Three address code, Static Single Assignment
- ❑ Differs on: Language and back-end machine agnostic
- ❑ Uses:
  - A common IR that connects front-ends and back-ends
  - Language / machine independent optimizations
    - Common subexpression elimination
    - Constant propagation
    - Loop invariant code motion
    - ...
  - Optimizations done in this common IR unless reason not to

# 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
    - Strength reduction (replacing op with cheaper op)
    - Vectorization (using CPU vector units if available)
    - ...



# Low-Level IRs

# Three Address Code

Generic form is  **$X = Y \text{ op } Z$**

where X, Y, Z can be variables, constants, or compiler-generated temporaries holding intermediate values

## □ Characteristics

- Assembly code for an 'abstract machine'
- Long expressions are converted to multiple instructions
- Control flow statements are converted to jumps
- Machine independent
  - Operations are generic (not tailored to specific machine)
  - Function calls represented as generic call nodes
  - Uses **symbolic names** rather than **register names**  
(Actual locations of symbols are yet to be determined)

## □ Why this form?

- Allows IR to be generated in a machine-agnostic way
- Optimizations on IR can be done much more easily  
(Optimizations don't worry about syntactic structure)

# Example

□ An example:

$x * y + z / w$

is translated to

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

$t2 = z / w$

$t3 = t1 + t2$

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

# Common Three-Address Statements (I)

- Assignment statement:

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

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

- Assignment statement:

**$x = \text{op } y$**

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

- Copy statement:

**$x = y$**

- Unconditional jump statement:

**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  $x_1$ , ..., param  $x_n$ , call  $F_y$ , n**

As an example, foo( $x_1$ ,  $x_2$ ,  $x_3$ ) is translated to

param  $x_1$

param  $x_2$

param  $x_3$

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  $x$

# Implementation of Three-Address Code

- ❑ There are three possible ways to store the code
  - quadruples
  - triples
  - indirect triples
- ❑ Using quadruples
  - op src1, src2, dest**
  - There are four fields at maximum
  - Src1 and src2 are optional
  - Src1, src2, dest are variables (including temporaries)

## Examples:

<code>x = a + b</code>	<code>=&gt; + a, b, x</code>
<code>x = - y</code>	<code>=&gt; - y, , x</code>
<code>goto L</code>	<code>=&gt; goto , , L</code>

# Using Triples

- ❏ Triples have only three fields. How?
  - Destination field of instruction is always a temporary (With the exception of assignments to variables)
  - 👉 Replace use of temporary with pointer to that instruction
  - 👉 Instruction no longer needs a destination field!

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

	Quadruples				Triples		
	op	src1	src2	dest	op	src1	src2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t1	*	b	(0)
(2)	-	c		t2	-	c	
(3)	*	b	t2	t2	*	b	(2)
(4)	+	t1	t2	t1	+	(1)	(3)
(5)	=	t1		a	=	a	(4)



# More About Triples

## Triples for array statements

$x[i] = y$

is translated to

(0)  $[] \times i$

(1)  $= (0) y$

➤ That is, one statement is translated to two triples

# Using Indirect Triples

## Problem with triples

- No code moving allowed because triple locations change (that would invalidate pointers to those locations, right?)

	Quadruples				Triples		
	op	src1	src2	dest	op	src1	src2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t1	*	b	(0)
(2)	-	c		t2	-	c	
(3)	*	b	t2	t2	*	b	(2)
(4)	+	t1	t2	t1	+	(1)	(3)
(5)	=	t1		a	=	a	(4)

# Using Indirect Triples

## Problem with triples

- No code moving allowed because triple locations change (that would invalidate pointers to those locations, right?)

	Quadruples				Triples		
	op	src1	src2	dest	op	src1	src2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t1	*	b	(0)
(2)	+	t1	t1	t1	+	(1)	(1)
(3)	=	t1		a	=	a	(4)

# Using Indirect Triples

- IR is now a listing of **pointers** to triples
- Triples are stored in a separate instruction storage  
(Typically triple objects are stored in program heap)
- Now code movement will not change locations of triples

	Indirect Triples
	(ptr to triple storage)
(0)	(0)
(1)	(1)
(2)	(2)
(3)	(3)
(4)	(4)
(5)	(5)

	Triple Storage		
	op	src1	src2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

# After Optimization

	Indirect Triples
	(ptr to triple database)
(0)	(0)
(1)	(1)
(2)	(4)
(3)	(5)

	Triple Storage		
	op	src1	src2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(1)
(5)	=	a	(4)

- After optimization, triple objects sometimes can be freed
  - Free space will get reused by heap management system

# After Optimization

	Indirect Triples
	(ptr to triple database)
(0)	(0)
(1)	(1)
(2)	(4)
(3)	(5)

	Triple Storage		
	op	src1	src2
(0)	-	c	
(1)	*	b	(0)
(2)	(free space)		
(3)	(free space)		
(4)	+	(1)	(1)
(5)	=	a	(4)

- After optimization, triple objects sometimes can be freed
  - Free space will get reused by heap management system

# After Optimization

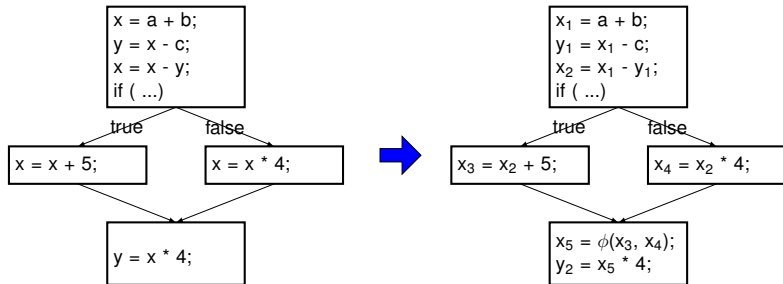
	Indirect Triples
	(ptr to triple database)
(0)	(0)
(1)	(1)
(2)	(4)
(3)	(5)

	Triple Storage		
	op	src1	src2
(0)	-	c	
(1)	*	b	(0)
(2)	(free space)		
(3)	(free space)		
(4)	+	(1)	(1)
(5)	=	a	(4)

- ❑ After optimization, triple objects sometimes can be freed
  - Free space will get reused by heap management system
- ❑ LLVM IR is represented in memory in this way as well

# Static Single Assignment (SSA)

- Developed by R. Cytron, J. Ferrante, *et al.* in 1980s
  - Every variable is assigned exactly once i.e. one **DEF**
  - Convert original variable name to name<sub>version</sub>  
e.g.  $x \rightarrow x_1, x_2$  in different places
  - Use  $\phi$ -function to combine two DEFs of same original variable on a control flow merge





# SSA helps compiler optimizations

## Dead Code Elimination (DCE):

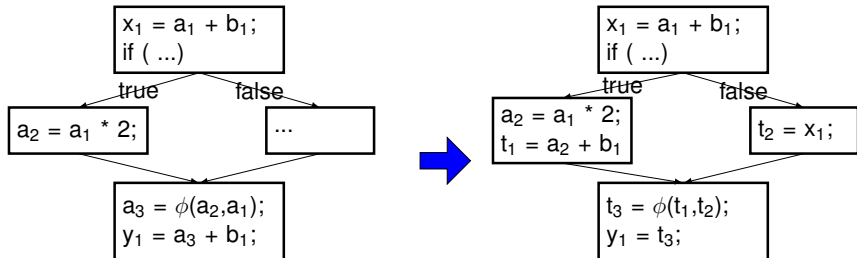
```
x = a + b;
x = c - d;
y = x * b;
```



```
x1 = a + b;
x2 = c - d;
y1 = x2 * b;
```

....  $x_1$  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

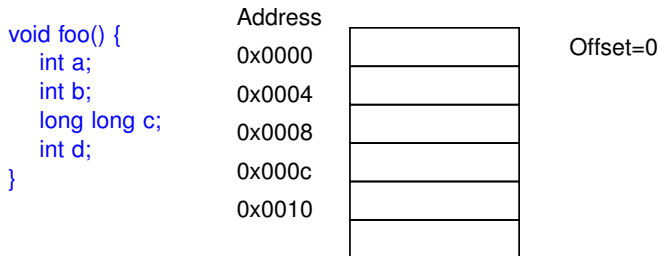
# Layout of Variables in Stack Memory

- 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
    - $\text{address}(x) \leftarrow \text{offset}$
    - $\text{offset} += \text{sizeof}(x.\text{type})$



# 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;  
}
```

Address

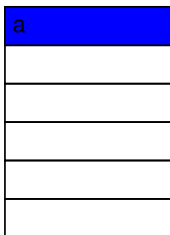
0x0000

0x0004

0x0008

0x000c

0x0010



Offset=0  
 $\text{Addr}(a) \leftarrow 0$

# 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
    - $\text{address}(x) \leftarrow \text{offset}$
    - $\text{offset} += \text{sizeof}(x.\text{type})$

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

Address

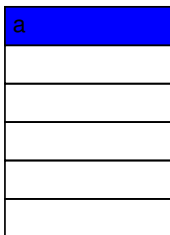
0x0000

0x0004

0x0008

0x000c

0x0010



Offset=4  
 $\text{Addr}(a) \leftarrow 0$

# 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
    - $\text{address}(x) \leftarrow \text{offset}$
    - $\text{offset} += \text{sizeof}(x.\text{type})$

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

Address

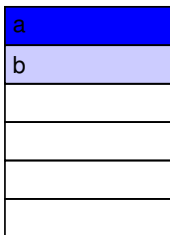
0x0000

0x0004

0x0008

0x000c

0x0010



Offset=8

$\text{Addr}(a) \leftarrow 0$

$\text{Addr}(b) \leftarrow 4$



# 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
    - $\text{address}(x) \leftarrow \text{offset}$
    - $\text{offset} += \text{sizeof}(x.\text{type})$

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

Address

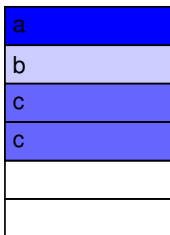
0x0000

0x0004

0x0008

0x000c

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
    - $\text{address}(x) \leftarrow \text{offset}$
    - $\text{offset} += \text{sizeof}(x.\text{type})$

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

Address

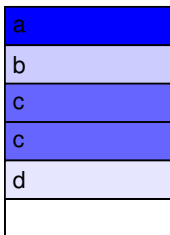
0x0000

0x0004

0x0008

0x000c

0x0010



Offset=20

$\text{Addr}(a) \leftarrow 0$

$\text{Addr}(b) \leftarrow 4$

$\text{Addr}(c) \leftarrow 8$

$\text{Addr}(d) \leftarrow 16$

# 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;
  }
}
  
```

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;
  }
}

```

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
  }

  {
    {
      int j;
    }
    int k;
  }
}
  
```

Symbol  
Table Stack

Offset  
Stack

	12
	8

a	0
b	4

i	8
---	---

# 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

	12
	8

i	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

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;
  }
}
  
```

Symbol  
Table Stack

Offset  
Stack

Symbol Table Stack	Offset Stack
	12
	8
	8

a	0
b	4

--	--

j	8
---	---

# 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
  }
}
  
```

Symbol Table Stack

Symbol	Offset
	12
	8
	8

a	0
b	4

--	--

j	8
---	---

# 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  
  }  
}
```

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;
      check point #4
    }
    int k;
    check point #5
  }
}
  
```

Symbol Table Stack

Symbol	Offset
	12
	8

a	0
b	4

k	8
---	---

# Consideration 1: Allocation Alignment

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

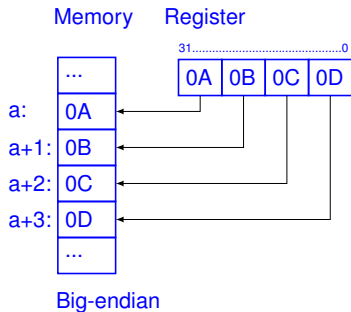
```
void foo() {  
    char a;      // addr(a) = 0;  
    int b;       // addr(b) = 4; /* instead of 1 */  
    int c;       // addr(c) = 8;  
    long long d; // addr(d) = 16; /* instead of 12 */  
}
```

- ❑ 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

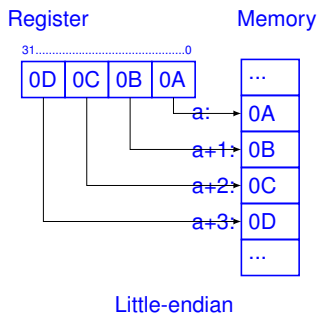
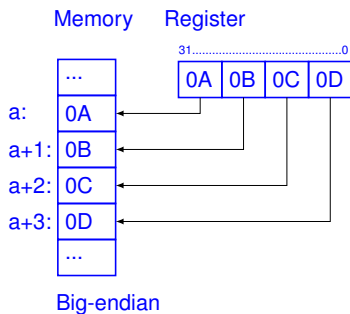




# Consideration II: Endianness

## 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**: synthesized attribute for temporary variable name to store result of E (for SSA, virtual register name)

# 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 \*/

$D \rightarrow D D$

$D \rightarrow T \text{ id};$  { 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); }

$D \rightarrow T \text{ id};$                 { 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[i1]`

2-dimension: `int x[100][200]; ..... x[i1][i2]`

3-dimension: `int x[100][200][300]; ..... x[i1][i2][i3]`

□ Row major: offset of a k-dimension array item

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


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

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

...

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

# Processing Array References

 Processing an array assignment (e.g.  $A[i] = B[j];$ )

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

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

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

$L \rightarrow L1 [ E ]$     {  $L.\text{base} = L1.\text{base}; L.\text{width} = L1.\text{width}; L.\text{dim} = L1.\text{dim} + 1;$   
                                $L.\text{place} = \text{newtemp}();$   
                                $\text{emit}(L.\text{place} '=' L1.\text{place} '*' L.\text{max}[L.\text{dim}]);$   
                                $\text{emit}(L.\text{place} '=' L.\text{place} '+' E.\text{place});$  }

# 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:

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

# Processing Boolean Expressions

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

### ➤ Processing simple boolean expressions:

$S \rightarrow \text{if } E \text{ then } S1$

$E \rightarrow a < b \quad \equiv \quad \begin{aligned} &E.\text{true} = \text{code for } S1; \\ &E.\text{false} = S.\text{next}; \\ &\text{emit( if } E \text{ goto } E.\text{true} ); \\ &\text{emit( goto } E.\text{false} ); \end{aligned}$

$S.\text{next}$ : address of code after  $S$

$E.\text{true}$ : address of code to execute on 'true'

$E.\text{false}$ : address of code to execute on 'false'

### ➤ Processing compound boolean expressions:

- Chain together multiple of above by updating  $E.\text{true}/E.\text{false}$
- $E \rightarrow E1 \ \&\& \ E2$ :  $E1.\text{true} = \text{code for } E2$ ,  $E1.\text{false} = S.\text{next}$
- $E \rightarrow E1 \ || \ E2$ :  $E1.\text{false} = \text{code for } E2$ ,  $E1.\text{true} = \text{code for } S1$

# Processing Boolean Expressions

## 2. With short circuiting (cont'd)

- A short circuited compound boolean expression

$S \rightarrow \text{if } E \text{ then } S1$

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

```
E.true = code for S1;  
E.false = S.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 S1 is generated  
E.false: Only available when code after S is generated
- Emitting any **forward jump** poses this problem

# Syntax Directed Translation

- ❑ Non-L-attributes preclude syntax directed translation
  - Even with 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

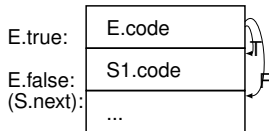
- ❑ Attributes for two pass based approach
  - Expression **E**
    - Synthesized attributes: **E.code**  
**E.holes\_truelist**, and **E.holes\_falselist**
  - Statement **S**
    - Synthesized attributes: **S.code** and **S.holes\_nextlist**
  
- ❑ 3 functions for implementing backpatching
  - **makelist(i)**: 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  $S \rightarrow \text{if } E \text{ then } S1$ , below is done in one-pass:
- (1). Gen **E.code**, making **E.holes\_truelist**, **E.holes\_falselist**
  - (2). Gen **S1.code**, filling in **E.holes\_truelist** and merging **S1.holes\_nextlist** with **E.holes\_falselist**
  - (3). Pass on merged list to **S.holes\_nextlist**

$S \rightarrow \text{if } E \text{ then } M \ S1$

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



$M \rightarrow \epsilon$

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

# Backpatching for if-then-else

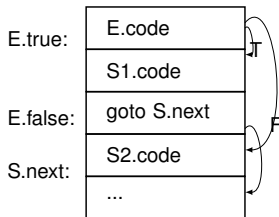
- Given rule  $S \rightarrow \text{if } E \text{ then } S1 \text{ else } S2$ :
- (1). Gen **E.code**, making **E.holes\_truelist**, **E.holes\_falselist**
  - (2). Gen **S1.code**, filling in **E.holes\_truelist**
  - (3). Emit goto to S.next before S2, to skip over S2
  - (4). Gen **S2.code**, filling in **E.holes\_falselist**
  - (5). Merge relevant holes into **S.holes\_nextlist**

$S \rightarrow \text{if } E \text{ then } M1 \ S1 \ N \ \text{else } M2 \ S2$

```
{
  backpatch(E.holes_truelist, M1.index);
  backpatch(E.holes_falselist, M2.index);
  templist = merge(S1.holes_nextlist, N.holes_nextlist);
  S.holes_nextlist = merge(templist, S2.holes_nextlist);
}
```

$N \rightarrow \epsilon$

```
{
  N.holes_nextlist = makelist(curIndex);
  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  $S \rightarrow S1\ S2$ :
  - (1). Gen **S1.code** making **S1.holes\_nextlist**
  - (2). Gen **S2.code** filling in **S1.holes\_nextlist** and making **S2.holes\_nextlist**
  - (3). Pass on **S2.holes\_nextlist** to **S.holes\_nextlist**

# Backpatching for Boolean Expressions

$E \rightarrow E1 \text{ or } M E2$	<pre>{ backpatch(E1.holes_falselist, M.index);   E.holes_truelist = merge(E1.holes_truelist, E2.holes_truelist);   E.holes_falselist = E2.holes_falselist; }</pre>
$E \rightarrow E1 \text{ and } M E2$	<pre>{ backpatch(E1.holes_truelist, M.index);   E.holes_falselist = merge(E1.holes_falselist, E2.holes_falselist);   E.holes_truelist = E2.holes_truelist; }</pre>
$M \rightarrow \varepsilon$	<pre>{ M.index = curIndex; }</pre>

# Backpatching for Boolean Expressions

$E \rightarrow \text{not } E1$	<pre>{ E.holes_truelist = E1.holes_falselist;   E.holes_falselist = E1.holes_truelist; }</pre>
$E \rightarrow (E1)$	<pre>{ E.holes_truelist = E1.holes_truelist;   E.holes_falselist = E1.holes_falselist; }</pre>
$E \rightarrow \text{id1 relop id2}$	<pre>{ E.holes_truelist = makelist(curlIndex);   E.holes_falselist = makelist(curlIndex+1);   emit('if' id1.place 'relop' id2.place 'goto ____');   emit('goto ____'); }</pre>
$E \rightarrow \text{true}$	<pre>{ E.holes_truelist = makelist(curlIndex);   emit('goto ____'); }</pre>
$E \rightarrow \text{false}$	<pre>{ E.holes_falselist = makelist(curlIndex);   emit('goto ____'); }</pre>

# Backpatching Example

□  $E \rightarrow (a < b) \text{ or } M1 \text{ (} c < d \text{ and } M2 \text{ } e < f \text{)}$

□ When reducing  $(a < b)$  to  $E1$ , we have

100: if( $a < b$ ) goto \_\_\_\_  
101: goto \_\_\_\_

$E1.\text{hole\_truelist} = (100)$   
 $E1.\text{hole\_falselist} = (101)$

□ When reducing  $\varepsilon$  to  $M1$ , we have

$M1.\text{index} = 102$

□ When reducing  $(c < d)$  to  $E2$ , we have

102: if( $c < d$ ) goto \_\_\_\_  
103: goto \_\_\_\_

$E2.\text{hole\_truelist} = (102)$   
 $E2.\text{hole\_falselist} = (103)$

□ When reducing  $\varepsilon$  to  $M2$ , we have

$M2.\text{index} = 104$

□ When reducing  $(e < f)$  to  $E3$ , we have

104: if( $e < f$ ) goto \_\_\_\_  
105: goto \_\_\_\_

$E3.\text{hole\_truelist} = (104)$   
 $E3.\text{hole\_falselist} = (105)$

# Backpatching Example (cont.)

- When reducing (E2 and M2 E3) to E4, we `backpatch((102), 104);`
- |                              |                             |
|------------------------------|-----------------------------|
| 100: if(a<b) goto ____       | E4.hole_truelist=(104)      |
| 101: goto ____               | E4.hole_falselist=(103,105) |
| 102: if(c<d) goto <b>104</b> |                             |
| 103: goto ____               |                             |
| 104: if(e<f) goto ____       |                             |
| 105: goto ____               |                             |
- When reducing (E1 or M1 E4) to E5, we `backpatch((101), 102);`
- |                        |                             |
|------------------------|-----------------------------|
| 100: if(a<b) goto ____ | E5.hole_truelist=(100, 104) |
| 101: goto <b>102</b>   | 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 (E2 and M2 E3) to E4, we `backpatch((102), 104);`  
100: if(a<b) goto \_\_\_\_                    E4.hole\_truelist=(104)  
101: goto \_\_\_\_                            E4.hole\_falselist=(103,105)  
102: if(c<d) goto **104**  
103: goto \_\_\_\_  
104: if(e<f) goto \_\_\_\_  
105: goto \_\_\_\_
  
- When reducing (E1 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 (E2 and M2 E3) to E4, we `backpatch((102), 104);`

```

100: if(a<b) goto ____      E4.hole_truelist=(104)
101: goto ____              E4.hole_falselist=(103,105)
102: if(c<d) goto 104
103: goto ____
104: if(e<f) goto ____
105: goto ____

```

- When reducing (E1 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

