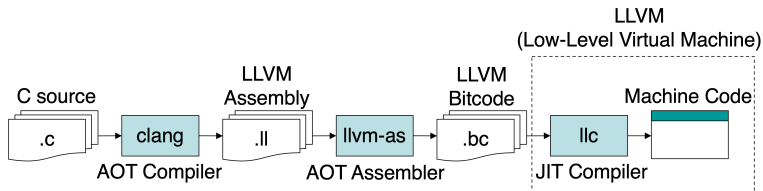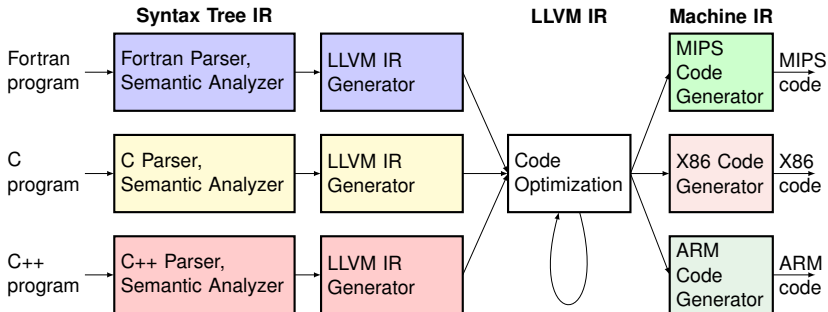# Code Generation

Multiple IRs in the Compiler

Modern Compiler Framework (Clang/LLVM)

❏ Remember this diagram from our first day?

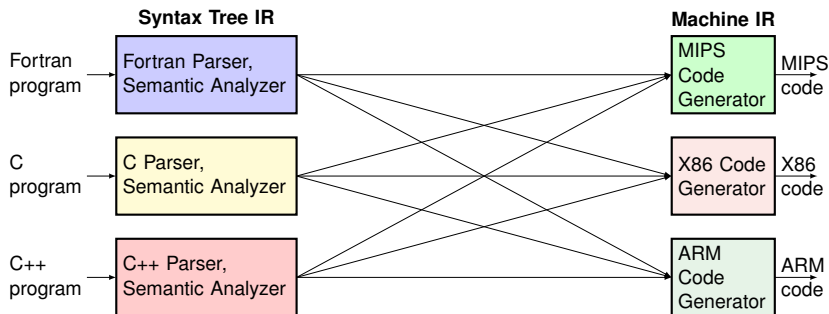❏ LLVM Bitcode is in LLVM IR (Intermediate Representation)

Modern Compiler Framework (Clang/LLVM)

❑ Common LLVM IR for all languages and backends means:
   ➤ Code optimizations need to be written only once
   ➤ Implementation complexity if 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 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 op z

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

❏ Assignment statement:

x = 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(x1, x2, x3) 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:

```
x = a + b        => + a, b, x
x = - y          => - y, , x
goto L           => goto , , L
```

Using Triples

❏ Tripes have only three fields. How?
- ➢ Destination field of instruction is always a temporary
  (With the exception of assignments to variables)
- ☞ Replace use of temporary wth 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) [] x i

  (1) = (0) y

   ➤ That is, one statement is translated to two triples

Using Indirect Triples

❑ Problem with triples
  ➤ Cannot move code around because instruction numbers will change

	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
  ➢ Cannot move code around because instruction numbers
    will change

	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 a listing of pointers to triples instead of triples themselves
➤ Triples are stored in a separate triple 'database'
➤ Can modify listing as long as the database does not change
➤ Slightly more overhead but allows optimizations

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

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

◻ After optimization, some entries in database can be reused

➢ i.e. Entries in triple database do not have to be contiguous

After Optimization

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

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

☐ After optimization, some entries in database can be reused

  ➤ i.e. Entries in triple database do not have to be contiguous

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$_{version}$
    e.g. $x \rightarrow x_1$, x_2 in different places
  ➤ Use ϕ-function to combine two DEFs of same original
    variable on a control flow merge

```
x = a + b;
y = x - c;
x = x - y;
if ( ...)
```
    true   false

```
x = x + 5;        x = x * 4;
```

```
y = x * 4;
```

➡

```
x₁ = a + b;
y₁ = x₁ - c;
x₂ = x₁ - y₁;
if ( ...)
```
    true   false

```
x₃ = x₂ + 5;        x₄ = x₂ * 4;
```

```
x₅ = φ(x₃, x₄);
y₂ = x₅ * 4;
```
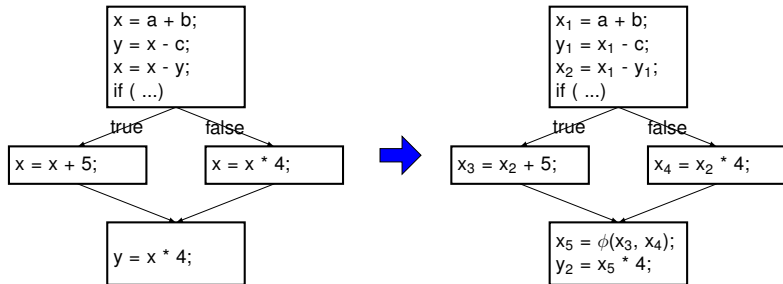
Benefits of SSA

❑ SSA can assist compiler optimizations
  ➢ e.g. remove dead code

$x = a + b;$ $x = c - d;$ $y = x * b;$	➡	$x_1 = a + b;$ $x_2 = c - d;$ $y_1 = x_2 * b;$
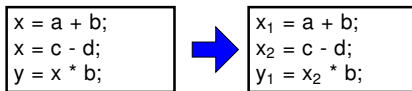
  .... x_1 is defined but never used, it is safe to remove

❑ Will discuss more in **compiler optimization** phase
❑ Intuition: Makes data dependency relationships between instructions more apparent in the IR

Generating IR using Syntax Directed Translation

Generating IR

Our next task is to translate **language constructs** to IR using **syntax directed translation scheme**

Generating IR

Our next task is to translate **language constructs** to IR using **syntax directed translation scheme**

❏ What is our parsing scheme?
- ➤ Bottom-up LR/LALR parsing
  - Natural to translate synthesized attributes
  - Hack to translate L-attributed inherited attributes

Generating IR

Our next task is to translate **language constructs** to IR using **syntax directed translation scheme**

❏ What is our parsing scheme?

➢ Bottom-up LR/LALR parsing

- Natural to translate synthesized attributes
- Hack to translate L-attributed inherited attributes

Generating IR

Our next task is to translate **language constructs** to IR using **syntax directed translation scheme**

❏ What is our parsing scheme?
  ➢ Bottom-up LR/LALR parsing
    ● Natural to translate synthesized attributes
    ● Hack to translate L-attributed inherited attributes

❏ What language structures do we need to translate?
  ➢ Declarations
    ● variables, procedures (need to enforce static scoping), ...
  ➢ Assignment statement
  ➢ Flow of control statement
    ● if-then-else, while-do, for-loop, ...
  ➢ Procedure call
  ➢ ...

Attributes to Evaluate in Translation

❏ Statement **S**

    ➤ **S.code** — a synthesized attribute that holds IR code of S

❏ Expression **E**

    ➤ **E.code** — a synthesized attribute that holds IR code for computing E

    ➤ **E.place** — a synthesized attribute that holds the location where the result of computing E is stored

❏ Variable declaration:

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

    ➤ Type information **T.type T.width**

    ➤ Variable information **V.type**, **V.offset**

Attributes to Evaluate in Translation

❑ Statement **S**

  ➢ **S.code** — a synthesized attribute that holds IR code of S

❑ Expression **E**

  ➢ **E.code** — a synthesized attribute that holds IR code for computing E

  ➢ **E.place** — a synthesized attribute that holds the location where the result of computing E is stored

❑ Variable declaration:

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

  ➢ Type information **T.type T.width**

  ➢ Variable information **V.type**, **V.offset**

  ..... What is **V.offset**?

Storage Layout of Variables in a Procedure

❑ When there are multiple variables defined in a procedure,
  ➢ we layout the variable sequentially
  ➢ use variable **offset**, to get address of **x**
    - address(x) ← offset
    - offset += sizeof(x.type)

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

Storage Layout of Variables in a Procedure

■ When there are multiple variables defined in a procedure,

➢ we layout the variable sequentially

➢ use variable **offset**, to get address of **x**

- address(x) ← offset
- offset += sizeof(x.type)

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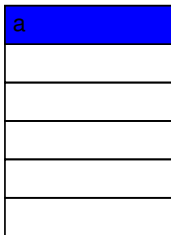
Address

0x0000

0x0004

0x0008

0x000c

0x0010
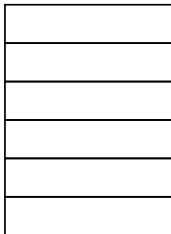
Offset=0

Storage Layout of Variables in a Procedure

◻ When there are multiple variables defined in a procedure,
  ➢ we layout the variable sequentially
  ➢ use variable **offset**, to get address of **x**
    • address(x) ← offset
    • offset += sizeof(x.type)

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

Address

0x0000	a
0x0004	
0x0008	
0x000c	
0x0010	
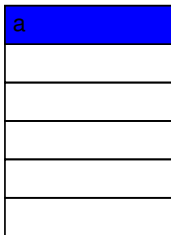
Offset=0
Addr(a)←0

Storage Layout of Variables in a Procedure

◻ When there are multiple variables defined in a procedure,

➢ we layout the variable sequentially
➢ use variable **offset**, to get address of **x**

  ● address(x) ← offset
  ● offset += sizeof(x.type)

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

Address	
0x0000	a
0x0004	
0x0008	
0x000c	
0x0010	

Offset=4
Addr(a)←0

Storage Layout of Variables in a Procedure

⬛ When there are multiple variables defined in a procedure,
  ➢ we layout the variable sequentially
  ➢ use variable **offset**, to get address of **x**
    ● address(x) ← offset
    ● offset += sizeof(x.type)

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

Address

0x0000	a
0x0004	b
0x0008	
0x000c	
0x0010	

Offset=8
Addr(a)←0
Addr(b)←4

Storage Layout of Variables in a Procedure

❑ When there are multiple variables defined in a procedure,
  ➢ we layout the variable sequentially
  ➢ use variable **offset**, to get address of **x**
    ● address(x) ← offset
    ● offset += sizeof(x.type)

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

Address	
0x0000	a
0x0004	b
0x0008	c
0x000c	c
0x0010	

Offset=16
Addr(a)←0

Addr(b)←4

Addr(c)←8

Storage Layout of Variables in a Procedure

❑ When there are multiple variables defined in a procedure,

➤ we layout the variable sequentially

➤ use variable **offset**, to get address of **x**

- address(x) ← offset
- offset += sizeof(x.type)

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

Address	
0x0000	a
0x0004	b
0x0008	c
0x000c	c
0x0010	d

Offset=20
Addr(a)←0

Addr(b)←4

Addr(c)←8

Addr(d)←16

More About Storage Layout (I)

❏ Allocation alignment

  ➢ Enforce **addr(x) mod sizeof(x.type) == 0**
  ➢ Most machine architectures are designed such that
    computation is most efficient at sizeof(x.type) boundaries

    • E.g. Most machines are designed to load integer values at
      integer word boundaries
    • If not on word boundary, 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 */
}
```
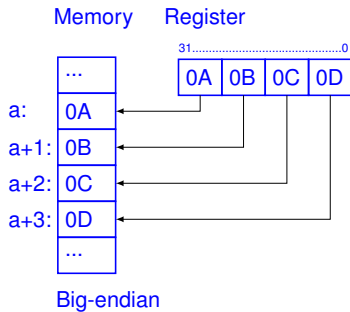
More About Storage Layout (II)

❑ Endianness

➢ Big endian stores **MSB** (most significant byte) in lowest address

➢ Little endian stores **LSB** (least significant byte) in lowest address

Big-endian

More About Storage Layout (II)
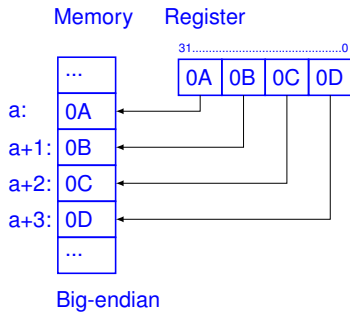
❏ Endianness

  ➤ Big endian stores **MSB** (most significant byte) in lowest address

  ➤ Little endian stores **LSB** (least significant byte) in lowest address

Big-endian   Little-endian
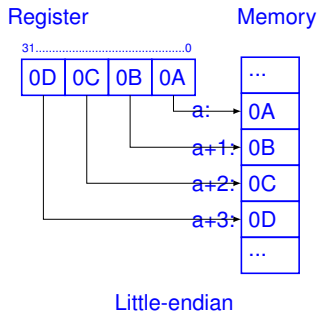
More About Storage Layout (III)

☐ Questions still unanswered
  ➢ How are non-local variables laid out?
  ➢ How dynamically allocated variables laid out?

Processing Declarations

❑ Translating the declaration in a single procedure
  ➢ enter(name, type, offset) — insert the variable into the symbol table

```
P → M D
M → ε          { offset=0; } /* reset offset before layout */
D → D ; D
D → T id       { enter(id.name, T.type, offset); offset += T.width; }
T → integer    { T.type=integer; T.width=4; }
T → real       { T.type=real; T.width=8;}
T → T1[num]    { T.type=array(num.val, T1.type);
                 T.width=num.val * T1.width; }
T → * T1       { T.type=ptr(T1.type); T.width=4; }
```

Processing Nested Declarations

☐ Need scope information for each level of nesting.

☐ When encountering a nested procedure declaration...

1. Create a new symbol table when encountering a sub-procedure declaration
   - mktable(ptr); — ptr points back to its parent table

2. Store procedure name in parent symbol table, with a pointer pointing to the new table
   - enterproc(parent_table_ptr, proc_id, child_table_ptr)

3. Suspend the processing of parent symbol table
   - Push new table in the **active symbol table stack**
   - Push the current offset in the **offset stack**

4. When done, resume the processing of parent symbol table
   - Pop entries in **active symbol table stack**, **offset stack** for nested procedure
   - Restore current offset from the **offset stack**
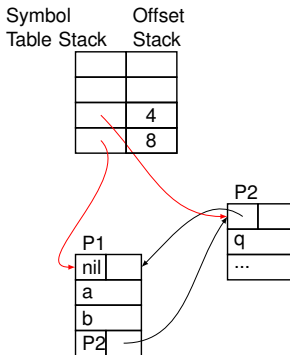
Nested Declaration Example

Nested Declaration Example

Nested Declaration Example

Nested Declaration Example

Nested Declaration Example
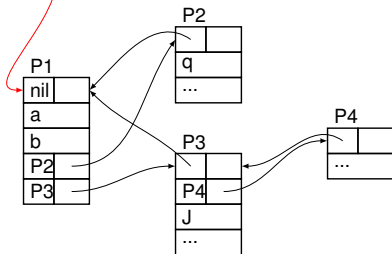
Processing Nested Declarations

❑ Syntax directed translation rules

P → M D   { pop(tblptr); pop(offset); }

M → ε   { t=mktable(nil); push(t, tblptr); push(0,offset); }
D → D1; D2
D → void pid() { N D1; S } { t=top(tblptr); pop(tblptr); pop(offset);
                             enter proc(top(tblptr), pid, t); } /* new symbol table */

D → T id;   { enter(top(tblptr), id, T.type, top(offset));
                             top(offset) = top(offset)+ T.width; }
N → ε   { t=mktable(top(tblptr));
                             push(t, tblptr); push(0, offset);

Processing Statements

▢ Statements are processed sequentially after processing declarations

➢ useful functions:

lookup (id) — search id in symbol table, return nil if none

emit() — print three address IR

newtemp() — get a new temporary variable

```
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: addressing a k-dimension array item
$(low_i = base = 0)$

1-dimension: $A_1 = a_1 * width$ $a_1 = i_1$
2-dimension: $A_2 = a_2 * width$ $a_2 = a_1 * N_2 + i_2$
3-dimension: $A_3 = a_3 * width$ $a_3 = a_2 * N_3 + i_3$
...
k-dimension: $A_k = a_k * width$ $a_k = a_{k-1} * N_k + i_k$

Processing Array References

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

S → L = E   { t = newtemp(); emit(t '=' L.place '*' L.width);
                    emit(t '=' L.base '+' t); emit ('*'t '=' E.place); }

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

L → id [E] { L.base = lookup(id).base; L.width = lookup(id).width; L.dim=1;
                    L.place = E.place; }

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

Processing Boolean Expressions

❑ Boolean expression: **a op b**

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

1. Compute just like any other arithmetic expression

  ➢ Good for languages with no *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 as usual

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

Processing Boolean Expressions

2. Implement as a series of jumps
    ➤ For languages with short circuiting (e.g. C/C++),
      evaluations sometimes have to be 'jumped'
    ➤ Processing a boolean expression:
      S→ if E then S1
      E→ a < b   ≡   E.true = S1.label;
                       E.false = S.next
                       if E goto E.true
                       goto E.false
      S1.label: label created at the address of code S1
      S.next: address of code after S
      E.true: code to execute on 'true'
      E.false: code to execute on 'false'
    ➤ Processing compound boolean expressions:
      ● Chain together multiple of above by updating E.true/E.false
      ● E→ E1 && E2: E1.true = code for E2, E1.false = S.next
      ● E→ E1 || E2: E1.false = code for E2, E1.true = S1.label

Processing Boolean Expressions

2. Implement as a series of jumps (cont'd)
   - ➢ A short circuited compound boolean expression
     E→ (a < b) or (c < d and e < f) ≡   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
   - ➢ Can apply to other control flow statements
     S → if E then S1 | if E then S1 else S2 | while E do S1

   - ➢ **Problem: E.true, E.false, S.next are non-L-attributes**
     - Depend on code that has not been generated yet
       S.next: Only available when code after S is generated
       E.true: Only available when S1 is generated

Syntax Directed Translation

❏ A non-L-attributed grammar may preclude a one pass syntax directed translation scheme

➤ Both top-down and bottom-up SDTS rely on L-attributed grammars

❏ How to handle non-L attributes?

➤ **E.true, E.false, S.next**

❏ 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
- Fill in holes when addresses is known later on
- Finish code generation in one pass

Two-Pass Based Syntax Directed Translation Scheme

❏ Attributes for two pass based approach
  ➢ Expression **E**
    —- Synthesized attributes: **E.code**
    —- non-L inherited attributes: **E.true**, **E.false**
  ➢ Statement **S**
    —- Synthesized attributes: **S.code**
    —- non-L inherited attributes: **S.next**

❏ Evaluation order:
  Given rule **S → if E then S1**, the two passes are:
  (1). Generate **E.code** and **S1.code** making a label for E.true
  (2). Replace label **E.true** with actual address of S1
       (Labels **E.false** and **S1.next** are inheried from **S.next**)
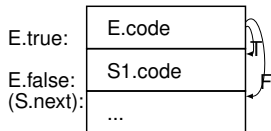
  Given rule **S → S1 S2**, the two passes are:
  (1). Generate **S1.code** and **S2.code** making a label for S1.next
  (2). Replace label **S1.next** with the actual address of S2
       (Label **S2.next** is inheried from **S.next**)

Two Pass based Rules

S → if E then S1
    { E.true = newlabel;
      E.false = S.next;
      S1.next = S.next;
      S.code = E.code || gen(E.true':') || S1.code; }

E.true:	E.code
E.false: (S.next):	S1.code
	...

S → if E then S1 else S2
    { S1.next = S2.next = S.next;
      E.true = newlabel;
      E.false = newlabel;
      S.code = E.code || gen(E.true':') ||
          S1.code || gen('goto ' S.next) ||
          gen(E.false ':') || S2.code; }

E.true:	E.code
	S1.code
E.false:	goto S.next
S.next:	S2.code
	...
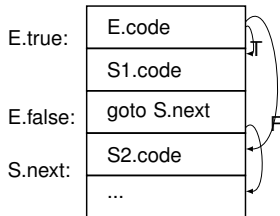
More Two Pass based SDT Rules

$E \rightarrow$ id1 relop id2   { E.code=gen('if' id1.place 'relop' id2.place 'goto' E.true) ||
                 gen('goto' E.false); }

$E \rightarrow$ E1 or E2   { E1.true = E2.true = E.true;
          E1.false = newlabel;
          E2.false = E.false;
          E.code = E1.code || gen(E1.false ':') || E2.code; }

$E \rightarrow$ E1 and E2   { E1.false = E2.false = E.false;
          E1.true = newlabel;
          E2.true = E.true;
          E.code = E1.code || gen(E1.true ':') || E2.code; }

$E \rightarrow$ not E1   { E1.true = E.false; E1.false = E.true; E.code = E1.code; }

$E \rightarrow$ true   { E.code = gen('goto' E.true); }

$E \rightarrow$ false   { E.code = gen('goto' E.false); }

Problem

☐ Try this at home. Refer to textbook Chapter 6.6.

☐ Write SDT rule (two pass) for the following statement

S → while (a<b) do
        if (c<d)
        then S
        endif
      endwhile

Backpatching

❑ If grammar contains L-attributes only, then it can be processed in one pass

❑ However, **we know** there are occasions for non-L attributes
  ➢ Example: E.true, E.false, S.next during code generation
  ➢ Is there a general solution to this problem?

Solution:

❑ Leave holes for non-L attributes, record their locations in holelists, and fill in the holes when we know the target values
  ➢ *holelist*: synthesized attribute of 'holes' to be filled in for a particular target value
  ➢ Holes are filled in one shot when target value is known
  ➢ All holes can be replaced at the end of code generation

One-Pass Based Syntax Directed Translation Scheme

❏ 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**

❏ Evaluation order:
Given rule **S → if E 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**

Given rule **S → S1 S2**, below is done in one-pass:
- (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 Rules for Boolean Expressions

❑ 3 functions for implementing backpatching for IR
generation

➤ makelist(i) — creates a new list with statement index i

➤ merge(p1, p2) — concatenates list p1 and list p2

➤ backpatch(p, i) — insert i as target label for each statement
in list p

$E \to E1$ or M E2   { backpatch(E1.holes_falselist, M.quad);
        E.holes_truelist = merge(E1.holes_truelist, E2.holes_truelist);
        E.holes_falselist = E2.holes_falselist; }

$E \to E1$ and M E2   { backpatch(E1.holes_truelist, M.quad);
        E.holes_falselist = merge(E1.holes_falselist, E2.holes_falselist);
        E.holes_truelist = E2.holes_truelist; }

$M \to \varepsilon$ { M.quad = nextquad; }

More One Pass SDT Rules

E → not E1   { E.holes_truelist = E1.holes_falselist;
                        E.holes_falselist = E1.holes_truelist; }

E → (E1)   { E.holes_truelist = E1.holes_truelist;
                        E.holes_falselist = E1.holes_falselist; }

E → id1 relop id2   { E.holes_truelist = makelist(nextquad);
                        E.holes_falselist = makelist(nextquad+1);
                        emit('if' id1.place 'relop' id2.place 'goto ___');
                        emit('goto ___'); }

E → true   { E.holes_truelist = makelist(nextquad);
                        emit('goto ___'); }

E → false   { E.holes_falselist = makelist(nextquad);
                        emit('goto ___'); }

Backpatching Example

☐ E → (a<b) or M1 (c<d and M2 e<f)

☐ When reducing (a<b) to E1, we have
     100: if(a<b) goto ___   E1.hole_truelist=(100)
     101: goto ___   E1.hole_falselist=(101)

☐ When reducing ε to M1, we have   M1.quad = 102

☐ When reducing (c<d) to E2, we have
     102: if(c<d) goto ___   E2.hole_truelist=(102)
     103: goto ___   E2.hole_falselist=(103)

☐ When reducing ε to M2, we have   M2.quad = 104

☐ When reducing (e<f) to E3, we have
     104: if(e<f) goto ___   E3.hole_truelist=(104)
     105: goto ___   E3.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 **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 ___	

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

Problem

❏ Try this at home. Refer to textbook Chapter 6.6, 6.7.

❏ Write SDT rule (one pass using backpatching) for the following statement

S → while E1 do
        if E2
        then S2
        endif
     endwhile

Solution Hint

❑ $S \rightarrow$ while E1 do if E2 then S2 endif endwhile

	Known Attributes	Attributes to Evaluate/Process
Two Pass	E1.code	E1.true, E1.false
	E2.code	E2.true, E2.false
	S2.code	S2.next
	S.next	S.code
One Pass	E1.code, E1.hole_truelist	S.code
	E1.hole_falselist	S.hole_nextlist
	E2.code, E2.hole_truelist	
	E2.hole_falselist	(E1.hole_truelist,E1.hole_falselist)
	S.code, S.hole_nextlist	(E2.hole_truelist,E2.hole_falselist)