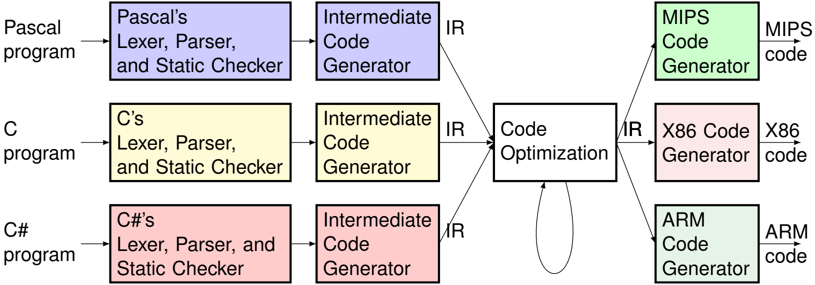
Code Generation

Modern Compiler Project



Why or Why not IR?

At the end of semantic analysis,

- If generating intermediate code, or IR,
  - IR is machine independent, and separates machine dependent/independent parts;
  - Front-end is retargetable;
  - Optimization at intermediate level is reusable;
  - Simply semantic routines in the code.
- If generating machine code directly
  - Avoid the overhead of extra code generation passes;
  - Can exploit the high level hardware features e.g. MMX;

Common Types of IR

- Postfix representation – used in earlier compilers

  a + b * c $\rightarrow$ c b * a +
- Abstract syntax tree

  Discussed before — attribute **tptr** attached to each non-terminal symbol
- Three address code

  Will discuss next
- Static Single Assignment (SSA)

  Assist many code optimization in modern compilers

Three Address Code

Generic form is **X := Y op Z**

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

- Characteristics
  - Similar to assembly code e.g. include statements of flow of control;
  - It is machine independent;
  - Statements use **symbolic names** rather than **register names**;
  - Actual locations of labels are yet to be determined.

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 representation of an AST
  - ➢ Temporaries may be replaced by pointers to the desired result

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 $\mathbf{x}_1$, ..., param $\mathbf{x}_n$, call $\mathbf{F}_y$, 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 a scalable variable and y is an array variable

- Address and pointer operation statement:

  x := & y ; a pointer x is set to location of y

  y := * x ; y is set to the content of the address

  ; stored in pointer x

  ***y := x** ; object pointed to by x gets value y

Implementation of Three-Address Code

- There are three possible ways to store the code
  - quadruples
  - triples
  - indirect triples
- Using quadruples

  op arg1, arg2, result

  - There are four(4) fields at maximum
  - Arg1 and arg2 are optional
  - Arg1, arg2, and result are usually pointers to the symbol table

Examples:

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

Using Triples

- To avoid putting temporaries into the symbol table, we can refer to temporaries by the positions of the statements that compute them

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

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
(2)	-	c		t3	-	c	
(3)	*	b	t3	t4	*	b	(2)
(4)	+	t2	t4	t5	+	(1)	(3)
(5)	:=	t5		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 statement numbers will change

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
(2)	-	c		t3	-	c	
(3)	*	b	t3	t4	*	b	(2)
(4)	+	t2	t4	t5	+	(1)	(3)
(5)	:=	t5		a	:=	a	(4)

Using Indirect Triples

- Problem with triples
  - Cannot move code around because statement numbers will change

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
(2)	+	t2	t2	t5	+	(1)	(1)
(3)	:=	t5		a	:=	a	(4)

Using Indirect Triples

- Listing pointers to triples instead of using triples directly
- Can move pointers around as long as the database (storing all triples) do not change
- Same amount of space as quadruples, i.e. more than triples

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

	Triples op	arg1	arg2
(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	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(1)
(5)	:=	a	(4)

- After optimization, some entries in triple database can be reused
  - That is, entries in the triple database are independent

After Optimization

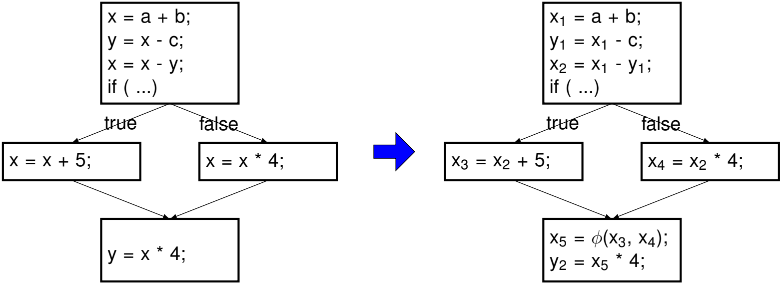
	Indirect Triples (ptr to triple database)
(0)	(0)
(1)	(1)
(2)	(4)
(3)	(5)

	Triple Database		
	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	(empty)		
(3)	(storing new triple)		
(4)	+	(1)	(1)
(5)	:=	a	(4)

- After optimization, some entries in triple database can be reused
  - That is, entries in the triple database are independent

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 $\text{name}_{version}$
    e.g. $x \rightarrow x_1, x_2$ in different places
  - Use ϕ-function to combine two DEFs of same original variable



Benefits of SSA

- SSA can assist compiler optimizations
  - e.g. remove dead code

x = a + b;
y = x - c;
y = x * b;
	⇒	x_1 = a + b;
		y_1 = x_1 - c;
		y_2 = x_1 * b;

.... y_1 is defined but never used, it is safe to remove

- Will discuss more in **compiler optimization** phase

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
    - To be solved: how to translate non-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
    - To be solved: how to translate non-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 E's value
- 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 E's value
- 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

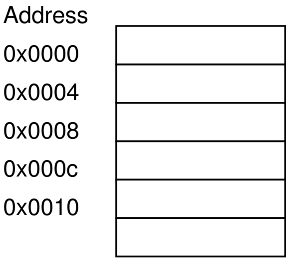
- When there are multiple variables defined in a procedure,
  - we layout the variable sequentially
  - use variable **offset**, to get address of **x**
    - address(x) $\leftarrow$ offset
    - offset += sizeof(x.type)

```
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;
}
```

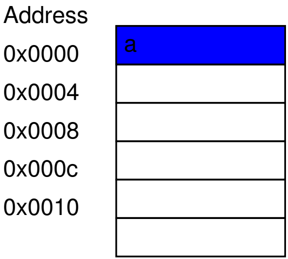


Offset=0

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;
}
```

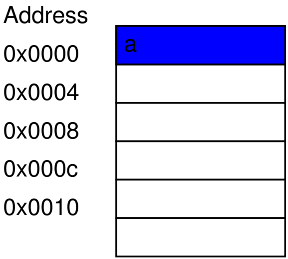


Offset=0
Addr(a)←0

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;
}
```

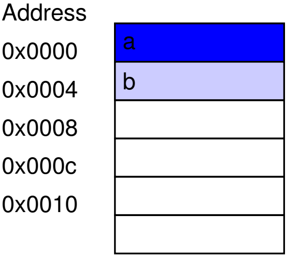


Offset=4
Addr(a)←0

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;
}
```



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) $\leftarrow$ 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)$\leftarrow$0
Addr(b)$\leftarrow$4
Addr(c)$\leftarrow$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) $\leftarrow$ 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)$\leftarrow$0
Addr(b)$\leftarrow$4
Addr(c)$\leftarrow$8
Addr(d)$\leftarrow$16

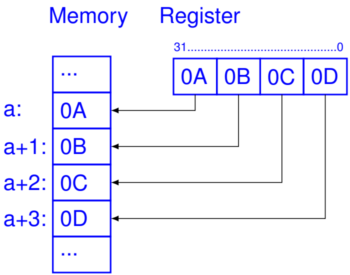
More About Storage Layout (I)

- Allocation alignment
  - In reality, **addr(x) mod sizeof(x.type) == 0**

```
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)

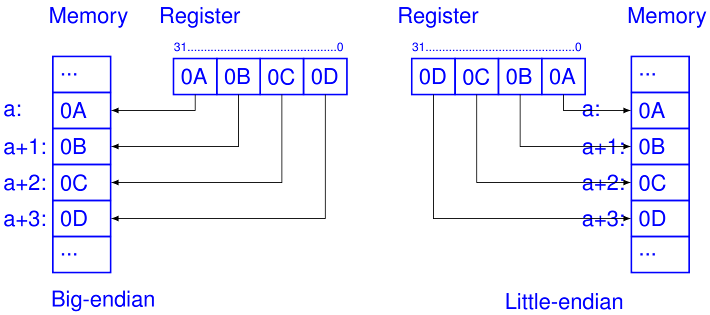
- Endianness
  - Big endian stores **MSB** (most significant byte) in lowest address
  - Little endian stores **LSB** (least significant byte) in lowest address



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



More About Storage Layout (III)

- To be solved in the future
  - How to layout non-local variables?
  - How to layout dynamically allocated variables?

Processing Declarations

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

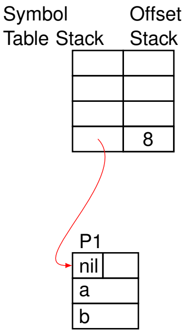
P $\rightarrow$ M D	
M $\rightarrow$ ε	{ offset=0; } /* reset offset before layout */
D $\rightarrow$ D ; D	
D $\rightarrow$ T id	{ enter(id.name, T.type, offset); offset += T.width; }
T $\rightarrow$ integer	{ T.type=interger; T.width=4; }
T $\rightarrow$ real	{ T.type=real; T.width=8;}
T $\rightarrow$ T1[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 Nested Declarations

- Need scope information
  - create a new symbol table when encounter a sub-procedure declaration
    - mktable(ptr); — ptr points back to its parent table
  - procedure name is stored in parent symbol table, with a pointer pointing to the new table
    - enterproc(parent_table_ptr, proc_id, child_table_ptr)
  - suspend the processing of parent symbol table
    - introducing an **offset stack**
  - track active variable names
    - introducing an **active symbol table stack**

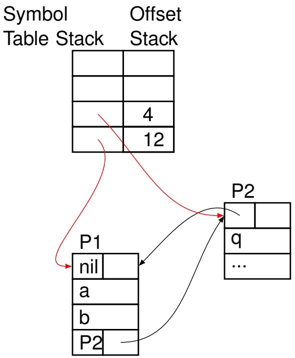
Nested Declaration Example

```
void P1() {
      int a;
      int b;
        check point #1
   void P2() {
         int q;
   }

   void P3() {
         void P4() {
            use a
         }
         int J;
   }
      use q
}
```



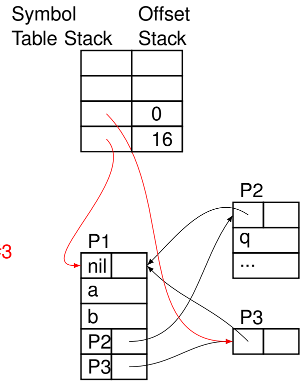
Nested Declaration Example

```
void P1() {
    int a;
    int b;
    check point #1
    void P2() {
        int q;
        check point #2
    }

    void P3() {
        void P4() {
            use a

        }
        int J;
    }
    use q

}
```



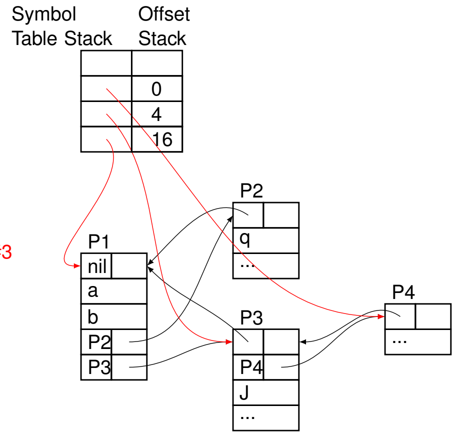
Nested Declaration Example

```
void P1() {
    int a;
    int b;
  check point #1
  void P2() {
      int q;
        check point #2
  }

  void P3() {  check point #3
    void P4() {
        use a
    }
      int J;
  }
    use q
}
```



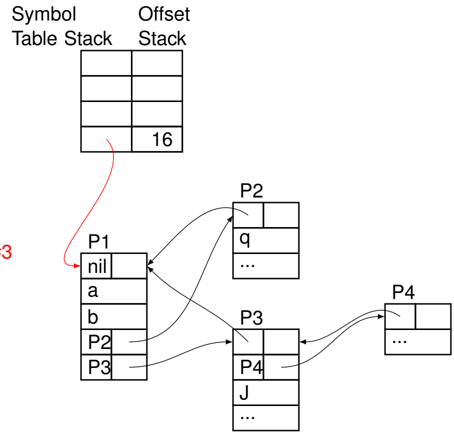
Nested Declaration Example

```
void P1() {
    int a;
    int b;
  check point #1
  void P2() {
      int q;
        check point #2
  }

  void P3() {  check point #3
      void P4() {
          use a
            check point #4
      }
      int J;
  }
  use q

}
```



Nested Declaration Example

```
void P1() {
      int a;
      int b;
        check point #1
   void P2() {
         int q;
           check point #2
   }

   void P3() { check point #3
      void P4() {
           use a
             check point #4
      }
         int J;
   }
      use q
        check point #5
}
```



Processing Nested Declarations

- Syntax directed translation rules

P $\rightarrow$ M D	{ pop(tblptr); pop(offset); }
M $\rightarrow$ ε	{ t=mktable(nil); push(t, tblptr); push(0,offset); }
D $\rightarrow$ D1; D2	
D $\rightarrow$ void pid() { N D1; S }	{ t=top(tblptr); pop(tblptr); pop(offset); enter proc(top(tblptr), pid, t); } /* new symbol table */
D $\rightarrow$ T id;	{ enter(top(tblptr), id, T.type, top(offset)); top(offset) = top(offset)+ T.width; }
N $\rightarrow$ ε	{ t=mktable(top(tblptr)); push(t, tblptr); push(0, offset);

Processing Assignment Statements

- After processing the declarations, let us translate sequential assignment statement
  - 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 Reference

- Recall the generalized form to compute the address of a n-dimensional array variable

```
S → L := E     { t= newtemp(); emit( t '=' L.addr '*' L.width);
                  emit(t '=' L.base '+' t); emit (t '=' E.addr);
E → L          { E.addr = newtemp(); t= newtemp();
                  emit( t '=' L.addr '*' L.width); emit ( E.addr '=' L.base '+' t );

L → id [ E1 ]  { L.array = lookup(id); L.dim=1;
                  emit(L.addr '=' E.addr); }

L → L1 [ E ]   { L.array = lookup(id); L.dim = L1.dim + 1;
                  emit( L.addr '=' L1.addr '*' L.max[L.dim]);
                  emit( L.addr '=' L.addr '+' E.addr); }
```

Recall Generalized Row/Column Major

- 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 $a_k = a_{k-1} {*} N_k + i_k$

- 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]; ..... $x[i_1][i_2][i_3]$

Processing Boolean Expressions

- Generic boolean expression: **a op b**
  - where op can be <, >, >=, ...
  - result is either true or false
- We can encode true and false such that the result is represented by its program location

  E$\rightarrow$ a < b $\equiv$ if (a<b) goto E.true
  goto E.false
- We can do *short circuiting*

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

```
    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
```

Processing Flow of Control Statements

- We have converted boolean expression flow of control statements
- Flow of control statements
  S $\rightarrow$ if E then S1 | if E then S1 else S2 | while E do S1
- S has an inherited attribute — S.next
  - the address of IR statement after S
- **E.true, E.false, S.next are non-L-attributed attributes**
  - they depend on the statements that have not been processed yet
    e.g. what is S1.next ?

Syntax Directed Translation

- Translating other attributes as we discussed in **semantic analysis phase**
- 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

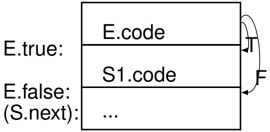
Two-Pass Based Syntax Directed Translation Scheme

- Attributes for two pass based approach
  - Expression **E**
    —- Synthesized attributes: **E.code**
    —- non-L attributes: **E.true**, **E.false**
  - Statement **S**
    —- Synthesized attributes: **S.code**
    —- non-L attributes: **S.next**
- Evaluation order:
  Given grammar rule **A** $\rightarrow$ **RHS**, we need to
  (1). Evaluate the synthesized attribute of **A**
  (2). Evaluate the inherited attributes of **RHS**
  (3). Use the synthesized attributes of **RHS** and inherited attributes of **A**

Two Pass based Rules

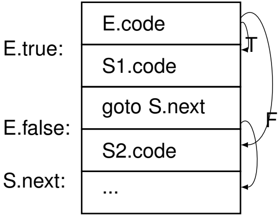
S $\rightarrow$ if E then S1

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



S $\rightarrow$ 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; }



More Two Pass based SDT Rules

S $\rightarrow$ 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

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

  S $\rightarrow$ while (a<b) do
          if (c<d)
          then S
          endif
      endwhile

Backpatching

- If a grammar contains synthesized attributes only, then its IR can be generated in one-pass
  - ... assuming LR/bottom-up parsing
- However, **we know** there are non-L-attributed inherited attributes in modern programs

Solution:

- We generate code using LR, leave holes in the code, record their locations in holelists, and fill in the holes when we know the target addresses
  - *holelist* is a synthesized attribute, we insert locations of holes to the list
  - All holes can be removed 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 grammar rule **A** $\rightarrow$ **RHS**, we need to
  (1). Evaluate the synthesized attribute of **A**
  (2). Use the synthesized attributes of **RHS**
  (3). Each holes_xxxlist might contain holes, we need process all holes_xxxlist of **RHS**
  .... either pass to A's holes_xxxlist, or backpatch the hole

Backpatching Rules for Boolean Expressions

- 3 functions for implementing backpatching
  - makelist(i) — creates a new list with statement index i in the list
  - merge(p1, p2) — concentrates list p1 and list p2
  - backpatch(p, i) — insert i as target label for each statement in list p

E $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ ε	{ M.quad = nextquad; }

More One Pass SDT Rules

$E \rightarrow$ not E1	{ E.holes_truelist = E1.holes_falselist; E.holes_falselist = E1.holes_truelist; }
$E \rightarrow$ (E1)	{ E.holes_truelist = E1.holes_truelist; E.holes_falselist = E1.holes_falselist; }
$E \rightarrow$ id1 relop id2	{ E.holes_truelist = makelist(nextquad); E.holes_falselist = makelist(nextquad+1); emit('if' id1.place 'relop' id2.place 'goto ___'); emit('goto ___'); }
$E \rightarrow$ true	{ E.holes_truelist = makelist(nextquad); emit('goto ___'); }
$E \rightarrow$ false	{ E.holes_falselist = makelist(nextquad); emit('goto ___'); }

Backpatching Example

- E $\rightarrow$ (a<b) or M1 (c<d and M2 e<f)

- When reading (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 104                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 104                  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.)

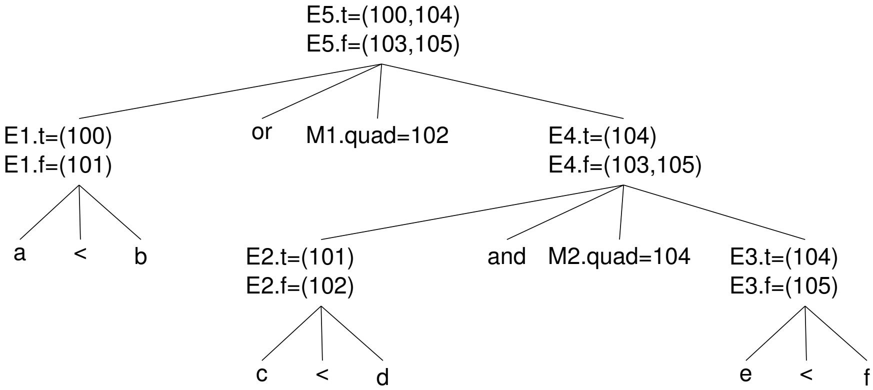
- When reducing (E2 and M2 E3) to E4, we backpatch((102), 104);

```
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 104                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


E5.t=(100,104)
E5.f=(103,105)
E1.t=(100)
E1.f=(101)
or
M1.quad=102
E4.t=(104)
E4.f=(103,105)
a
<
b
E2.t=(101)
E2.f=(102)
and
M2.quad=104
E3.t=(104)
E3.f=(105)
c
<
d
e
<
f

Problem

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

  S $\rightarrow$ 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 E2.code S2.code S.next	E1.true, E1.false E2.true, E2.false S2.next S.code
One Pass	E1.code, E1.hole_truelist E1.hole_falselist E2.code, E2.hole_truelist E2.hole_falselist S.code, S.hole_nextlist	S.code S.hole_nextlist (E1.hole_truelist,E1.hole_falselist) (E2.hole_truelist,E2.hole_falselist)