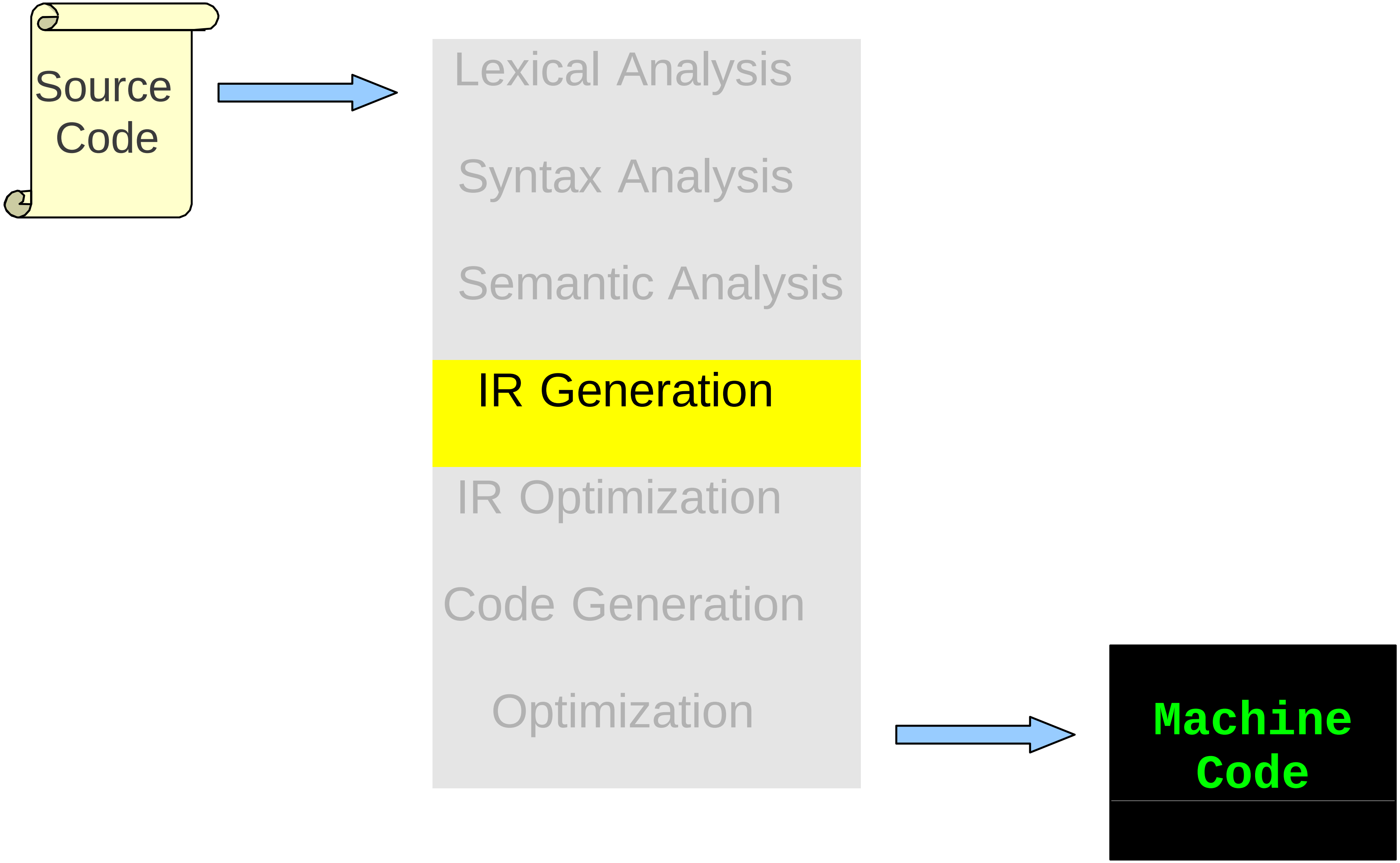
Compilers and Interpreters

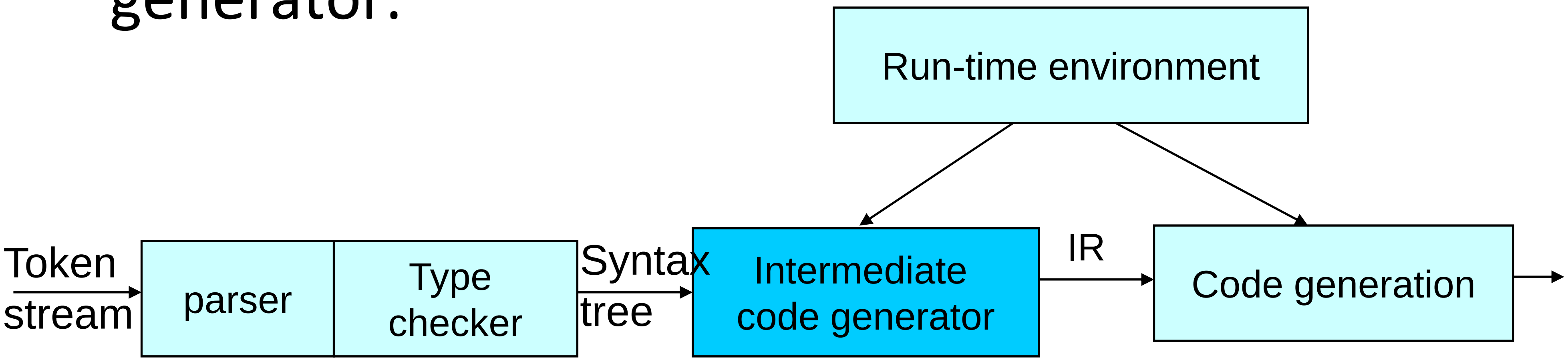
Intermediate Code Generation

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Intermediate Code Generation

- **Intermediate codes** are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an **equivalent program** in an intermediate language by the intermediate code generator.



Keywords

- Intermediate Representation
  - Directed Acyclic Graph (DAG)
  - Three-address Code
  - Postifx
  - …
- Checking and Translation
  - Type Checking
  - Translation of Expressions
  - Boolean Expressions Translation
  - Flow-of-Control Statements Translation

Intermediate language

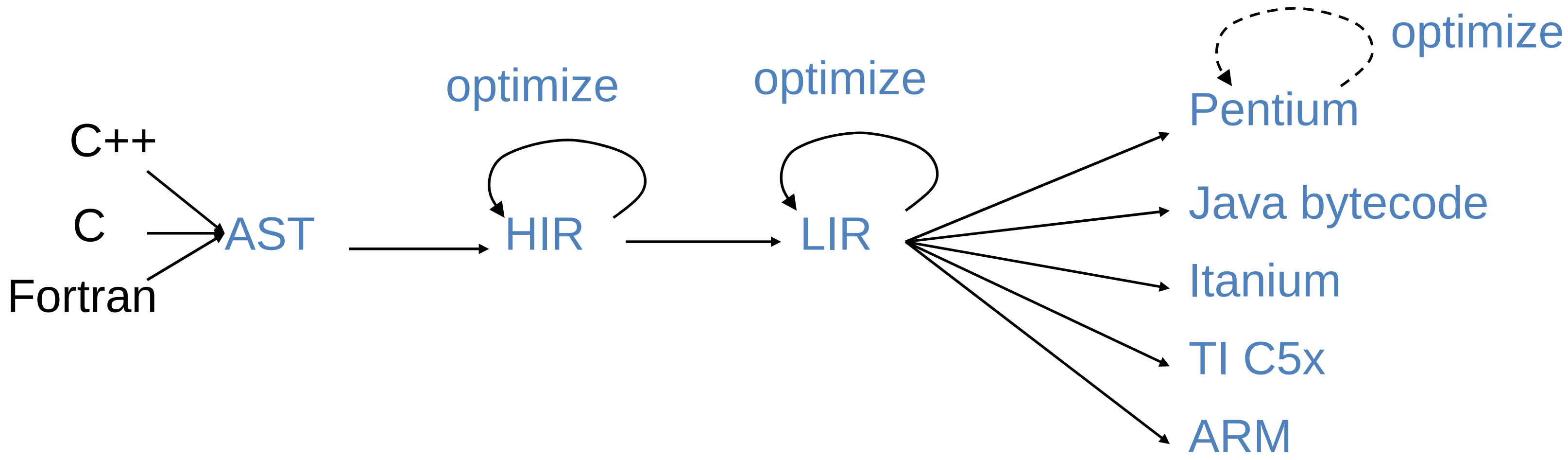
- Intermediate language can be many different languages
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadruples) can be used as an intermediate language
    - we will use quadruples to discuss intermediate code generation
    - quadruples are close to machine instructions, but they are not actual machine instructions.
  - some programming languages have well defined intermediate languages.
    - java – java virtual machine
    - In fact, there are byte-code emulators to execute instructions in these intermediate languages.

What Makes a Good IR?

- Captures high-level language constructs
  - –Easy to translate from AST
  - –Supports high-level optimizations
- Captures low-level machine features
  - –Easy to translate to assembly
  - –Supports machine-dependent optimizations
- Narrow interface: small number of node types (instructions)
  - –Easy to optimize
  - –Easy to retarget

Multiple IRs

- Most compilers use 2 IRs:
  - –High-level IR (HIR): Language independent but closer to the language
  - –Low-level IR (LIR): Machine independent but closer to the machine
  - –A significant part of the compiler is both language and machine independent!



Intermediate Representation Categories

- **Structural (High-level IR)**
  - graphically oriented
    - e.g.: trees, DAGs
  - nodes, edges tend to be large
- **Linear (Low-level IR)**
  - pseudo-code for abstract machine
  - large variation in level of abstraction
  - simple, compact data structures
- **Hybrids**
  - combination of graph & linear code
  - examples: control flow graphs

IR Category example

Source

```
float a[10][20];
a[i][j+2];
```

Low ←→ High

High-level IR

```
t1 = a[i, j+2]
```

Middle-level IR

```
t1 = j + 2
t2 = i * 20
t3 = t1 + t2
t4 = 4 * t3
t5 = addr a
t6 = t5 + t4
t7 = *t6
```

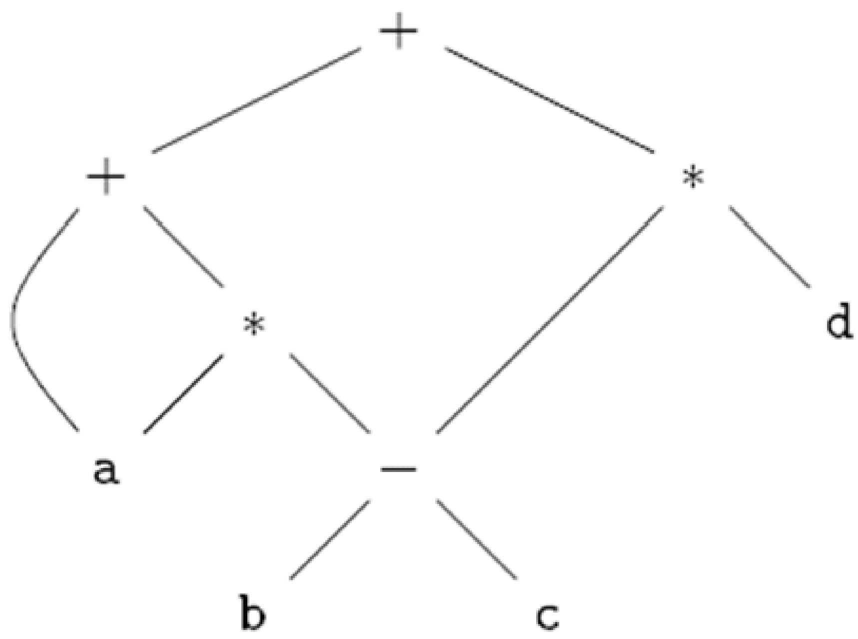
Low-level IR

```
r1 = [fp - 4]
r2 = [r1 + 2]
r3 = [fp - 8]
r4 = r3 * 20
r5 = r4 + r2
r6 = 4 * r5
r7 = fp -216
f1 = [r7 + r6]
```

Directed Acyclic Graphs

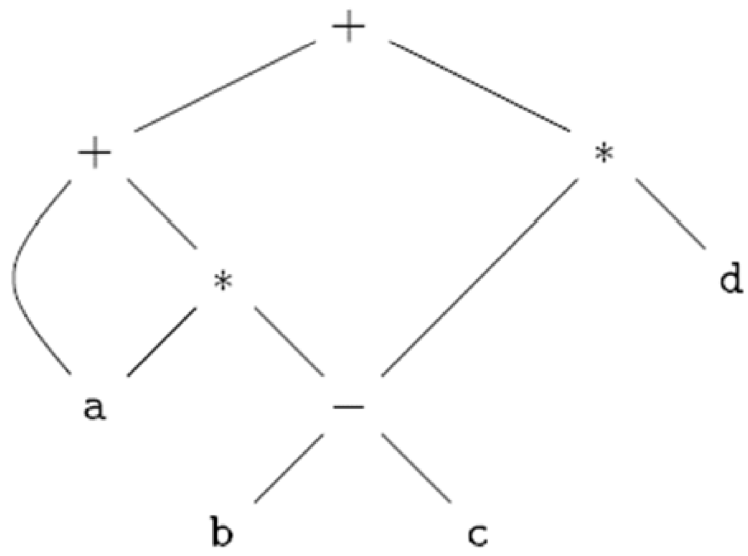
- A DAG has leaves corresponding to atomic operands and interior codes corresponding to operators.
- A DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

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



Directed Acyclic Graphs (Cont.)

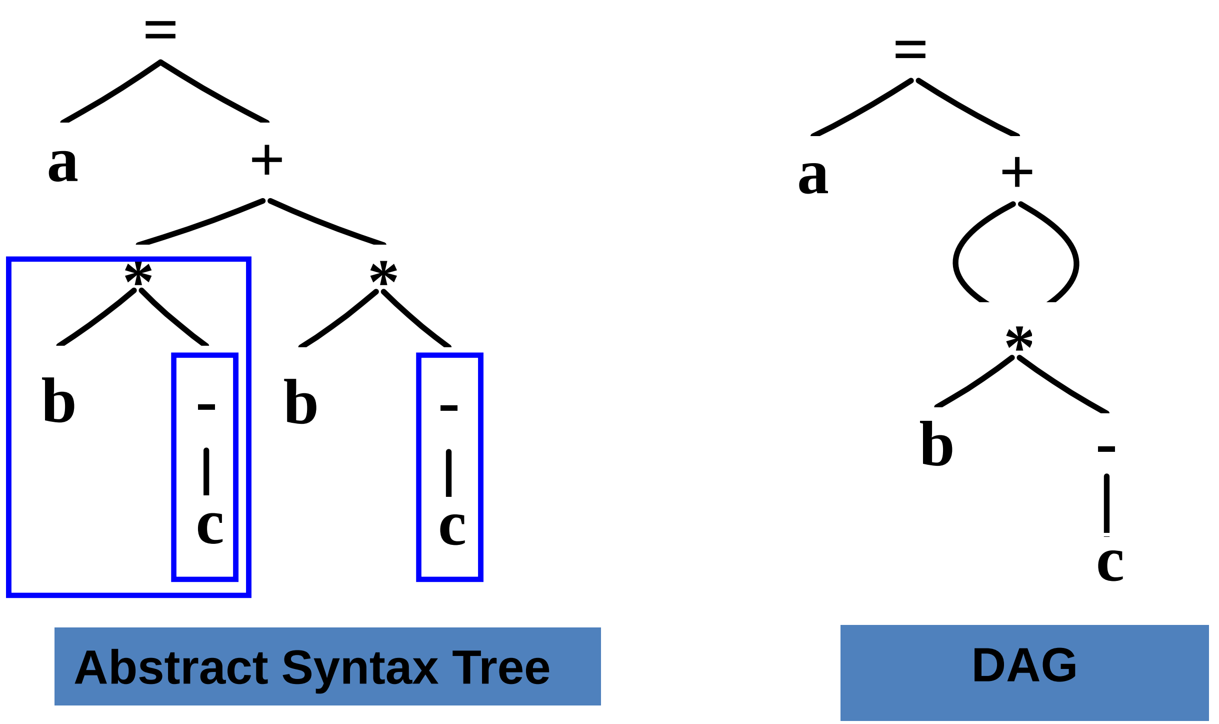
a + a * (b - c) + (b - c) * d



	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Directed Acyclic Graphs

- a=(b∗(-c))+ (b∗(-c))
- Postfix : a b c uminus ∗ b c uminus ∗ + =



Postfix Notation

- $a*b+c \quad \Rightarrow \quad ab * c +$

$E \rightarrow T + E_1$	$E.code = T.code \,\|\|\, E_1.code \,\|\|\, +$
$E \rightarrow T$	$E.code = T.code$
$T \rightarrow F * T_1$	$T.code = F.code \,\|\|\, T_1.code \,\|\|\, *$
$T \rightarrow F$	$E.code = F.code$
$F \rightarrow (E)$	$E.code = E.code$
$F \rightarrow id$	$F.code = id$

- $(E, E.code) \Rightarrow (T+E_1,\ T.code || E_1.code || +)$

$\Rightarrow (F * T_1 + E_1,\ F.code || T_1.code || * || E_1.code || +)$

$\Rightarrow (a * T_1 + E_1,\ a || T_1.code || * || E_1.code || +)$

$\Rightarrow (a * F + E_1,\ a || F.code || * || E_1.code || +)$

$\Rightarrow (a * b + E_1,\ a || b || * || E_1.code || +)$

$\Rightarrow (a * b + T,\ a || b || * || T.code || +)$

$\Rightarrow (a * b + F,\ a || b || * || F.code || +)$

$\Rightarrow (a * b + c,\ a || b || * || c || +)$

Three-Address Code (Quadruples)

- `OP    y, z, x`

  operation operands result

- Has three names/addresses (`x,y,z`), or less
- A single operator (`OP`)

- We will write as: $x \leftarrow y \text{ OP } z$

<u>Example:</u>

$x \leftarrow (y + z)*(-r);$

$t1 \leftarrow y + z$

$t2 \leftarrow -r$

$t3 \leftarrow t1 * t2$

Three-Address Code (Quadruples)

- A quadruple is:

  $x := y$ ***op*** z

  where x, y and z are names, constants or compiler-generated temporaries; **op** is any operator.
- But we may also the following notation for quadruples (much better notation because it looks like a machine code instruction)

  op x, y, z

  apply operator op to y and z, and store the result in x.
- We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

Three-Address Code

Binary Operator:

`op result,y,z` **or** **`result := y op z`**

where `op` is a binary arithmetic or logical operator. This binary operator is applied to `y` and `z`, and the result of the operation is stored in `result`.

Ex:

```
add  a,b,c
addi a,b,c
gt   a,b,c
```

Unary Operator:

`op result,,y` **or** **`result := op y`**

where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in `result`.

Ex:

```
uminus    a,,c
not       a,,c
inttoreal a,,c
```

Three-Address Code(cont.)

Move Operator:

`mov result,,y` **or** **`result := y`**

where the content of y is copied into `result`.

Ex:
```
mov    a,,c
movi   a,,c
movr   a,,c
```

Unconditional Jumps:

`jmp ,,L` **or** **`goto L`**

We will jump to the three-address code with the label L, and the execution continues from that statement.

Ex:
```
jmp   ,,L1  // jump to L1
jmp   ,,7   // jump to the statement 7
```

Three-Address Code(cont.)

Conditional Jumps:

`jmp` ***`relop`*** **`y,z,L`** or

`if y` ***`relop`*** **`z goto L`**

We will jump to the three-address code with the label `L` if the result of y `relop z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Ex:

```
jmpgt   y,z,L1    // jump to L1 if y>z
jmpge   y,z,L1    // jump to L1 if y>=z
jmpeq   y,z,L1    // jump to L1 if y==z
jmpne   y,z,L1    // jump to L1 if y!=z
```

Our relational operator can also be a unary operator.

```
jmpnz   y,,L1     // jump to L1 if y is not zero
jmpz    y,,L1     // jump to L1 if y is zero
jmpt    y,,L1     // jump to L1 if y is true
jmpf    y,,L1     // jump to L1 if y is false
```

Three-Address Code(cont.)

Procedure Parameters:

`param x,,` or **`param x`**

Procedure Calls:

`call p,n,` or **`call p,n`**

where x is an actual parameter, we invoke the procedure p with n parameters.

Ex: param x_1,,

param x_2,,

➔ $p(x_1, \ldots, x_n)$

param x_n,,

call p,n,

```
f(x+1,y) ➔   add    t1,x,1
             param  t1,,
             param  y,,
             call   f,2,
```

Three-Address Code(cont.)

Indexed Assignments:

```
move x,,y[i] or  x := y[i]
move y[i],,x or  y[i] := x
```

Address and Pointer Assignments:

```
moveaddr x,,y  or  x := &y
movecont x,,y  or  x := *y
```

Three-Address Code Example 1

```
n = 0;
while ( n < 10 ) {
    n = n + 1;
}
```

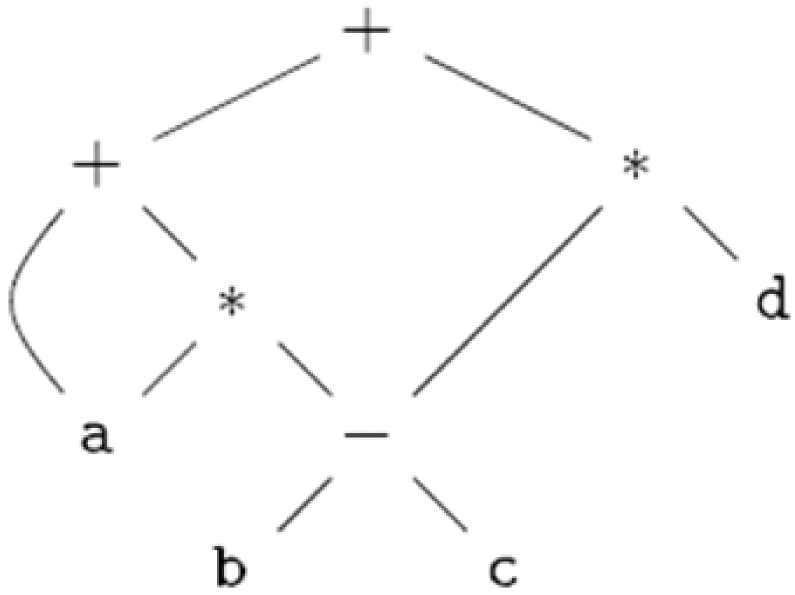
```
n ← 0
label lTEST
t2 ← n < 10
t3 ← NOT t2
cjmp t3 lEND
label lBODY
n ← n + 1
jump lTEST
label lEND
```

Three-Address Code Example 2

```
m = 0;
if ( c == 0 ) {
    m = m + n*n;
}  else {
    m = m + n;
}
```

```
m ← 0
t1 ← c == 0
cjmp t1 lTRUE
m ← m + n
jump lEND
label lTRUE
t2 ← n * n
m ← m + t2
label lEND
```

Different IRs



(a) DAG

$$
\begin{aligned}
t_1 &= b - c \\
t_2 &= a * t_1 \\
t_3 &= a + t_2 \\
t_4 &= t_1 * d \\
t_5 &= t_3 + t_4
\end{aligned}
$$

(b) Three-address code

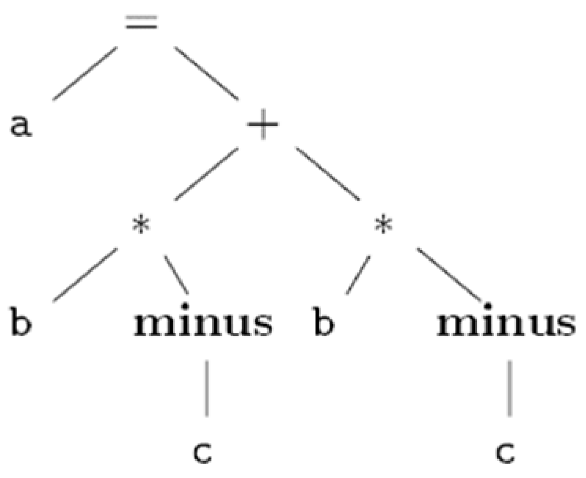
$a=(b*(-c))+(b*(-c))$

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

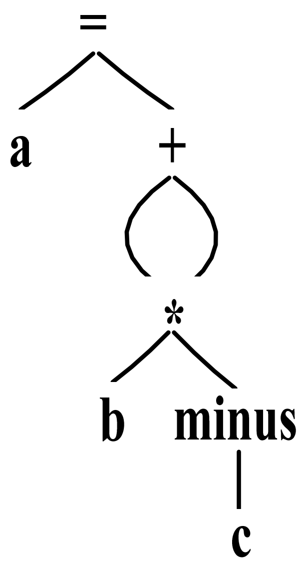
Three-address code

	op	arg1	arg2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		. . .		

Quadruples



Syntax tree



DAG

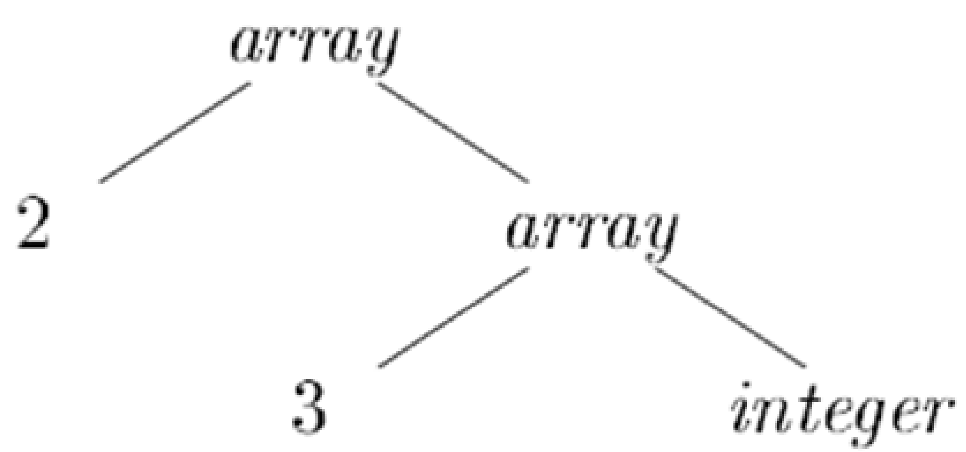
op *arg*$_1$ *arg*$_2$ *result*

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

Triples

Type Expressions

- A type expression is either a basic type or is formed by applying an operator called a type constructor to a type expression. The sets of basic types and constructors depend on the language to be checked.



Type expression for `int[2][3]`

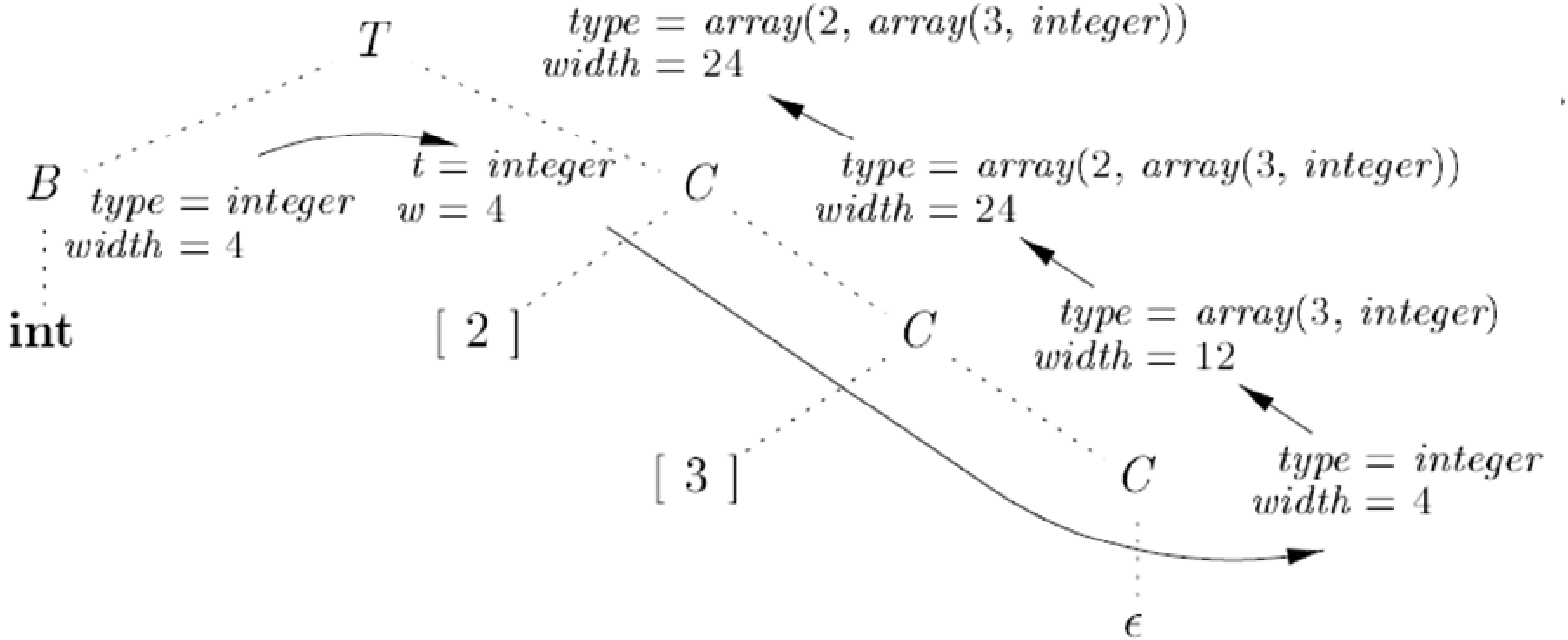
Declarations

- At compile time, we can use these amounts to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name .
- Data of varying length, such as strings, or data whose size cannot be determined until run time , such as dynamic arrays , is handled by reserving a known fixed amount of storage for a pointer to the data.

$$
\begin{array}{lcll}
T & \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \{\ T.type = C.type;\ T.width = C.width;\ \} \\
B & \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ C.type = array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Storage Layout for Local Names

- Syntax-directed translation of array types



Type Checking

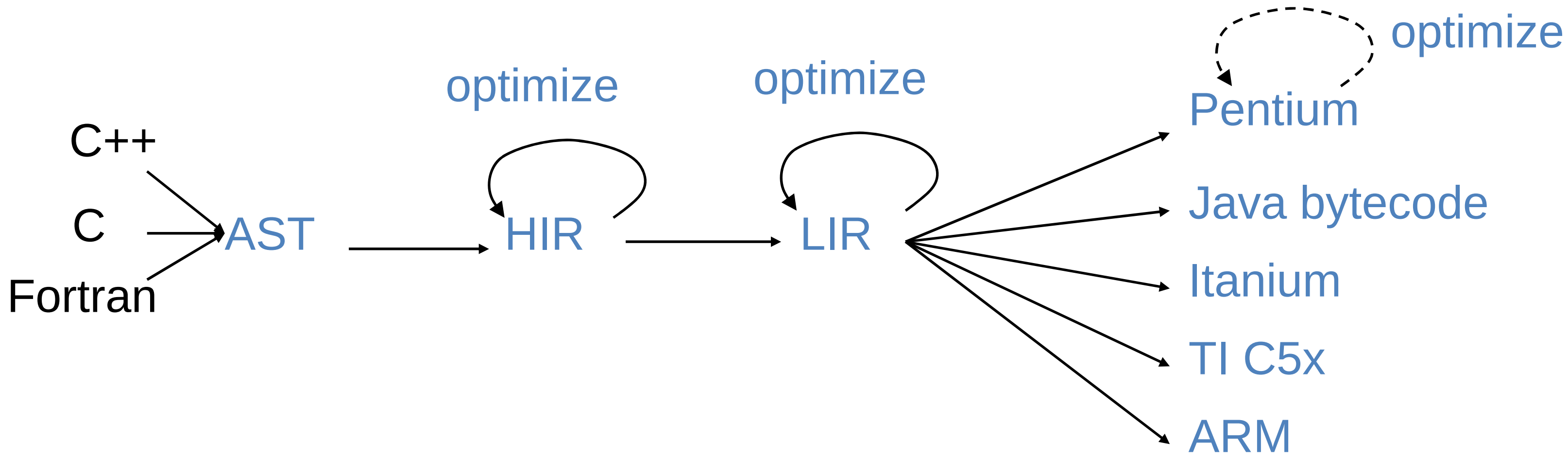
- To do type checking a compiler needs to assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logic al rules that is called the type

Type Systems

- A type system is a collection of rules for assigning type expressions to the parts of a program.
- A type checker implements a type system.
- A sound type system eliminates run-time type checking for type errors.
- A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.
  - In practice, some of type checking operations are done at run-time (so, most of the programming languages are not strongly-typed).
  - Ex: int x[100]; … x[i] most of the compilers cannot guarantee that i will be between 0 and 99

Multiple IRs

- Most compilers use 2 IRs:
  - High-level IR (HIR): Language independent but closer to the language
  - Low-level IR (LIR): Machine independent but closer to the machine
  - A significant part of the compiler is both language and machine independent!



Type Expressions

- The type of a language construct is denoted by a ***type expression***.
- A *type expression* can be:
  - **A basic type**
    - a primitive data type such as *integer, real, char, boolean*, …
    - *type-error* to signal a type error
    - *void* : no type
  - **A type name**
    - a name can be used to denote a type expression.

A Simple Type Checking System

- The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub-expressions.

P → D; E

D → D; D

D → **id**: T { addtype(id.entry,T.type) }

T → char { T.type=char }

T → int { T.type=int }

T → real { T.type=real }

T → ↑T_1 { T.type=pointer(T_1.type) }

T → array[**intnum**] of T_1 { T.type=array(0..intnum.val,T_1.type) }

Type Checking of Expressions

$E \rightarrow$ **id** { E.type=lookup(id.entry) }

$E \rightarrow$ **charliteral** { E.type=char }

$E \rightarrow$ **intliteral** { E.type=int }

$E \rightarrow$ **realliteral** { E.type=real }

$E \rightarrow E_1 + E_2$ { if (E_1.type=int and E_2.type=int) then E.type=int
else if (E_1.type=int and E_2.type=real) then E.type=real
else if (E_1.type=real and E_2.type=int) then E.type=real
else if (E_1.type=real and E_2.type=real) then E.type=real
else E.type=type-error }

$E \rightarrow E_1 [E_2]$ { if (E_2.type=int and E_1.type=array(s,t)) then E.type=t
else E.type=type-error }

$E \rightarrow \uparrow E_1$ { if (t=pointer(E1.type)) then E.type=t
else E.type=type-error }

Type Checking of Functions

$E \rightarrow E_1 (E_2)$ { if (E_2.type=s and E_1.type=s$\rightarrow$t) then E.type=t
else E.type=type-error }

Ex: int f(double x, char y) { ... }

f: double ***x*** char $\rightarrow$ int

argument types return type

Structural Equivalence of Type Expressions

- How do we know that two type expressions are equal?
- As long as type expressions are built from basic types (no type names), we may use **structural equivalence** between two type expressions

Structural Equivalence Algorithm (sequiv):

if (s and t are same basic types) then return true

else if ($s=array(s_1,s_2)$ and $t=array(t_1,t_2)$) then return ($sequiv(s_1,t_1)$ and $sequiv(s_2,t_2)$)

else if ($s = s_1 \times s_2$ and $t = t_1 \times t_2$) then return ($sequiv(s_1,t_1)$ and $sequiv(s_2,t_2)$)

else if ($s=pointer(s_1)$ and $t=pointer(t_1)$) then return ($sequiv(s_1,t_1)$)

else if ($s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$) then return ($sequiv(s_1,t_1)$ and $sequiv(s_2,t_2)$)

else return false

Type Coercions

x + y **what is the type of this expression (int or double)?**

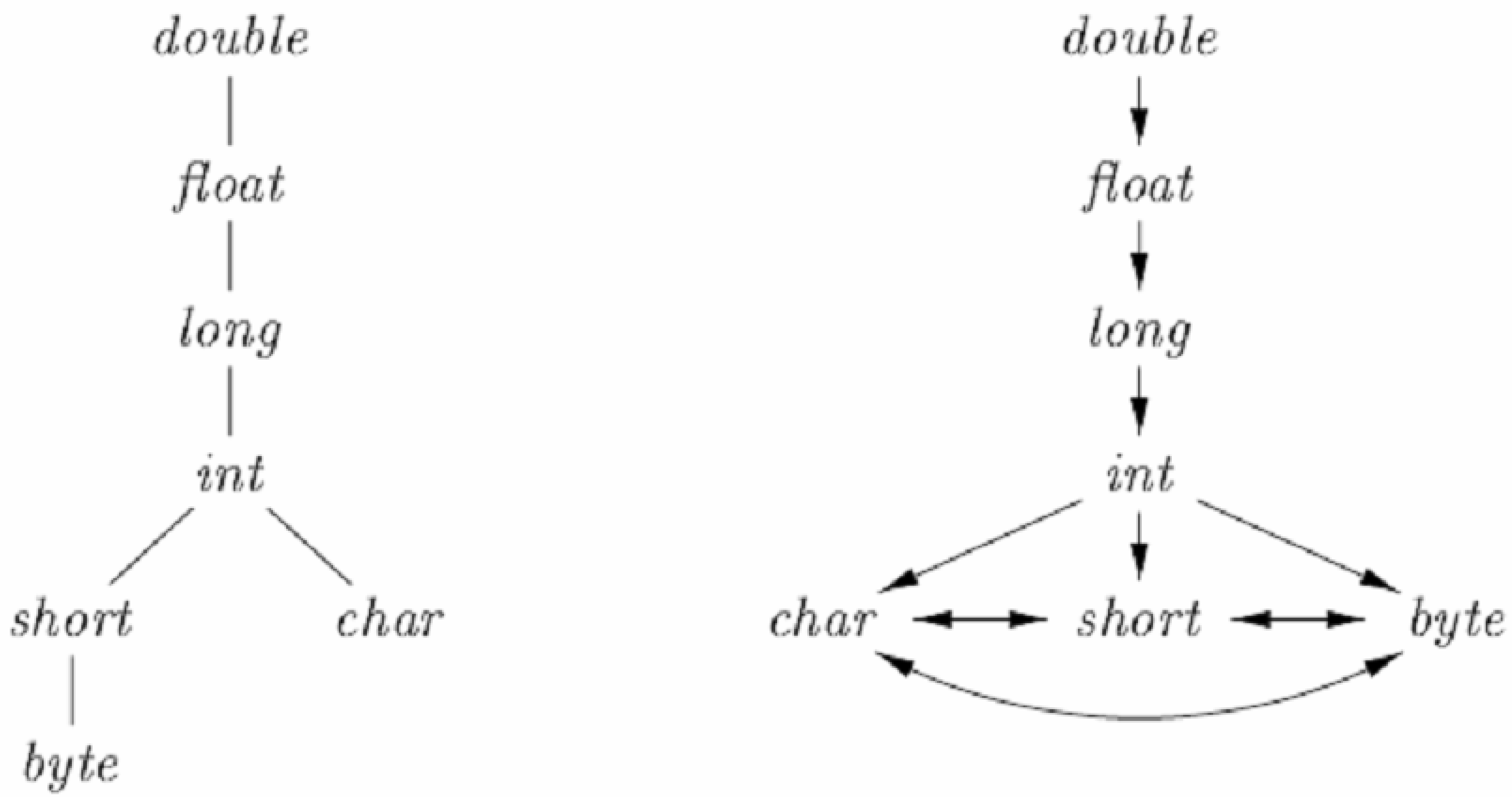
- What kind of codes we have to produce, if the type of x is double and the type of y is int?

```
inttoreal  t1,,y
real+      t2,x,t1      (addf $f3,$f1,$f2)
```

```
                             t2     x     t1
```

Conversion from one type to another is said to be implicit if it is done automatically by the compiler. Implicit type conversions, also called coercions. Conversion is said to be explicit if the programmer must write something to cause the conversion. Explicit conversions are also called casts.

Type Coercions (cont.)



(a) Widening conversions (b) Narrowing conversions

Overloading

- Expressions might have a multitude of types.

$\mathbf{E} \to \mathbf{id}$ $\{\mathbf{E}.\textit{\textbf{types}} = \textit{\textbf{lookup}}(\mathbf{id}.\textit{\textbf{entry}})\}$

$\mathbf{E} \to \mathbf{E_1 + E_2}$ $\{\mathbf{E}.\textit{\textbf{types}} = \{\ \mathbf{t}\ |$ there exists a type in $\mathbf{E_1}.\textit{types}$ and $\mathbf{E_2}.\textit{types}$ simultaneously $\}$

$\mathbf{E} \to \mathbf{E_1(E_2)}$ $\{\mathbf{E}.\textit{\textbf{types}} = \{\ \mathbf{t}\ |$ there exists a s in $\mathbf{E_2}.\textit{types}$ such that $s \to t$ belongs to $\mathbf{E_1}.\textit{types}$ $\}$

- An overloaded symbol is one that has different meanings depending on its context.
  - The type of E is determined by looking at the possible types of E_1 and E_2.
- Type error will occur when E.*types* becomes empty for some expression.

SDD of Expressions

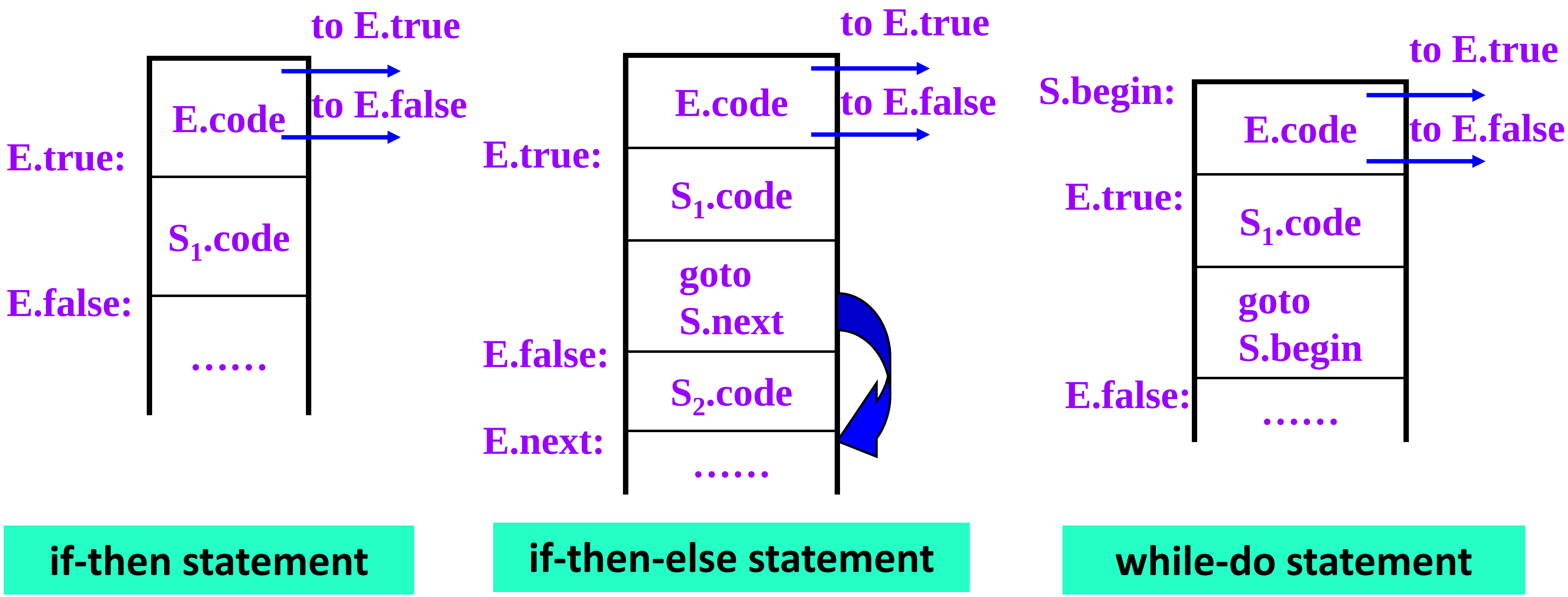
$S \rightarrow$ **id** := E	S.code = E.code || gen('mov' id.place ',,' E.place)
$E \rightarrow E_1 + E_2$	E.place = newtemp();
	E.code = E_1.code || E_2.code || gen('add' E.place ','E_1.place ',' E_2.place)
$E \rightarrow E_1 * E_2$	E.place = newtemp();
	E.code = E_1.code || E_2.code || gen('mult' E.place ',' E_1.place ',' E_2.place)
$E \rightarrow - E_1$	E.place = newtemp();
	E.code = E_1.code || gen('uminus' E.place ',,' E_1.place)
$E \rightarrow (E_1)$	E.place = E_1.place;
	E.code = E_1.code
$E \rightarrow$ **id**	E.place = **id**.place;
	E.code = null

Translation Scheme to Produce Three-Address Code -- Assignment Statements

$S \rightarrow$ **id** := E	{ p= lookup(id.name);
	if (p is not nil) then emit('mov' p',,' E.place)
	else error("undefined-variable") }
$E \rightarrow E_1 + E_2$	{ E.place = newtemp();
	emit('add' E.place ',' E_1.place ',' E_2.place) }
$E \rightarrow E_1 * E_2$	{ E.place = newtemp();
	emit('mult' E.place ',' E_1.place ',' E_2.place) }
$E \rightarrow - E_1$	{ E.place = newtemp();
	emit('uminus' E.place ',,' E_1.place) }
$E \rightarrow (E_1)$	{ E.place = E_1.place; }
$E \rightarrow$ **id**	{ p= lookup(id.name);
	if (p is not nil) then E.place = **id**.place
	else error("undefined-variable") }

Flow-of-Control Statement

- $S \rightarrow$ if E then S_1 | if E then S_1 else S_2 | while E do S_1



PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \; ! \; B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \; \mathbf{rel} \; E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\quad \mid\mid gen(\text{'}\texttt{if}\text{'} \; E_1.addr \; \mathbf{rel}.op \; E_2.addr \; \text{'}\texttt{goto}\text{'} \; B.true)$ $\quad \mid\mid gen(\text{'}\texttt{goto}\text{'} \; B.false)$
$B \rightarrow \mathbf{true}$	$B.code = gen(\text{'}\texttt{goto}\text{'} \; B.true)$
$B \rightarrow \mathbf{false}$	$B.code = gen(\text{'}\texttt{goto}\text{'} \; B.false)$

Figure 6.37: Generating three-address code for booleans

```
      if x < 100 goto L2
      goto L3
L3:   if x > 200 goto L4
      goto L1
L4:   if x != y goto L2
      goto L1
L2:   x = 0
L1:
```

Figure 6.38: Control-flow translation of a simple if-statement

1)	$B \rightarrow B_1 \; \texttt{||} \; M \; B_2$	$\{$ $backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist;$ $\}$
2)	$B \rightarrow B_1 \; \&\& \; M \; B_2$	$\{$ $backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);$ $\}$
3)	$B \rightarrow \; ! \; B_1$	$\{$ $B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist;$ $\}$
4)	$B \rightarrow (\; B_1 \;)$	$\{$ $B.truelist = B_1.truelist;$ $B.falselist = B_1.falselist;$ $\}$
5)	$B \rightarrow E_1 \; \mathbf{rel} \; E_2$	$\{$ $B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $gen('\texttt{if}' \; E_1.addr \; \mathbf{rel}.op \; E_2.addr \; '\texttt{goto \_}');$ $gen('\texttt{goto \_}');$ $\}$
6)	$B \rightarrow \mathbf{true}$	$\{$ $B.truelist = makelist(nextinstr);$ $gen('\texttt{goto \_}');$ $\}$
7)	$B \rightarrow \mathbf{false}$	$\{$ $B.falselist = makelist(nextinstr);$ $gen('\texttt{goto \_}');$ $\}$
8)	$M \rightarrow \epsilon$	$\{$ $M.instr = nextinstr;$ $\}$

Figure 6.43: Translation scheme for boolean expressions

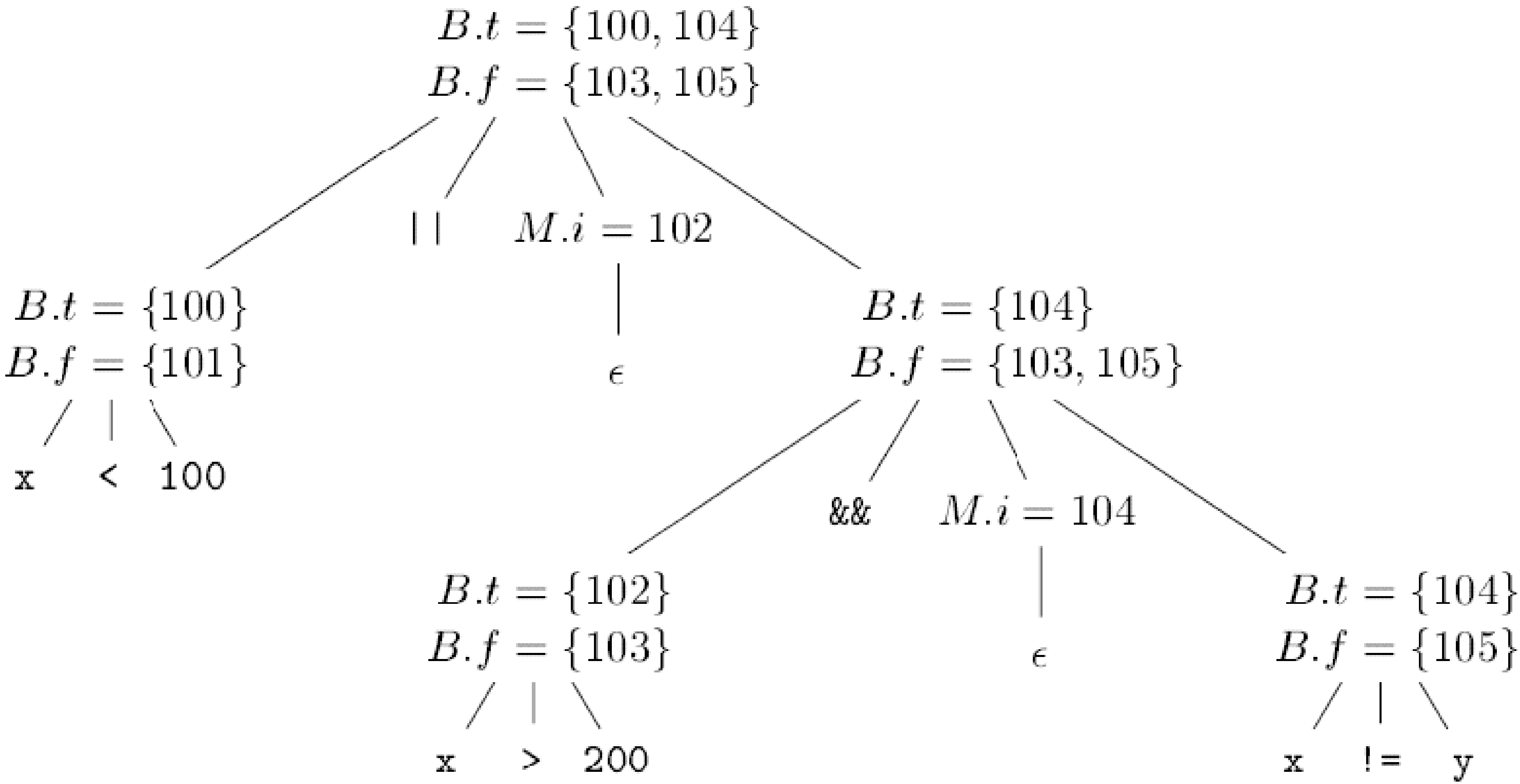


Figure 6.44: Annotated parse tree for $x < 100$ || $x > 200$ && x != y

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(a) After backpatching 104 into instruction 102.

```
100:  if x < 100 goto _
101:  goto 102
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(b) After backpatching 102 into instruction 101.

Figure 6.45: Steps in the backpatch process

1) $S \rightarrow \textbf{if} \, (\, B \,) \, M \, S_1$ $\{$ $backpatch(B.truelist, \; M.instr);$
$S.nextlist \; = \; merge(B.falselist, \; S_1.nextlist);$ $\}$

2) $S \rightarrow \textbf{if} \, (\, B \,) \, M_1 \, S_1 \, N \, \textbf{else} \, M_2 \, S_2$
$\{$ $backpatch(B.truelist, \; M_1.instr);$
$backpatch(B.falselist, \; M_2.instr);$
$temp \; = \; merge(S_1.nextlist, \; N.nextlist);$
$S.nextlist \; = \; merge(temp, \; S_2.nextlist);$ $\}$

3) $S \rightarrow \textbf{while} \, M_1 \, (\, B \,) \, M_2 \, S_1$
$\{$ $backpatch(S_1.nextlist, \; M_1.instr);$
$backpatch(B.truelist, \; M_2.instr);$
$S.nextlist \; = \; B.falselist;$
$gen('\texttt{goto}' \; M_1.instr);$ $\}$

4) $S \rightarrow \{ \, L \, \}$ $\{$ $S.nextlist \; = \; L.nextlist;$ $\}$

5) $S \rightarrow A \; ;$ $\{$ $S.nextlist \; = \; \textbf{null};$ $\}$

6) $M \rightarrow \epsilon$ $\{$ $M.instr \; = \; nextinstr;$ $\}$

7) $N \rightarrow \epsilon$ $\{$ $N.nextlist \; = \; makelist(nextinstr);$
$gen('\texttt{goto \_}');$ $\}$

8) $L \rightarrow L_1 \; M \; S$ $\{$ $backpatch(L_1.nextlist, \; M.instr);$
$L.nextlist \; = \; S.nextlist;$ $\}$

9) $L \rightarrow S$ $\{$ $L.nextlist \; = \; S.nextlist;$ $\}$

Figure 6.46: Translation of statements

Reference

- Compilers Principles, Techniques & Tools (Second Edition) Alfred Aho., Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007
  - Chapter 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7.1, 6.7.2, 6.7.3

- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu