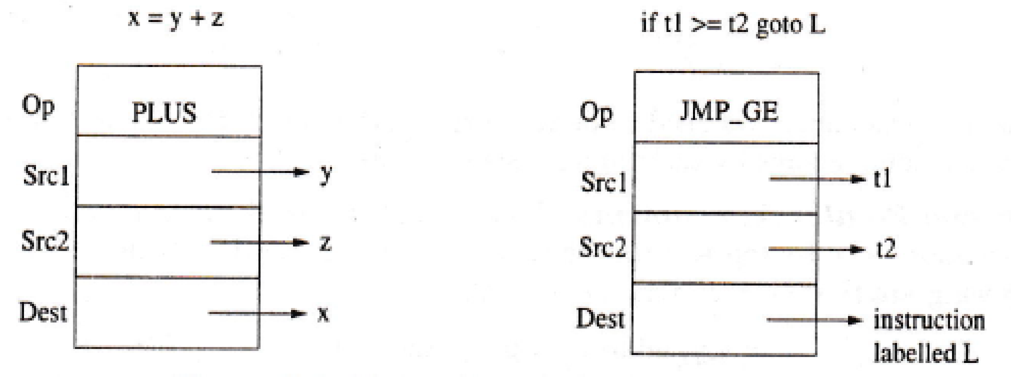
Miscellaneous Statements

- More statements may be needed depending upon the source language
- One such statement is to define jump target as,

label L

Three-Address Instruction Implementation

- Quadruple representation – each instruction has at most four fields:
  - Operation – identifying the operation to be carried out
  - Upto two operands – a bit is used to indicate whether it is a constant or a pointer
  - Destination



Example

if x+2 > 3*(y-1)+4 then z = 0

```
t1 = x + 2
t2 = y – 1
t3 = 3 * t2
t4 = t3 + 4
if t1 ≤ t4 goto L
z = 0
Label L
```

Three Address Code Generation - Assignment

Grammar:

S → id := E

E → E + E | E * E | -E | (E) | id

Grammar Rule	Semantic Actions
$S \rightarrow \mathbf{id} := E$	$S.code := E.code \| \| gen(\mathbf{id}.place \text{ ':=' } E.place)$
$E \rightarrow E_1 + E_2$	$E.place :=$ newtemp(); $E.code := E_1.code \| \| E_2.code \| \| gen(E.place \text{ ':=' } E_1.place \text{ '+' } E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place :=$ newtemp(); $E.code := E_1.code \| \| E_2.code \| \| gen(E.place \text{ ':=' } E_1.place \text{ '*' } E_2.place)$
$E \rightarrow -E_1$	$E.place :=$ newtemp(); $E.code := E_1.code \| \| gen(E.place \text{ ':=' } \text{'uminus'} E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow \mathbf{id}$	$E.place := \mathbf{id}.place;$ $E.code :=$ ' '

Attributes for non-terminal E:

- E.place – name that will hold value of E
- E.code – sequence of three address statements corresponding to evaluation of E

Attributes for non-terminal S:

- S.code – sequence of three-address statements

Attributes for terminal symbol id:

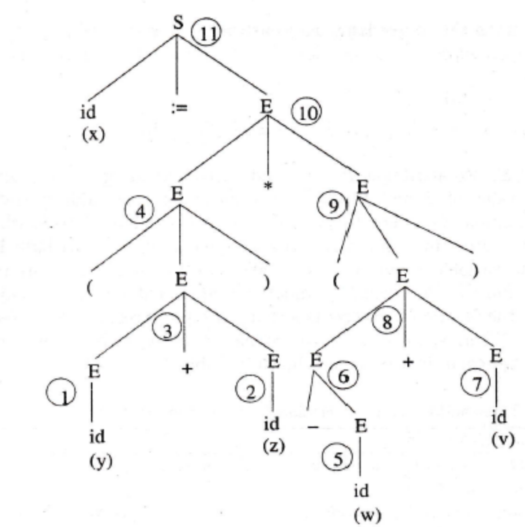
- id.place – contains the name of the variable to be assigned

- Function *newtemp* returns a unique new temporary variable.
- Function *gen* accepts a string and produces it as a three-address quadruple.
- '||' concatenates two three-address code segments

Example

x := (y+z)*(-w+v)

Reduction No.	Action
1	$E.place = y$
2	$E.place = z$
3	$E.place = t_1$
	$E.code = \{t_1 := y + z\}$
4	$E.place = t_1$
	$E.code = \{t_1 := y + z\}$
5	$E.place = w$
6	$E.place = t_2$
	$E.code = \{t_2 := uminus \;\; w\}$
7	$E.place = v$
8	$E.place = t_3$
	$E.code = \{t_2 := uminus \;\; w, t_3 := t_2 + v\}$
9	$E.place = t_3$
	$E.code = \{t_2 := uminus \;\; w, t_3 := t_2 + v\}$
10	$E.place = t_4$
	$E.code = \{t_1 := y + z, t_2 := uminus \;\; w, t_3 := t_2 + v, t_4 := t_1 * t_3\}$
11	$S.code = \{t_1 := y + z, t_2 := uminus \;\; w, t_3 := t_2 + v, t_4 := t_1 * t_3, x := t_4\}$



Code Generation for Arrays

- Consider array element *A[i]*
- Assume lowest and highest indices of *A* are *low* and *high*, width of each element *w* and start address of *A, base*
- Element A[i] starts at location *(base + (i – low)*w) = ((base – low*w) + i*w)*
- First part of the expression can be precomputed into a constant and added to the offset *i*w*

Code Generation for Arrays

- For two-dimensional array with row-major storage, $A[i_1,i_2]$ starts at location

$$base + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$$

$$= \quad base - ((\mathrm{low}_1 * \mathrm{n}_2) + low_2) * w) + ((i_1 * n_2) + i_2) * w$$

where n_2 is the size of the second dimension

- This can be extended to higher dimensions

Array Translation Scheme

Grammar:

S → L := E

E → E + E | (E) |L

L → Elist] | id

Elist → Elist, E | id[E

Attributes:

- L.place: holds name of the variable (may be array name also)
- L.offset: null for simple variable, offset of the element for array
- E.place: name of the variable holding value of expression E
- Elist.array: holds the name of the array referred to
- Elist.place: name of the variable holding value for index expression
- Elist.dim: holds current dimension under consideration for array

Semantic Actions for Arrays

```
S → L := E
    { if L.offset = null then
          emit(L.place ':=' E.place);
      else
          emit(L.place '[' L.offset ']' ':=' E.place
    }
```

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

```
E → (E1)
      { E.place := E1.place}
```

Semantic Actions for Arrays

```
E → L
   { if L.offset = null then
         E.place = L.place
     else
         E.place = newtemp()
         emit(E.place ':=' L.place '[' L.offset ']')
   }
```

```
L → id
   { L.place = id.place
     L.offset ':=' null
   }
```

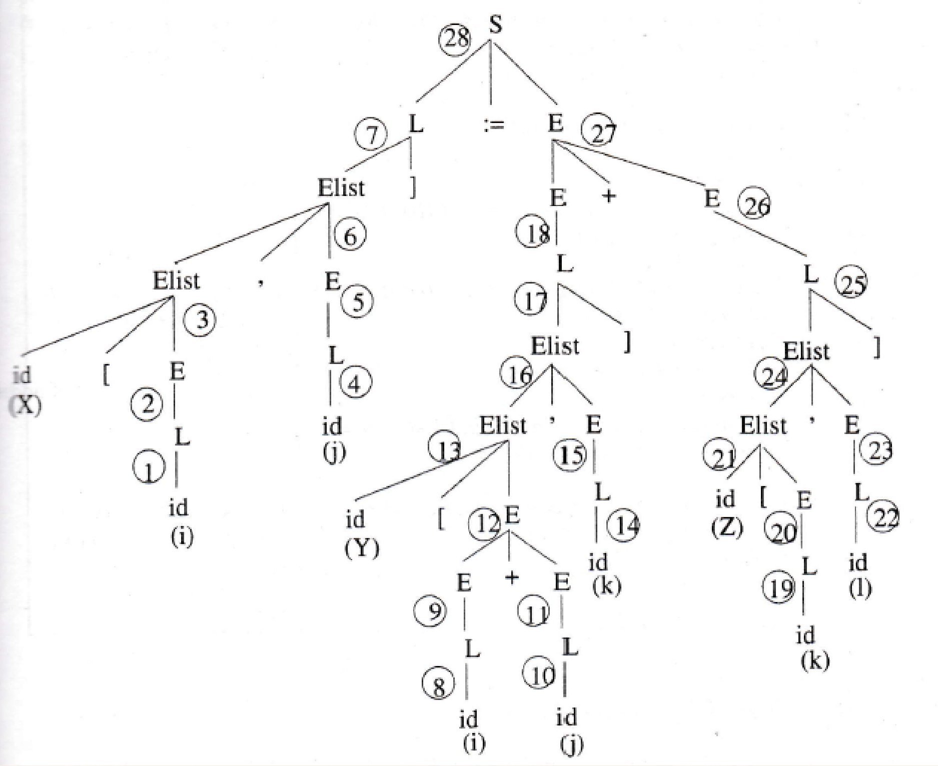
```
L → Elist ]
   { L.place = newtemp()
     L.offset = newtemp()
     emit(L.place ':=' c(Elist.array) /* c returns constant part of the array */
     emit(L.offset ':=' Elist.place * width(Elist.array))
   }
```

Semantic Actions for Arrays

```
Elist → Elist1, E
    { t = newtemp()
      m = Elist1.dim + 1
      emit(t ':=' Elist1.place '*' limit(Elist1.array, m))
      emit(t ':=' t '+' E.place
      Elist.array = Elist1.array
      Elist.place = t
      Elist.dim = m
    }
```

```
Elist → id [E
    { Elist.array = id.place
      Elist.place = E.place
      Elist.dim = 1
    }
```

Example



X[i, j] := Y[i + j, k] + Z[k, l]

Array dimensions:

$X[d_1, d_2], Y[d_3, d_4], Z[d_5, d_6]$

Each element of width *w*

Example

Step No.	Attribute assignment	Code generated
1	$L.place = i, L.offset = null$	
2	$E.place = i$	
3	$Elist.array = X, Elist.place = i, Elist.dim = 1$	
4	$L.place = j, L.offset = null$	
5	$E.place = j$	
6	$Elist.array = X, Elist.place = t_1, Elist.dim = 2$	$t_1 = i * d_2, t_1 := t_1 + j$
7	$L.place = t_2, L.offset = t_3$	$t_2 := C(X), t_3 := t_1 * w$
8	$L.place = i, L.offset = null$	
9	$E.place = i$	
10	$L.place = j, L.offset = null$	
11	$E.place = j$	
12	$E.place = t_4$	$t_4 := i + j$
13	$Elist.array = Y, Elist.place = t_4, Elist.dim = 1$	
14	$L.place = k, L.offset = null$	
15	$E.place = k$	
16	$Elist.array = Y, Elist.place = t_5, Elist.dim = 2$	$t_5 := t_4 * d_4, t_5 := t_5 + k$
17	$L.place = t_6, L.offset = t_7$	$t_6 := C(Y), t_7 := t_5 * w$
18	$E.place = t_8$	$t_8 := t_6[t_7]$
19	$L.place = k, L.offset = null$	
20	$E.place = k$	
21	$Elist.array = Z, Elist.place = k, Elist.dim = 1$	
22	$L.place = l, L.offset = null$	
23	$E.place = l$	
24	$Elist.array = Z, Elist.place = t_9, Elist.dim = 2$	$t_9 := k * d_6, t_9 := t_9 + l$
25	$L.place = t_{10}, L.offset = t_{11}$	$t_{10} := C(Z), t_{11} := t_9 * w$
26	$E.place = t_{12}$	$t_{12} := t_{10}[t_{11}]$
27	$E.place = t_{13}$	$t_{13} := t_8 + t_{12}$
28		$t_2[t_3] := t_{13}$

Translation of Boolean Expressions

Attributes of Boolean expression *B*:

1. *B.true*: defines place, control should reach if B is true
2. *B.false*: defines place, control should reach if B is false

Grammar:

B → *B* **or** *B*
  | *B* **and** *B*
  | **not** *B*
  | (*B*)
  | **id relop id**
  | **true**
  | **false**

- Assumed true and false transfer points for entire expression B is known
- If B1 is true, B is true ➔ need not evaluate B2 ➔ called short-circuit evaluation
- If B1 is false, B2 needs to be evaluated
- Thus, B1.false assigned a new label marking beginning of evaluation of B2
- Function newlabel() generates new label

```
B → B1 or B2
  { B1.true = B.true
    B1.false = newlabel()
    B2.true = B.true
    B2.false = B.false
    B.code = B1.code ||
          gen(B1.false, ':') ||
          B2.code
  }
```

Translation of Boolean Expressions

B → B1 and B2
{ B1.true = newlabel()
B1.false = B.false
B2.true = B.true
B2.false = B.false
B.code = B1.code ||
gen(B1.true, ‘:’) ||
B2.code
}

B → not B1
{ B1.true = B.false
B1.false = B.true
B.code = B1.code
}

B → (B1)
{ B1.true = B.true
B1.false = B.false
B.code = B1.code
}

B → id1 relop id2
{ B.code = gen(‘if’ id1.place relop id2.place ‘goto’ B.true) || gen(‘goto’ B.false)}

B → true
{ B.code = gen(‘goto’ B.true) }

B → false
{ B.code = gen(‘goto’ B.false) }

Disadvantages

- Makes the scheme inherently two-pass procedure
- All jump targets are computed in the first pass
- Actual code generation done in second pass
- A single-pass approach can be developed by
  - Modifying the grammar a bit, and
  - Introducing a few more attributes
  - A few new procedures
  - Generated code can be visualized as an array of quadruples

Attributes

- *B.truelist*
  - List of locations within the generated code for *B*, at which *B* definitely true
  - Once defined, all these points should transfer control to *B.true*
- *B.falselist*
  - List of locations within the generated code for *B*, at which *B* definitely false
  - Once defined, all these points should transfer control to *B.false*

Extra Functions

- *makelist(i)*: creates a new list with a single entry *i* – an index into the array of quadruples
- *mergelist(list1, list2)*: returns a new list containing *list1* followed by *list2*
- *backpatch(list, target)*: inserts the *target* as the target label into each quadruple pointed to by entries in the *list*
- *nextquad()*: returns the index of the next quadruple to be generated

Modified Grammar

$B \rightarrow B$ **or** MB
    | B **and** MB
    | **not** B
    | (B)
    | **id relop id**
    | **true**
    | **false**
$M \rightarrow \varepsilon$

M is a dummy nonterminal with attribute *M.quad*, that can hold index of a quadruple

Consider the rule $B \rightarrow B1$ *or* $MB2$:
Before the reduction of B2 starts, reduction $M \rightarrow \varepsilon$ has already taken place. Hence, M.quad points to the index of the first quadruple of B2

Translation Rules

```
B → B1 or MB2
    { backpatch(B1.falselist, M.quad)
      B.truelist = mergelist(B1.truelist, B2.truelist)
      B.falselist = B2.falselist
    }
```

```
B → B1 and MB2
    { backpatch(B1.truelist, M.quad)
      B.truelist = B2.falselist
      B.falselist = mergelist(B1.falselist, B2.falselist)
    }
```

```
B → not B1
    { B.truelist = B1.falselist
      B.falselist = B1.truelist
    }
```

```
B → (B1)
    { B.truelist = B1.truelist
      B.falselist = B1.falselist
    }
```

```
B → true
    { B.truelist = makelist(nextquad())
      emit('goto' ...)
    }
```

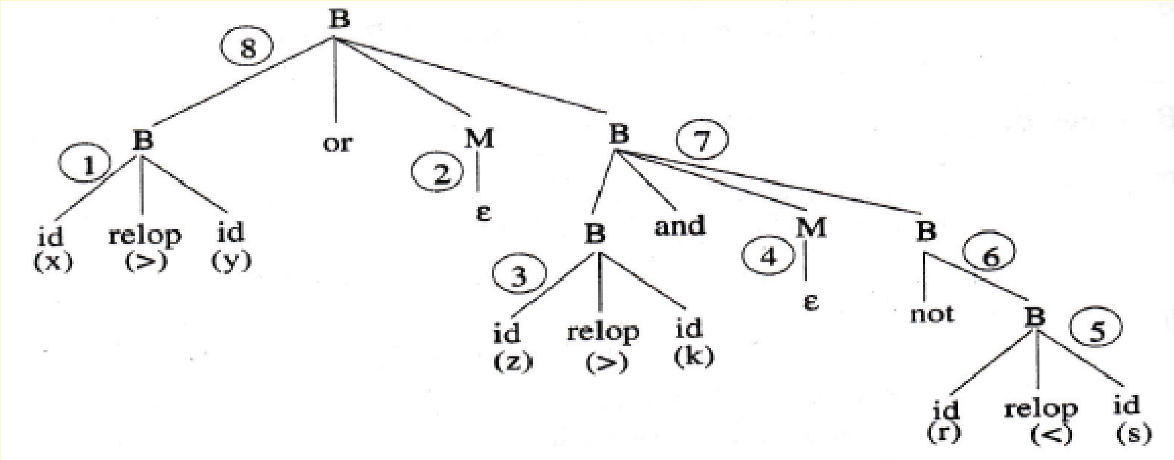
```
B → id1 relop id2
    { B.truelist = nextquad()
      B.falselist = nextquad()
      emit('if' id1.place relop id2.place 'goto' ...)
      emit('goto' ...)
    }
```

```
B → false
    { B.falselist = makelist(nextquad())
      emit('goto' ...)
    }
```

```
M → ε
    { M.quad = nextquad() }
```

Example

x > y or z > k and not r < s



Translation Example (Contd.)

Reduction	Action	Code generated
1	B.truelist = {1} B.falselist = {2}	1: if x > y goto ... 2: goto ...
2	M.quad = 3	
3	B.truelist = {3}, B.falselist = {4}	3: if z > k goto ... 4: goto ...
4	M.quad = 5	
5	B.truelist = {5} B.falselist = {6}	5: if r > s goto ... 6: goto ...
6	B.truelist = {6}, B.falselist = {5}	
7	Backpatches list {3} with 5	3: if z > k goto 5
8	Backpatches list {2} with 3 B.truelist = {1,6}, B.falselist = {4,5}	2: goto 3

Full Code:

1: if x > y goto ...

2: goto 3

3: if z > k goto 5

4: goto ...

5: if r < s goto ...

6: goto ...

1, 6 true exit,

4, 5 false exit

Control Flow Statements

- Most programming languages have a common set of statements
  - Assignment: assigns some expression to a variable
  - If-then-else: control flows to either then-part or else-part
  - While-do: control remains within loop until a specified condition becomes false
  - Block of statements: group of statements put within a *begin-end* block marker

Grammar

S → **if** *B* **then** *M S*
  | **if** *B* **then** *M S N* **else** *M S*
  | **while** *M B* **do** *M S*
  | **begin** *L* **end**
  | *A* **/* for assignment */**
L → *L M S*
  | *S*
M → ε
N → ε

Attributes:

- *S.nextlist*: list of quadruples containing jumps to the quadruple following S
- *L.nextlis*t: Same as S.nextlist for a group of statements

Nonterminal *N* enables to generate a jump after the *then*-part of *if-then-else* statement. *N.nextlist* holds the quadruple number for this statement

Translation Rules

```
S → if B then M S1
    { backpatch(B.truelist, M.quad)
      S.nextlist = mergelist(B.falselist, S1.nextlist)
    }
```

```
S → if B then M1 S1 N else M2 S2
    { backpatch(B.truelist, M1.quad)
      backpatch(B.falselist, M2.quad)
       S.nextlist = mergelist(S1.nextlist,
                         mergelist(N.nextlist, S2.nextlist))
    }
```

```
S → while M1 B do M2 S1
    { backpatch(S1.nextlist, M1.quad)
      backpatch(B.truelist, M2.quad)
       S.nextlist = B.falselist
       emit( 'goto' M1.quad)
    }
```