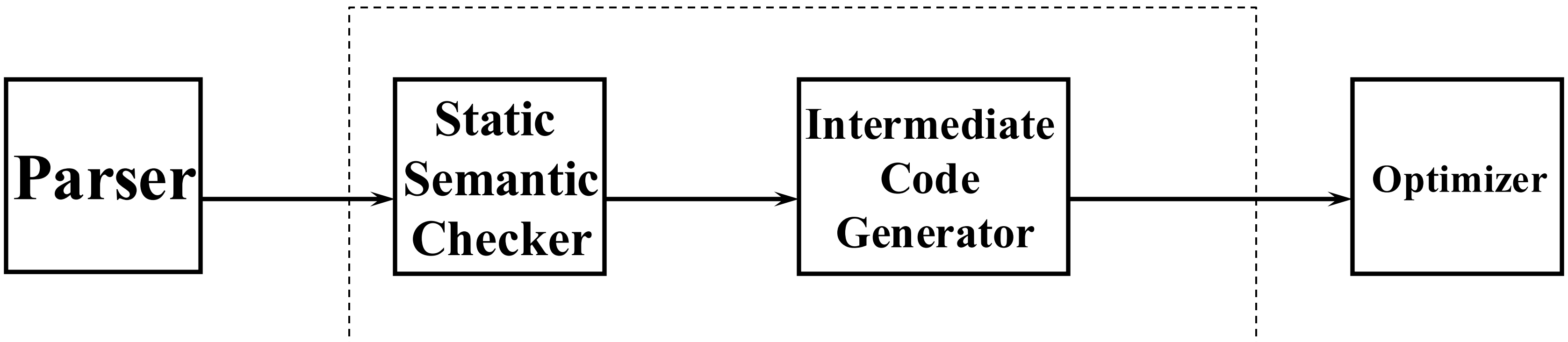
Chapter 7: Semantic Analysis and Intermediate Code Generation

Zhen Gao
gaozhen@Tongji.edu.cn

Outline

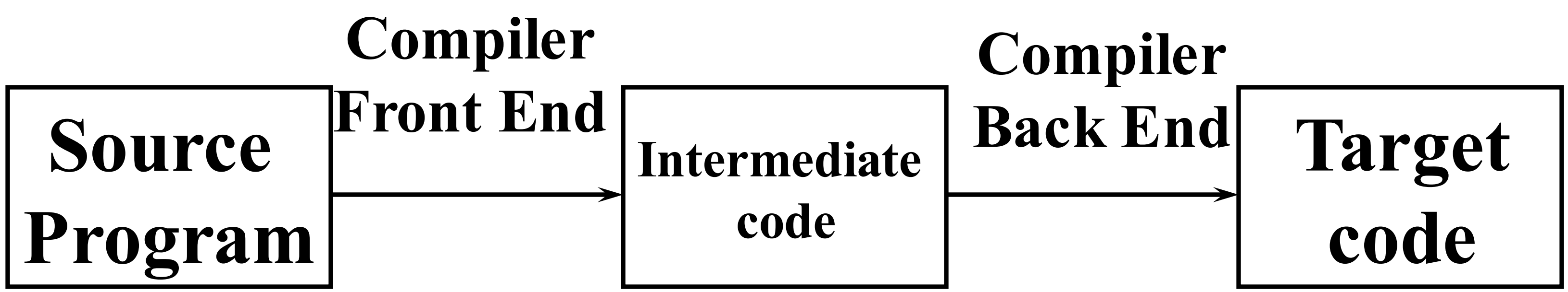
- Intermediate Language
- Translation of Declaration Statements
- Translation of Assignment Statements
- Translation of Boolean Expressions
- Translation of Flow-of-Control Statements
- Procedure Call Handling

Semantic Analysis and Intermediate Code Generation



- **Static Semantic Checks**
  - Type checking
  - Control flow checking
  - Consistency checking
  - Related name checking
  - Name scope analysis

- **Benefits of Intermediate Language** (complexity between source and target languages):
  - Facilitates **machine-independent code optimization**
  - Easier **portability**
  - Makes the **compiler structure logically simpler and clearer**



Intermediate Language

- **Postfix notation**
  - Reverse Polish Notation (RPN)
- **Graph representation**
  - DAG (Directed Acyclic Graph)
  - Abstract Syntax Tree (AST)
- **Three-Address Code (TAC)**
  - Triple
  - Quadruple
  - Indirect Triple

1-Postfix notation

Production	Semantic Rules
$E \rightarrow E^{(1)}$ op $E^{(2)}$	**$E.code := E^{(1)}.code \mid\mid E^{(2)}.code \mid\mid op$**
$E \rightarrow (E^{(1)})$	**$E.code := E^{(1)}.code$**
$E \rightarrow id$	**$E.code := id$**

- **E.code** represents the **postfix** form of expression E
- **op** denotes any binary **operator**
- || denotes **concatenation** of postfix expressions

Quiz

- **Postfix notation**

(1) a+b*(c+d/e)

(2) (A and B) or(not C or D)

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

(4) (A or B) and (C or not D and E)

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

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

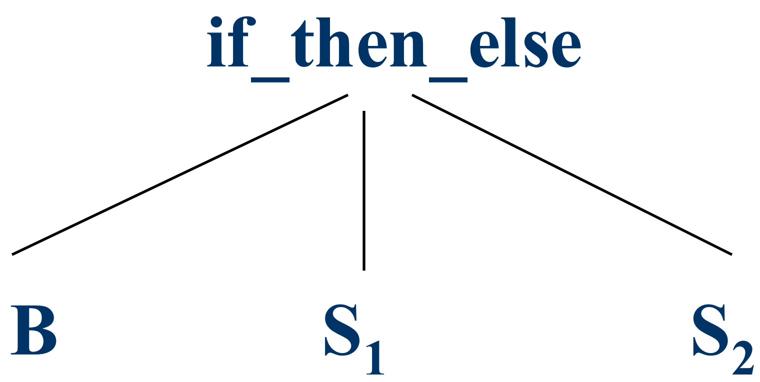
2-Graph Representations

- Abstract Syntax Tree (AST)
- DAG

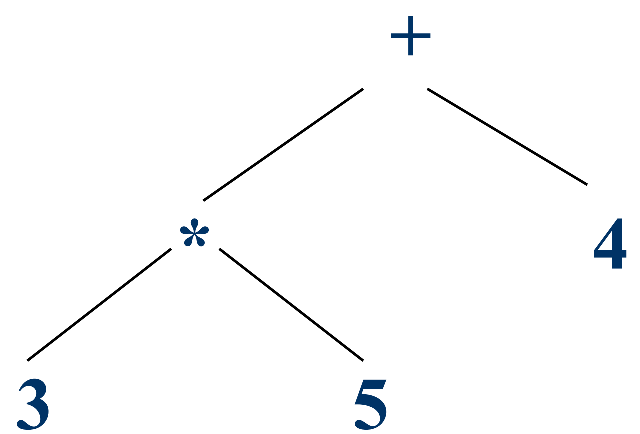
Abstract Syntax Tree (AST)

- In a syntax tree, remove information unnecessary for translation to obtain a more **efficient intermediate representation** of the source program
  - This transformed tree is called an **Abstract Syntax Tree (AST)**
  - **Operators** and **keywords** appear as **internal nodes**, not as leaves

- **$S \rightarrow$ if B then S_1 else S_2**



- **3*5+4**

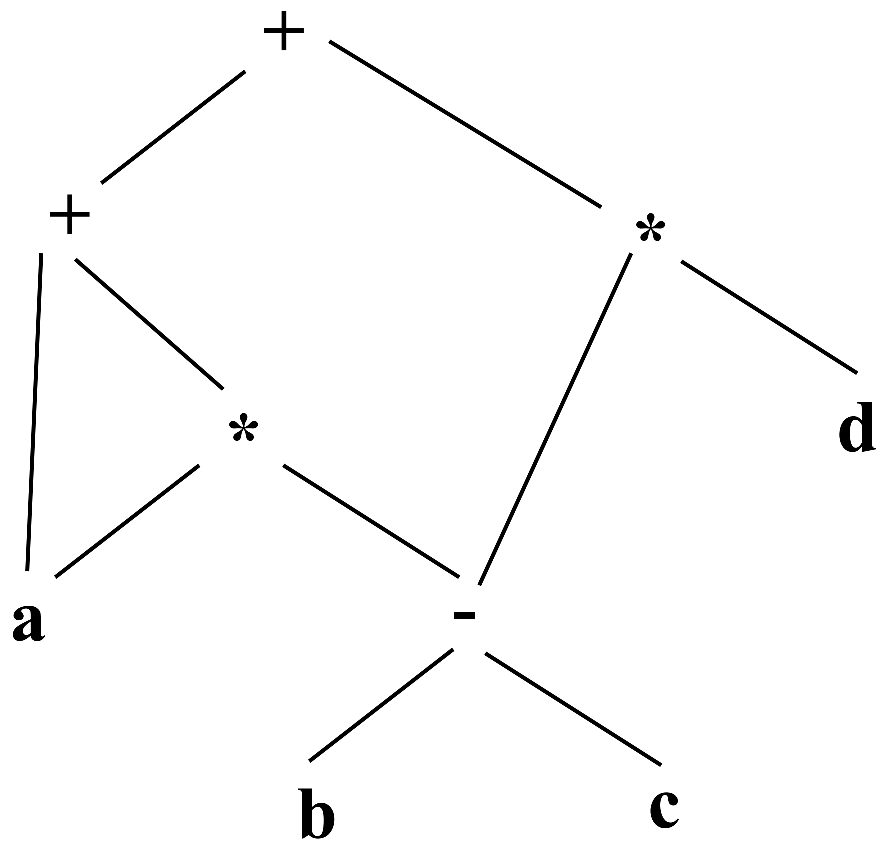


*

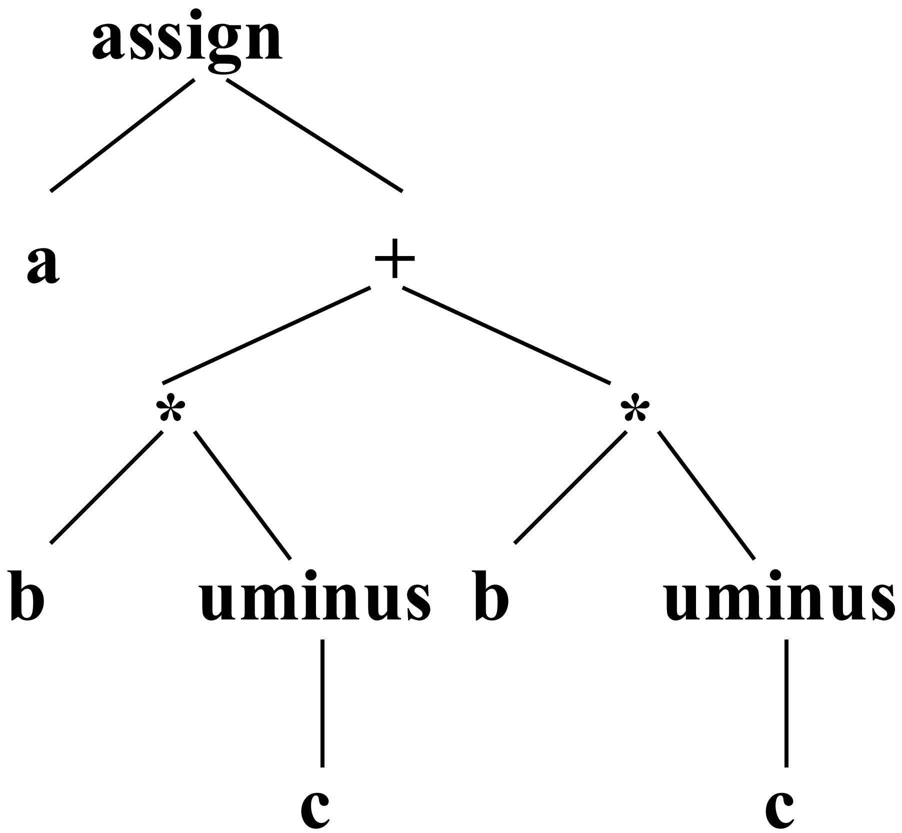
DAG

- **Directed Acyclic Graph (DAG)**
  - Each **subexpression** of an expression corresponds to a **node** in the DAG
  - An **internal node** represents an **operator**, and its children represent **operands**
  - Nodes representing **common subexpressions** have **multiple parents** in the DAG

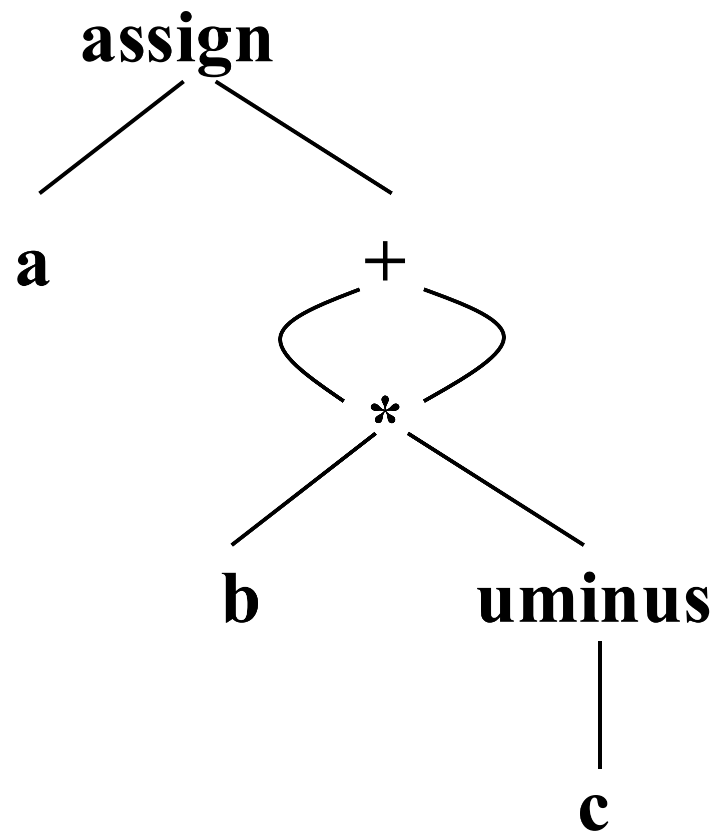
a+a*(b-c)+(b-c)*d



a:=b*(-c)+b*(-c)的图表示法



AST



DAG

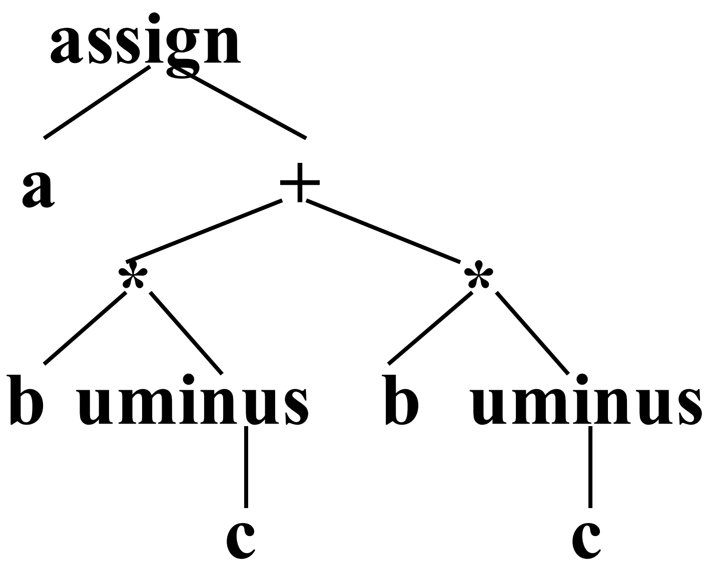
3-Three-Address Code (TAC)

- **General form x:=y op z**

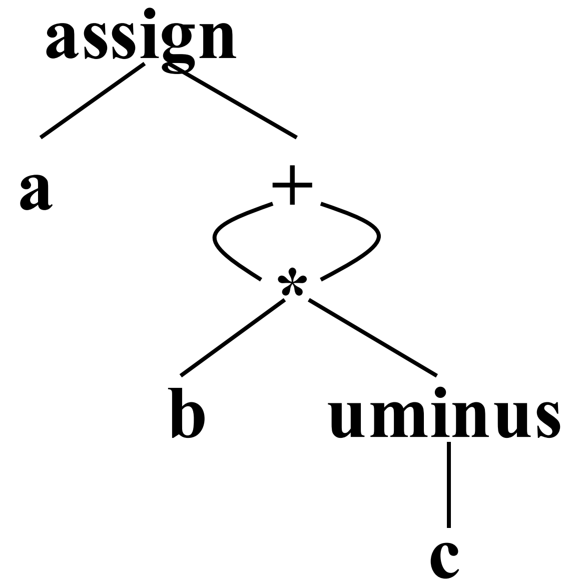
  How to code x+y*z ？

- **TAC** can be viewed as a **linear representation** of an **AST** or a **DAG**

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



AST



DAG

$T_1:=-c$
$T_2:=b*T_1$
$T_3:=-c$
$T_4:=b*T_3$
$T_5:=T_2+T_4$
$a:=T_5$

$T_1:=-c$
$T_2:=b*T_1$
$T_5:=T_2+T_2$
$a:=T_5$

Types of Three-Address Statements

x := y op z — binary operation

x := op y — unary operation

x := y — simple assignment

goto L — unconditional jump

if x relop y goto L or **if a goto L** — conditional jump

param x and **call p, n,** and **return y** — procedure call and return

x := y[i] and **x[i] := y** — indexed assignment (arrays)

x := &y, x := *y, and ***x := y** — address and pointer assignments

Three-Address Statements

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

- **Quadruples**
- **A quadruple is a record structure with four fields, named op, arg1, arg2 and result**

	op	arg1	arg2	result
(0)	uminus	c		T_1
(1)	*	b	T_1	T_2
(2)	uminus	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	:=	T_5		a

- Connection between quadruples is implemented via temporary variables.
- Unary operations use only the arg1 field.
- Jump statements store the target label in the result field.
- arg1, arg2, and result are usually pointers to entries in the symbol table, with temporary variables also recorded in the symbol table.

Three-Address Statements

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

- **Triple**
  - Temporary variable is referenced by the position of the statement that computes it
  - Three fields： op, arg1 and arg2

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

-(a+b)*(c+d)-(a+b+c) Three-Address Statements：

T_1: =a+b

T_2: =-T_1

T_3: =c+d

T_4: =T_2* T_3

T_5: =a+b

T_6: = T_5+c

T_7: = T_4-T_6

Quadruples

	op	arg1	arg2	result
(1)	+	a	b	T_1
(2)	uminus	T_1		T_2
(3)	+	c	d	T_3
(4)	*	T_2	T_3	T_4
(5)	+	a	b	T_5
(6)	+	T_5	c	T_6
(7)	-	T_4	T_6	T_7

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

T_1: $=a+b$

T_2: $=-T_1$

T_3: $=c+d$

T_4: $=T_2 * T_3$

T_5: $=a+b$

T_6: $= T_5+c$

T_7: $= T_4-T_6$

Triple

	op	arg1	arg2
(1)	+	a	b
(2)	uminus	(1)	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	a	b
(6)	+	(5)	c
(7)	-	(4)	(6)

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

T_1: =a+b

T_2: =-T_1

T_3: =c+d

T_4: =T_2* T_3

T_5: =a+b

T_6: = T_5+c

T_7: = T_4-T_6

	op	arg1	arg2
(1)	+	a	b
(2)	uminus	(1)	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	(1)	c
(6)	-	(4)	(5)

Instruction
(1)
(2)
(3)
(4)
(1)
(5)
(6)

Indirect Triple

Three-Address Statements

- **Indirect Triple**
  - To facilitate code optimization, **use a triple table plus an indirect table** to represent intermediate code.
  - **Indirect table**: an index table that lists the positions of triples in the triple table according to the order of operations.
- **Advantages**:
  - Facilitates **code optimization**
  - **Saves space**

Comparison

- **Triples** use pointers to refer to other triples, so modifications during optimization are difficult.
- **Indirect triples** only require changes to the indirect table for optimization, saving storage space in the triple table.
- **Quadruples** are easier to modify, but temporary variables must be added to the symbol table, occupying some storage space.

Outline

- Intermediate Language
- Translation of Declaration Statements
- Translation of Assignment Statements
- Translation of Boolean Expressions
- Translation of Flow-of-Control Statements
- Procedure Call Handling

Translation scheme

$S \rightarrow id:=E$	$S.code:=E.code \;\|\|\; gen(id.place\ `:=`\ E.place)$
$E \rightarrow E_1+E_2$	$E.place:=newtemp;$ $E.code:=E_1.code \;\|\|\; E_2.code \;\|\|gen(E.place\ `:=`\ E_1.place\ `+`\ E_2.place)$
$E \rightarrow E_1*E_2$	$E.place:=newtemp;$ $E.code:=E_1.code \;\|\|\; E_2.code \;\|\|\; gen(E.place\ `:=`\ E_1.place\ `*`\ E_2.place)$

$S \rightarrow id:=E$ { p:=lookup(id.name);
if p≠nil then
emit(p ':=' E.place)
else error }

$E \rightarrow E_1+E_2$ { E.place:=newtemp;
emit(E.place ':=' E_1.place '+' E_2.place)}

$E \rightarrow E_1*E_2$ { E.place:=newtemp;
emit(E.place ':=' E $_1$.place '*' E $_2$.place)}

$E \rightarrow -E_1$	E.place:=newtemp; E.code:=E_1.code || gen(E.place ':=' 'uminus' 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= ' '

$E \rightarrow -E_1$ { E.place:=newtemp;

emit(E.place':=''uminus'E_1.place)}

$E \rightarrow (E_1)$ { E.place:=E_1.place}

$E \rightarrow id$ { p:=lookup(id.name);

if p≠nil then

E.place:=p

else error }

*

Translation scheme

- Attribute **id.name** represents the name itself of the identifier id.
- Procedure **lookup(id.name)** checks if there is an entry for this name in the symbol table. If found, it returns a **pointer** to the entry; otherwise, it returns **nil** to indicate not found.
- Procedure **emit** sends the generated three-address code statements to the output file.

Type conversion $E \rightarrow E_1 + E_2$

```
{ E.place:=newtemp;
  if E1.type=integer and
     E2.type=integer then   begin
      emit (E.place  ':='  E
     1.place  'int+'  E 2.place);
      E.type:=integer
  end
  else if E1.type=real and
     E2.type=real then begin
      emit (E.place  ':='  E
     1.place  'real+'  E 2.place);
      E.type:=real
  end
else if E1.type=integer and E2.type=real
      then begin
      u:=newtemp;
      emit (u  ':='  'inttoreal'  E
      1.place);
      emit (E.place  ':='  u  'real+'  E
      2.palce);
      E.type:=real
end
else if E1.type=real and E2.type=integer
      then begin
      u:=newtemp;
      emit (u  ':='  'inttoreal'  E
      2.place);
      emit (E.place  ':='  E 1.place
       'real+'  u);
      E.type:=real
end
 else E.type:=type_error}
```

*

Type conversion $E \rightarrow E_1 + E_2$

- Example: x := y + i * j
  Here, x and y are of real type; i and j are of integer type. The three-address code generated for this assignment statement is:

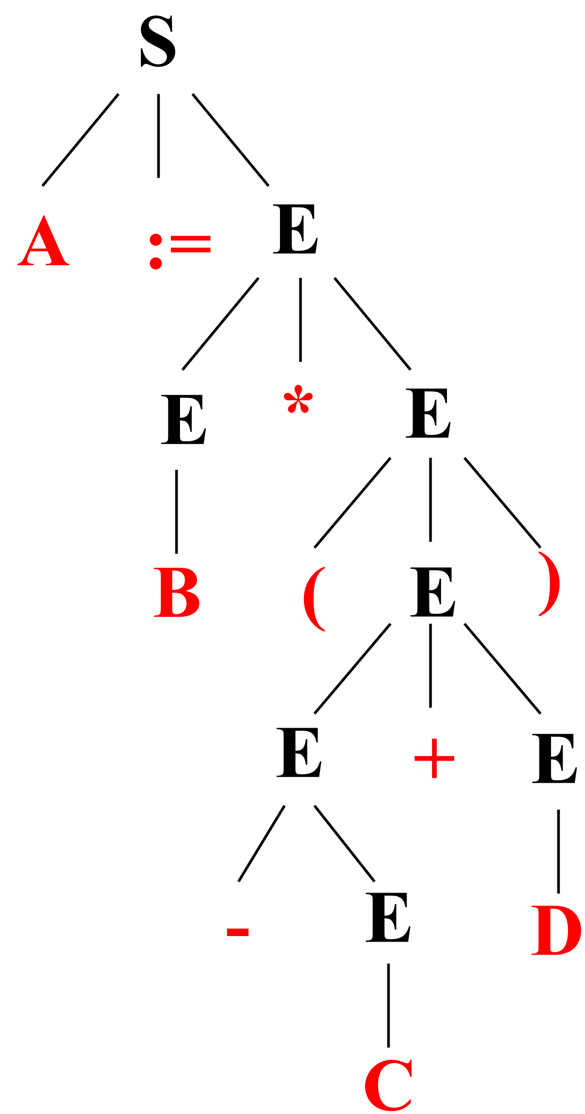
```
T1 := i * j              // integer multiplication
T3 := inttoreal T1       // type conversion from integer to real
T2 := y + T3             // real addition
x := T2                  // assignment
```

$A:=B*(-C+D)$

$S \rightarrow id:=E$	{ p:=lookup(id.name); if p≠nil then emit(p ':=' E.place) else error }
$E \rightarrow E_1+E_2$	{ E.place:=newtemp; emit(E.place ':=' E_1.place '+' E_2.place)}
$E \rightarrow E_1*E_2$	{ E.place:=newtemp; emit(E.place ':=' E_1.place '*' E_2.place)}
$E \rightarrow -E_1$	{ E.place:=newtemp; emit(E.place':=''uminus'E_1.place)}
$E \rightarrow (E_1)$	{ E.place:=E_1.place}
$E \rightarrow id$	{ p:=lookup(id.name); if p≠nil then E.place:=p else error }

A:=B*(-C+D), Bottom-up

$S \rightarrow id:=E$	{ p:=lookup(id.name); if p≠nil then emit(p ':=' E.place) else error }
$E \rightarrow E_1+E_2$	{ E.place:=newtemp; emit(E.place ':=' E_1.place '+' E_2.place)}
$E \rightarrow E_1*E_2$	{ E.place:=newtemp; emit(E.place ':=' E_1.place '*' E_2.place)}
$E \rightarrow -E_1$	{ E.place:=newtemp; emit(E.place':=''uminus'E_1.place)}
$E \rightarrow (E_1)$	{ E.place:=E_1.place}
$E \rightarrow id$	{ p:=lookup(id.name); if p≠nil then E.place:=p else error }



Quiz-Canvas 2min

- **ch7 Intermediate Code Generation: Assignment Statement**

Outline

- **Intermediate Language**
- **Translation of Declaration Statements**
- **Translation of Assignment Statements**
- **Translation of Boolean Expressions**
- **Translation of Flow-of-Control Statements**
- **Procedure Call Handling**

Translation of Boolean Expressions

- **Boolean expression**: An expression formed by connecting Boolean values and relational expressions using Boolean operators.
- **Boolean operators**: and, or, not
- **Relational operators**: $<$, $\leq$, $=$, $\neq$, $>$, $\geq$

Translation of Boolean Expressions

- Two main uses of Boolean expressions:
  - Logical computation to produce a Boolean value.
  - Conditional expressions in Flow-of-Control Statements.
- Grammar for Boolean expressions:
  - **E→E or E | E andE | ¬ E | (E) | id rop id | id**
  - Operator precedence:
    - **not > and > or, left-associative**
    - **Relational operators: > Boolean operators**

There are usually two methods to evaluate Boolean expressions

(1) Step-by-step evaluation, similar to arithmetic expressions

1 or (not 0 and 0) or 0

=1 or (1 and 0) or 0

=1 or 0 or 0

=1 or 0

=1

(2) Using optimization (short-circuit evaluation)

A or B:	if A then true else B
A and B:	if A then B else false
$\neg$ A:	if A then false else true

Correspondingly, there are two translation methods

*

Boolean Expression Evaluation Method (1)

- **a or b and not c**

  T_1:=not c

  T_2:=b and T_1

  T_3:=a or T_2

- **a<b can equivalently be written as : if a<b then 1 else 0**

```
100:  if a<b goto 103
101:  T:=0
102:  goto 104
103:  T:=1
```

Boolean Expression Evaluation Method (1)

$E \rightarrow E_1$ or E_2 {E.place:=newtemp;
emit(E.place ':=' E_1.place 'or' E_2.place)}

$E \rightarrow E_1$ and E_2 {E.place:=newtemp;
emit(E.place ':=' E_1.place 'and' E_2.place)}

$E \rightarrow$ not E_1 {E.place:=newtemp;
emit(E.place ':=' 'not' E_1.place)}

$E \rightarrow (E_1)$ {E.place:=E_1.place}

Boolean Expression Evaluation Method (1)

a<b translates to

100:	if a<b goto 103
101:	T:=0
102:	goto 104
103:	T:=1
104:	

$E \rightarrow id_1$ relop id_2 { E.place:=newtemp;
emit(‘if’ id_1.place relop. op id_2. place ‘goto’ nextstat+3);
emit(E.place ‘:=’ ‘0’);
emit(‘goto’ nextstat+2);
emit(E.place ‘:=’ ‘1’) }

$E \rightarrow id$ { E.place:=id.place }

a<b or c<d and e<f

100: if a<b goto 103

101: T_1:=0

102: goto 104

103: T_1:=1

104: if c<d goto 107

105: T_2:=0

106: goto 108

107: T_2:=1

108: if e<f goto 111

109: T_3:=0

110: goto 112

111: T_3:=1

112: T_4:=T_2 and T_3

113: T_5:=T_1 or T_4

E →id_1 relop id_2

{ E.place:=newtemp;

emit('if' id_1.place relop. op id_2. place 'goto' nextstat+3);

emit(E.place ':=' '0');

emit('goto' nextstat+2);

emit(E.place ':=' '1) }

E→id

{ E.place:=id.place }

E→E_1 or E_2

{ E.place:=newtemp;

emit(E.place ':=' E_1.place 'or' E_2.place)}

E→E_1 and E_2

{ E.place:=newtemp;

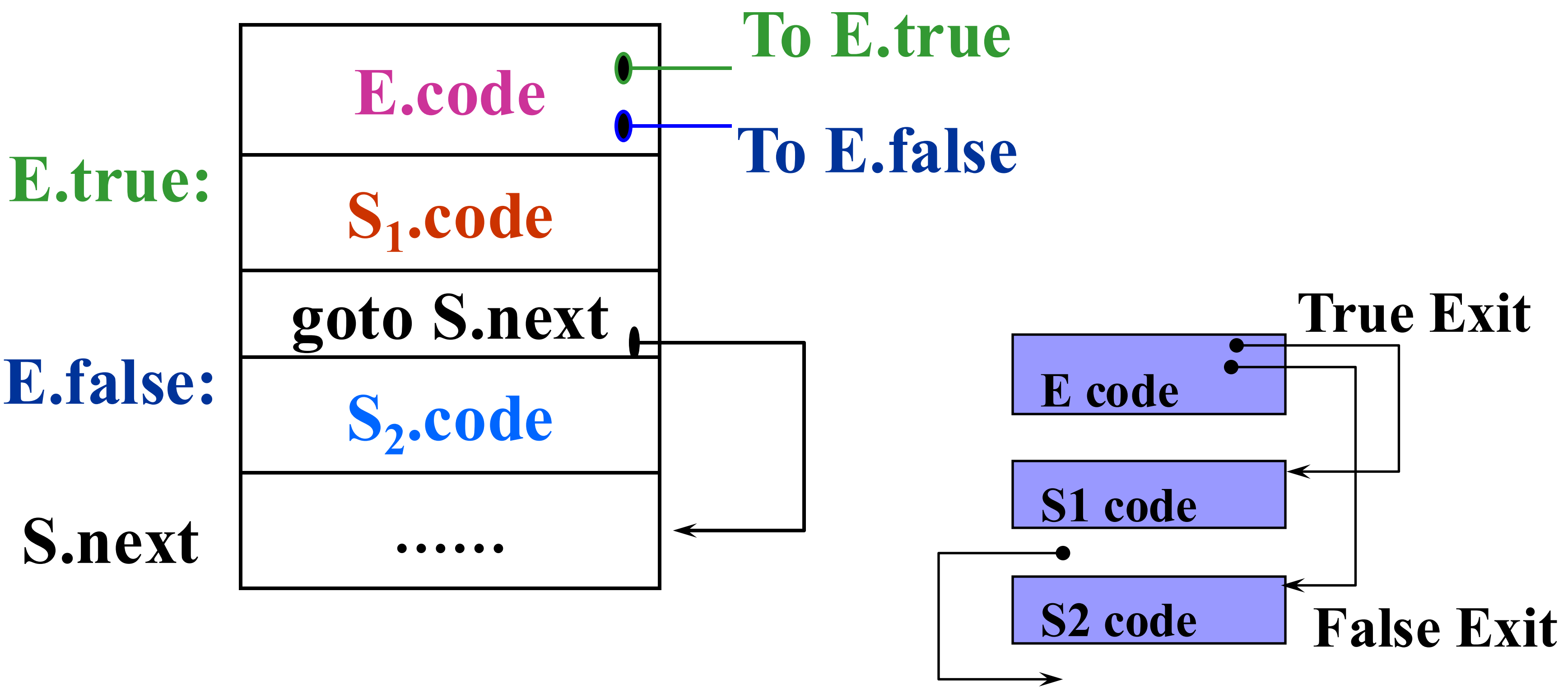
emit(E.place ':=' E_1.place 'and' E_2.place)}

Exercise

- a>b and c>d

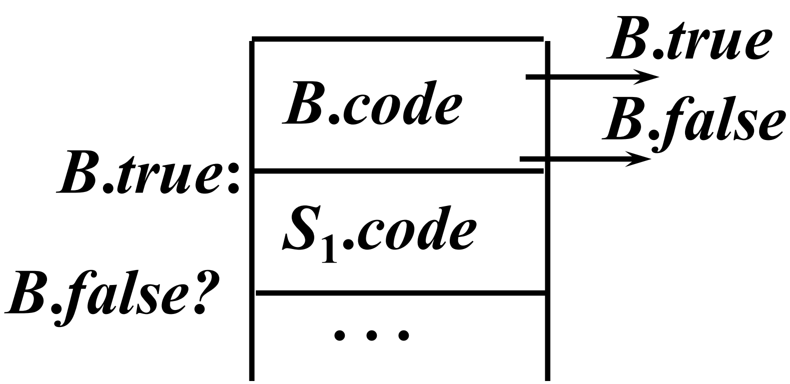
Translation of Boolean expressions as conditional control

- For the conditional statement if E then S1 else S2, assign **two exits** to E: one for true, one for false.

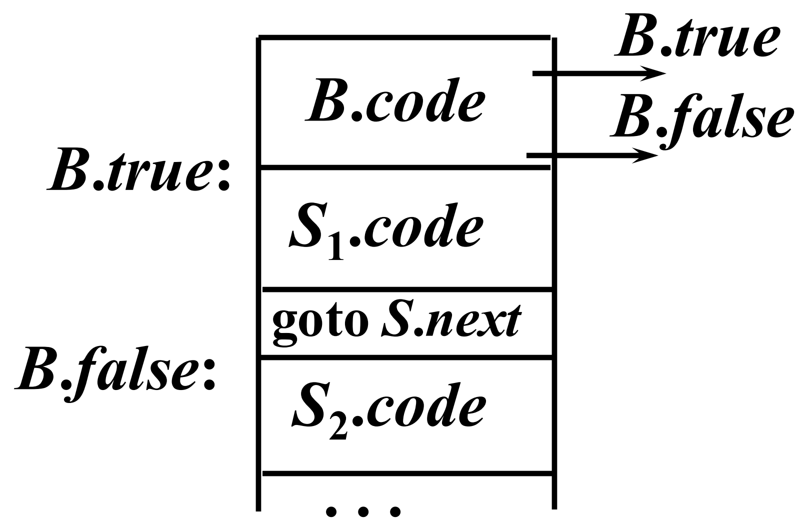


*

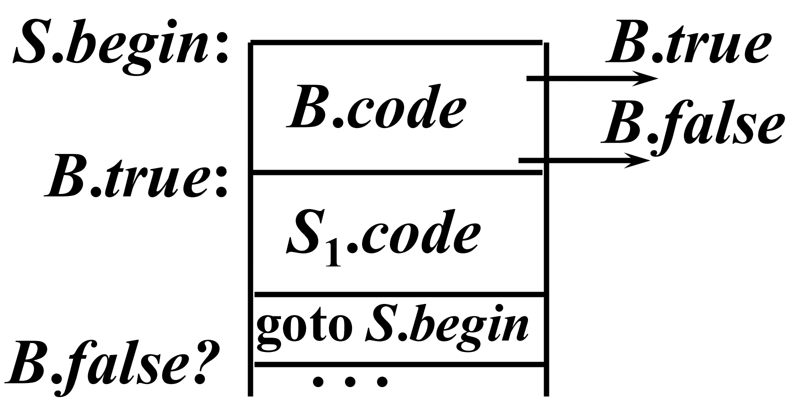
Boolean Expressions and Control Flow Statements



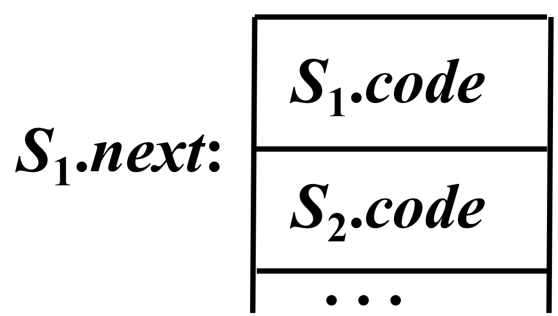
(a) if-then



(b) if-then-else



(c) while-do



(d) S_1; S_2

Translation of Boolean Expressions

- Two-pass scanning
  - Construct a **syntax tree** for the given input string.
  - Perform a **depth-first traversal** of the syntax tree and apply the translations specified by the semantic rules.
- **One-pass scanning**

Grammar of Boolean Expressions

(1) $E \longrightarrow E_1$ **or** $M\ E_2$

(2) $\quad\quad | \ E_1$ **and** $M\ E_2$

(3) $\quad\quad |$ **not** E_1

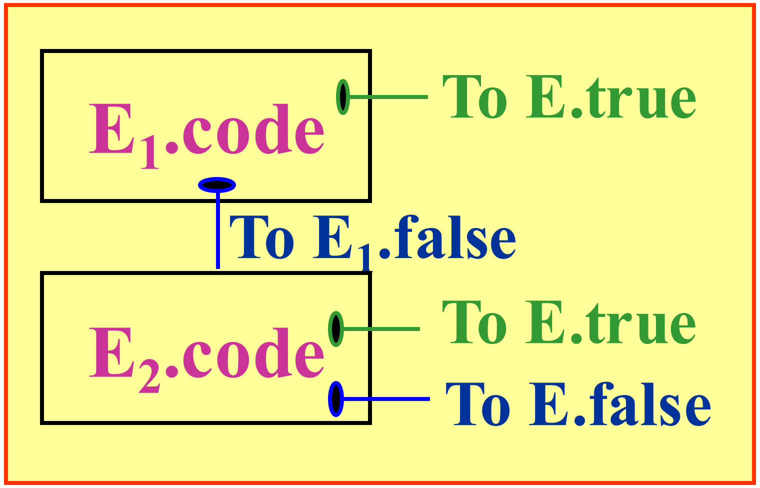
(4) $\quad\quad | \ (E_1)$

(5) $\quad\quad | \ \mathbf{id}_1$ **relop** $\mathbf{id}_2$

(6) $\quad\quad | \ \mathbf{id}$

(7) $M \longrightarrow \varepsilon$

Translation Scheme for Boolean Expressions




```
if (A or B)
then V=0
else V=1
```

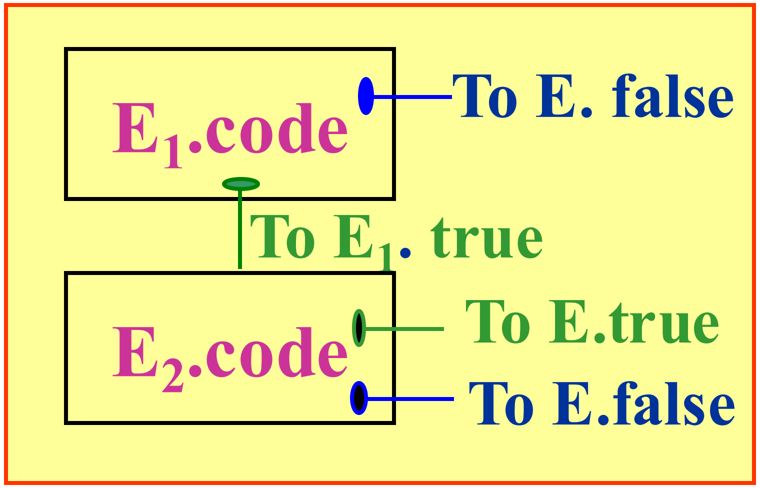
(1) $E \rightarrow E_1$ or $M\ E_2$

{ backpatch(E_1.falselist, M.quad);

E.truelist:=merge(E_1.truelist, E_2.truelist);

E.falselist:=E_2.falselist }

Translation Scheme for Boolean Expressions



if (A and B)
then V=0
else V=1

(2) $E \rightarrow E_1$ and M E_2

{ backpatch(E_1.truelist, M.quad);

E.truelist:=E_2.truelist;

E.falselist:=merge(E_1.falselist,E_2.falselist) }

*

Translation Scheme for Boolean Expressions

(3) $E \rightarrow$ not E_1

{ E.truelist:=E_1.falselist;

E.falselist:=E_1.truelist}

(4) $E \rightarrow (E_1)$

{ E.truelist:=E_1.truelist;

E.falselist:=E_1. falselist}

Translation Scheme for Boolean Expressions

(5) $E \rightarrow id_1$ relop id_2

{ E.truelist:=makelist(nextquad);

E.falselist:=makelist(nextquad+1);

emit('j', relop.op ',' id_1.place ',' id_2.place ',' 0');

emit('j, —, —, 0') }

(6) $E \rightarrow id$

{ E.truelist:=makelist(nextquad);

E.falselist:=makelist(nextquad+1);

emit('jnz' ',' id .place ',' '—,' 0') ;

emit(' j, -, -, 0') }

(7) $M \rightarrow \varepsilon$

{ M.quad:=nextquad }

*

One-Pass Translation

- Quadruples
  - Store in an array; the array index represents the quadruple's number.
- Conventions

  (jnz, a, -, p) $\rightarrow$ **if a goto p**

  (jrop, x, y, p) $\rightarrow$ **if x rop y goto p**

  (j, -, -, p) $\rightarrow$ **goto p**
- Sometimes the jump target is **unknown**. Save the unfinished quadruple as E's semantic value for later **backpatching**.

*

- Assign two synthesized attributes to the nonterminal E:
  - E.truelist — list of quadruple numbers whose *true exits* need backpatching.
  - E.falselist — list of quadruple numbers whose *false exits* need backpatching.

(p) (x, x, x, 0)

Tail of the list

…

(q) (x, x, x, p)

…

(r) (x, x, x, q)

E. truelist =r

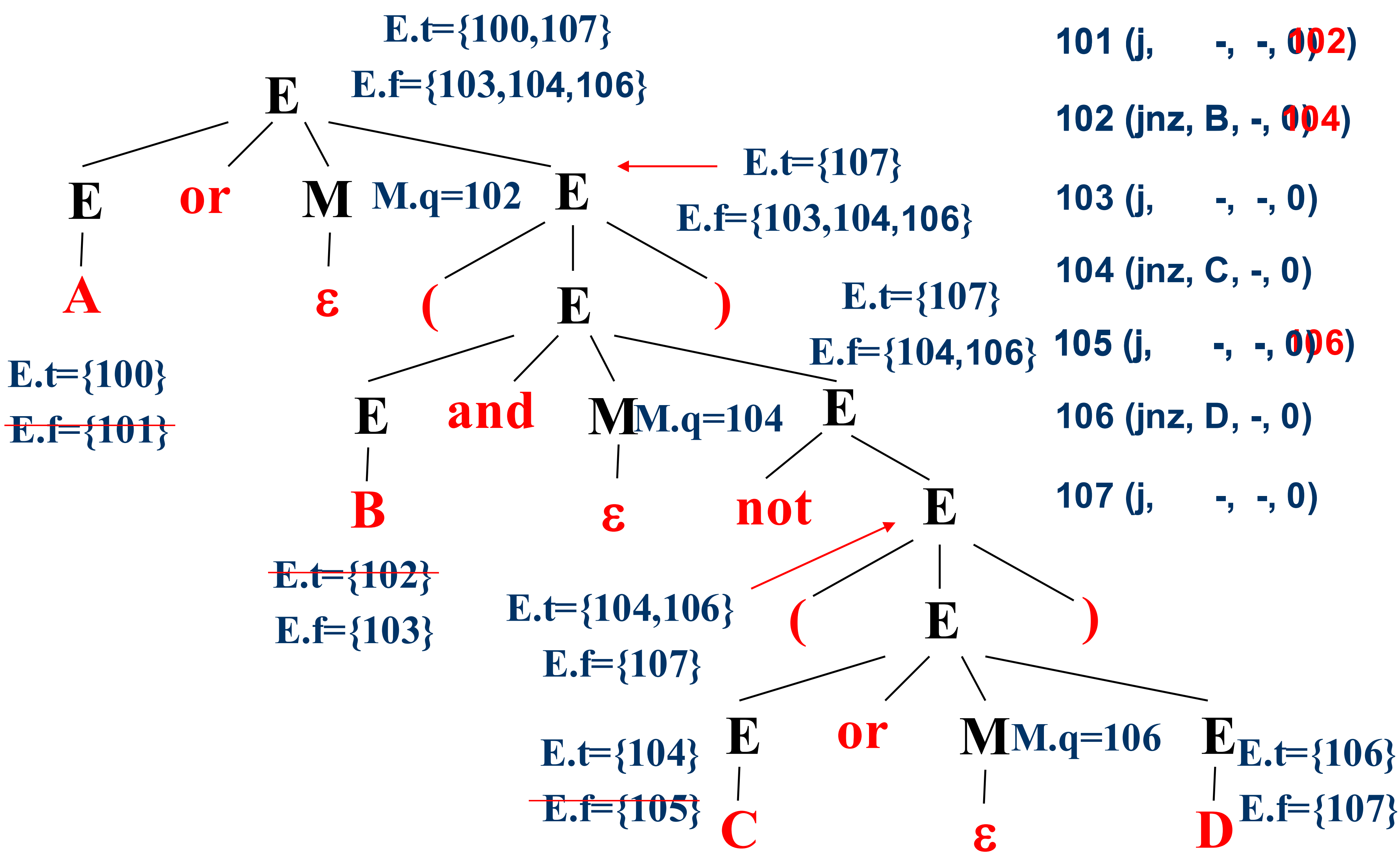
To handle E.truelist and E.falselist, introduce the following semantic variables and procedures:

- **nextquad**: points to the next quadruple; auto-increment after each emit.
- **makelist**(i): create a list with index i.
- **merge**(p_1,p_2): merge two lists, return head.
- **backpatch**(p,t): fill target field of list p with t.

Exercise

- **A or B**
- **A or (B and not (C or D))**

A or (B and not (C or D))



100 (jnz, A, -, 0)

101 (j, -, -, 102)

102 (jnz, B, -, 104)

103 (j, -, -, 0)

104 (jnz, C, -, 0)

105 (j, -, -, 106)

106 (jnz, D, -, 0)

107 (j, -, -, 0)

a<b or c<d and e<f

```
100  (j<, a, b, 0)
101  (j, -, -, 102)
102  (j<, c, d, 104)
103  (j, -, -, 0)
104  (j<, e, f, 100)    truelist
105  (j, -, -, 103)     falselist 5
```

The true and false exits of the Boolean expression still need **backpatching**

Quiz-Canvas

- **3min**
- **ch7 Intermediate Code Generation: Boolean Expressions**

Outline

- **Intermediate Language**
- **Translation of Declaration Statements**
- **Translation of Assignment Statements**
- **Translation of Boolean Expressions**
- **Translation of Flow-of-Control Statements**
- **Procedure Call Handling**

One-pass translation of Flow-of-Control Statements

- **Consider the statements defined by the following productions:**

(1) S→if E then S

(2) | if E then S else S

(3) | while E do S

(4) | begin L end

(5) | A

(6) L→L;S

(7) | S

S → statement
L → statement list
A → assignment statement
E → Boolean expression

Translation of if Statements

Productions

$$S \rightarrow \text{if } E \text{ then } S^{(1)}$$

$$\mid \text{if } E \text{ then } S^{(1)} \text{ else } S^{(2)}$$

Rewritten Productions

$$S \rightarrow \text{if } E \text{ then } M\ S_1$$

$$S \rightarrow \text{if } E \text{ then } M_1\ S_1\ N \text{ else } M_2\ S_2$$

$$M \rightarrow \varepsilon$$

$$N \rightarrow \varepsilon$$

Translation Scheme:

1. $S \rightarrow$ if E then M S_1
{ backpatch(E.truelist, M.quad);
S.nextlist:=merge(E.falselist, S_1.nextlist) }

2. $S \rightarrow$ if E then M_1 S_1 N else M_2 S_2
{ backpatch(E.truelist, M_1.quad);
backpatch(E.falselist, M_2.quad);
S.nextlist:=merge(S_1.nextlist, N.nextlist, S_2.nextlist) }

3. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

4. $N \rightarrow \varepsilon$ { N.nextlist:=makelist(nextquad);
emit(‘j,—,—,0’) }

*

1. $S \rightarrow$ if E then M S_1
{ backpatch(E.truelist, M.quad);
S.nextlist:=merge(E.falselist, S_1.nextlist) }

2. $S \rightarrow$ if E then M_1 S_1 N else M_2 S_2
{ backpatch(E.truelist, M_1.quad);
backpatch(E.falselist, M_2.quad);
S.nextlist:=merge(S_1.nextlist, N.nextlist, S_2.nextlist) }

3. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

4. $N \rightarrow \varepsilon$ { N.nextlist:=makelist(nextquad);
emit(‘j, - , - ,0’) }

Translate into intermediate code

if a then b:=1 else b:=2

*

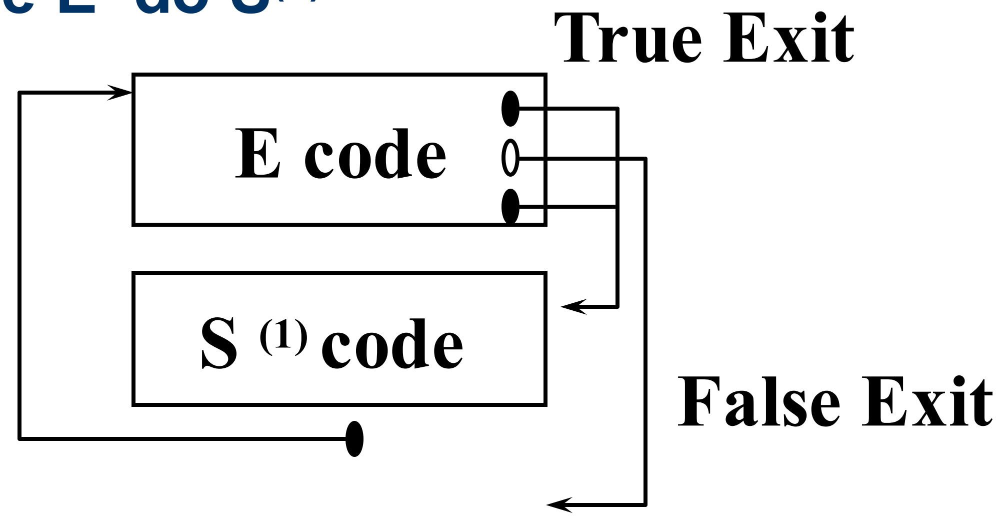
Quiz-Canvas

- **1min**
- **ch7 Intermediate Code Generation: Conditional Statements**

Translation of while Statements

Productions

$S \rightarrow$ while E do $S^{(1)}$



To facilitate backpatching, rewrite the production as:

$S \rightarrow$ while M_1 E do M_2 S_1

$M \rightarrow \varepsilon$

Translation Scheme:

1. $S \rightarrow$ while M_1 E do M_2 S_1

    {backpatch(S_1.nextlist, M_1.quad);

    backpatch(S_1.nextlist, nextquad);

    backpatch(E.truelist, M_2.quad);

    S.nextlist:=E.falselist

    emit('j,—,—,' M_1.quad) }

Which is better?

2. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

3. $S \rightarrow A$ { S.nextlist:=makelist( ) }

1. $S \rightarrow$ while M_1 E do M_2 S_1

{backpatch(S_1.nextlist, M_1.quad);

backpatch(E.truelist, M_2.quad);

S.nextlist:=E.falselist

emit('j, - , - ,' M_1.quad) }

2. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

Translate into intermediate code

while a>b do a:=a-1

Translation of $L \to L;S$

Production

$L \to L;S \mid S$

Rewritten Production

$L \to L_1;\ M\ S \mid S$

$M \to \varepsilon$

Translation Scheme

1. $L \to L_1;\ M\ S$ { backpatch(L_1.nextlist, M.quad);
   L.nextlist:=S.nextlist }
2. $M \to \varepsilon$ { M.quad:=nextquad }
3. $L \to S$ { L.nextlist:=S.nextlist }

Translation of other statements

$S \rightarrow$ begin L end

{ S.nextlist:=L.nextlist }

$S \rightarrow A$

{ S.nextlist:=makelist( ) }

*

1. $L \rightarrow L_1$; M S	{ backpatch(L_1.nextlist, M.quad); L.nextlist:=S.nextlist }
2. $M \rightarrow \varepsilon$	{ M.quad:=nextquad }
3. $L \rightarrow S$	{ L.nextlist:=S.nextlist }

Translation into intermediate code

a:=a+1;

b:=b+1

while (a<b) do
if (c<d) then x:=y+z;

P190

$E \rightarrow id_1$ relop id_2

{ E.truelist:=makelist(nextquad);

E.falselist:=makelist(nextquad+1);

emit(‘j’ relop.op ‘,’ id_1.place ‘,’ id_2.place ‘,’ ‘0’);

emit(‘j, - , - , 0’) }

P179

$S \rightarrow A$	{ S.nextlist:=makelist( ) }
$A \rightarrow id:=E$	{ p:=lookup(id.name); if p≠nil then emit(p ‘:=’ E.place) else error }
$E \rightarrow E_1+E_2$	{ E.place:=newtemp; emit(E.place ‘:=’ E_1.place ‘+’ E_2.place)}

*

while (a<b) do

if (c<d) then x:=y+z;

P195

$S \to$ if E then M S_1

{ backpatch(E.truelist, M.quad);

S.nextlist:=merge(E.falselist, S_1.nextlist) }

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

$S \to A$ { S.nextlist:=makelist( ) }

P195

$S \to$ while M_1 E do M_2 S_1

{ backpatch(S_1.nextlist, M_1.quad);

backpatch(E.truelist, M_2.quad);

S.nextlist:=E.falselist

emit(‘j, - , - ,’ M_1.quad) }

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

*

```
while (a<b) do
      if (c<d) then x:=y+z;
```

```
100   (j<, a, b, 102)
101   (j, -, -, 107)
102   (j<, c, d, 104)
103   (j, -, -, 100)
104   (+, y, z, T)
105   (:=, T, -, x)
106   (j, -, -, 100)
107
```

Labels and goto Statements

- Label definition:

  L: S;

  - After processing, label L is considered defined
  - In the symbol table, L's address field records the address of the first quadruple of statement S
- Label reference:

  goto L;

Backward jump	Forward jump
L1: ……	goto L1;
……	……
goto L1;	L1: ……

Symbol Table

Backward jump

```
L1:   ......
      ......
      goto L1;
```

Forward jump

```
      goto L1;
      ......
L1:   ......
```

Name	Type	…	Defined	Address
…	…	…	…	…
L	Label		**false**	r

(p) (j, -, -, 0)

…

(q) (j, -, -, p)

…

(r) (j, -, -, q)

Semantic Actions for the Production S' → goto L:

- **Look up L in the symbol table.**
- **If L exists and is defined:**
  - GEN(J, -, -, P) # P is the address from L's entry.
- **Else if L does not exist:**
  - Insert L into the table.
  - Set "defined" = false, "address" = nextquad.
  - GEN(J, -, -, 0)
- **Else if L exists but not yet defined:**
  - q := L's current address.
  - Update L's address to nextquad.
  - GEN(J, -, -, q)

- Production for Label Statements:

$S_L \rightarrow$label S **label $\rightarrow$ i:**

- Semantic Actions of **label $\rightarrow$ i:**
  - If identifier i (assume L) is not in the symbol table:
    - Insert L
    - Set type = "label", defined = true, address = nextquad
  - If L exists but type ≠ "label" or defined = true:
    - Report an error
  - If L exists:
    - Change defined = true
    - Retrieve the head of the address chain (q) from L
    - Fill nextquad in the chain
    - Execute BACKPATCH(q, nextquad)

*

Exercise

Backward jump	Forward jump
L1: a:=1	goto L1;
goto L1;	L1: a:=1

Show the intermediate code and the changes in the symbol table.

Translation of CASE Statements

- **Statement structure:**

case E of

C_1: S_1;

C_2: S_2;

…

C_{n-1}: S_{n-1};

otherwise: S_n

end

E is the expression, called the **selector**
E is usually an integer expression or a character variable

- **Method (1):**

```
        T:=E
L1:     if T≠C1 goto L2
        S1 code
        goto next
L2:     if T≠C2 goto L3
        S2 code
        goto next
L3:
        …
Ln-1:   if T≠Cn-1 goto Ln
        Sn-1 code
        goto next
Ln:     Sn code
next:
```

- **Method (2):**

```
        T:=E
        goto test
L1:     S1 code
        goto next
…
Ln-1:   Sn-1 code
        goto next
Ln:     Sn code
        goto next
test:   if T=C1 goto L1
        if T=C2 goto L2
        …
        if T=Cn-1 goto Ln-1
        goto Ln
next:
```

*

Exercise

```
case  a+1  of
    1:      b:=2;
    2:      b:=4;
    3:      b:=9;
    otherwise:      b:=0
end
```

- Method (2):

	$T:=E$
	goto test
L_1:	S_1 code
	goto next
…	
L_{n-1}:	S_{n-1} code
	goto next
L_n:	S_n code
	goto next
test:	if $T=C_1$ goto L_1
	if $T=C_2$ goto L_2
	…
	if $T=C_{n-1}$ goto L_{n-1}
	goto L_n
next:	

Outline

- Intermediate Language
- Translation of Declaration Statements
- Translation of Assignment Statements
- Translation of Boolean Expressions
- Translation of Flow-of-Control Statements
- Procedure Call Handling

Handling of Procedure Calls

- Procedure calls mainly involve two tasks
  - Passing parameters
  - Control transfer to procedure
- Passing by address
  - Pass the address of the actual parameter to the corresponding formal parameter
  - Copy the address of the actual parameter into the corresponding formal unit
  - References and assignments to formal parameters in the procedure body are handled as indirect accesses to the formal unit
- Note: Only the "pass by address" method is discussed

Grammar for Procedure Calls

(1) S → call id (Elist)

(2) Elist → Elist, E

(3) Elist → E

- **Parameter handling:**
  - The addresses of arguments are stored in a **queue**
  - For each item in the queue, generate a **param** statement

Translation of Procedure Calls

- **Method:** Place the addresses of actual arguments one by one **before the call instruction**.
- **Example:** CALL S(A, X+Y)
- Intermediate code:

```
T := X + Y      // compute second argument
param A         // address of first argument
param T         // address of second argument
call S          // transfer control to procedure S
```

*

- **Translation Actions:**

1. S→call id (Elist)

{ for each item p in the queue

emit(‘param’ p);

emit(‘call’ id.place) }

2. Elist→Elist, E

{ Append E.place to the end of the queue }

3. Elist→E

{ Initialize the queue to contrain only E.place }

Exercise

- **CALL f(a， b)**

Survey

- Intermediate Language
- Translation of Declaration Statements
- Translation of Assignment Statements
- Translation of Boolean Expressions
- Translation of Flow-of-Control Statements
- Procedure Call Handling

Dank u
Dutch

Merci
French

Спасибо
Russian

Gracias
Spanish

شكراً
Arabic

धन्यवाद
Hindi

감사합니다
Korean

Tack så mycket
Swedish

תודה רבה
Hebrew

Obrigado
Brazilian Portuguese

谢谢
Chinese

Dankon
Esperanto

Thank You !

ありがとうございます
Japanese

Trugarez
Breton

Danke
German

Tak
Danish

Grazie
Italian

நன்றி
Tamil

děkuji
Czech

ขอบคุณ
Thai

go raibh maith agat
Gaelic

*

Canvas作业

- 作业6-中间代码生成
  - P218(7)

    While A<C and B<D do

    If A=1 then C:=C+1 else

    while A<=D do A:=A+2;

1 后缀式

- 后缀式表示法：波兰逻辑学家Lukasiewicz发明的一种表示表达式的方法，又称逆波兰表示法。
- 一个表达式E的后缀形式可以如下定义：

1. 如果E是一个变量或常量，则E的后缀式是E自身。
2. 如果E是E_1 op E_2形式的表达式，其中op是任何二元操作符，则E的后缀式为E_1' E_2' op，其中E_1' 和E_2' 分别为E_1 和E_2的后缀式。
3. 如果E是(E_1)形式的表达式，则E_1的后缀式就是E的后缀式。

- 逆波兰表示法不用括号。只要知道每个算符的目数，对于后缀式，不论从哪一端进行扫描，都能对它进行唯一分解。
- 后缀式的计算
  - 用一个栈实现。
  - 一般的计算过程是：自左至右扫描后缀式，每碰到运算量就把它推进栈。每碰到k目运算符就把它作用于栈顶的k个项，并用运算结果代替这k个项。

四元式、三元式和间接三元式比较

- 三元式中使用了指向三元式的指针，优化时修改较难。
- 间接三元式优化只需要更改间接码表，并节省三元式表存储空间。
- 修改四元式表也较容易，只是临时变量要填入符号表，占据一定存储空间。