# Unit 11
# Intermediate Code Generation

# Summary

- **Three-Address Code**
- **Code for Assignments**
- **Code for Boolean Expressions**
- **Code for Flow-of-Control Statements**

# Introduction

- A source program can be translated into intermediate code by intermediate code generator
- Intermediate code is machine-independent

# Benefits of Intermediate Code

- Retargetting is facilitated
- A machine-independent code optimizer can be applied to intermediate representation

# Intermediate Code

- To be done by an Intermediate Code Generator.

- Intermediate representation is decided by compiler designer
- Typical intermediate representations are:
  - Syntax Tree
  - Postfix Notation
  - Three Address Code

# Intermediate Code

- As **intermediate code** we consider the *three-address* ***code***, similar to assembly: sequence of instructions with at most *three* operands such that:
  1. There is at most one operator, in addition to the assignment (we make explicit the operators precedence).
  2. The general form is: x := y *op* z
- x,y,z are called **addresses**, i.e. either identifiers, constants or **compiler**-generated temporary names.
  - Temporary names must be generated to compute **intermediate** operations.
  - Addresses are implemented as pointers to their symbol-table entries

# Three Address Code of x + y * z

- $t_1$ := y*z
- $t_2$ := x+$t_1$

# Types of Three-Address Statements

- Three-Address statements are akin to assembly **code**: Statements can have *labels*
- There are statements for flow-of-control.
  1. *Assignment Statements*: x := y *op* z.
  2. *Unary Assignment Statements*: x := *op* y.
  3. *Copy Statements*: x := y.
  4. *Unconditional Jump*: goto L, with L a label of a statement.
  5. *Conditional Jump*: if x *relop* y goto L.

# Types of Three-Address Statements

- *Procedure Call*: param x, and call p,n for calling a procedure, p, with n parameters. return y is the returned value of the procedure:

  param $x_1$

  param $x_2$

  . . .

  param $x_n$

  Call p,n

- *Indexed assignments*: x := y[i] or x[i] := y. Note: x[i] denotes the value in the location i memory units beyond location x.

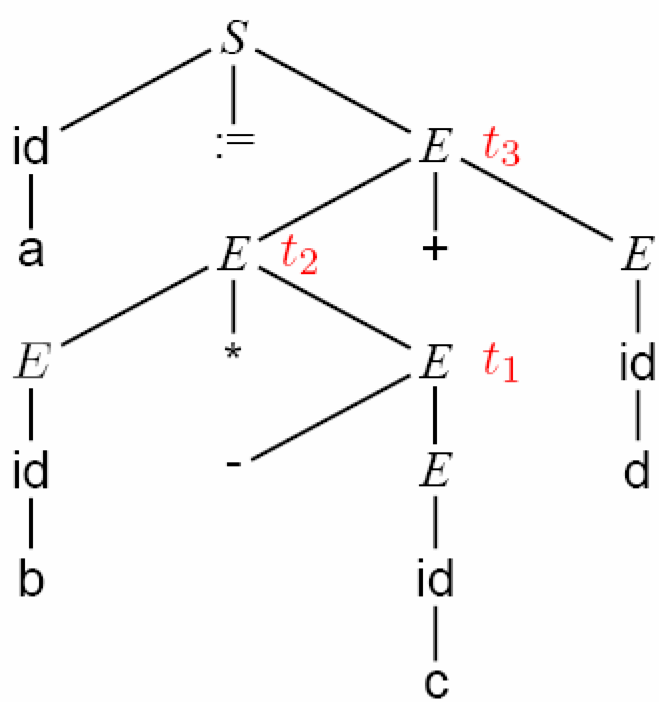
# Syntax Directed Definition into Three Address Code

- The synthesized S.code represents the three address code
- Template names are generated for intermediate calculations
- The nonterminal E has two attributes
- E.place the name that will hold the value of E
- E.code the sequence of three-address statements evaluating E
- The function newtemp returns a sequence of distinct names t1, t2,. .
- The function *gen* generates three-address **code** such that:
  1. Everything quoted is taken literally;
  2. The rest is evaluated.
- In practice, **code** can be sent to an output file instead of being assigned to the ***code*** attribute.

# SDD to Produce Three Address Code for Assignments

| Productions | Semantic Rules |
|---|---|
| $S \rightarrow$ 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*) } |
| $E \rightarrow - E_1$ | {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* = '' } |

# Three Address Code of a := b * -c + d




$$
\begin{aligned}
t_1 &:= \text{uminus } c \\
t_2 &:= b * t_1 \\
t_3 &:= t_2 + d \\
a &:= t_3
\end{aligned}
$$

# Implementation of Three Address Statements

**Quadruples**

$t_1 := -\ c$

$t_2 := b\ *\ t_1$

$t_3 := -\ c$

$t_4 := b\ *\ t_3$

$t_5 := t_2\ +\ t_4$

$a := t_5$

| | *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 |

**Template names must be inserted into the symbol table when they are generated.**

# Implementation of Three Address Statements

- Triples

$t_1 := - \ c$

$t_2 := b \ * \ t_1$

$t_3 := - \ c$

$t_4 := b \ * \ t_3$

$t_5 := t_2 \ + \ t_4$

$a := t_5$

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

**Template names are not inserted into symbol tables**

# More Triple Representations

- Example
  x[i]:=y                                    x:=y[i]
- Use triple structures

| | *op* | *arg1* | *arg2* |
|---|---|---|---|
| (0) | [ ] | x | i |
| (1) | := | (0) | y |

| | *op* | *arg1* | *arg2* |
|---|---|---|---|
| (0) | [ ] | y | i |
| (1) | := | x | (0) |

# Implementation of Three Address Statements

- Indirect triples is considered a listing of pointers to triples.

| | *op* |
|---|---|
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

| | *op* | *arg1* | *arg2* |
|---|---|---|---|
| (14) | uminus | c | |
| (15) | * | b | (14) |
| (16) | uminus | c | |
| (17) | * | b | (16) |
| (18) | + | (15) | (17) |
| (19) | assign | a | (18) |

# Intermediate Code for Declarations

***offset*** is a global variable can keep track of the next available relative address***.***

Attribute ***Type*** represent a type expression constructed from basic types, attribute ***Width*** indicate number of memory units taken by object of that type

| **Productions** | **Semantic Rules** |
|---|---|
| **P → M D** | **{ }** |
| **M → ε** | **{*offset*:=0 }** |
| **D → D; D** | |
| **D → id : T** | **{ *enter*(id*.name,* T.*type*, *offset*)** <br> ***offset*:=*offset* + T*.width* }** |
| **T → integer** | **{T.*type* = *integer*; T.*width* = 4 }** |
| **T → real** | **{T.*type* =*real*; T.*width* = *8* }** |
| **T → array [ num ] of $T_1$** | **{T.*type*=*array*(1..num.*val*,$T_1$.*type*)** <br> **T.*width* = num.val * $T_1$.*width*}** |

# Keeping track of scope information

- In a language with nested procedure, when a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended
- Grammar for this type of declaration:

  **P → D**

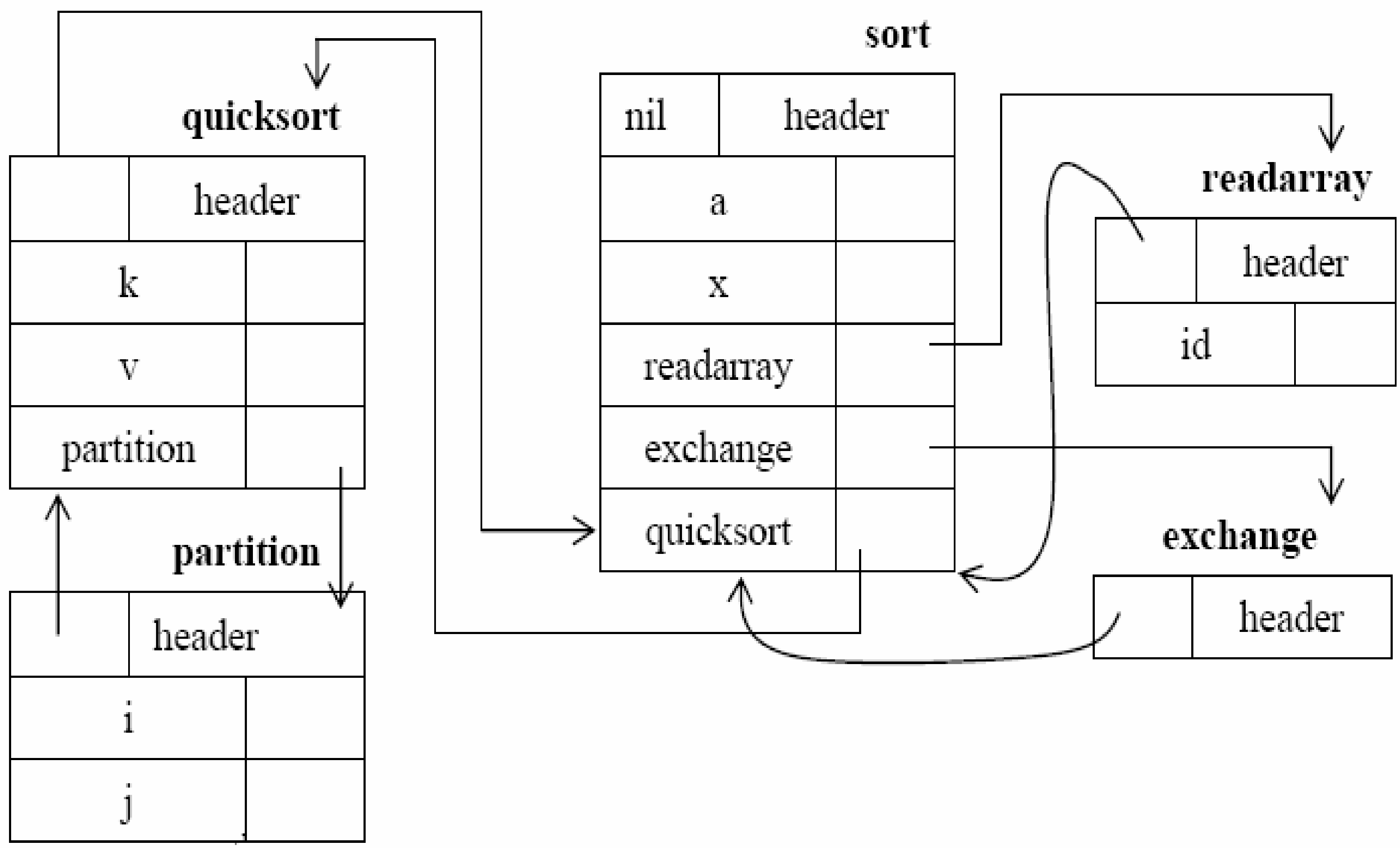
  **D → D; D | id : T | proc id ; D ; S**

  should be added more semantic rules
- A new table is created when a procedure declqration D→ proc id D1; is seen

# Nested procedures for quick sorting

```
Program sort;
Var  a: array[0..10] of integer;
         x: integer;
       Procedure readarray;
          Var i: integer;
          Begin …a… end {readarray};
      Procedure exchange(i, j: integer);
          Begin {exchange} end;
      Procedure quicksort(m, n: integer);
          Var k, v: integer;
        Function partition(y,z: integer): integer;
          Begin ..a..v..exchange(i,j) end; {partition}
      Begin … end; {quicksort}
Begin … end; {sort}
```

# Five Symbol Tables of Sort



# Procedures of semantic rules

**mktable(previous)** – creates a new symbol table and returns a pointer to the new table. The argument *previous* point to previously created symbol table.

**enter(table,name,type,offset)** – creates a new entry for name name in the symbol table pointed to by table,places type *type* and relative address offset in fields within the entry

**enterproc(table,name,newbtable)** – creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*

**addwidth(table,width)** – records the cummulative width of all the entries in table in the header associate with this symbol table.

# Declaration Processing in Nested Procedures

**P → M D** { ***addwidth***(***top***(tblptr), ***top***(offset)); ***pop***(tblptr);
***pop***(offset) }

**M → ε** { t:=***mktable***(null); ***push***(t, tblptr); ***push***(0, offset)}

**D → $D_1$ ; $D_2$**

**D → proc id ; N $D_1$ ; S** { t:=***top***(tblpr); ***addwidth***(t,***top***(offset));
***pop***(tblptr); ***pop***(offset);
***enterproc***(***top***(tblptr), id.name, t)}

**N → ε** {t:=***mktable***(***top***(tblptr)); ***push***(t,tblptr); ***push***(0,offset);}

**D → id : T** {***enter***(***top***(tblptr), id.name, T.***type***, ***top***(offset);
***top***(offset):=***top***(offset) + T.***width***

# Declarations in nested blocks

- All semantic actions in the subtrees for for B and C in $A \rightarrow BC$ $\{action_A\}$ are done before $action_{A,}$ at the end of the production occurs A.
- Sản xuất **M** $\rightarrow \varepsilon$ khởi tạo stack ***tblptr*** với một bảng kí hiệu cho phạm vi ngoài cùng (chương trình sort) bằng lệnh ***mktable(nil)*** đồng thời đặt offset = 0.
- N đóng vai trò tương tự M khi một khai báo chương trình con xuất hiện, nó dùng lệnh ***mktable*(*top*(tblptr))** để tạo ra một bảng mới, tham số ***top*(tblptr)** cho giá trị con trỏ tới bảng lại được đẩy vào đỉnh stack tblptr và 0 được đẩy vào stack offset.
- Với mỗi khai báo **id: T** một ô mới được tạo ra cho id trong bảng kí hiệu hiện hành, stack tblptr không đổi, giá trị top(offset) được tăng lên bởi T.width.
- Khi **D** $\rightarrow$ **proc id ; N $D_1$ ; S** diễn ra thì kích thước của tất cả các đối tượng dữ liệu khai báo trong $D_1$ sẽ nằm trên đỉnh stack offset. Nó được lưu trữ bằng cách dùng Addwidth, các stack tblptr và offset bị đẩy và chúng ta thao tác trên các khai báo của chương trình con.

# Syntax directed definition for assignment statements

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

# Names in the symbol table

- Names in an assignment generated by S must have been declared in either the procedure that S appears in or in some enclosing procedures.
- The *lookup* operation first check if there is an entry for attribute id.name in the symbol table. If so, a pointer to the entry is returned. If the name cannot be found, then lookup returns nil.
- Emit procedure used to emit three-address statements to an output file rather than building up *code* attributes for some nonterminals. Translation can be done by emitting to an output file if the *code* attributes of the nonterminals on the left sides of productions are formed by concatenating the code attributes of the nonterminals on the right side, perhaps with some additional strings in between

# Names in the symbol table (cont'd)

- Consider productions: $D \rightarrow$ proc id; $ND_1$; S
- Names in an assignment generated by S must have been declared in either the procedure that S appears in or in some enclosing procedure.
- When apply to *name*, lookup operation checks if name appears in the current symbol table accessible through *top(tblptr).* If not, lookup uses the pointer in the header of the table to find the symbol table for the enclosing procedure and looks for the name there. If the name cannot be found in any of these scopes, *lookup* will return nil

# Addresing array element

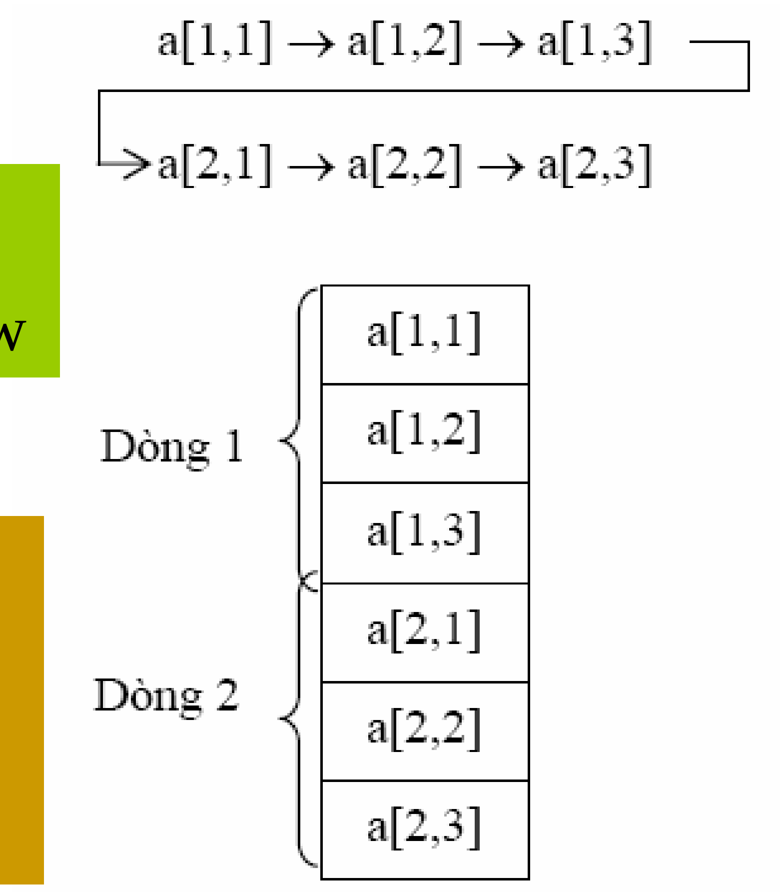
- Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is w then the $i^{th}$ element of array A begins in location:
- A[i] = base + (i-low)*w
- Where:
  - Low is the lower bound of the subscript
  - Base is the relative address of the storage allocated for the array(the relative address of A[low])
- A[i] = i*w + (base – low*w)
- Where c = base – low*w can be evaluated when the declaration of the array is seen. Assume c is saved in the symbol table entry for A,
- $\Rightarrow$ A[i] = i*w + c

# Layouts for a two dimensional array

- By row

Relative address of $A[i_1,i_2]$ =

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

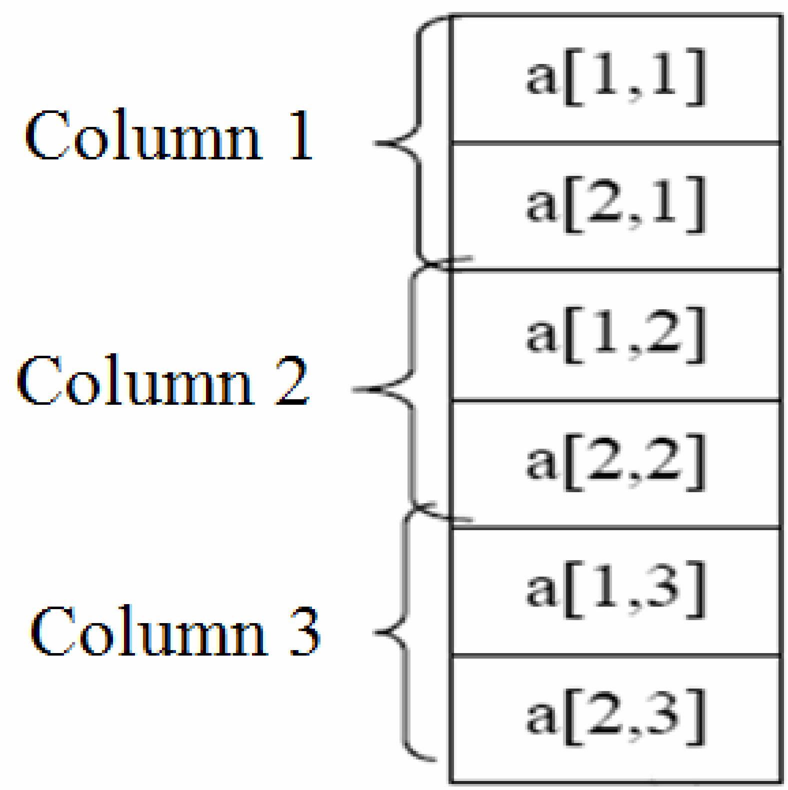



$low_1$, $low_2$: lower bound of $i_1$ và $i_2$

$n_2$: the number of values that $i_2$ can take. If $high_2$ is upper bound of $i_2$ then $n_2 = high_2 – low_2 + 1$

# Layouts for a two dimensional array

- By column



# Boolean Expressions

- **Boolean Expressions** are used to either compute logical values or as conditional expressions in flow-of-control statements.
- We consider Boolean Expressions with the following grammar:
- $E \to E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false}$
- There are two methods to evaluate Boolean Expressions
  1. *Numerical Representation*. Encode true with ’1’ and false with ’0’ and we proceed analogously to arithmetic expressions.
  2. *Jumping* ***Code***. We represent the value of a Boolean Expression by a position reached in a program.

# Numerical Representation of Boolean Expressions

- Expressions will be evaluated from left to right assuming that: or and and are left-associative, and that or has lowest precedence, then and, then not
- Example: The translation for *a or b and not c* is

```
t1 = not c
t2 = b and t1
t3 = a or t2
```

- A relational expression such as a<b is equivalent to the conditional statement if a<b then 1 else 0 and its translation involves *jumps to labeled statements*:

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

# Numerical Representation: The Translation

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

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

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

$E \rightarrow id_1 \text{ relop } id_2 \; \{E.place := newtemp;$

$emit(\text{'if' } id_1.place \; relop.op \; id_2.place \text{ 'goto' } nextstat + 3);$

$emit(E.place \text{ ':=' '0'}); \quad emit(\text{'goto' } nextstat + 2);$

$emit(E.place \text{ ':=' '1'})\}$

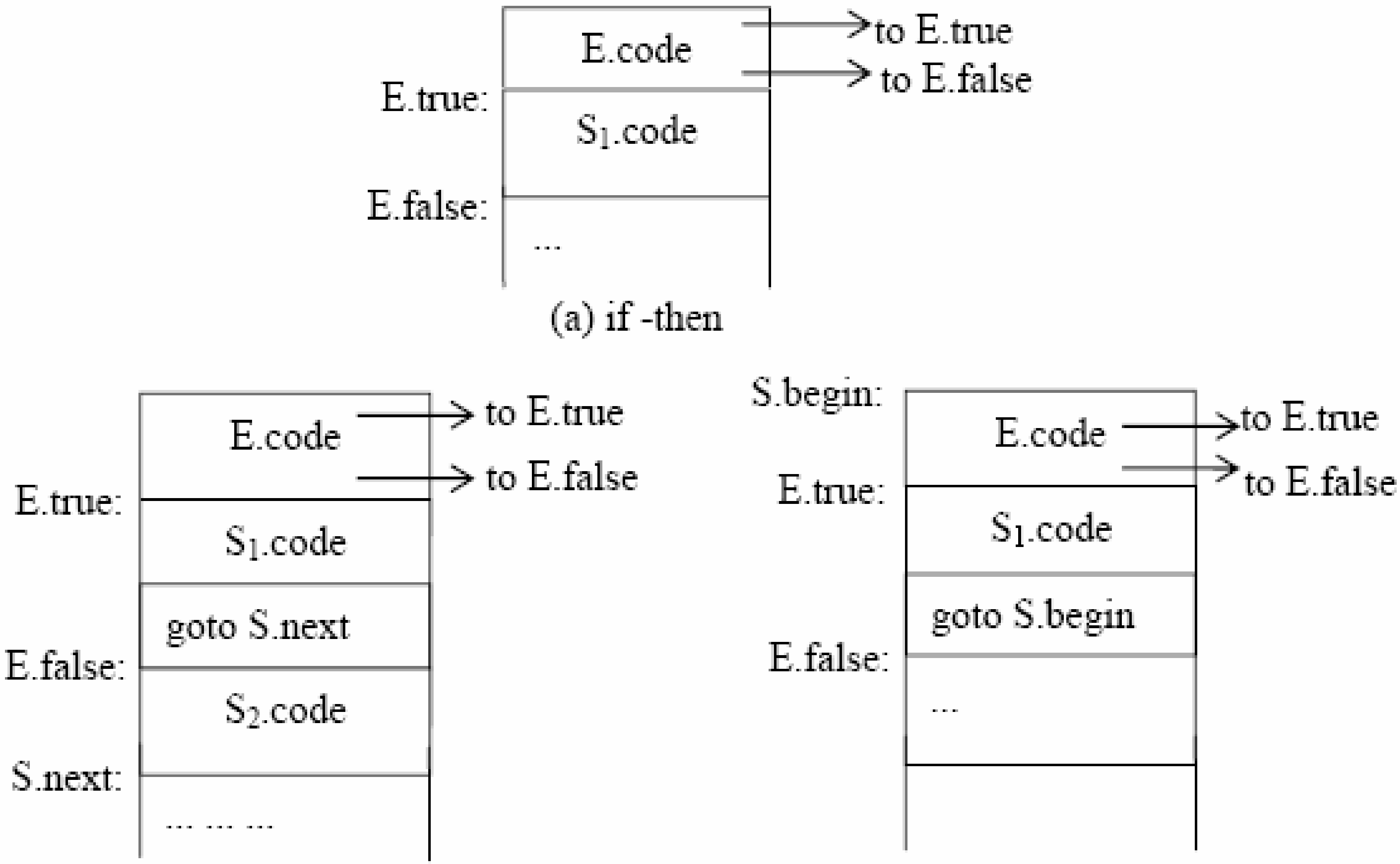
$E \rightarrow \text{true} \quad \{E.place := newtemp; \quad emit(E.place \text{ ':=' '1'})\}$

$E \rightarrow \text{false} \quad \{E.place := newtemp; \quad emit(E.place \text{ ':=' '0'})\}$

# Jumping Code for Boolean Expressions

- The value of a Boolean Expression is represented by a position in the **code**.
- Consider Example 2: We can tell what value t will have by whether we reach statement 101 or statement 103.
- Jumping **code** is extremely useful when Boolean Expressions are in thecontext of flow-of-control statements.
- We start by presenting the translation for flow-of-control statements generated by the following grammar:

# Flow of Control Statements




(a) if -then



(b) if -then-else



(c) while-do

# Flow-of-Control Statements

- In the translation, we assume that a three-address **code** statement can have a *symbolic label*, and that the function *newlabel* generates such labels.
- We associate with E two labels using inherited attributes:
  1. *E.true*, the label to which control flows if E is true;
  2. *E.false*, the label to which control flows if E is false.
- We associate to S the inherited attribute *S.next* that represents the label attached to the first statement after the **code** for S.

# SDT for flow of control statements

| PRODUCTIONS | SEMANTIC RULES |
|---|---|
| $S \rightarrow$ if E then $S_1$ | *E.true := newlabel;*<br>*E.false := S.next;*<br>*$S_1$.next := S.next;*<br>*S.code := E.code \|\| gen(E.true ':') \|\| $S_1$.code* |
| $S \rightarrow$ if E then $S_1$ else $S_2$ | *E.true := newlabel;*<br>*E.false := newlabel;*<br>*$S_1$.next := S.next;*<br>*$S_2$.next := S.next;*<br>*S.code := E.code \|\| gen(E.true ':') \|\| $S_1$.code \|\|*<br>*gen('goto' S.next) \|\|*<br>*gen(E.false ':') \|\| $S_2$.code* |
| $S \rightarrow$ while E do $S_1$ | *S.begin := newlabel;*<br>*E.true := newlabel;*<br>*E.fasle := S.next;*<br>*$S_1$.next := S.begin;*<br>*S.code:= gen(S.begin':') \|\| E.code \|\| gen(E.true ':') \|\|*<br>*$S_1$.code \|\| gen('goto' S.begin)* |

# Flow-of-Control and Boolean Expressions

- **Example.** Translate the following statement:

```
while a < b do
if c < d then
x := y + z
else
x := y - z
```

## Translating Boolean expressions

- If E has form : a<b the the code generated is

  If a<b then goto E.true else goto E.false
- If E has form: E1 or E2 then
  - If E1 is true then E is true
  - If E1is false then evaluate E2; Value of E is true or false depends onE2
- Similarly for E1 and E2

# Translating Boolean expressions

| SẢN XUẤT | QUY TẮC NGỮ NGHĨA |
|---|---|
| $E \rightarrow E_1$ or $E_2$ | $E_1.true := E.true;$<br>$E_1.false := newlabel;$<br>$E_2.true := E.true;$<br>$E_2.false := E.false;$<br>$E.code := E_1.code \mid\mid gen(E.false\ ':') \mid\mid E_2.code$ |
| $E \rightarrow E_1$ and $E_2$ | $E_1.true := newlabel;$<br>$E_1.false := E.false;$<br>$E_2.true := E.true;$<br>$E_2.false := E.false;$<br>$E.code := E_1.code \mid\mid gen(E.true\ ':') \mid\mid E_2.code$ |
| $E \rightarrow$ not $E_1$ | $E_1.true := E.false;$<br>$E_1.false := E.true;$<br>$E.code := E_1.code$ |
| $E \rightarrow (E_1)$ | $E_1.true := E.true;$<br>$E_1.false := E.false;$<br>$E.code := E_1.code$ |
| $E \rightarrow id_1$ relop $id_2$ | $E.code := gen('if'\ id_1.place\ relop.op\ id_2.place\ 'goto'\ E.true) \mid\mid gen('goto'\ E.false)$ |
| $E \rightarrow$ true | $E.code := gen('goto'\ E.true)$ |
| $E \rightarrow$ false | $E.code := gen('goto'\ E.false)$ |