

# Compilers—Lecture VII

## Intermediate Code Generation

Alessandro Artale

Faculty of Computer Science – Free University of Bolzano

POS Building – Room: 2.03

`artale@inf.unibz.it`

`http://www.inf.unibz.it/~artale/`

## Summary of Lecture VII

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

## Intermediate Code Generation

- An intermediate code is generated as a program for an abstract machine.
- 1. The intermediate code should be easy to translate into the target program.
- 2. A machine-independent *Code Optimizer* can be applied before generating the target 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:  $x := y \ op \ z$   
where  $x, y, z$  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.

## Types of Three-Address Statements

Three-Address statements are akin to assembly code: Statements can have *labels* and 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 (Cont.)

6. *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 x1`

`param x2`

...

`param xn`

`call p, n`

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

8. *Pointer Assignments*: `x := &y`, `x := *y`, or `*x := y`; where `&y` stands for the address of `y`, and `*y` the value of `y`.

## Summary

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

## Code for Assignments

- The following S-attributed definition generates three-address code for assignments ( $\parallel$  means string concatenation).

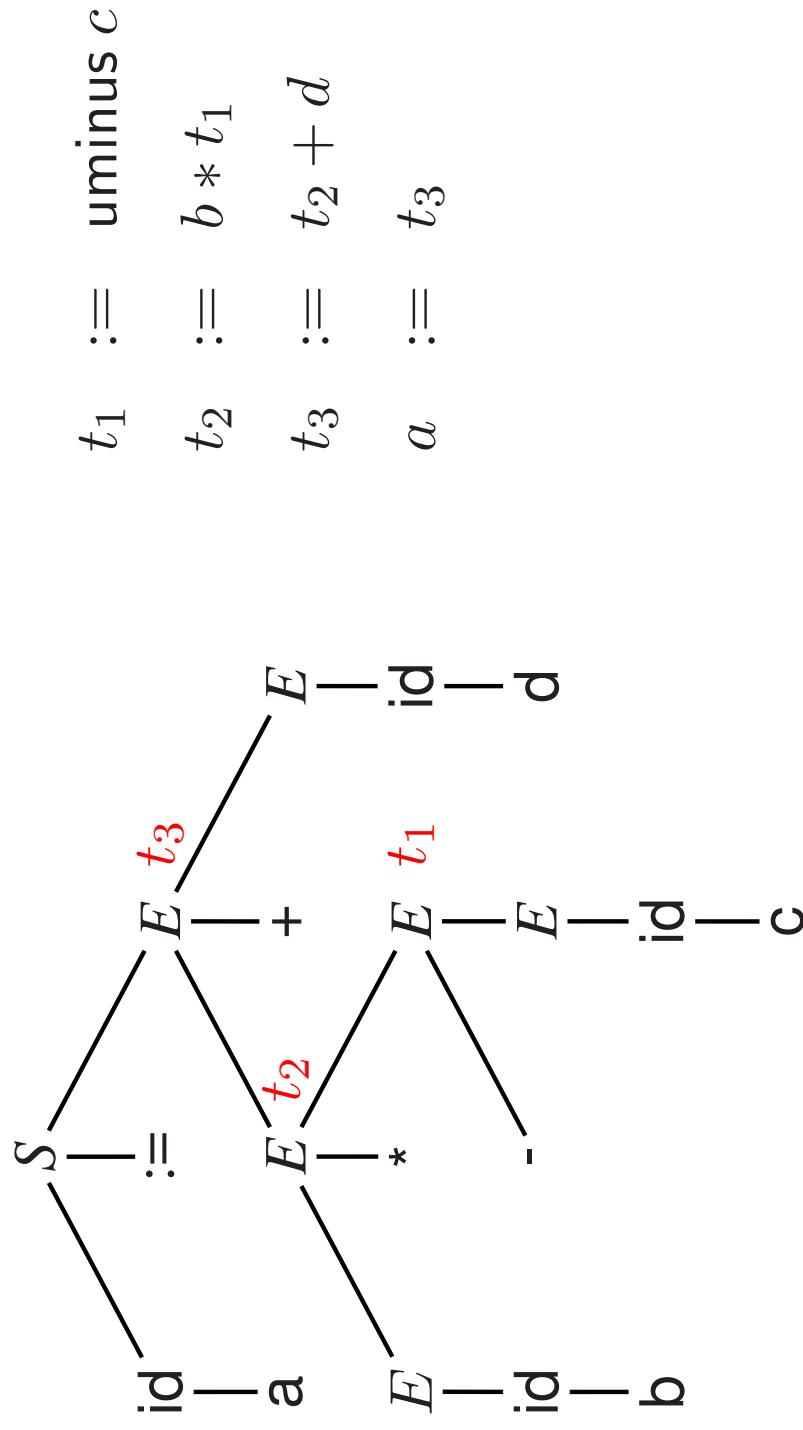
PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{id} := E$	$S.code := E.code \parallel \text{gen}(\text{id}.addr' := E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr := \text{newtemp};$ $E.code := E_1.code \parallel E_2.code \parallel$ $\text{gen}(E.addr' := E_1.addr' + E_2.addr)$
$E \rightarrow E_1 * E_2$	$E.addr := \text{newtemp};$ $E.code := E_1.code \parallel E_2.code \parallel$ $\text{gen}(E.addr' := E_1.addr' * E_2.addr)$
$E \rightarrow -E_1$	$E.addr := \text{newtemp};$ $E.code := E_1.code \parallel \text{gen}(E.addr' := 'uminus' E_1.addr)$
$E \rightarrow (E_1)$	$E.addr := E_1.addr; E.code := E_1.code$
$E \rightarrow \text{id}$	$E.addr := \text{id}.addr; E.code := ''$

## Code for Assignments (Cont.)

- The synthesized attribute *S.code* represents the three-address code for the assignment.
- Temporary names are generated for intermediate computations.
- To expressions we associate two synthesized attributes:
  1. *E.addr*, holds the address for the value of *E*;
  2. *E.code*, the three-address code for evaluating *E*.
- The function *newtemp* generates distinct temporary names  $t_1, t_2, \dots$
- 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.

## Code for Assignments: An Example

Given the assignment  $a := b * -c + d$  the code generated by the above grammar is:



## Assignments and Symbol Tables

- We assumed that names/addresses stand for pointers to their symbol table entries since other info are needed for the final code generation (in particular, the storage address for the variable).
- **Note. Under this assumption Temporary Names must be also entered into the symbol table as they are created by the *newtemp* function.**
- The function *lookup(id.name)* return *nil* if the entry is not found, otherwise a pointer to the entry is returned.
  - The *lookup(id.name)* can be easily modified to account for scope: If *name* does not appear in the current symbol table the enclosing symbol table is checked (see the Lecture on “Symbol Table”).
- Instead of using a *code* attribute we *emit* three-address code to an output file
  - This is always possible if the *code* of the non-terminal on the left is obtained by concatenating the *code* attributes of the non-terminals on the right in the same order (perhaps with some additional strings in between).

## Assignments and Symbol Tables: The Translation

PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{id} := E$	$p := \text{lookup}(\text{id.name});$ if $p \neq \text{nil}$ then $\text{emit}(p' := E.addr)$ else <i>error</i>
$E \rightarrow E_1 + E_2$	$E.addr := \text{newtemp};$ $\text{emit}(E.addr' := E_1.addr' + E_2.addr)$
$E \rightarrow E_1 * E_2$	$E.addr := \text{newtemp};$ $\text{emit}(E.addr' := E_1.addr' * E_2.addr)$
$E \rightarrow -E_1$	$E.addr := \text{newtemp};$ $\text{emit}(E.addr' := 'uminus' E_1.addr)$
$E \rightarrow (E_1)$	$E.addr := E_1.addr$
$E \rightarrow \text{id}$	$p := \text{lookup}(\text{id.name});$ if $p \neq \text{nil}$ then $E.addr := p$ else <i>error</i>

## Summary

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

## 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:

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id rel op id} \mid \text{true} \mid \text{false}$$

- 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 1.** The translation for “**a or (b and (not c))**” is:

$t_1 := \text{not } c$

$t_2 := b \text{ and } t_1$

$t_3 := a \text{ or } t_2$

- **Example 2.** 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

- The following S-Attributed Definition makes use of the global variable *nextstat* that gives the index of the next three-address code statement and is incremented by *emit*.
- We use the attribute *op* to determine which of the comparison operators is represented by *relop*.

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 \text{ or } E_2$	$E.addr := newtemp;$ $emit(E.addr' := E_1.addr' \text{ or } E_2.addr')$
$E \rightarrow E_1 \text{ and } E_2$	$E.addr := newtemp;$ $emit(E.addr' := E_1.addr' \text{ and } E_2.addr')$
$E \rightarrow \text{not } E_1$	$E.addr := newtemp;$ $emit(E.addr' := 'not' E_1.addr)$
$E \rightarrow (E_1)$	$E.addr := E_1.addr$
$E \rightarrow id_1 \text{ relop } id_2$	$E.addr := newtemp;$ $emit('if' id_1.addr \text{ relop } id_2.addr \text{ goto } nextstat + 3);$ $emit(E.addr' := '0');$ $emit('goto' nextstat + 2);$ $emit(E.addr' := '1')$
$E \rightarrow \text{true}$	$E.addr := newtemp; emit(E.addr' := '1')$
$E \rightarrow \text{false}$	$E.addr := newtemp; emit(E.addr' := '0')$

## 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 the context of flow-of-control statements.
- We start by presenting the translation for flow-of-control statements generated by the following grammar:

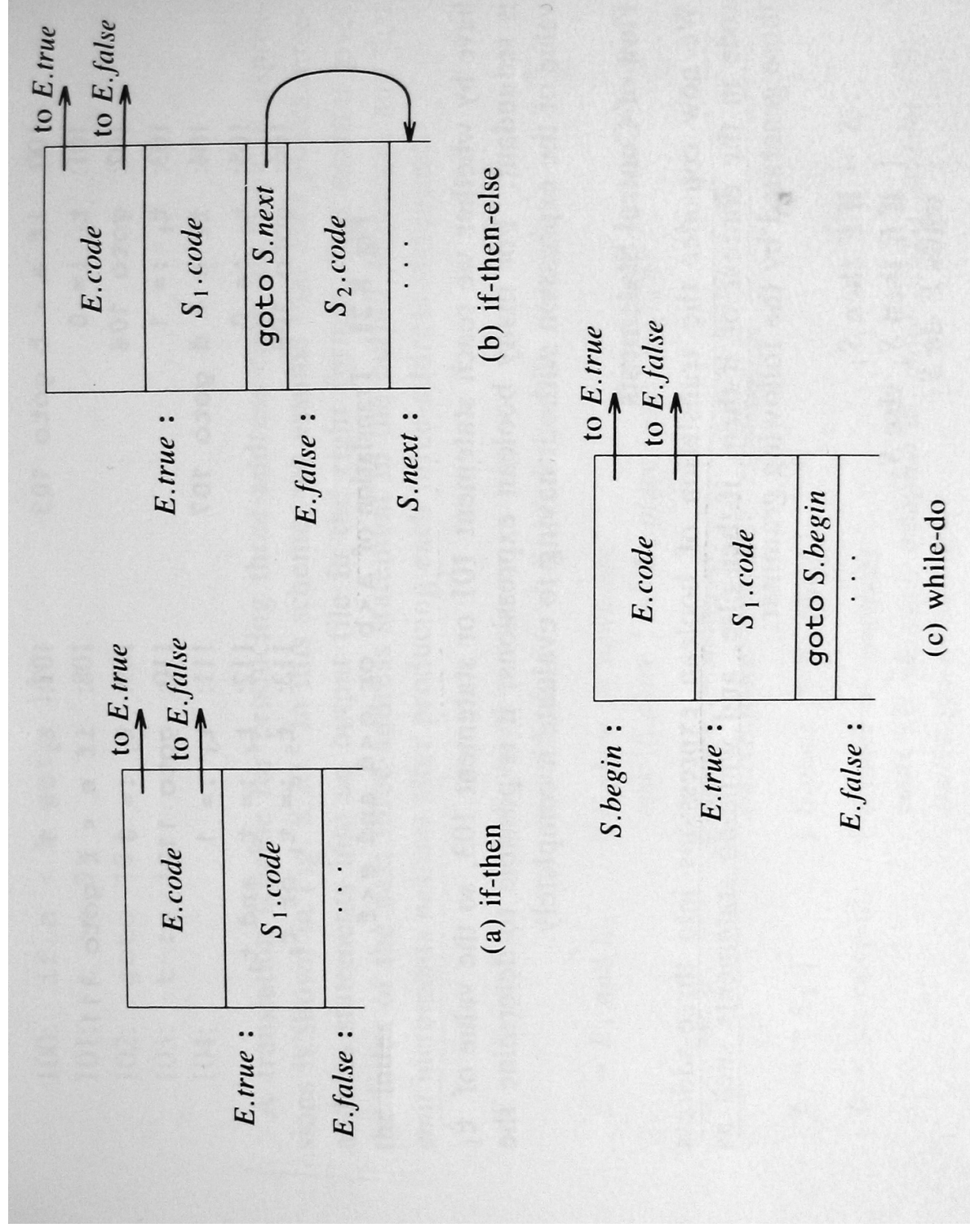
$$\begin{array}{l} S \rightarrow \text{if } E \text{ then } S \\ \quad | \text{if } E \text{ then } S_1 \text{ else } S_2 \\ \quad | \text{while } E \text{ do } S \end{array}$$

## 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  $S$ .
- **Note 1.** This method of generating labels associated to  $S.next$  can lead to a proliferation of labels: The *backpatching* method (see the Book) creates labels only when needed.
- **Note 2.** In the following translation the initialization of  $S.next$  is not shown.

## Flow-of-Control Statements (Cont.)

- The following figures show how the flow-of-control statements are translated.



PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{if } E \text{ then } S_1$	$E.true := \text{newlabel}; E.false := S.next;$ $S_1.next := S.next;$
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$S.code := E.code \parallel \text{gen}(E.true \text{ ' :'}) \parallel S_1.code$ $E.true := \text{newlabel}; E.false := \text{newlabel};$ $S_1.next := S.next; S_2.next := S.next;$ $S.code := E.code \parallel \text{gen}(E.true \text{ ' :'}) \parallel S_1.code \parallel$ $\quad \text{gen('goto' } S.next) \parallel$ $\quad \text{gen}(E.false \text{ ' :'}) \parallel S_2.code$
$S \rightarrow \text{while } E \text{ do } S_1$	$E.begin := \text{newlabel};$ $E.true := \text{newlabel}; E.false := S.next;$ $S_1.next := E.begin;$ $S.code := \text{gen}(E.begin \text{ ' :'}) \parallel E.code \parallel$ $\quad \text{gen}(E.true \text{ ' :'}) \parallel S_1.code \parallel$ $\quad \text{gen('goto' } E.begin)$

## Jumping Code for Boolean Expressions (Cont.)

- Boolean Expressions are translated in a sequence of conditional and unconditional jumps to either  $E.true$  or  $E.false$ .
- $a < b$ . The code is of the form:  
if  $a < b$  then goto  $E.true$   
goto  $E.false$
- $E_1$  or  $E_2$ . If  $E_1$  is true then  $E$  is true, so  $E_1.true = E.true$ . Otherwise,  $E_2$  must be evaluated, so  $E_1.false$  is set to the label of the first statement in the code for  $E_2$ .
- $E_1$  and  $E_2$ . Analogous considerations apply.
- not  $E_1$ . We just interchange the true and false with that for  $E$ .
- **Note.** Both the *true* and *false* attributes are inherited and the translation is an L-attributed grammar.

# Jumping Code for Boolean Expressions: The Translation

(22)

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 \text{ or } E_2$	$E_1.true := E.true; E_1.false := newlabel;$ $E_2.true := E.true; E_2.false := E.false;$ $E.code := E_1.code \parallel gen(E_1.false ' :) \parallel E_2.code$
$E \rightarrow E_1 \text{ and } E_2$	$E_1.true := newlabel; E_1.false := E.false;$ $E_2.true := E.true; E_2.false := E.false;$ $E.code := E_1.code \parallel gen(E_1.true ' :) \parallel E_2.code$
$E \rightarrow \text{not } E_1$	$E_1.true := E.false; E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true; E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow id_1 \text{ relop } id_2$	$E.code := gen('if' id_1.addr \text{ relop } id_2.addr 'goto' E.true) \parallel$ $gen('goto' E.false)$
$E \rightarrow \text{true}$	$E.code := gen('goto' E.true)$
$E \rightarrow \text{false}$	$E.code := gen('goto' E.false)$

## Flow-of-Control and Boolean Expressions: An Example

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

while  $a < b$  do

  if  $c$  or  $d$  then

$x := y + z$

  else

$x := y - z$

## Mixed-Mode Boolean Expressions

- Boolean Expressions often contain Arithmetic sub-expressions—e.g.  $(a+b) < c$ .
- On the other hand, if `true = 1` and `false = 0`, then  $(a < b) + (b < a)$  can be an Arithmetic expression—with value 0 if  $a=b$  and 1 otherwise.
- The method of representing Boolean Expressions by Jumping code is still a good option.

## Mixed-Mode Boolean Expressions (Cont.)

- Consider the following Grammar:

$$E \rightarrow E + E \mid E \text{ and } E \mid E \text{ relop } E \mid \text{id}$$

- $E + E$ , produces an arithmetic result, and the arguments can be mixed;
- $E$  and  $E$ , produces a Boolean result, and both arguments must be Boolean;
- $E$  relop  $E$ , produces a Boolean result, and the arguments can be mixed;
- $\text{id}$  is assumed of type arithmetic.

## Mixed-Mode Boolean Expressions: The Translation

- To generate code we use a synthesized attribute  $E.type$ , that will be either *arith* or *bool*.
- Boolean Expressions will have inherited attributes  $E.true$  and  $E.false$  useful for the jumping code.
- Arithmetic Expressions will have the synthesized attribute  $E.addr$  standing for the (temporary) variable holding the value of  $E$ .
- The global variable *nextstat* gives the index of the next three-address code statement and is incremented by *gen*.

**Mixed-Mode Boolean Expressions: The Translation (Cont.)**

(27)

- Let just show the semantic rule for  $E \rightarrow E + E$ .

$$E \rightarrow E_1 + E_2$$

$$E.type := arith;$$

if  $E_1.type := arith$  and  $E_2.type := arith$  then begin

$$E.addr := newtemp;$$

$$E.code := E_1.code \parallel E_2.code \parallel$$

$$\quad gen(E.addr' :=' E_1.addr' +' E_2.addr)$$

end

else if  $E_1.type := arith$  and  $E_2.type := bool$  then begin

$$E.addr := newtemp;$$

$$E_2.true := newlabel; E_2.false := newlabel;$$

$$E.code := E_1.code \parallel E_2.code \parallel$$

$$\quad gen(E_2.true' ! E.addr' :=' E_1.addr + 1) \parallel$$

$$\quad gen('goto'nextstat + 1) \parallel$$

$$\quad gen(E_2.false' ! E.addr' :=' E_1.addr)$$

end

else if ...

## Summary of Lecture VII

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