Intermediate Representations

- A variety of intermediate representations are used in compilers
- Most common intermediate representations are:
  - Abstract Syntax Tree
  - Directed Acyclic Graph (DAG)
  - Three-Address Code
  - Code for a simplified stack-based virtual machine (Example: P-code)
- Abstract Syntax is adequate representation of the source code
  - However, it is not linear and does not resemble target code
  - High-level constructs, such as **if** and **while**, should be translated into jumps
- Directed acyclic graph is an optimization of a syntax tree

Three-Address Code

- Generalized assembly code for a virtual 3-address machine
- 3-address code represents a **linearization** of the syntax tree
- 3-address code can be:
  - **High level**: representing all operations as abstractly as a syntax tree
  - **Low level**: closely resembling target code
- Basic 3-address instruction consists of an operator and 3 addresses
  - Two addresses for the two operands and one address for the result
  - General form `x := y op z`
  - *op* is an **operator code**
  - *x*, *y*, and *z* are typically implemented as **pointers to symbols**
  - *x* is either an **identifier** or a **temporary** symbol
  - *y* and *z* can be an **identifier**, a **literal**, or a **temporary** symbol

Compiler Design – © Muhammed Mudawwar

Types of 3-Address Instructions

- Arithmetic, logical, and shift instructions are of the form:

  `x := y op z` **Binary Operator**

  op can be any of the following operators:

  `ADD, SUB, MUL, DIV, MOD,` **Binary Arithmetic**

  `AND, OR, XOR, SHL, SHR, SHRA,` **Logical and shift**

  `EQ, NE, LT, LE, GT, GE` **Relational operators**

  The above operators are **generic**

  Type information can be also added to each operator

  - To distinguish between integer and floating-point operations

- Unary *op* instructions are of the form:

  `x := op y` **Unary Operator**

  op can be **`PLUS`**, **`MINUS`**, or **`NOT`** operator

  op can also be conversion operators to convert between integer and FP

Example on Translating an Expression

- Consider the translation of $(2 + a * (b - c / d)) / e$

```
t1 := c / d
t2 := b - t1
t3 := a * t2
t4 := 2 + t3
t5 := t4 / e
```

- Compiler **generates temporaries** when translating into 3AC
- *t1*, *t2*, *t3*, *t4*, and *t5* are generated temporaries
- Temporaries are **identified by number**
- Temporaries are stored in symbols, like identifiers
- Type information is also added to temporary symbols

Compiler Design – © Muhammed Mudawwar

3-Address Instructions for Copy and Jumps

- Move or copy instruction is of the form:

  `x := y`

- Unconditional jump and label instructions are of the form:

  `GOTO Ln` — **`Ln`** is a **label** identified by a number *n*

  `LABEL Ln`

  A label is not an instruction; It is the address of the following instruction

- Conditional branch instructions are of the form:

  `IF x goto Ln` — Branch if *x* is true

  `IFNOT x goto Ln` — Branch if *x* is false

  x is a Boolean variable that evaluate to **true** or **false**

- The general conditional branch instruction is of the form:

  `if x relop y goto Ln`

  Conditional branches are used to implement conditional and loop statements

Compiler Design – © Muhammed Mudawwar

Example on Translating a While Loop

❖ Consider the following while loop:

```
sum:=0; i:=1;
while (i<n) {
  sum := sum + i;
  i := i+1;
}
```

❖ A translation of the above loop into 3AC is shown below:

```
sum := 0
i := 1
goto  L2
label L1
sum := sum + i
i := i + 1
label L2
if i < n goto L1
```


Compiler Design – © Muhammed Mudawwar

Implementation of 3-Address Code

- A 3-address instruction is implemented as a **quadruple**:
  - An **operator code**
  - Two pointers to **operand symbols**
  - A pointer to **result symbol**, **goto target**, or **label number**
- A code sequence can be implemented as an array or a linked list
- A linked list is preferable because it …
  - Facilitates reordering and concatenation of instructions
  - Grows dynamically as required
- A 3-address instruction has a **link** to the next instruction

opcode	**result target label**	**first**	**second**	**link**

Three-Address Instruction Structure

- The structure of a 3-address instruction is given below:

```
struct Inst {
  Inst(OpType op, Symbol* r=0, Symbol* f=0, Symbol* s=0);
  Inst(OpType op, Inst* t,     Symbol* f=0, Symbol* s=0);
  Inst(OpType op, unsigned l);
  OpType  opcode;     // Operation Code
  union {             // Anonymous union
    Symbol*  result; // Either a symbol pointer, or
    Inst*    target; // Target label used with GOTO
    unsigned label;  // Label number used with LABEL
  };
  Symbol* first;      // First operand symbol
  Symbol* second;     // Second operand symbol
  Inst*   link;       // Link to next instruction
};
```


Compiler Design – © Muhammed Mudawwar

Generating Temporaries and Labels

- *newtemp*() allocates and returns a new temporary symbol
- The static variable *num* ensures a unique number for every call

```
Symbol* newtemp() {
  static int num = 1;
  Symbol* temp = new Symbol(TEMP,num);
  num++;
  return temp;
}
```

- *newlabel*() allocates and returns a new label instruction

```
Inst* newlabel() {
  static int num = 1;
  Inst* label = new Inst(LABEL,num);
  num++;
  return label;
}
```

Concatenating Instructions and Code

- A code sequence is a linear linked list of instructions
- We identify the **first** and **last** instructions in a code sequence

```
struct Code {       // Code sequence structure
  Inst* first;      // First instruction in code sequence
  Inst* last;       // Last instruction in code sequence
};
```

- The + operator is overloaded to mean concatenation
  - Four + operators will concatenate code sequences and instructions
  - The result of concatenation is a code sequence

```
Code operator+(Code  a, Code  b);
Code operator+(Code  c, Inst* i);
Code operator+(Inst* i, Code  c);
Code operator+(Inst* i, Inst* j);
```


Compiler Design – © Muhammed Mudawwar

Concatenating Two Code Sequences

❖ The + operator links the pointers of two code sequences:

```
Code operator+(Code a, Code b) {
  Code c;
  if (a.first == 0) {            // Code a is NULL
    c = b;
  }
  else if (b.first == 0) {       // Code b is NULL
    c = a;
  }
  else {                         // General Case
    c.first = a.first;
    c.last  = b.last;
    a.last->link = b.first;
  }
  return c;
}
```


Compiler Design – © Muhammed Mudawwar

Translating Expressions into 3-Address Code

Synthesized Attributes

E.code:	Code sequence evaluating *E*
E.sym:	Symbol representing value of *E*
addop.*op*:	addition operator: **ADD** or **SUB**
mulop.*op*:	multiplication operator: **MUL**, **DIV**, or **MOD**

Grammar Rules	Semantic Rules
$E \rightarrow E^1$ **addop** E^2	*E.sym* := newtemp(); *AddInst* := **new** Inst(**addop**.*op*, *E.sym*, E^1*.sym*, E^2*.sym*); *E.code* := E^1*.code* + E^2*.code* + *AddInst*;
$E \rightarrow E^1$ **mulop** E^2	*E.sym* := newtemp(); *MulInst* := **new** Inst(**mulop**.*op*, *E.sym*, E^1*.sym*, E^2*.sym*); *E.code* := E^1*.code* + E^2*.code* + *MulInst*;
$E \rightarrow$ *UnaryOp* E^1	*E.sym* := newtemp(); *UnaryInst* := **new** Inst(*UnaryOp.op*, *E.sym*, E^1*.sym*); *E.code* := E^1*.code* + *UnaryInst*;

Compiler Design – © *Muhammed Mudawwar*

Translating Expressions – cont'd

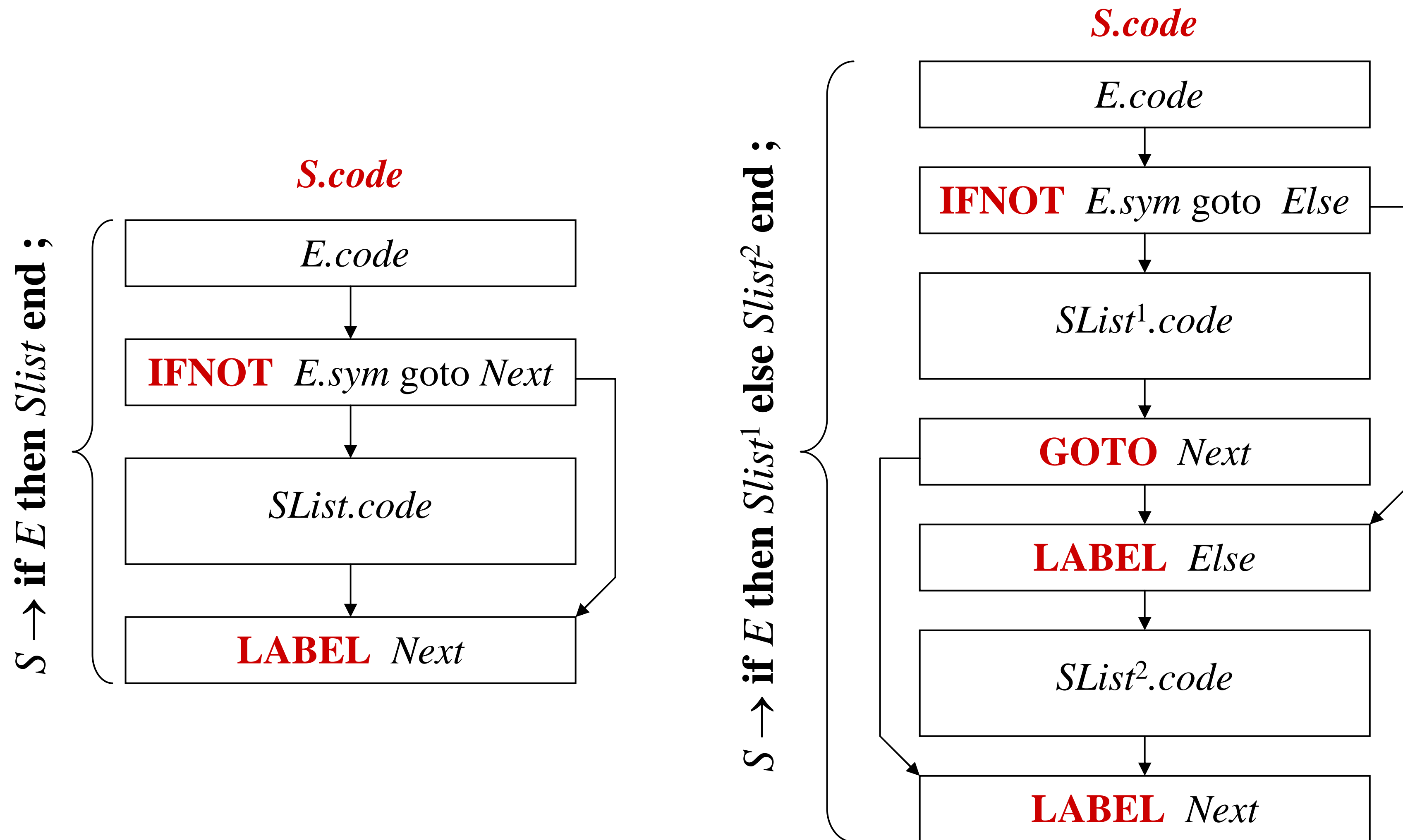
Synthesized Attributes

UnaryOp.op:	Unary operator: **PLUS**, **MINUS**, **NOT**
id.*name*:	Identifier name
num.*sym*:	Literal symbol holding number value

Grammar Rules	Semantic Rules
$E \rightarrow (\; E^1 \;)$	$E.sym := E^1.sym$; $E.code := E^1.code$;
$E \rightarrow$ **id**	$E.sym := idTable$.lookup(**id**.*name*); $E.code.first := E.code.last = 0$;
$E \rightarrow$ **num**	$E.sym :=$ **num**.*sym*; $E.code.first := E.code.last = 0$;
UnaryOp $\rightarrow$ **addop**	**if addop**.*op* = **ADD then** *UnaryOp.op* := **PLUS** **else** *UnaryOp.op* := **MINUS**
UnaryOp $\rightarrow$ **not**	*UnaryOp.op* := **NOT**

Compiler Design – © *Muhammed Mudawwar*

Translating If-Statement into 3-Address Code



Compiler Design – © Muhammed Mudawwar

Translating If-Statement – cont'd

Synthesized Attributes

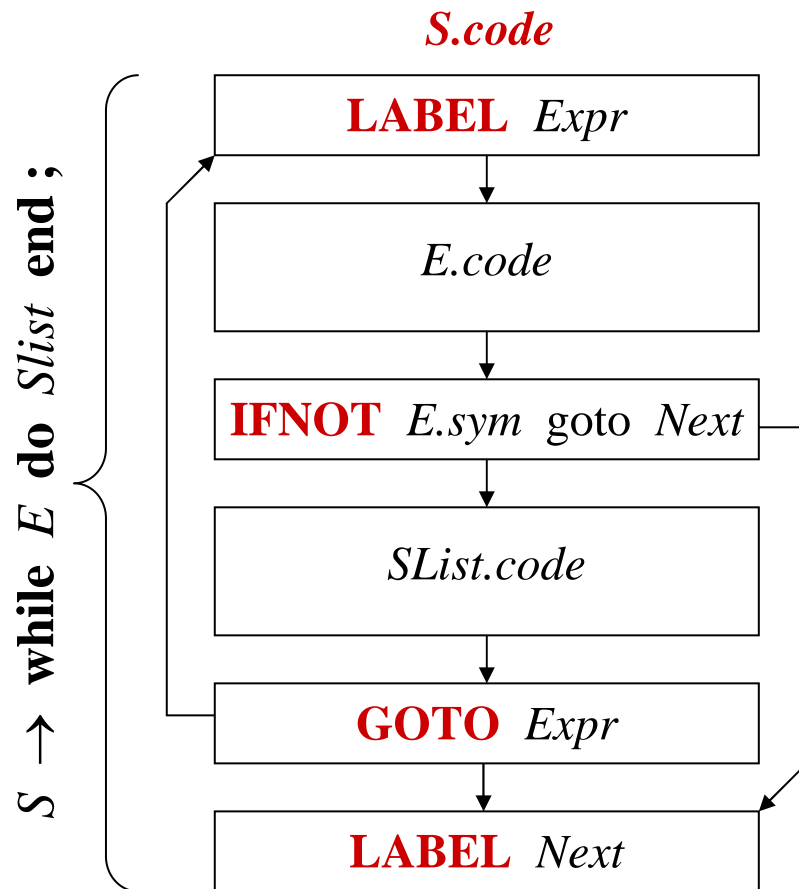
E.code:	Code sequence for *E*
E.sym:	Symbol representing *E*
S.code:	Code sequence of a statement
Slist.code:	Code sequence of a statement list

Grammar Rules	Semantic Rules
S → **if** *E* **then** *Slist* **end ;**	*Next* := newlabel(); *IfNotNext* := **new** Inst(**IFNOT**, *Next*, *E.sym*); *S.code* := *E.code* + *IfNotNext* + *Slist.code* + *Next*;
S → **if** *E* **then** $Slist^1$ **else** $Slist^2$ **end ;**	*Else* := newlabel(); *Next* := newlabel(); *IfNotElse* := **new** Inst(**IFNOT**, *Else*, *E.sym*); *GotoNext* := **new** Inst(**GOTO**, *Next*); *S.code* := *E.code* + *IfNotElse* + $Slist^1$*.code* + *GotoNext* + *Else* + $Slist^2$*.code* + *Next*;

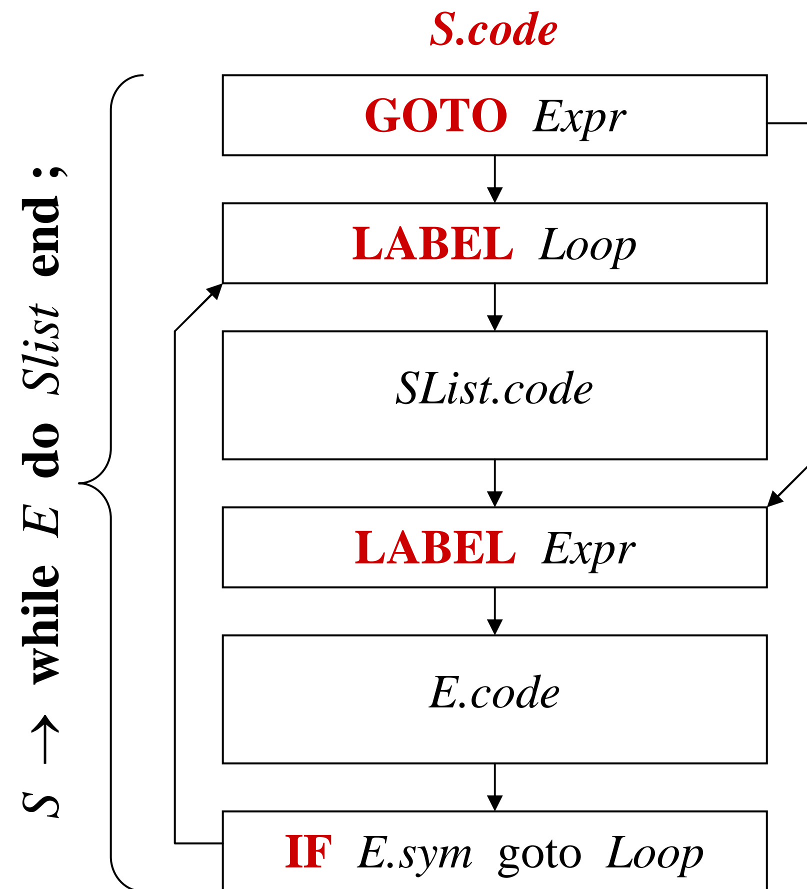
Compiler Design – © Muhammed Mudawwar

Translating While Statement

Possible translation



Better translation



Translating While and Statement Lists

Synthesized Attributes

E.code:	Code sequence evaluating *E*
S.code:	Code sequence of statement *S*
Slist.code:	Code sequence of statement list *Slist*
E.sym:	Symbol representing value of *E*

Grammar Rules	Semantic Rules
S → **while** *E* **do** *Slist* **end ;**	*Loop* := newlabel(); *Expr* := newlabel(); *GotoExpr* := **new** Inst(**GOTO**, *Expr*); *IfNotLoop* := **new** Inst(**IFNOT**, *Loop*, *E.sym*); *S.code* := *GotoExpr* + *Loop* + *Slist.code* + *Expr* + *E.code* + *IfNotLoop*;
Slist → $Slist^1$ *S*	*Slist.code* := $Slist^1$*.code* + *S.code*;
Slist → ε	*Slist.code.first* := 0; *Slist.code.last* := 0;

Compiler Design – © Muhammed Mudawwar