Translation of Expressions

- Here we consider the translation of expressions and statements. We begin with translation of expressions into three-address code. An expression with more than one operator, like a + b * c, will translate into instructions with at most one operator per instruction.
- The syntax-directed definition in the figure given below builds up the three-address code for an assignment statement S using attribute 'code' for S and attributes 'addr' and 'code' for an expression E. Attributes 'S.code' and 'E.code' denote the three-address code for S and E, respectively. Attribute 'E.addr' denotes the address that will hold the value of E. An address can be a name, a constant, or a compiler generated temporary.

PRODUCTION	SEMANTIC RULES
$S \rightarrow id = E$	S.code = E.code || gen (id.lexeme '=' E.addr)
$E \rightarrow E_1 + E_2$	E.addr = new Temp() E.code = E_1.code || E_2.code || gen (E.addr '=' E_1.addr '+' E_2.addr)
$E \rightarrow - E_1$	E.addr = new Temp() E.code = E_1.code || gen (E.addr '=' 'minus' E_1.addr)
$E \rightarrow (E_1)$	E.addr = E_1.addr E.code = E_1.code
$E \rightarrow id$	E.addr = id.Lexeme E.code = ''

Fig: Three-address code for expressions.

- When an expression is a single identifier, i.e., $E \Rightarrow id$, where id is x, then x itself holds the value of the expression. E.code is empty.
- For convenience we use the notation gen (x '=' y '+' z) to represent the three-address instruction $x = y + z$.
- A sequence of distinct temporary names $t_1, t_2, \cdots$ is created by successively executing 'new Temp()'.
- In practice, three-address statements might be sent to an output file, rather than built up into the 'code' attributes. For example,

  $E \rightarrow E_1 + E_2$ emit (E.addr '=' E_1.addr '+' E_2.addr)

- Example:

  The syntax-directed definition given above translates the assignment statement $a = b + -c$ into the three-address code sequence

  $t_1 = -c$

  $t_2 = b + t_1$

  $a = t_2$.

[Start with example in page 82]

Translation of Control Flow Statements * [Introduce numerical representation first]?

- The translation of statements such as if-else statements and while-statements is tied to the translation of boolean expressions.
- In programming languages, boolean expressions are often used to
  1. Alter the flow of control: Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expressions is implicit in a position reached in a program. For example, in 'if (E) S', the expression E must be true if statement S is reached.
  2. Compute logical values: A boolean expression can represent 'true' or 'false' as values. Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

* It looks better to introduce translation of Boolean expressions using numerical representation first. "Short circuit code" can be considered as an optimized version. Ref ASU, Page 489.

83(a).

May cause confusion using S.next?
Hence, skip 83(a)&(b).

Skip 83(a) & (b).

TRANSLATION OF CONTROL-FLOW STATEMENTS (INTRODUCTION): Start using this.

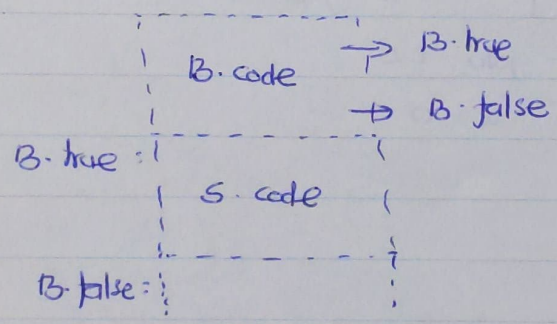
- Example:

  S : if (a < b) x = 0

- Intermediate code for S is:

  if a < b goto L1
  goto L2
  L1 : x = 0
  L2 :

- How to generate intermediate code for "if (B) S"?
  - Design:



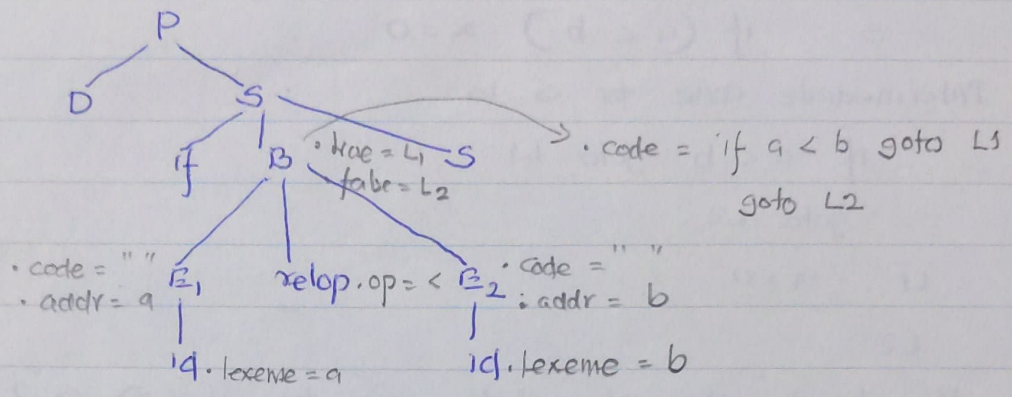
  - Implementation: (Semantic actions)

S → if (B) S_1 { B.true = newLabel();
B.false = newLabel();
S.code = B.code || label(B.true) ||
S_1.code || label(B.false)

B → E_1 relop E_2 { B.code = E_1.code || E_2.code ||
gen('if' E_1.addr relop.op E_2.addr 'goto' B.true)
gen('goto' B.false) }

- Example :

if (a < b) x = 0



S . code = if a < b goto L1
goto L2
L1 : x = 0
L2 :

- Note :
  - We can use inherited attribute "next" for S and have an alternative method for intermediate code generation for control constructs. The solution using this approach is given below.