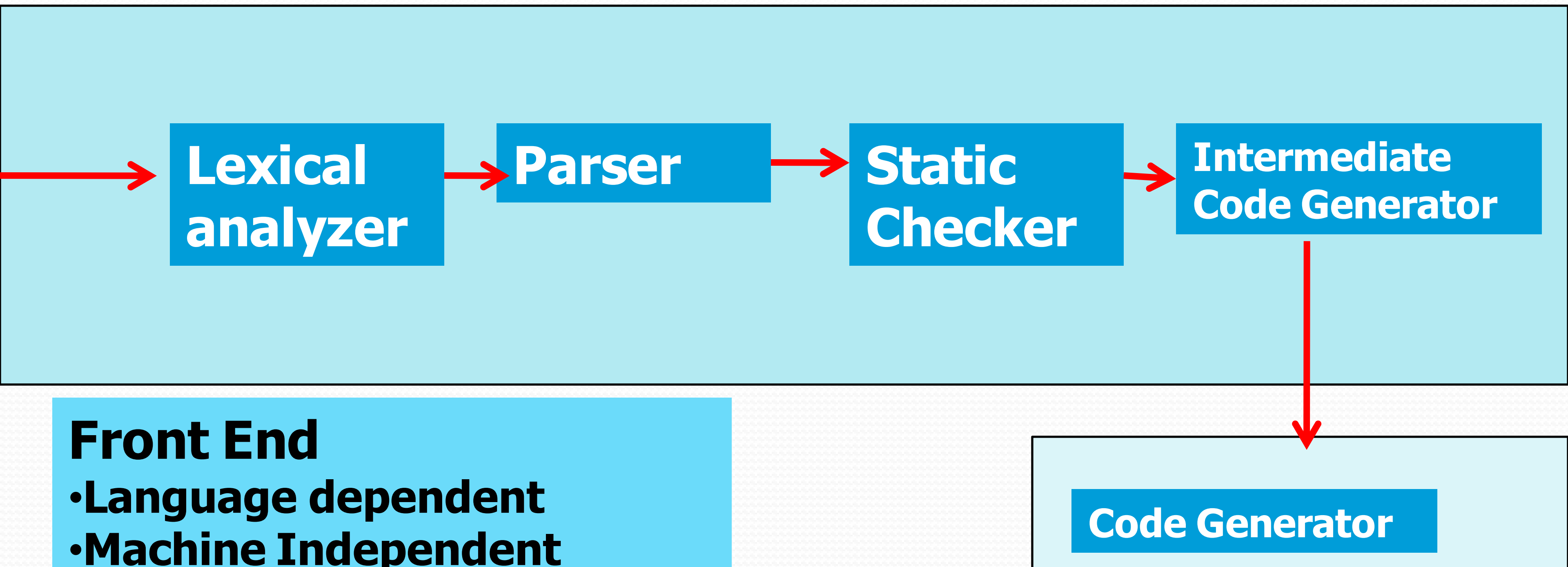
Intermediate Code Generation

Compiler

- The objective of a compiler is to analyze a source program and produce target code
- Front end analyzes the source program and generates an intermediate code
- Back end takes the Intermediate code as input and generates the target code

Language Vs. Target machine

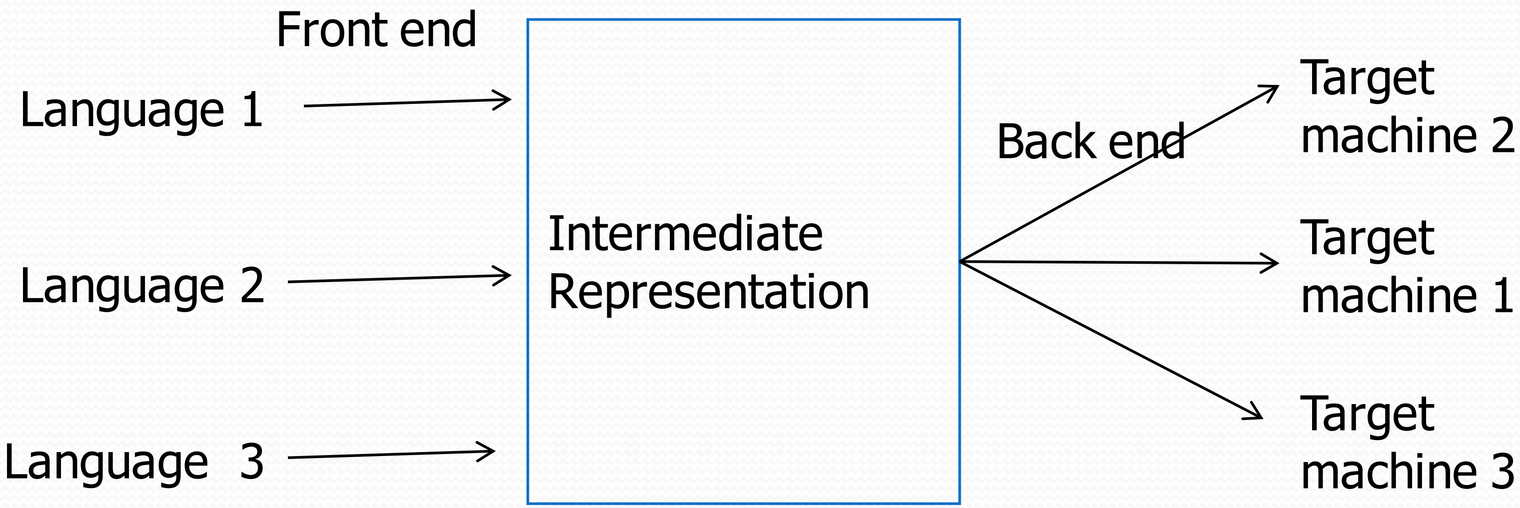


Front End
- Language dependent
- Machine Independent
- Input : source code
- Output: Intermediate Code

Back End
- Language independent
- Machine dependent
- Input: Intermediate Code
- Output : Target code

Intermediate Language

- IL is the language of an abstract machine
- Ease in retargeting to a different machine
- Optimization of the code at intermediate level



Intermediate Representation

- The intermediate representation is a machine- and language-independent version of the original source code.
- Advantages
  - increased abstraction,
  - Cleaner separation between the front and back ends,
  - adds possibilities for retargeting/ cross-compilation.
  - support advanced compiler optimizations and most optimization is done on this form of the code.

Intermediate Languages of GCC

- GENERIC Very high level. Generated by most front ends
- GIMPLE A simplified GENERIC in Static Single Assignment (SSA) form
- RTL Register Transfer Language. A low level representation used in the back ends

Sample C program and generated IR

```
#include<stdio.h>
main()
{
	int a,d;
	int b,c;
	a=3;
	b=4;
	d=1;
	c=a+b+d;
	printf("c=%d\n",c);
}
```

gcc -fdump-tree-gimple-raw test1.c
File test1.c.t03.gimple is created

Partial only

```
@45     function_decl    name: @49      type: @48      srcp: stdio.h:327
                         body: undefined               link: extern
@46     addr_expr        type: @50      op 0: @51
@47     tree_list        valu: @36
@48     function_type    size: @6       algn: 8        retn: @7
                         prms: @52
@49     identifier_node  strg: printf   lngt: 6
@50     pointer_type     size: @13      algn: 32       ptd : @53
@51     array_ref        type: @53      op 0: @54      op 1: @55
                         op 2: @55      op 3: @56
@52     tree_list        valu: @57
@53     integer_type     name: @58      size: @6       algn: 8
                         prec: 8        sign: signed   min : @59
                         max : @60
@54     string_cst       type: @61      strg: c=%d
   lngt: 6
@55     integer_cst      type: @62      low : 0
@56     integer_cst      type: @62      low : 1
@57     pointer_type     qual:   r      unql: @63      size: @13
                         algn: 32       ptd : @64
```

$gcc –S test1.c

```
.file   "test1.c"
        .section        .rodata
.LC0:
        .string "c=%d\n"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $36, %esp
        movl    $3, -20(%ebp)
        movl    $4, -12(%ebp)
        movl    $1, -16(%ebp)
```

```
movl    -12(%ebp), %eax
        addl    -20(%ebp), %eax
        addl    -16(%ebp), %eax
        movl    %eax, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        addl    $36, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.1.2 20061115
(prerelease) (Debian 4.1.1-21)"
        .section        .note.GNU-
stack,"",@progbits
```

```c
int main()
{
        int a = 10;
        int b = 5;
        int c;
        c = a+b;
        return(c);
}
```

C2suif
(Suif IR generator
module.)

```
PROC  @t5:"main”
   Proc   BasicSymbolTable: t9:
     Explicit Super Scope:   @t8:symtab
     var_sym:t10:  "a" with t:@t6:q.(@t3:i.32) addrTaken:0

     var_sym:t11:  "b" with t:@t6:q.(@t3:i.32) addrTaken:0

     var_sym:t12:  "c" with t:@t6:q.(@t3:i.32) addrTaken:0
…
   Body:
     ASSIGN <dst> = <src>
       ["line": 1 "test.c"]
           <dst>:@t10:(@t6:q.(@t3:i.32)) "a"
         <src>:  (@t3:i.32) 10
     ASSIGN <dst> = <src>
       ["line": 1 "test.c"]
           <dst>:@t11:(@t6:q.(@t3:i.32)) "b"
         <src>:  (@t3:i.32) 5
     ASSIGN <dst> = <src>
       ["line": 6 "test.c"]
           <dst>:@t12:(@t6:q.(@t3:i.32)) "c"
         <src>:  (@t3:i.32) <e1> add <e2>
             <e1>:(@t3:i.32) @t10:(@t6:q.(@t3:i.32)) "a"
           <e2>:  (@t3:i.32) @t11:(@t6:q.(@t3:i.32)) "b"
     RET <retval>
       ["line": 7 "test.c"]
         <retval>:(@t3:i.32) @t12:(@t6:q.(@t3:i.32)) "c"

   PROC END   @t5:"main"
```

Stanford University Intermediate Format

```
PROC  @t5:"main"
   Proc   BasicSymbolTable: t9:
     Explicit Super Scope:   @t8:symtab
     var_sym:t10:  "a" with t:@t6:q.(@t3:i.32)
addrTaken:0
     var_sym:t11:  "b" with t:@t6:q.(@t3:i.32)
addrTaken:0
     var_sym:t12:  "c" with t:@t6:q.(@t3:i.32)
addrTaken:0
…
   Body:
     ASSIGN <dst> = <src>
       ["line": 1 "test.c"]
          <dst>:@t10:(@t6:q.(@t3:i.32)) "a"
        <src>:  (@t3:i.32) 10
     ASSIGN <dst> = <src>
       ["line": 1 "test.c"]
          <dst>:@t11:(@t6:q.(@t3:i.32)) "b"
        <src>:  (@t3:i.32) 5
     ASSIGN <dst> = <src>
       ["line": 6 "test.c"]
          <dst>:@t12:(@t6:q.(@t3:i.32)) "c"
        <src>:  (@t3:i.32) <e1> add <e2>
            <e1>:(@t3:i.32) @t10:(@t6:q.(@t3:i.32)) "a"
          <e2>:  (@t3:i.32) @t11:(@t6:q.(@t3:i.32)) "b"
     RET <retval>
       ["line": 7 "test.c"]
        <retval>:(@t3:i.32) @t12:(@t6:q.(@t3:i.32)) "c"

PROC END   @t5:"main"
```

Procedure declaration : main

Variable declaration (a,b,c)

Procedure body .

Assign a = 10

Assign b = 5

Assign c = a + b

Return value (c)

Procedure End : main

SUIF- a compiler infrastructure

- Fortran and C front ends -> SUIF
- SUIF -> Fortran and C
- Data dependence analysis
- A basic parallelizer
- A loop-level locality optimizer
- A visual SUIF code browser

What is an Intermediate code instruction?

- Any representation of the small units of HLL instruction, which can easily be translated into target code
- Example: Consider a HLL instruction

  x=a+b*c;

- Considering the limitation of the target machine to execute one operation at a time, this must be broken down to instructions

  t=b*c;

  x=a+t;

Example target machine

- Available instructions
  - Load /store
  - Add
  - Mul etc.
- Addressing modes
  - Name x refers to the location holding value of x
  - a(R) to fetch the contents(a+contents(R))
  - ....

t=b*c;
x=a+t;

Intermediate code

Load R0,b
Load R1,c
Mul R2,R1,R0
Store t,R2
Load R0,t
Load R1,a
Add R2,R0,R1
Store R2, x

Target code

Can be optimized

- **In terms of cost of instructions**
- **In terms of less no of registers used**

High level IR, Mid level IR and Low level IR

Original

```
float a[10][20];
x= a[i][j+2];
```

High IR

```
t1 = a[i, j+2]
```

Mid IR

```
t1 = j + 2
t2 = i * 20
t3 = t1 + t2
t4 = 4 * t3
t5 = addr a
t6 = t5 + t4
t7 = *t6
```

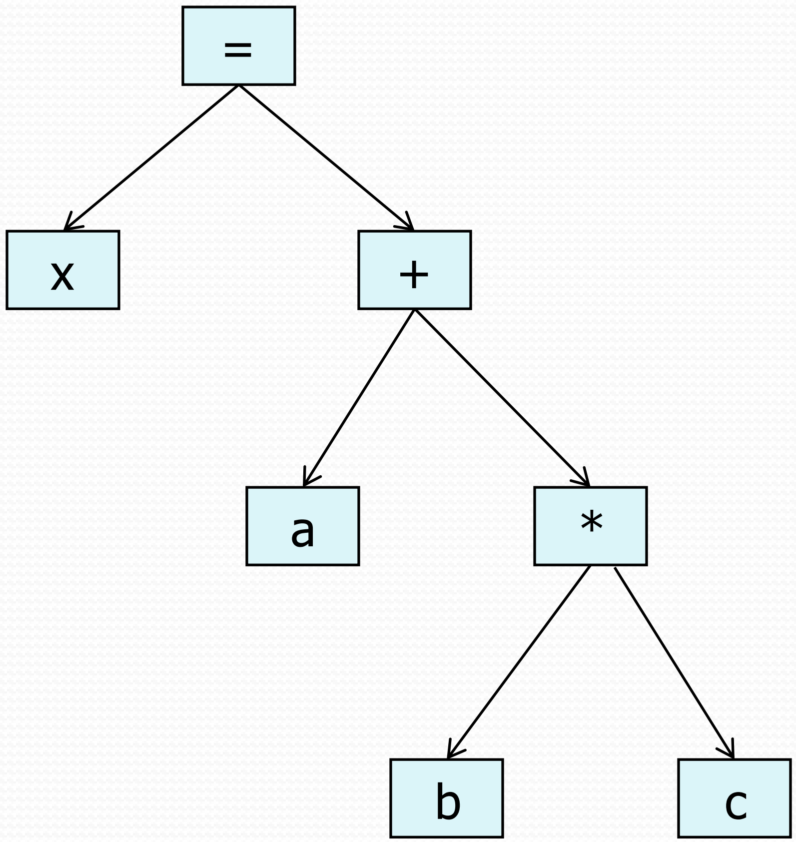
Low IR

```
r1 = [fp - 4]
r2 = [r1 + 2]
r3 = [fp - 8]
r4 = r3 * 20
r5 = r4 + r2
r6 = 4 * r5
r7 = fp - 216
f1 = [r7 + r6]
```

Abstract syntax tree for x=a+b*c;

t=b*c;
x=a+t;

Intermediate code



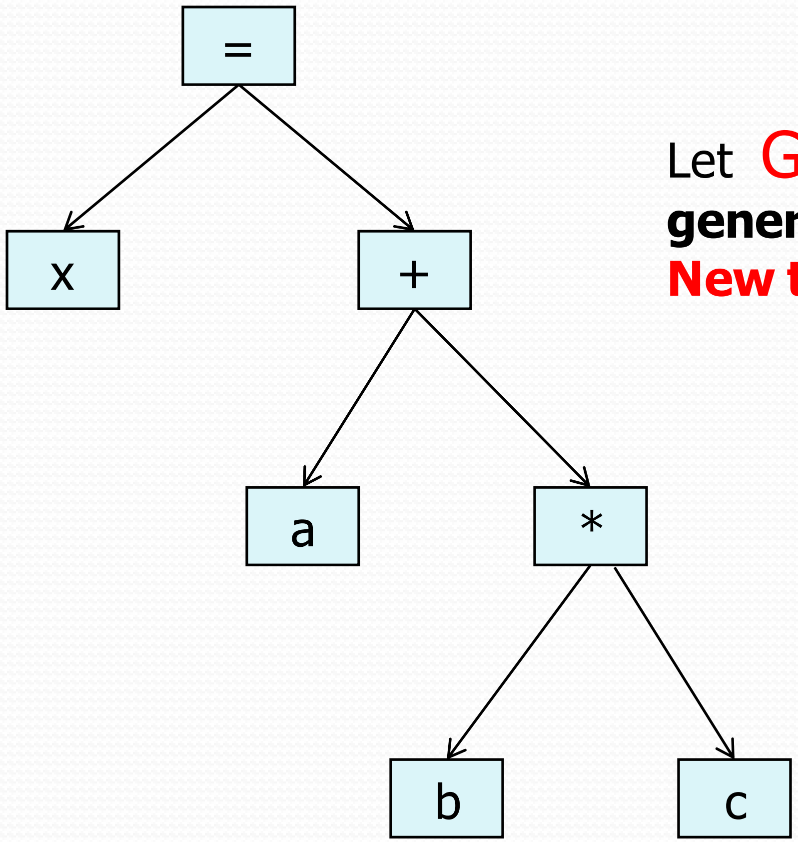
Does this AST give any information?

1. We need to associate semantic rules with each node
2. We need to associate appropriate attributes with each node
3. We need to traverse the AST for the purpose of generating information using the defined semantic rules

Abstract syntax tree for x=a+b*c;

t=b*c;
x=a+t;

Intermediate code



Let GenerateCode(op,left,right)
generate
New temp = left.lexeme op right.lexeme

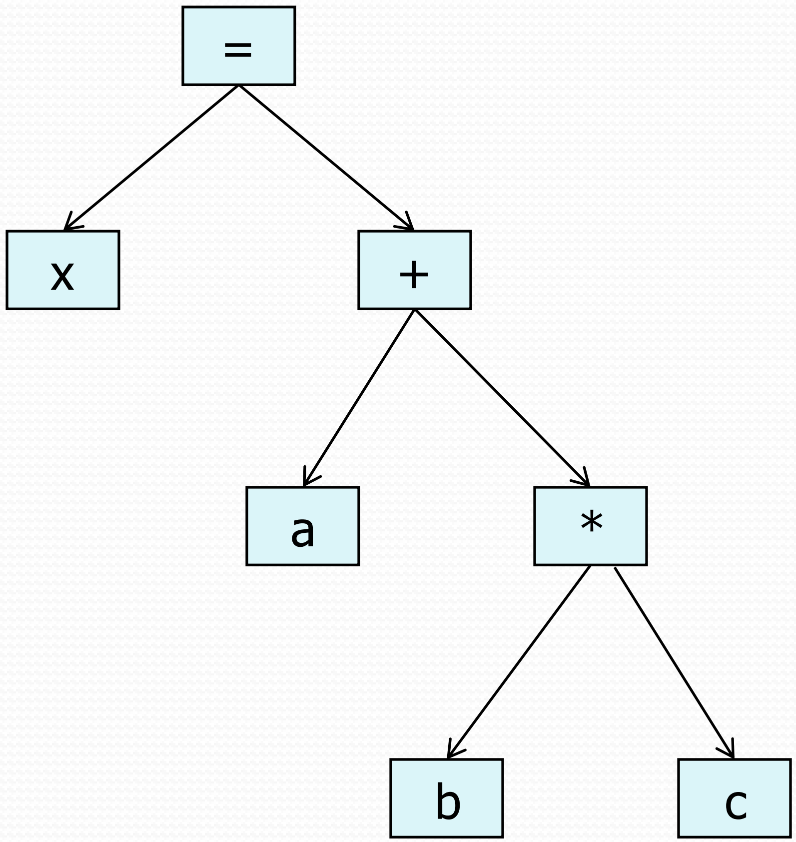
Question: How would you prefer to traverse so that the desired code is generated?

- Preorder?
- Inorder?
- Postorder?

Example : Intermediate code generation using AST

t=b*c;
x=a+t;

Intermediate code



t1=b * c

a=b * c

lexeme	token	type	width	offset
x	TK_ID	Int	4	0
a	TK_ID	Int	4	4
b	TK_ID	Int	4	8
c	TK_ID	Int	4	12

All names are just the pointers to the corresponding symbol table

Three address code

- It is linearized representation of an AST
- At most one operator on the right side of an instruction is permissible
- Temporary names can be generated to store temporary results
- At most three addresses (any combination of the following) are permitted
  - **Names of variables (representing memory locations)**
  - **A constant**
  - **Names of temporaries (may be mapped to registers or to memory)**

Symbolic three address instructions (Intermediate Language used in the text book)

1. x = y op z //op is binary operator
2. x = op y //op is unary operator
3. x = y //copy instruction
4. goto L //unconditional jump to L
5. if x goto L //conditional jump
6. if x relop y goto L //conditional jump
7. param x //parameters in function
8. call p,n //function p with n parameters
9. return y //y being the return value

Symbolic three address instructions (Intermediate Language used in the text book)

10. x=a[i] //indexed copy
11. b[i]= y //indexed copy
12. x=&y //address and pointer
13. x=*y //assignments
14. *x= y

Example :

Create intermediate code for the following C like code

x=a[i]+b[i];

$t1= a[i]$

$t2= b[i]$

$x= t1+t2$

The code generation will require exact offsets of the elements

Equivalent target code may be as follows

Load R0,i
Mul R1,i,4
Load R2,a(R1) **//contents(a+contents(R1))**
Load R3,b(R1)
Add R2,R2,R3

Class assignment: Interpret the symbol table, AST and call stack for accessing the data from the memory locations

Syntax-Directed Translation for generating 3-address code.

- Attributes
  - E.addr: the name that will hold the value of E
  - E.code: holds the three address code statements that evaluate E
- Use functions
  - Newtemp()
  - gen()

Translation of Expressions

1) $S \rightarrow id = E$ $\{ S.code = E.code \| gen(id.addr \text{ '=' } E.addr \text{ ';'}) \}$

2) $E \rightarrow E_1 + E_2$ $\{E.addr = newtemp()$

$E.code = E_1.code \| E_2.code \|$

$\| gen(E.addr \text{'='} E_1.addr \text{ '+' } E_2.addr) \}$

3) $E \rightarrow E_1 * E_2$ $\{E.addr = newtemp()$

$E.code = E_1.code \| E_2.code \|$

$\| gen(E.addr \text{'='} E_1.addr \text{'*'} E_2.addr) \}$

4) $E \rightarrow - E_1$ $\{E.addr = newtemp()$

$E.code = E_1.code \|$

$\| gen(E.addr \text{ '=' 'uminus' } E_1.addr) \}$

5) $E \rightarrow (E_1)$ $\{E.addr = E_1.addr ; E.code = E_1.code\}$

6) $E \rightarrow id$ $\{E.addr = id.lexeme$

$E.code = \text{' '} \}$

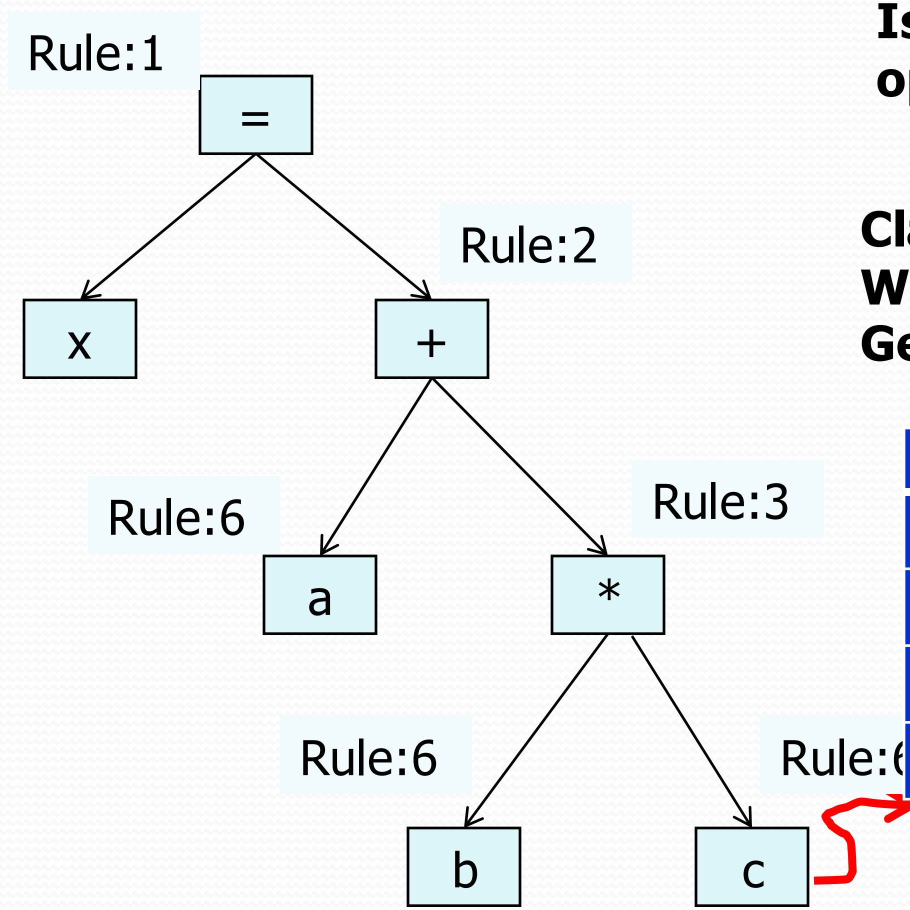
NOTE: E.addr represents the name of the value holder e.g. a,b,c,t1,t2 etc..

Recall following Example : create Intermediate code generation using AST and SDT (previous slide)

t=b*c;
x=a+t;

Intermediate code

Is the intermediate code optimized?



Class Assignment:
Work out the SDT scheme
Generate 3 address code

lexeme	token	type	width	offset
x	TK_ID	Int	4	0
a	TK_ID	Int	4	4
b	TK_ID	Int	4	8
c	TK_ID	Int	4	12

What are the attributes?
Are they synthesized or inherited?
What is the evaluation order?

Implementations of 3-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		
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Translation of control flow statements

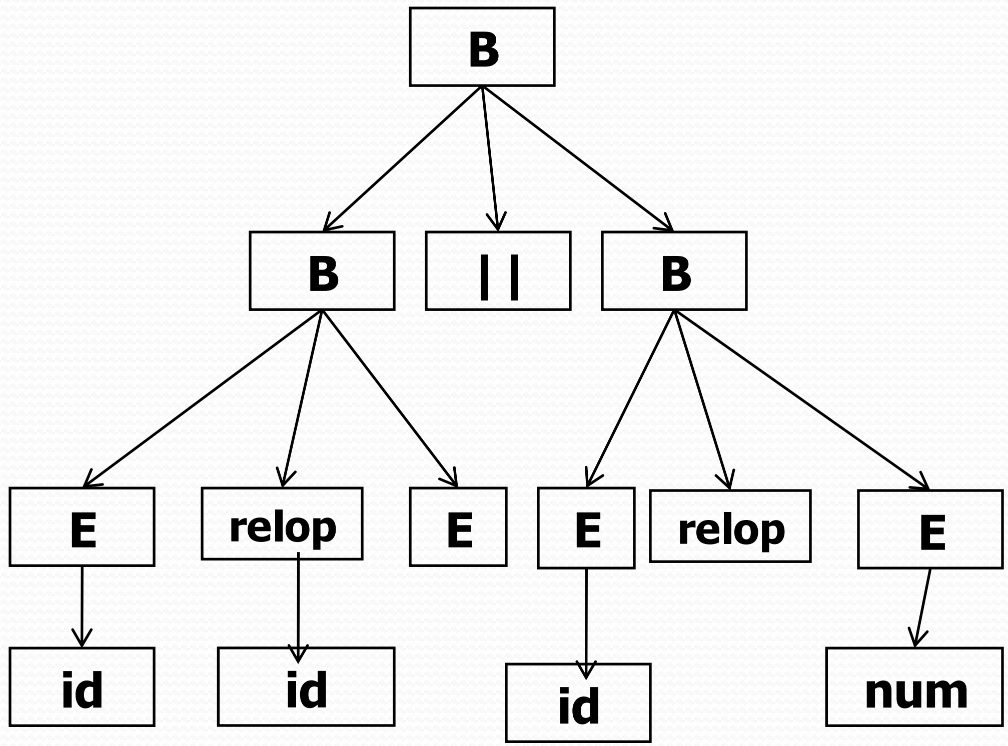
- Example statements
- S→if (B) S
  - **If ((x<y) || (x>30)) z=x+y;**
- S→if (B) S else S
  - **If ((x<y) || (x>30)) z=x+y; else z=x-y;**

Boolean grammar

1. B → B || B
2. B → B && B
3. B → ~B
4. B → E relop E
5. E → id
6. E → num

Parse the following input Boolean expression

x<y || x>30



Statement grammar with control flow instructions

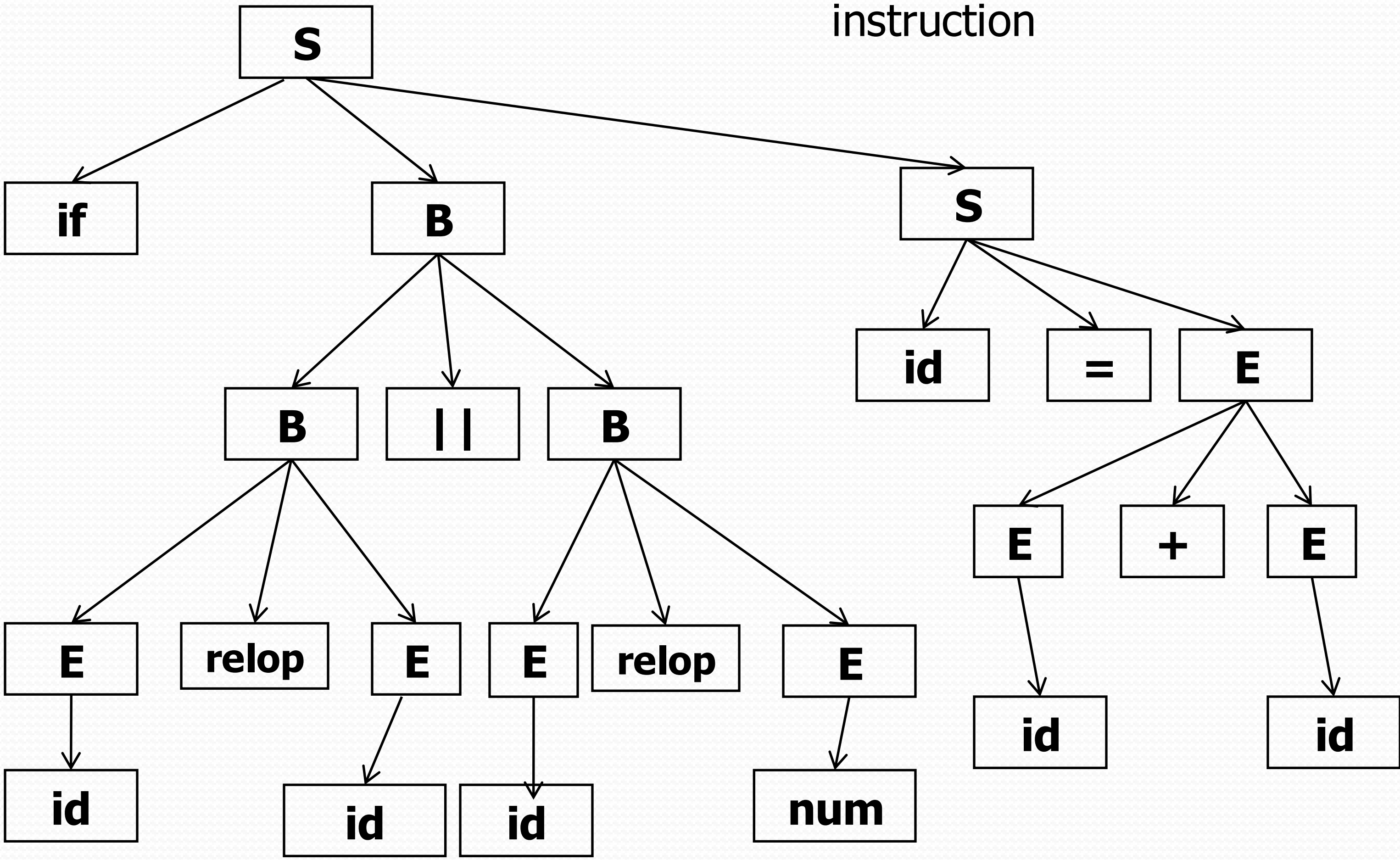
1. S → if (B) S
2. S → if (B) S else S
3. S → while(B) S
4. S → S S
5. S → id = E

+

1. B → B || B
2. B → B && B
3. B → ~B
4. B → E relop E
5. E → id
6. E → num

if x<y || x>30 z=x+y

Think about the semantic rules that generate 3 address code for this HLL instruction



Expected intermediate code for HLL instruction

if **x<y || x>30 z=x+y**

```
          if  x < y goto L1
          if  x > 30  goto L1
          goto L2
L1:       z=x+y
L2:
```

Expected intermediate code for HLL instruction

if x<y || x>30 z=x+y else z=x-y

```
        if  x < y goto L1
        if  x > 30  goto L1
        goto L2
L1:     z=x+y
L2:     z=x-y
```

Expected intermediate code for HLL instruction

if x<y || x>30 z=x+y

if **x < y** goto L1

if **x > 30** goto L1

goto L2

L1: z=x+y

L2:

Analyse the rule
$B \rightarrow B_1 \mid\mid B_2$

Associate a label say
L1 when B_1 is true
Or when B_2 is true

Expected intermediate code for HLL instruction

if $x<y \;||\; x>30 \;\; z=x+y$

if **x < y** goto L1

if **x > 30** goto L1

goto L2

L1: z=x+y

L2: z=x-y

B is a non terminal

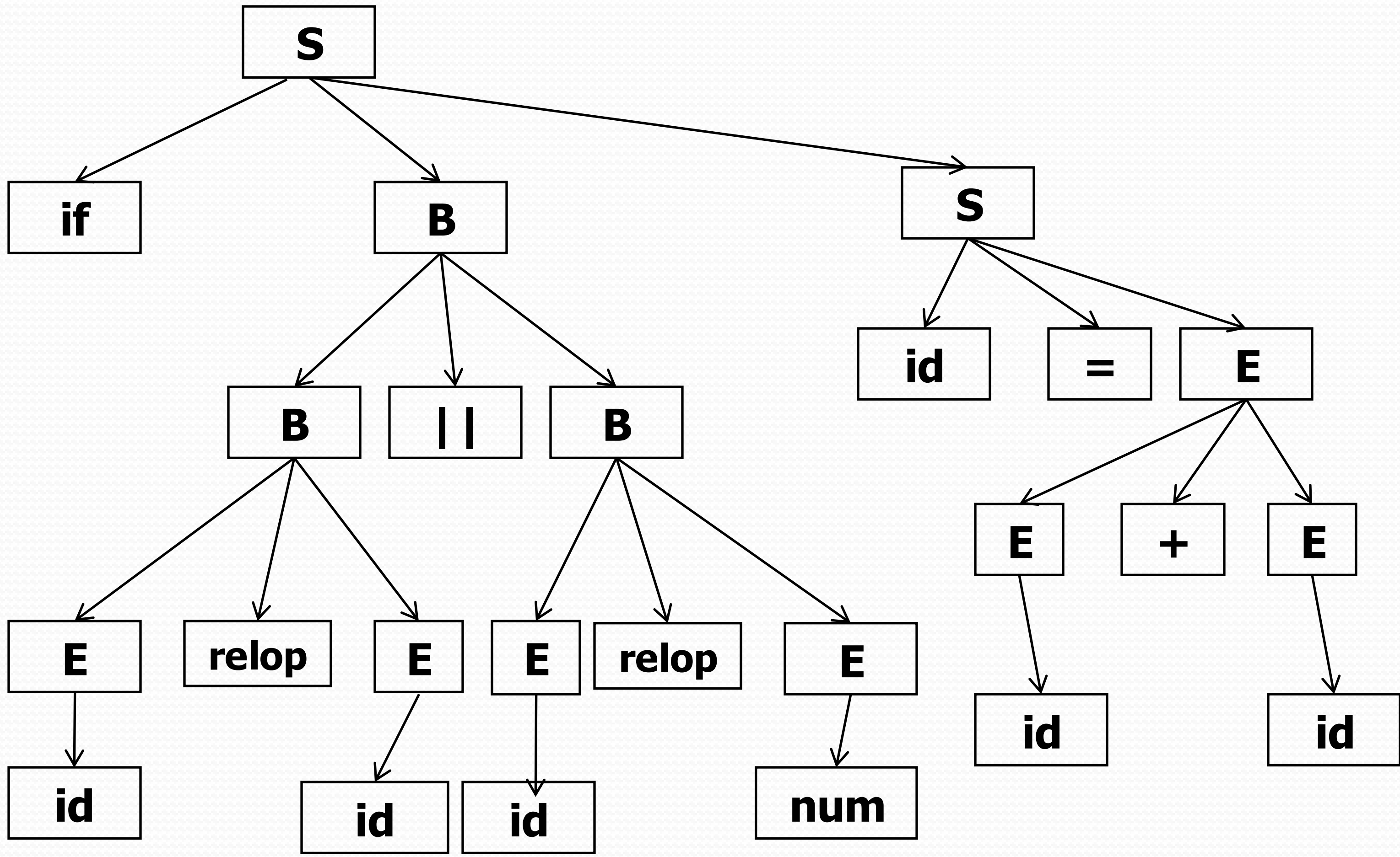
Associate attributes true and false such that B_1.true = L1
or B_2.true = L1
or B_2.false = L2
B_1.false = ????? think

Generating three address code for $S \rightarrow$ if(B) S_1

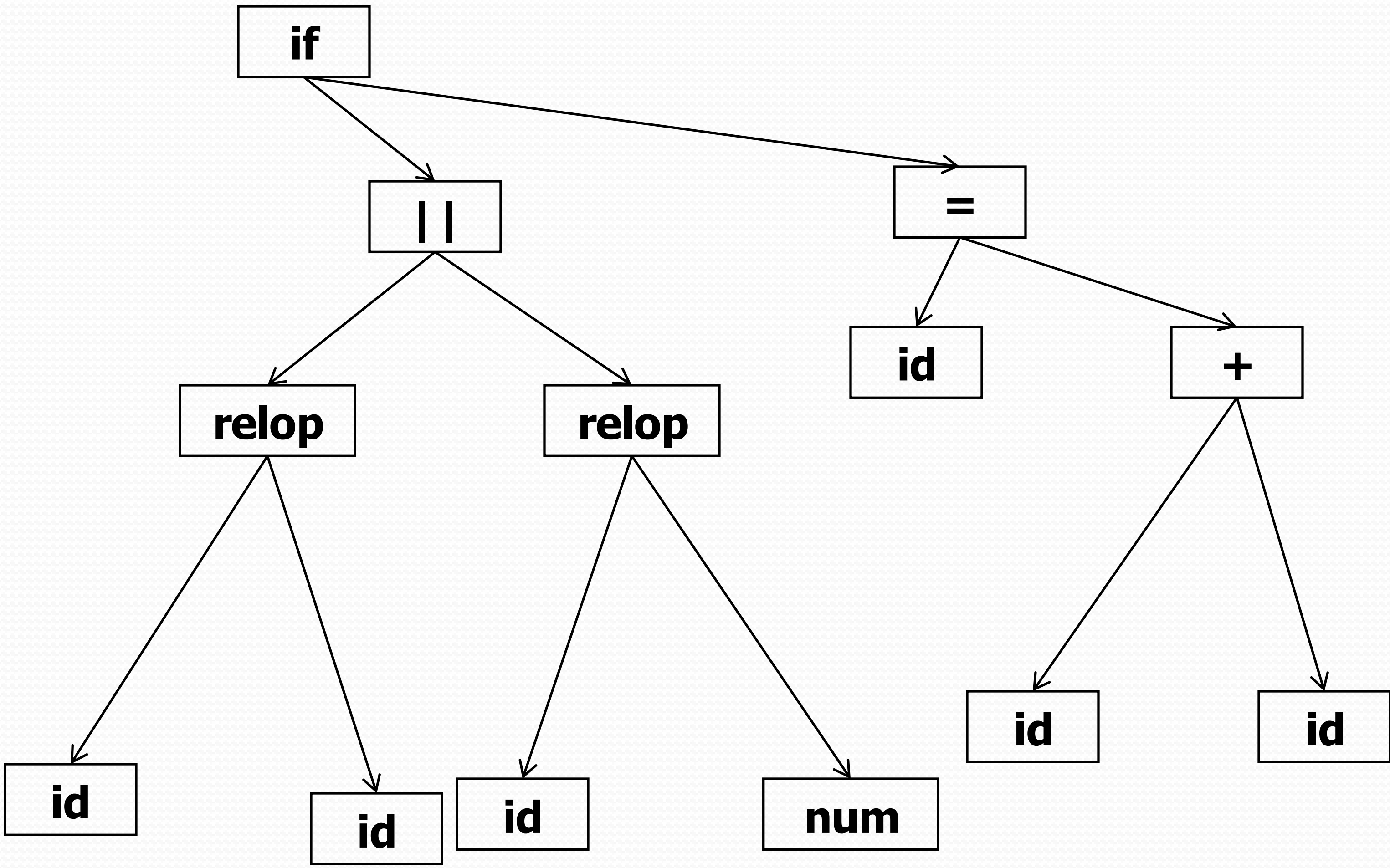
$S \rightarrow$ if (B) S_1	B.true=newlabel() B.false=S_1.next=S.next S.code=B.code || label(B.true) || S_1.code
$B \rightarrow B_1 \|\| B_2$	B_1.true = B.true B_1.false = newlabel() B_2.true = B.true B_2.false= B.false B.code = B_1.code || label(B_1.false) || B_2.code
$B \rightarrow E_1$ relop E_2	B.Code = E_1.code || E_2.code || gen('if' E_1.addr relop.lexeme E_2.addr 'goto' B.true) || gen('goto' B.false)
$B \rightarrow$ true	B.code = gen('goto' B.true)
$B \rightarrow$ false	B.code = gen('goto' B.false)

Parse tree for

if x<y || x>30 z=x+y



Abstract Syntax tree for if x<y || x>30 z=x+y



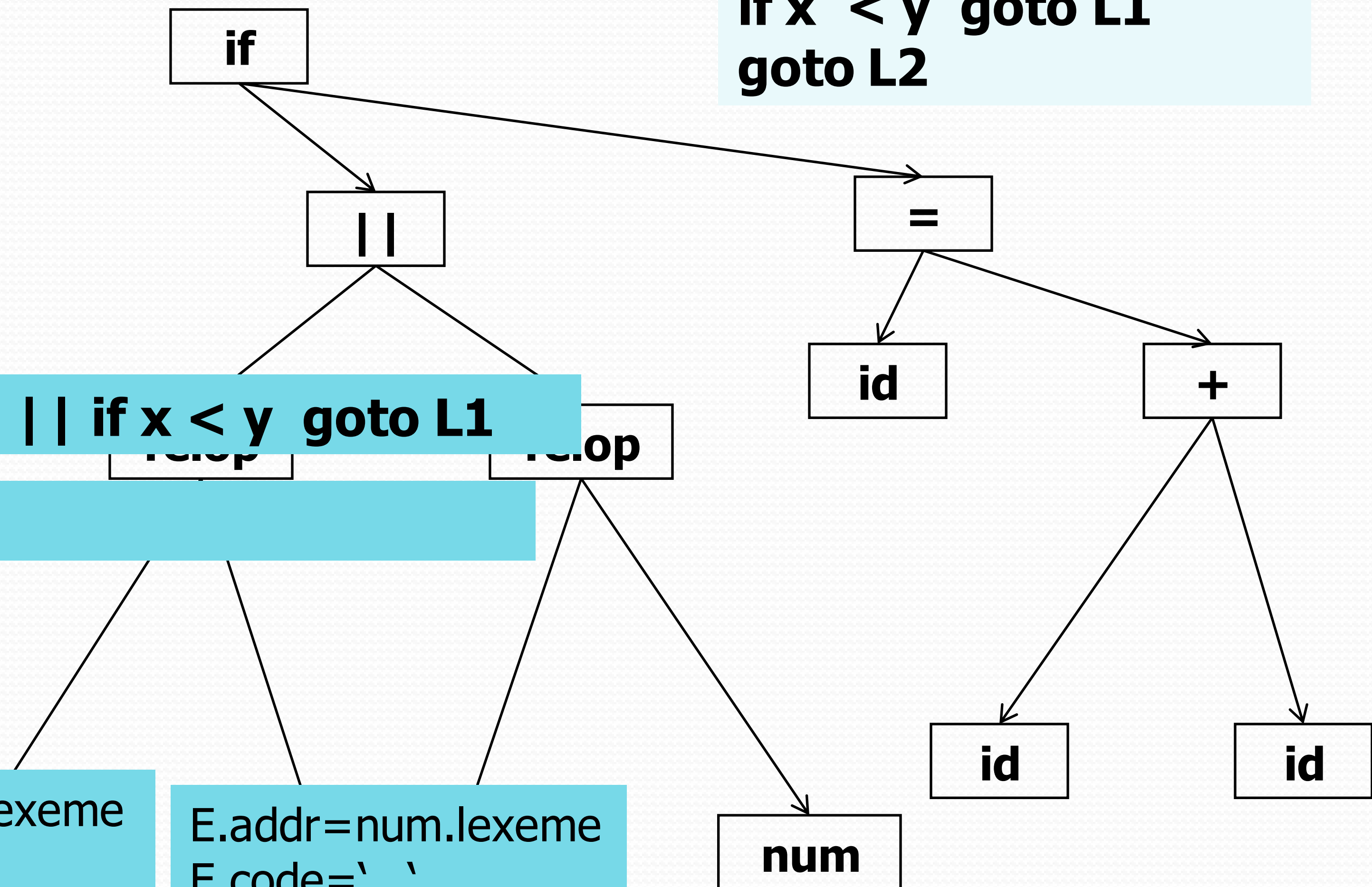
$B \rightarrow E_1$ relop E_2

B.Code = E_1.code || E_2.code ||
gen('if' E_1.addr relop.lexeme E_2.addr 'goto' B.true)
|| gen('goto' B.false)

Final code

if x < y goto L1
goto L2

Input
x < y



E.code=' ' || ' ' || if x < y goto L1

|| goto L2

E.addr=id.lexeme
E.code=' '

E.addr=num.lexeme
E.code=' '

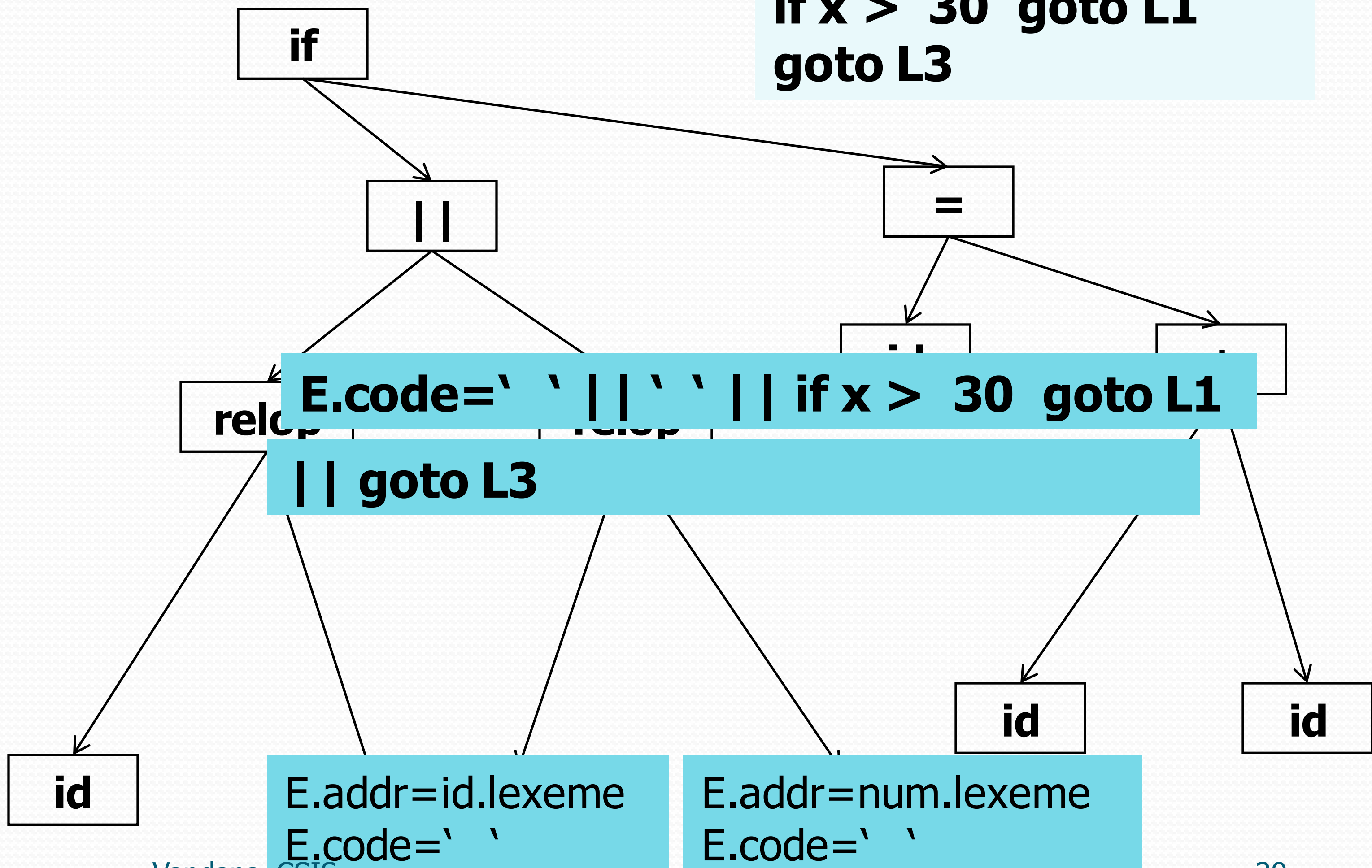
B → E_1 relop E_2

B.Code = E_1.code || E_2.code ||
gen('if' E_1.addr relop.lexeme E_2.addr 'goto' B.true)
|| gen('goto' B.false)

Final code

if x > 30 goto L1
goto L3

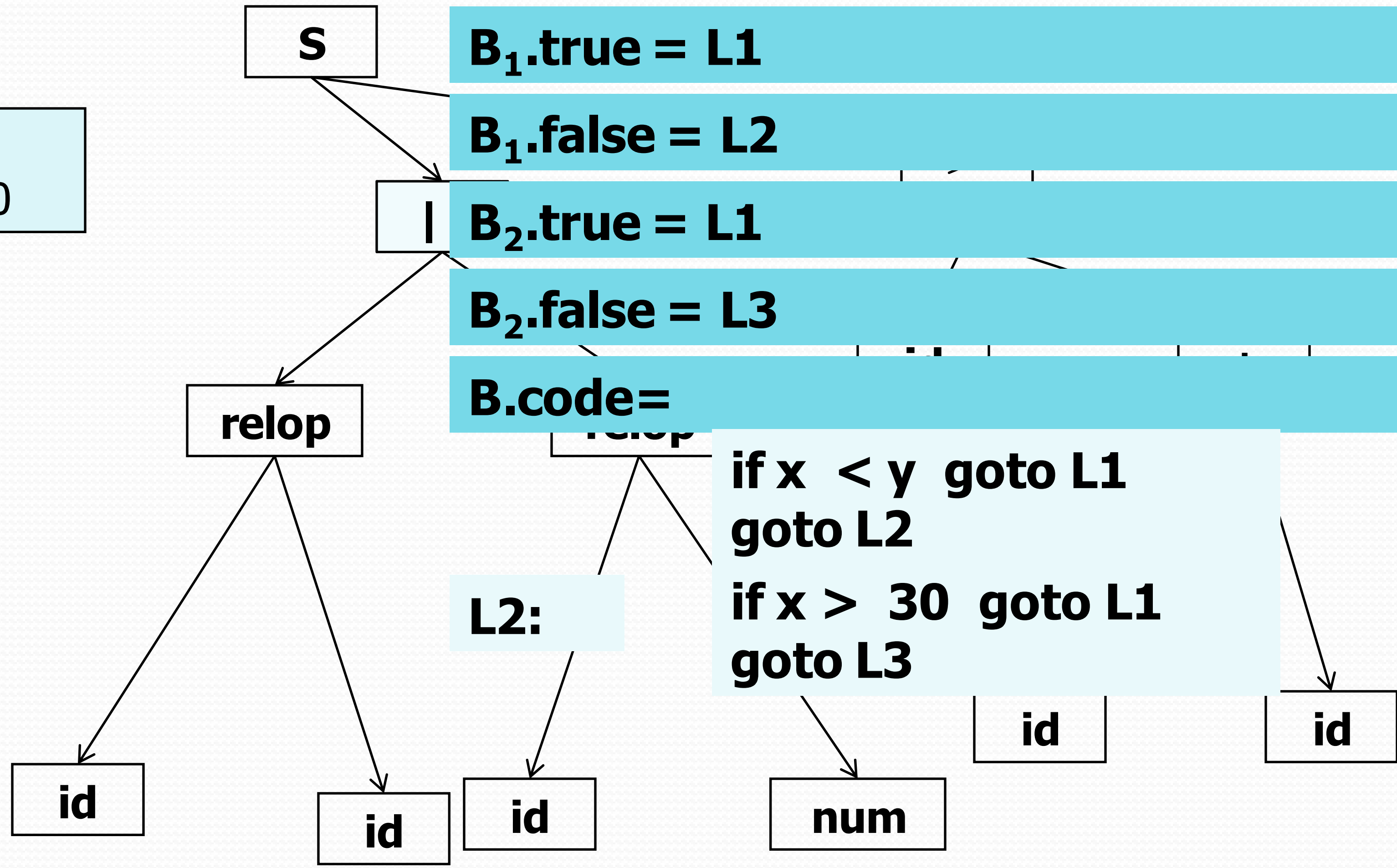
Input
x > 30



$B \rightarrow B_1 \mid\mid B_2$

$B_1.true = B.true$
$B_1.false = newlabel()$
$B_2.true = B.true$
$B_2.false = B.false$
$B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

Input
$x < y \ \mid\mid x > 30$



B_1.true = L1

B_1.false = L2

B_2.true = L1

B_2.false = L3

B.code=

if x < y goto L1
goto L2

L2:

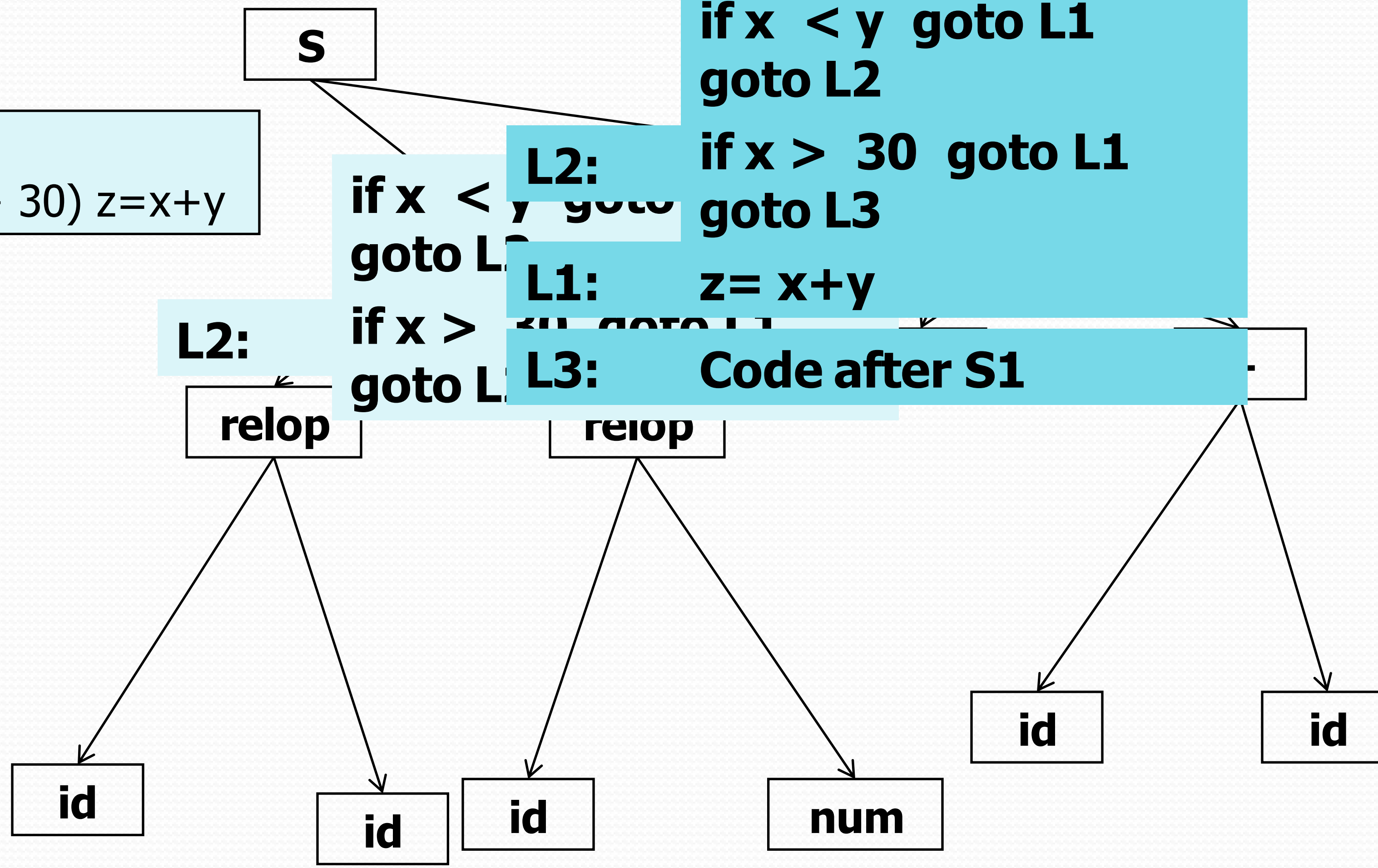
if x > 30 goto L1
goto L3

S → if (B) S_1

B.true=newlabel()
B.false=S_1.next=S.next
S.code=B.code || label(B.true) || S_1.code

Already seen (inherited from here)
B.true=L1
B.false=L3

Input
if (x < y || x > 30) z=x+y



if x < y goto L1
goto L2
L2: if x > 30 goto L1
goto L3
L1: z= x+y
L3: Code after S1

Generated Intermediate code using semantic rule

Input
if (x < y || x > 30) z=x+y

	if x < y goto L1 **goto L2**
L2:	**if x > 30 goto L1** **goto L3**
L1:	**z= x+y**
L3:	**Code after S1**

Generated Intermediate code using semantic rule

Input
if (x < y || x > 30) z=x+y + v

	if x < y goto L1
	goto L2
L2:	**if x > 30 goto L1**
	goto L3
L1:	**t1= x + y**
	t2= t1 + v
	z = t2
L3:	**Code after S1**

Redundant copy instruction

Data structure for internal representation of the generated IR

- Quadruples
- Triples

Class assignment

- Design the semantic rules for generating the Intermediate code for a C like input

```
if ((x<=5) && !(c==10))
    y=z+c-x;
```

- How many passes of the AST are required to generate the intermediate code