CS536

Control Flow

A Sahu

CSE, IIT Guwahati

Outline

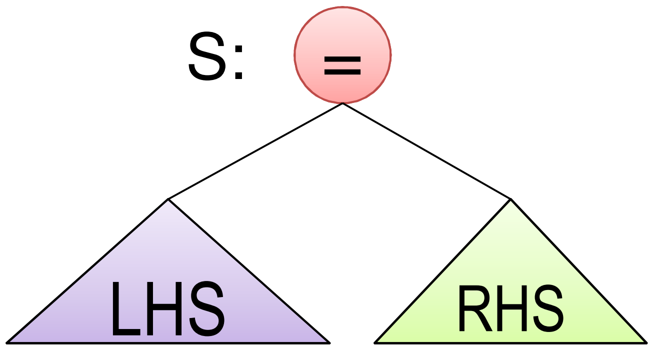
- Control Flow: Flow Graph
- Short-Circuit Code
- Flow of Control Statement
- Back Patching
- Translation of Switch Statement
- Translation of function call

Control Flow

Boolean expressions are often used to:

- *Alter the flow of control.*
- *Compute logical values.*

Assignments



Code structure:

evaluate LHS

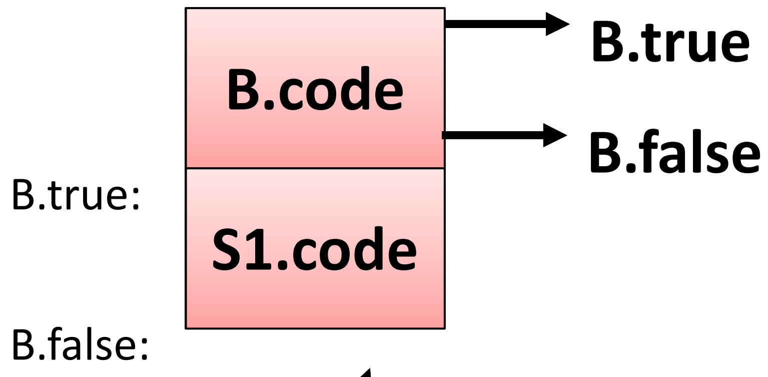
evaluate RHS

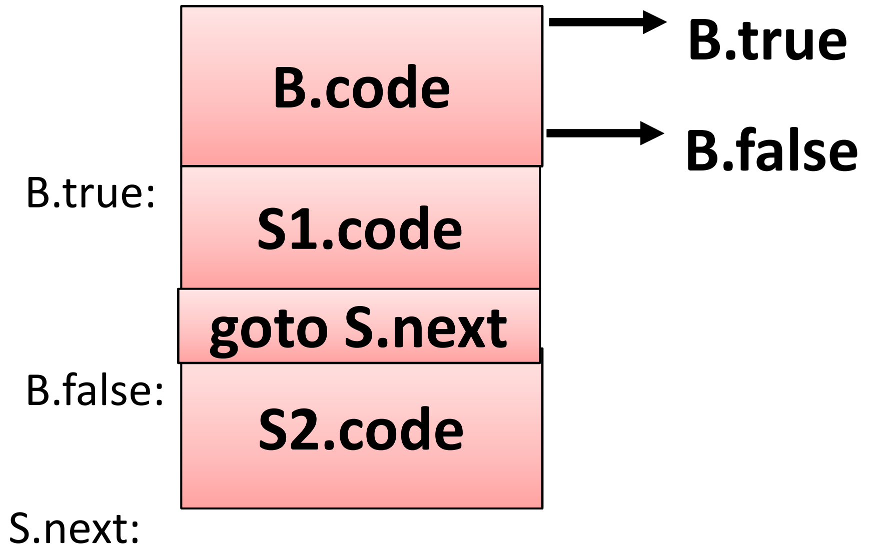
copy value of RHS into LHS

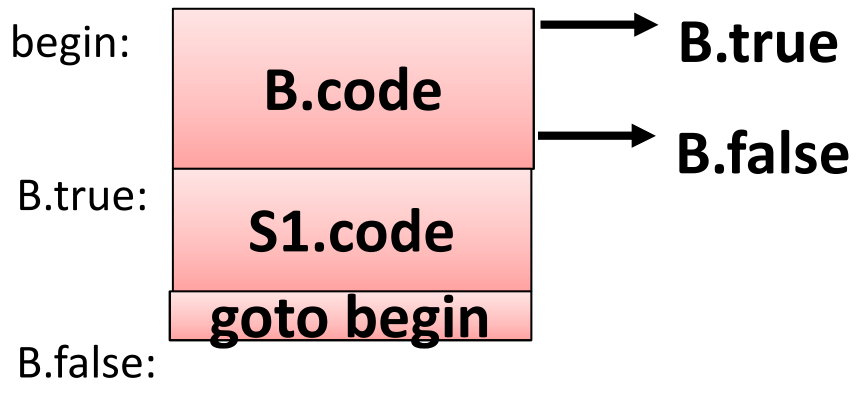
```
codeGen_stmt(S):
/* base case: S.nodetype = ‘S’ */
codeGen_expr(LHS);
codeGen_expr(RHS);
S.code = LHS.code
             || RHS.code
             || newinstr(ASSG,
                          LHS.place,
                          RHS.place) ;
```

Flow-of-Control Statements

S→ if (B) S1
S→ if (B) S1 else S2
S→ while (B) S1







Syntax-directed definition

Production	Semantic Rules

P→S	S.next=newlabel()
	P.code= S.code ||label(S.next)

S→assign	S.code=assign.code

S→ if (B) S1	B.true=newlabel();
	B.false=S1.next=S.next
	S.code=B.code||Label(B.true)||S1.code

S→ if (B) S1 else S2	B.true=newlabel();
	B.false=newlabel();
	S1.next=S2.next=S.next
	S.code=B.code||lebel(B.true)||S1.code
	||gen('goto' S.next) ||label(B.false)
	||S2.code

Syntax-directed definition

Production	Semantic Rules

S→ while (B) S1	begin=newlabel();
	B.true=newlabel();
	B.false=newlabel();
	S1.next=begin
	S.code=label(begin)||B.code||lebel(B.true)
	||S1.code ||gen('goto' begin)

S→ S1 S2	S1.next=newlabel()
	S2.next=S.next
	S.code= S1.code || label(S1.next)||S2.code

Translation of a simple if-statement

```
if ( x<100 || x>200 && x != y ) x =0
```

```
       if x < 100 goto L2
       goto L3
L3:    if x > 200 goto L4
       goto L1
L4:    if x != y goto L2
       goto L1
L2:    x = 0
L1:
```

Short-Circuit Code

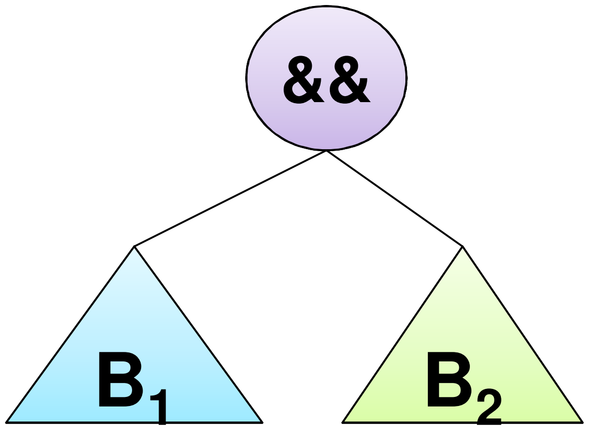
```
if ( x<100 || x>200 && x != y ) x = 0
```

```
        if   x<100           goto L2
        ifFalse x > 200  goto L1
        ifFalse x!=y        goto L1
L2:     x =0 ;
L1:
```

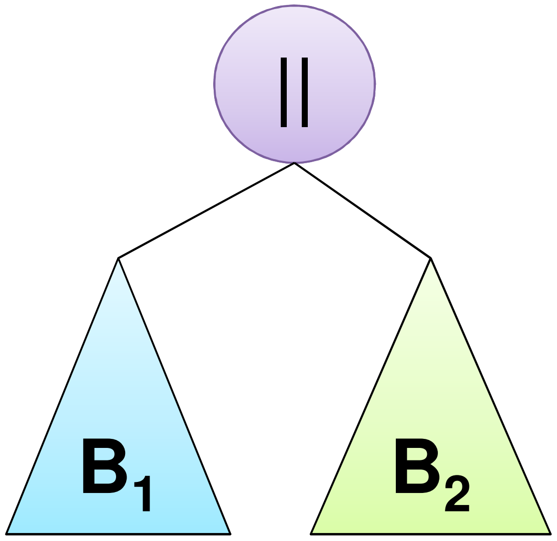
Short-Circuit Code

- Given an expression B1 || B2
  - If we determine B1 is true, we can conclude B is true, without evaluating B2
- Given an expression B1 && B2
  - If we determine B1 is false, we can conclude B is false, without evaluating B2
- If the language permit portion of a Boolean expr to go unevaluated: to optimize
  - **Side effect: if B2 contain a function that change global variable, then unexpected answer be obtained**

Short Circuit Evaluation

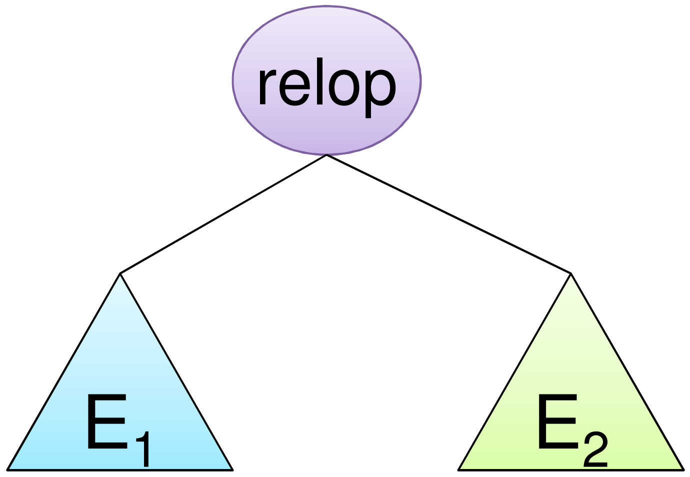


```
codeGen_bool (B, trueDst, falseDst):
/* recursive case 1: B.nodetype == ‘&&’ */
    $L_1$ = newlabel( );
    codeGen_bool($B_1$, $L_1$, falseDst);
    codeGen_bool($B_2$, trueDst, falseDst);
    $B$.code = $B_1$.code $\oplus$ $L_1$ $\oplus$ $B_2$.code;
```



```
codeGen_bool (B, trueDst, falseDst):
/* recursive case 2: B.nodetype == ‘||’ */
    $L_1$ = newlabel( );
    codeGen_bool($B_1$, trueDst, $L_1$);
    codeGen_bool($B_2$, trueDst, falseDst);
    $B$.code = $B_1$.code $\oplus$ $L_1$ $\oplus$ $B_2$.code;
```

Logical Expressions 1



Naïve but Simple Code (TRUE=1, FALSE=0):

```
        t1 = { evaluate $E_1$
        t2 = { evaluate $E_2$
        t3 = 1          /* TRUE */
        if ( t1 relop t2 ) goto L
        t3 = 0          /* FALSE */
    L: …
```

Disadvantage: lots of unnecessary memory references.

Logical Expressions 2

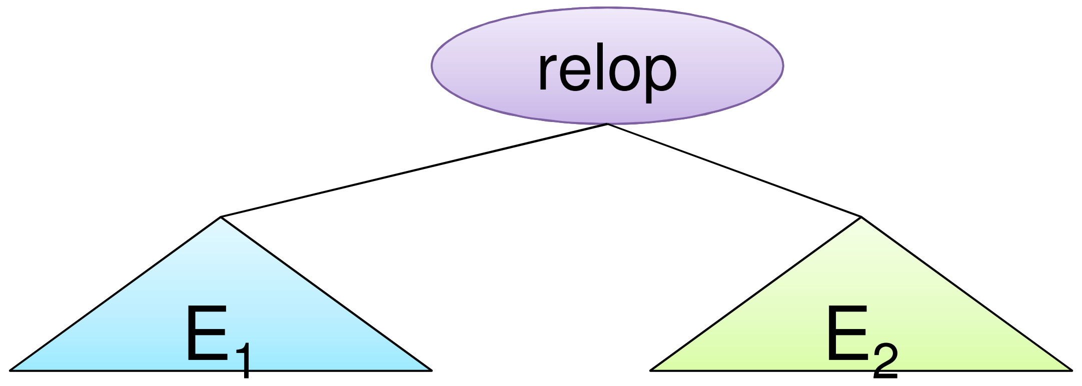
- **Observation**: Logical expressions are used mainly to direct flow of control.
- **Intuition**: "tell" the logical expression where to branch based on its truth value.
  - When generating code for *B,* use two inherited attributes, *trueDst* and *falseDst*. Each is (a pointer to) a *label* instruction.

    E.g.: for a statement `if ( B ) S1 else S2` :

    B.trueDst = start of S_1

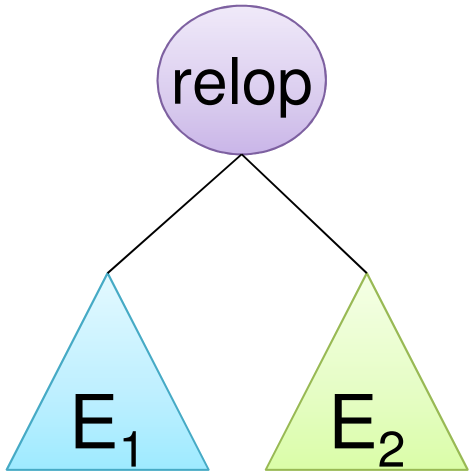
    B.falseDst = start of S_2
  - The code generated for *B* jumps to the appropriate label.

Logical Expressions 2: cont'd



```
codeGen_bool(B, trueDst, falseDst):
/* base case: B.nodetype == relop */
    B.code = E1.code
        ⊕ E2.code
        ⊕ newinstr(relop, E1.place, E2.place, trueDst)
        ⊕ newinstr(GOTO, falseDst, NULL, NULL);
```

Logical Expressions 2: cont'd



```
codeGen_bool(B, trueDst, falseDst):
  /* base case: B.nodetype == relop */
  B.code = E1.code
      ⊕ E2.code
      ⊕ newinstr(relop, E1.place, E2.place, trueDst)
      ⊕ newinstr(GOTO, falseDst, NULL, NULL);
```

Example: $B \Rightarrow$ x+y > 2*z.

Suppose *trueDst* = Lbl1, *falseDst* = Lbl2.

$E_1 \equiv$ x+y, E_1.place = tmp_1, E_1.code $\equiv \langle$ 'tmp_1 = x + y' $\rangle$

$E_2 \equiv$ 2*z, E_2.place = tmp_2, E_2.code $\equiv \langle$ 'tmp_2 = 2 * z' $\rangle$

B.code = E_1.code $\oplus$ E_2.code $\oplus$ 'if (tmp_1 > tmp_2) goto Lbl1' $\oplus$ goto Lbl2

= $\langle$ 'tmp_1 = x + y' , 'tmp_2 = 2 * z', 'if (tmp_1 > tmp_2) goto Lbl1' , goto Lbl2 $\rangle$

Translation of Boolean Expression

Production	Semantic Rules

B→ B1 || B2	B1.true=B.true
	B1.false=newlabel()
	B2.true=B.true
	B2.false=B.false
	B.code=B1.code||label(B1.false)||B2.code

B→ B1 && B2	B1.true=newlabel()
	B1.false=B.false
	B2.true=B.true
	B2.false=B.false
	B.code=B1.code||label(B1.true)||B2.code

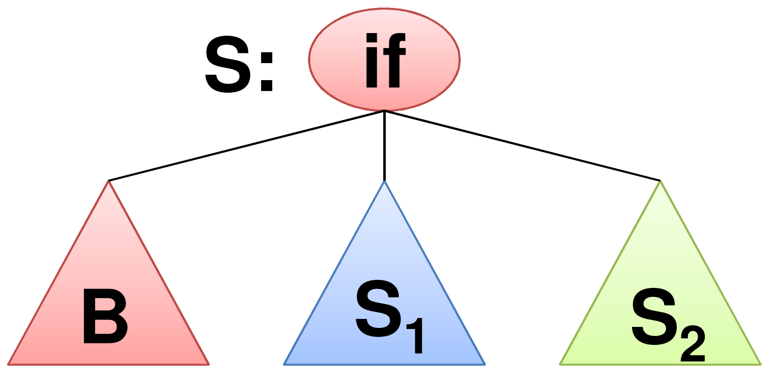
Translation of Boolean Expression

Production	Semantic Rules

B→!B1	B1.true=B.false
	B1.false=B.true
	B.Code =B1.code

B→ E1 rel E2	B.code=E1.code||B2.code ||gen('if E1.addr rel.op E2.addr 'goto' B.true) || gen('goto' B.false)
B→true	B.code=gen('goto' B.true)
B→false	B.code=gen('goto' B.false)

Conditionals : if then else

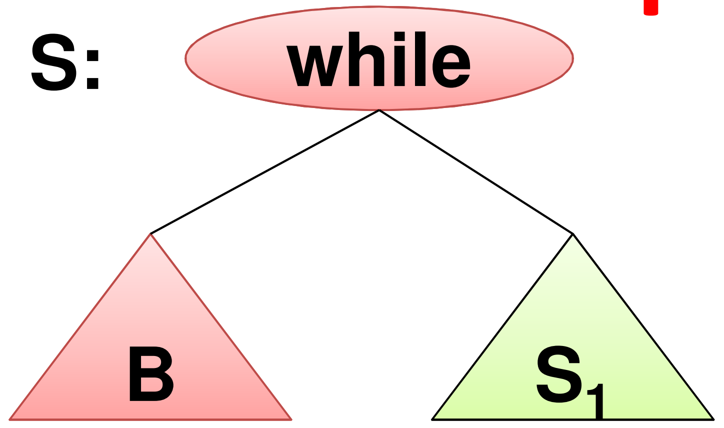


- **Code Structure:**

```
        code to evaluate B
L_then: code for S1
        goto L_after
 L_else: code for S2
 L_after : …
```

```
codeGen_stmt(S):
/* S.nodetype == 'IF' */
   $L_{then}$ = newlabel();   $L_{else}$ = newlabel();
   $L_{after}$ = newlabel();
   codeGen_bool(B, $L_{then}$, $L_{else}$);
   codeGen_stmt($S_1$);
   codeGen_stmt($S_2$);
   S.code = B.code
             ⊕ $L_{then}$
             ⊕ $S_1$.code
             ⊕ newinstr(GOTO, $L_{after}$)
             ⊕ $L_{else}$
             ⊕ $S_2$.code
              ⊕ $L_{after}$ ;
```

Loops : While Loop

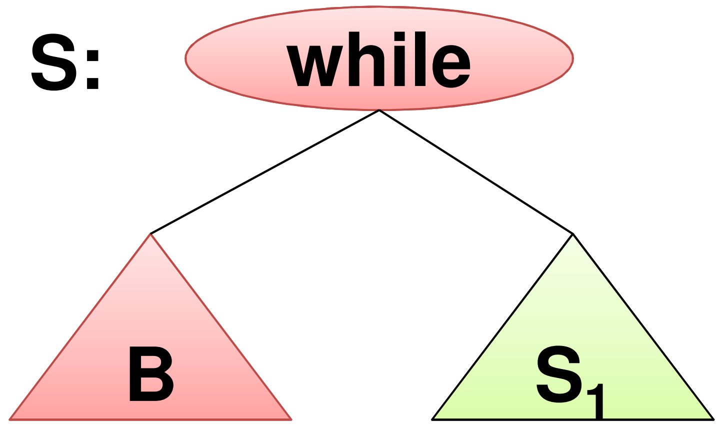


Code Structure:

```
L_top  : code to evaluate B
         if ( !B ) goto L_after
L_body:  code for S_1
         goto L_top
L_after:  …
```

```
codeGen_stmt(S):
/* S.nodetype == 'WHILE' */
L_top = newlabel();
L_body = newlabel();
L_after = newlabel();
codeGen_bool(B, L_body, L_after);
    codeGen_stmt(S_1);
S.code = L_top
        ⊕ B.code
        ⊕ L_body
        ⊕ S1.code
        ⊕ newinstr(GOTO, L_top)
        ⊕ L_after ;
```

Loops : While Loop-Improved Code



Code Structure:

```
        goto $L_{eval}$
$L_{top}$ : code for $S_1$
$L_{eval}$: code to evaluate B
        if ( B ) goto $L_{top}$
$L_{after}$:
```

This code executes fewer branchs

```
codeGen_stmt(S):
$L_{top}$ = *newlabel*();
$L_{eval}$ = *newlabel*();
$L_{after}$ = *newlabel*();
codeGen_bool(B, $L_{top}$, $L_{after}$);
codeGen_stmt($S_1$);
S.code =
        *newinstr*(GOTO, $L_{eval}$)
        $\oplus$ $L_{top}$
        $\oplus$ $S_1$.code
        $\oplus$ $L_{eval}$
        $\oplus$ B.code
        $\oplus$ $L_{after}$ ;
```