CS 5500 – The Structure of a Compiler

Intermediate Code Generation (continued)

Boolean Expressions

- Used to **compute logical values**; use 3-address code for *relops*, *and*, *or*, *not* similar to what we do for arithmetic operators
- Need to know whether PL uses **lazy evaluation** (i.e., expression not evaluated unless actually required for result); important if side-effects!
- In ***short-circuit code*** (a.k.a. ***jumping code***) &&, ||, and ! translate into jumps

  Ex: if ((x < 100) || ((x > 200) && (x !=y))) x = 0;

  ifTrue x < 100 goto L2

  ifFalse x > 200 goto L1

  ifFalse x != y goto L1

  L2: x = 0

  L1:

Syntax-Directed 3-Address Code Generation for Boolean Expressions

- B.true is label to jump to if B is true
- B.false is label to jump to if B is false
- (we -addr. code stmt.)

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_1 && B_2 { B_1.true = newLabel( );
B_1.false = B.false;
B_2.true = B.true;
B_2.false = B.false;
B.code = B_1.code + label(B_1.true) + B_2.code; }

| ! B_1 { B_1.true = B.false;
B_1.false = B.true;
B.code = B_1.code; }

| E_1 relop E_2 { B.code = E_1.code + E_2.code +
$_1$.addr, relop, E_2
; }

| true {

| false {

Ex: (x < 100) || ((x > 200) && (x !=y))

Assume that whatever production referenced

$B \rightarrow E_1$ relop E_2 **B.code** = E_1.code + E_2.code +
gen(“if”, E_1.addr, relop, E_2.addr, “goto”, B.true) +
gen(“goto”, B.false);
= “if x != y goto L2
goto L1”

Output for ((x < 100) || ((x > 200) && (x !=y))):

```
if x < 100 goto L2
goto L3
L3:
if x > 200 goto L4
goto L1
L4:
if x != y goto L2
goto L1
L2:         // B.true
…
L1:         // B.false
…
```

Flow-of-Control Statements

- Boolean expressions also used to **alter flow of control** (e.g., if-stmt, loops, etc.)

Syntax-Directed 3-Addr. Code Generation for Flow-of-Control Statements

P S { S.next = newLabel( );
P.code = S.code + label(S.next); }

assign { S.code = assign.code; }

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

| if (B) S_1 else S_2
{ B.true = newLabel( );
B.false = newLabel( );
S_1.next = S.next;
S_2.next = S.next;
S.code = B.code + label(B.true) + S_1.code +
$_2$.code; }

| while (B) S_1
{ begin = newLabel( );
B.true = newLabel( );
B.false = S.next;
S_1.next = begin;
S.code = label(begin) + B.code + label(B.true) + S_1.code +

| S_1 S_2 { S_1.next = newLabel( );
S_2.next = S.next;
S.code = S_1.code + label(S_1.next) + S_2.code; }

Ex: fact = 1; while (n > 1) { fact = fact * n; n = n 1; }

P → S **S.next = newLabel( ) = L1**
P.code = S.code + label(S.next)
= "fact = 1;
L2:
L3: t2 = n;
…
goto L3;
L1:"

S → S_1 S_2 **S_1.next = newLabel( ) = L2**
S_2.next = S.next = L1
S.code = S_1.code + label(S_1.next) + S_2.code
= "fact = 1; L2:" + S_2.code
= "fact = 1;
L2:
L3: t2 = n;
t3 = 1;
if t2 > t3 goto L4;
goto L1;
L4: fact = fact * n;
L5: n = n – 1;
goto L3;"

S → while (B) S_3
begin = newLabel( ) = L3
B.true = newLabel( ) = L4
B.false = S.next = L1
S_3.next = begin = L3
S.code = label(begin) + B.code + label(B.true) +
S_3.code + gen("goto", begin)
= "L3: t1 = n > 1; L4: " + S_3.code + "goto L3"
= "L3: t2 = n;
t3 = 1;
if t2 > t3 goto L4;
goto L1;
L4: fact = fact * n;
L5: n = n – 1;
goto L3;"

S_3 → S_4 S_5 **S_4.next = newLabel( ) = L5**
S_5.next = S.next = L3
S.code = S_4.code + label(S_4.next) + S_5.code
= "fact = fact * n; L5: n = n – 1"

$B \rightarrow E_1$ relop E_2 B.code = E_1.code + E_2.code +
gen("if", E_1.addr, relop, E_2.addr, "goto", B.true)
+ gen("goto", B.false)
= "t2 = n;
t3 = 1;
if t2 > t3 goto L4;
goto L1;"

So final code is:

```
        fact = 1;
L2:
L3:     t2 = n;
        t3 = 1;
        if t2 > t3 goto L4;
        goto L1;
L4:     fact = fact * n;
L5:     n = n – 1;
        goto L3;
L1:
```