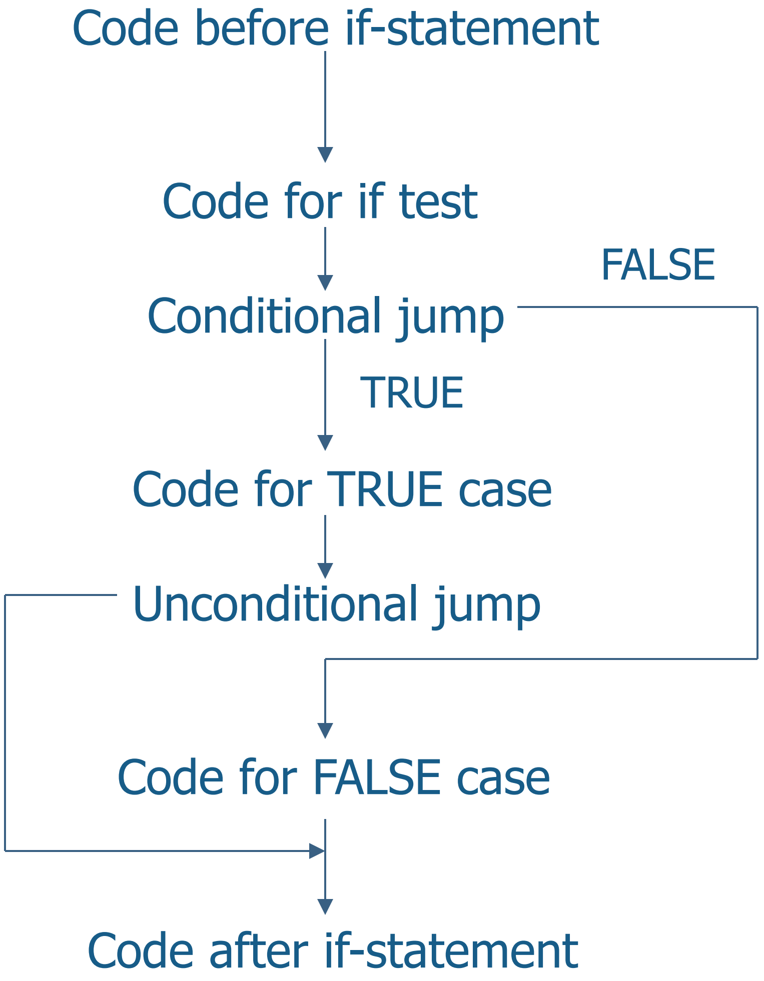
Intermediate Code Generation (II)

CIS*4650 (Winter 2020)

Code for If-statements



Code before if-statement

Code for if test

FALSE

Conditional jump

TRUE

Code for TRUE case

Unconditional jump

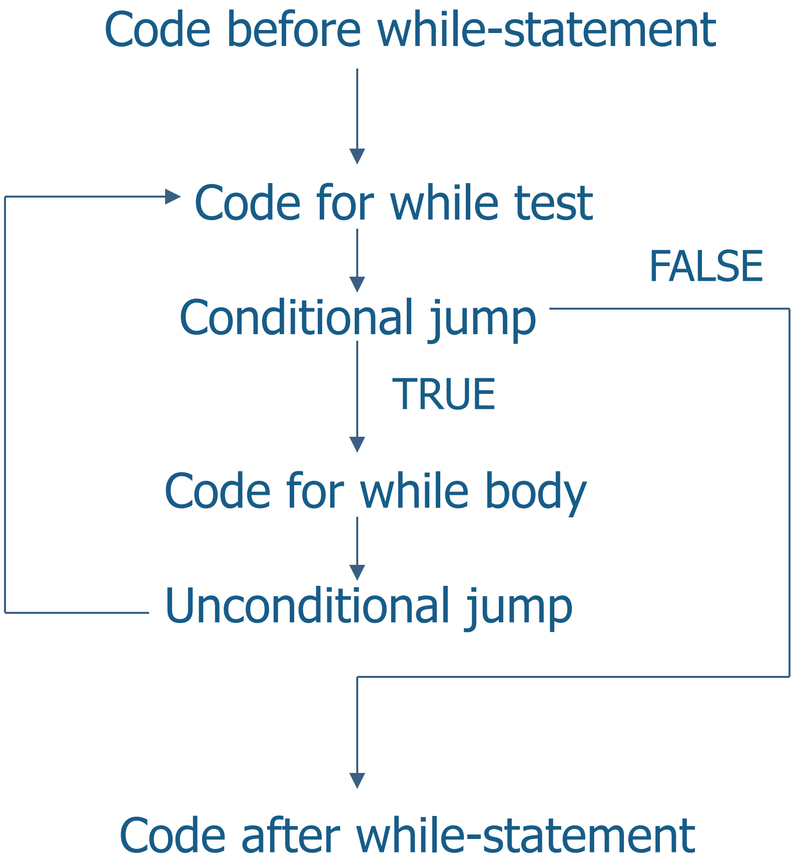
Code for FALSE case

Code after if-statement

if (E) S1 else S2

```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
```

Code for While-statements



while (E) S

```
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```

Label Generation and Backpatching

- Jumps to a label may need to be generated before the label definition
  - Intermediate code: generate a label for a forward jump and save it until the label location is known
  - Executable code: labels must be resolved to absolute or relative addresses
- Backpatching: leave a gap in the code for a forward jump or create a dummy jump to a fake location, and then go back to fix the location when the actual label is known
  - Keep the generated code in a buffer or a temporary file

Code for Logical Expressions

- Short circuit:
  - If a is false then (a and b) is also false
  - If a is true then (a or b) is also true
- If-expressions: equivalent to if-statements except that they return values
  - a and b ≡ if a then b else false
  - a or b ≡ if a then true else b

```
(x != 0) && (y == x)

if( x != 0 ) then (y == x)
else false
```

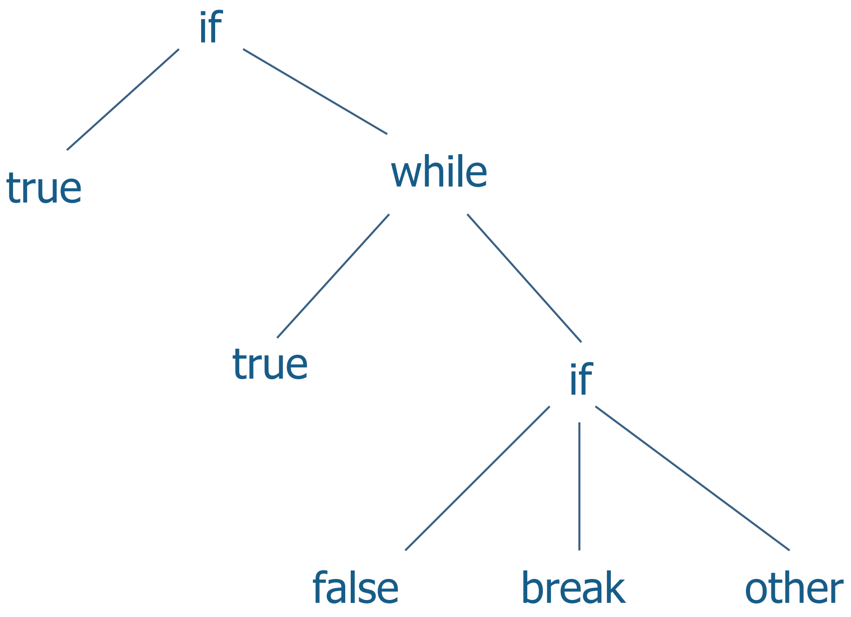
```
t1 = (x != 0)
if_false t1 goto L1
t2 = (y == x)
goto L2
label L1
t2 = FALSE
label L2
```

Code Generation for Control Stmts

stmt -> if-stmt | while-stmt | **break** | **other**
if-stmt -> **if** (exp) stmt | **if** (exp) stmt **else** stmt
while-stmt -> **while** (exp) stmt
exp -> **true** | **false**

```
if_false true goto L1

label L2
if_false true goto L3

if_false false goto L4
goto L3
goto L5
label L4
other
label L5

goto L2
label L3

label L1
```

e.g., if (true) while (true) if (false) break else other



Code Generation for Control Stmts

```
void genCode( Exp tree, String label ) {
    String codestr = “”;
    String lab1, lab2;
    if( tree != null ) {
        if( tree instanceof IntExp ) {
          // do nothing
        } else if( tree instanceof IfExp ) {
          // refer to the related fragment
        } else if( tree instanceof WhileExp ) {
          // refer to the related fragment
        } else if( tree instanceof BreakExp ) {
          codestr += “goto  “ + label;
          emitCode( codestr );
        } else if( tree instanceof OtherExp ) {
            emitCode( “Other” );
        } else
    }
}
```

```
// code fragment for WhileExp
lab1 = genLabel();
codestr += “label” + lab1;
emitCode( codestr );
genCode( tree.test, label );
lab2 = genLabel();
if( tree.test .value == 0 )
    codestr += “if_false false goto ” + lab2;
else
   codestr += “if_false true goto ” + lab2;
emitCode( codestr );
genCode( tree.body, lab2 );
codestr += “goto ” + lab1;
emitCode( codestr );
codestr += “label ” + lab2;
emitCode( codestr );
```

Code Generation for Control Stmts

```
// code fragment for IfExp
genCode( tree.test, label );
lab1 = genLabel();
if( tree.test.value == 0 )
    codestr += “if_false false goto ” + lab1;
else
    codestr += “if_false true goto ” + lab1;
emitCode( codestr );
genCode( tree.then, label );
if( tree.else != null ) {
    lab2 = genLabel();
    codestr += “goto ” + lab2;
    emitCode( codestr );
}
```

```
// continued from left
codestr += “label ” + lab1;
emitCode( codestr );
if( tree.else != null ) {
    genCode( tree.else, label );
    codestr += “label ” + lab2;
    emitCode( codestr );
}
```

Function Definitions and Calls

- Function definition: create a function name, parameters, the return type, and the code
- Function call: create actual values for parameters (called arguments), perform a jump to the function code, and return to the caller
- The runtime environment is not known at the definition time, but the general record structure is clear
  - The runtime environment is built by the calling sequence (partially by the caller and partially by the callee)

Intermediate Code for Functions

e.g., function definition:

```
int f( int x, int y ) {
    return x + y + 1;
}
```

Three-address code:

```
entry f
t1 = x + y
t2 = t1 + 1
return t2
```

e.g., function call:

```
x = f( 2 + 3, 4 );
```

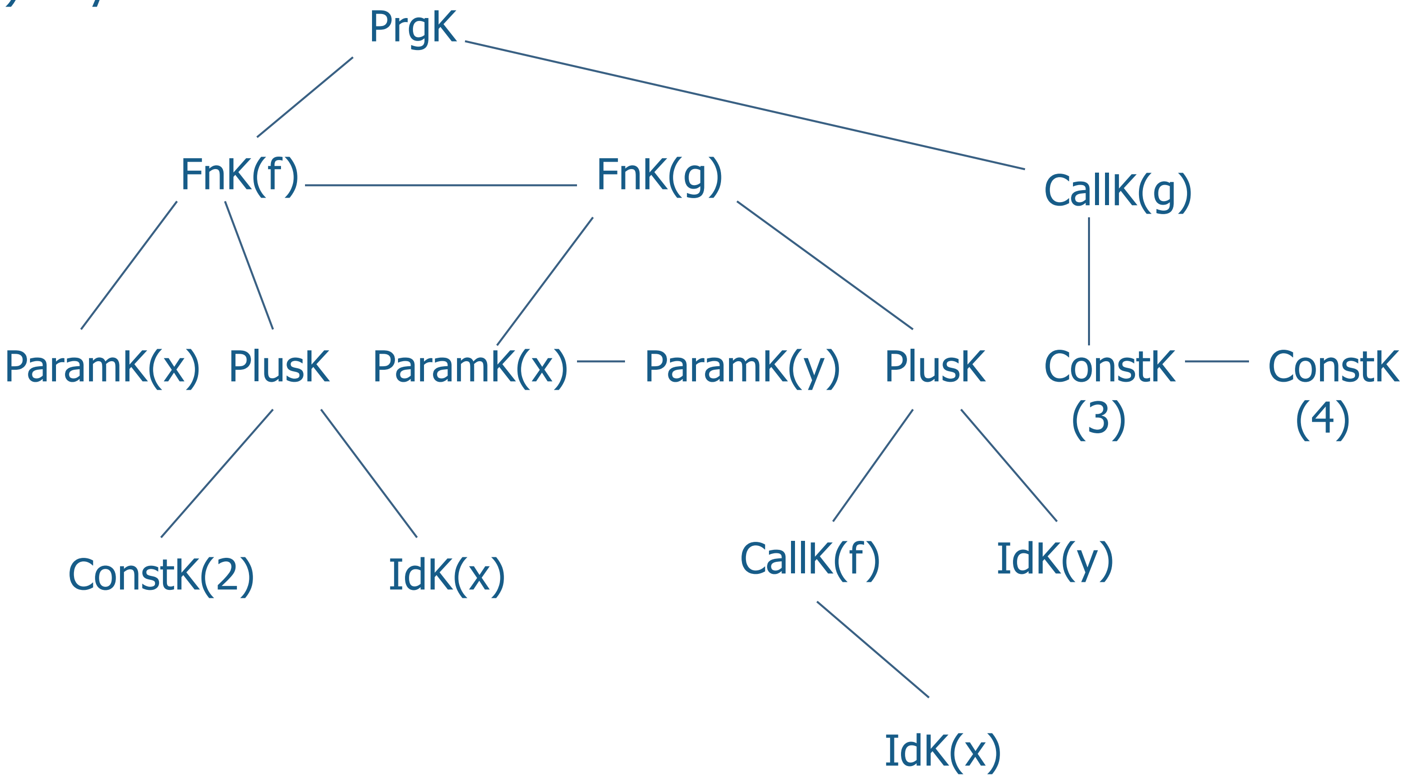
Three-address code:

```
begin_args
t1 = 2 + 3
arg t1
arg 4
x = call f
```

Code Generation for Functions

```
fn f(x) = 2 + x
fn g(x, y) = f(x) + y
g(3, 4)
```

```
entry f
t1 = 2 + x
return t1

entry g
begin_args
arg x
t2 = call f
t3 = t2 + y
return t3

begin_args
arg 3
arg 4
call g
```



PrgK

FnK(f)

FnK(g)

CallK(g)

ParamK(x)

PlusK

ParamK(x)

ParamK(y)

PlusK

ConstK (3)

ConstK (4)

ConstK(2)

IdK(x)

CallK(f)

IdK(y)

IdK(x)