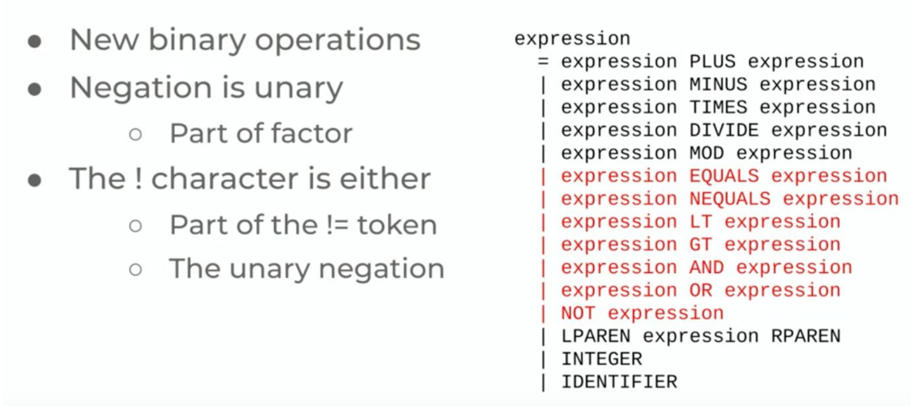
COP-3402 Systems Software
10.24 Thu

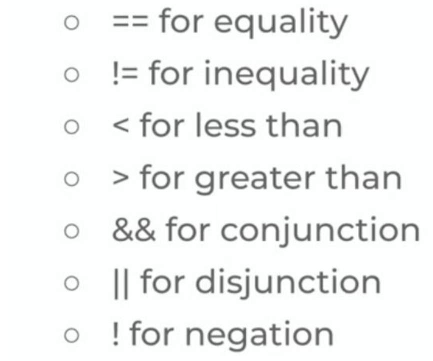
Please refer to the notes of the previous lecture for the new Project3 grammar.
Some updates on the grammar: (LT:less than, GT:greater than, etc.)
Refer also to */syllabus/projects/lexical_specification.md*:



- New binary operations
- Negation is unary
    - Part of factor
- The ! character is either
    - Part of the != token
    - The unary negation

```
expression
  = expression PLUS expression
  | expression MINUS expression
  | expression TIMES expression
  | expression DIVIDE expression
  | expression MOD expression
  | expression EQUALS expression
  | expression NEQUALS expression
  | expression LT expression
  | expression GT expression
  | expression AND expression
  | expression OR expression
  | NOT expression
  | LPAREN expression RPAREN
  | INTEGER
  | IDENTIFIER
```

SimpleC now has:



- == for equality
- != for inequality
- < for less than
- > for greater than
- && for conjunction
- || for disjunction
- ! for negation

The *icmp* instruction (integer compare) does integer comparisons. SimpleC variables are always 32-bit signed integers. 1 represents true, zero represents false in LLVM.
Ex:
*%t2 = icmp slt i32 %t1, 10*

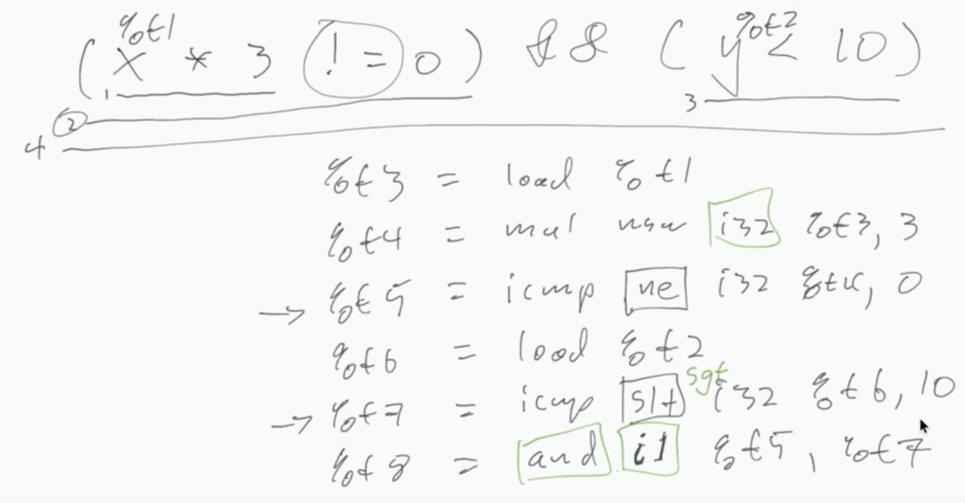
%t2 holds a 1-bit integer. icmp returns a 1-bit integer. *slt* is less-than for signed integers.

LLVM Instructions for Booleans:
Booleans are 1-bit integers in LLVM.
Ex:
*%t3 = and i1 %t1, %t2*
*%t4 = or i1 %t3, %t1*

LLVM has no unary negation. Use xor instead.
Ex:
*%t5 = xor i1 %t4, 1*

Demo: Boolean Operations and Comparisons
Ex:
(x * 3 != 0) && (y < 10)

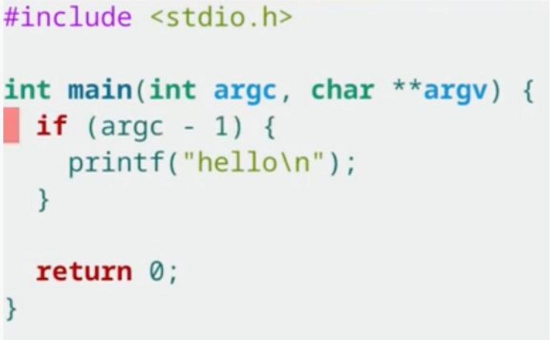
// x * 3 is executed first, and then first parentheses, next is y - 10 and lastly the and operator.
Let’s assume x is allocated to %t1 and y is allocated to %t2.



```
   %t3 = load %t1
   %t4 = mul nsw i32 %t3, 3
-> %t5 = icmp ne i32 %t4, 0
   %t6 = load %t2
-> %t7 = icmp slt i32 %t6, 10
   %t8 = and i1 %t5, %t7
```

How do we guarantee booleans are i1 types?
The OR operator takes a signed integer and an integer
We can use a type system to make this formal

Ex: The value zero then it is true. Nonzero then false.

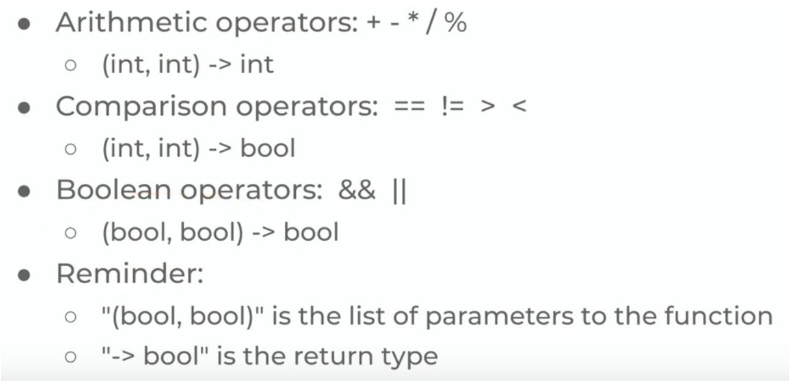


```
#include <stdio.h>

int main(int argc, char **argv) {
  if (argc - 1) {
    printf("hello\n");
  }

  return 0;
}
```

Compile:
gcc -o test test.c
./test                         // hello

To see the llvm:
gcc -S -00 test.c
less test.s

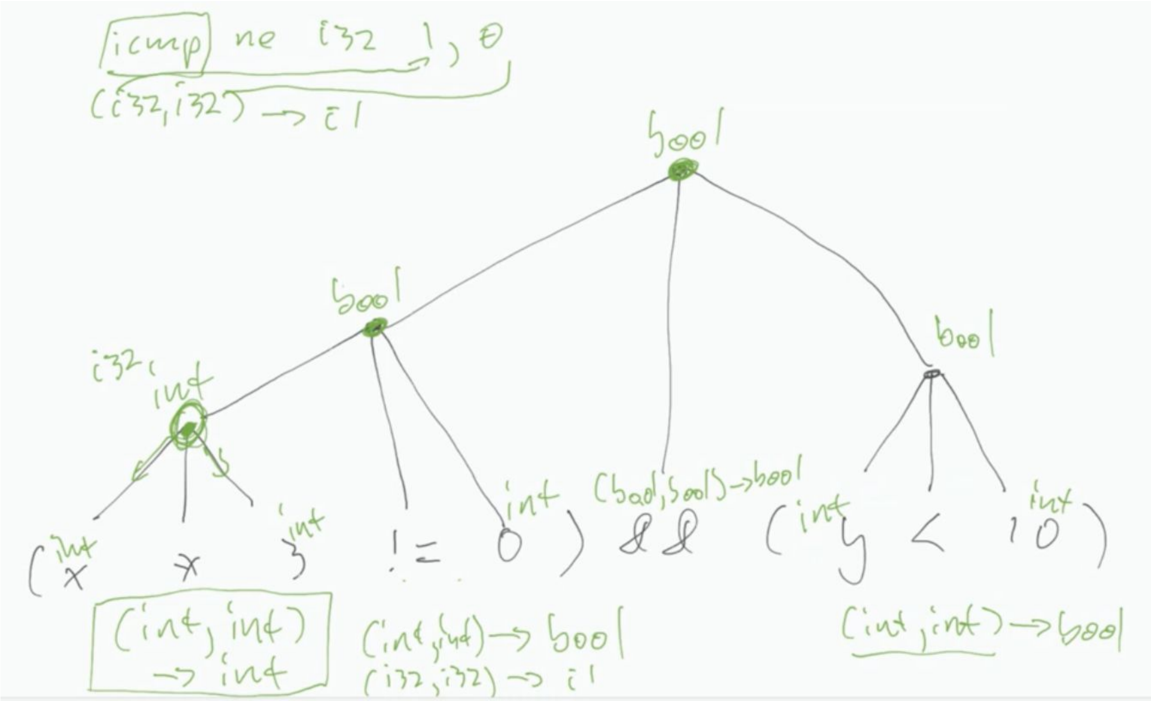
Operator Types:



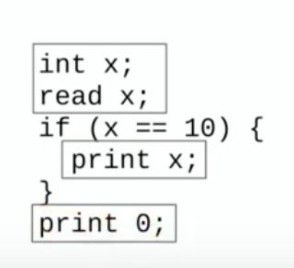
- Arithmetic operators: + - * / %
  - (int, int) -> int
- Comparison operators: == != > <
  - (int, int) -> bool
- Boolean operators: && ||
  - (bool, bool) -> bool
- Reminder:
  - "(bool, bool)" is the list of parameters to the function
  - "-> bool" is the return type

Type checking guarantees correct types, i1 is used only in booleans.
Type checker can reject the program, require a rewrite

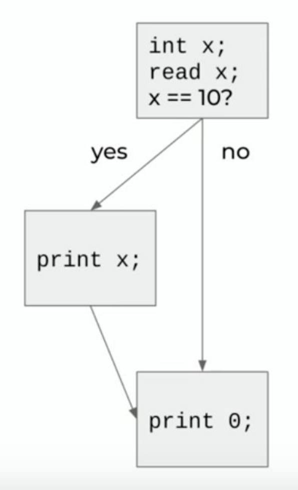
Demo: Type-checking expressions
Type of x is stored in symbol table as int.
3 is just a number as int;
x*3 (int, int) -> int.       Result of operation satisfies the boolean (!=)  and zero is number (int)



Implementing if-statements:
Each branch means a mutually exclusive choice
Only one branch to be executed



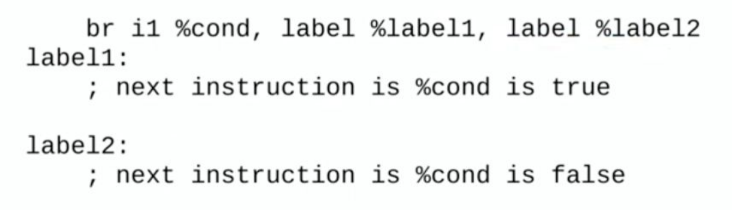
```
int x;
read x;
if (x == 10) {
  print x;
}
print 0;
```



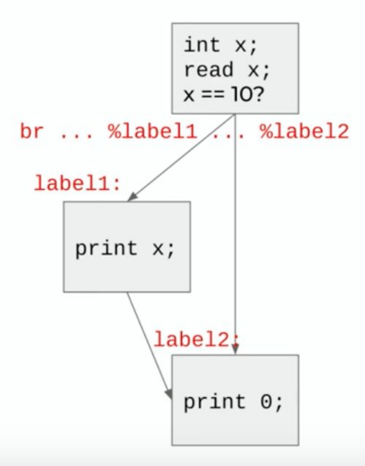
If x is 11 it will print 0.
If it is 10, it will print 10 and 0.

Multiple out edges can be a branch
In-edges are targets of the branch

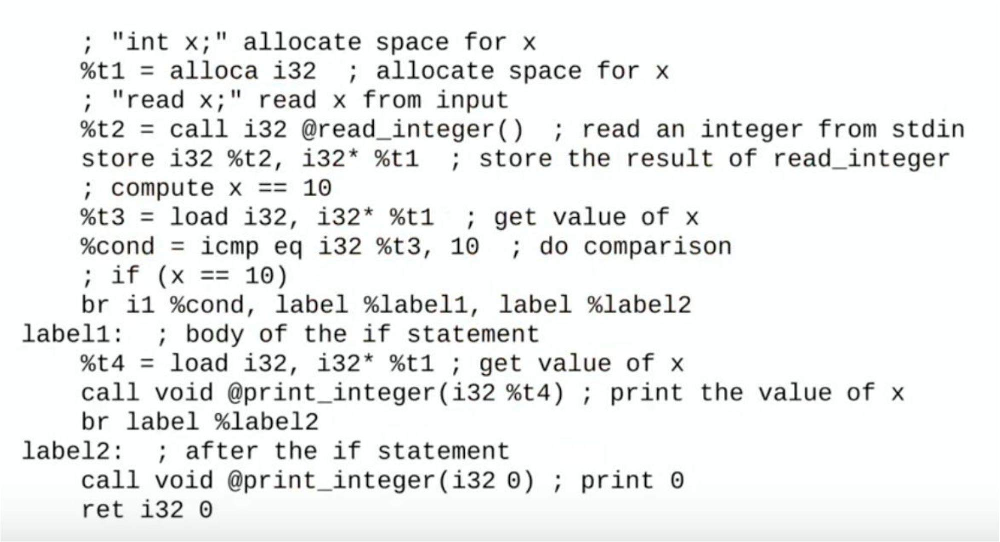
Branching in LLVM:



```
    br i1 %cond, label %label1, label %label2
label1:
    ; next instruction is %cond is true

label2:
    ; next instruction is %cond is false
```



The branch instruction goes right after comparison and labels get inserted right before “print x” and “print 0”



```
    ; "int x;" allocate space for x
    %t1 = alloca i32  ; allocate space for x
    ; "read x;" read x from input
    %t2 = call i32 @read_integer()  ; read an integer from stdin
    store i32 %t2, i32* %t1  ; store the result of read_integer
    ; compute x == 10
    %t3 = load i32, i32* %t1  ; get value of x
    %cond = icmp eq i32 %t3, 10  ; do comparison
    ; if (x == 10)
    br i1 %cond, label %label1, label %label2
label1:  ; body of the if statement
    %t4 = load i32, i32* %t1 ; get value of x
    call void @print_integer(i32 %t4) ; print the value of x
    br label %label2
label2:  ; after the if statement
    call void @print_integer(i32 0) ; print 0
    ret i32 0
```