UCF | College of Engineering and Computer Science
UNIVERSITY OF CENTRAL FLORIDA

Control Flow Target Language

COP-3402 Systems Software

Paul Gazzillo

Boolean Expressions

- SimpleC (now) has
  - == for equality
  - != for inequality
  - < for less than
  - > for greater than
  - && for conjunction
  - || for disjunction
  - ! for negation

Same Syntax for Booleans and Arithmetic

- 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
```

LLVM Instructions for Comparisons

- The icmp instruction does integer comparisons
  - Equals, not equals, less than, etc
  - SimpleC's variables are always 32-bit signed integers
- For instance, (y < 10), where y's value is in %t1

```
%t2 = icmp slt i32 %t1, 10
```

Save result in register

Less-than for signed integers

Takes two operands and their type (just like add)

- icmp returns a 1-bit integer (LLVM's Boolean)

LLVM Instructions for Booleans

- Booleans are 1-bit integers in LLVM, i.e.,
  - `i1` as opposed to `i32`
- `and` and `or` work just like arithmetic instructions

```
%t3 = and i1 %t1, %t2
%t4 = or i1 %t3, %t1
```

- LLVM has no unary negation
- Use `xor` instead

```
%t5 = xor i1 %t4, 1
```

Demo: Boolean Operations and Comparisons

```
(x * 3 != 0) && (y < 10)
```

How Do We Guarantee Booleans Are `i1` Types?

- What happens with

  ```
  (10 - foo) || (bar == 4)
  ```

- The OR operator takes a signed integer and a Boolean
- We can use a type system to make this formal
  - And automatically checkable

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 to Guarantee Correct Types

- Goal: guarantee only `i1` is used in Boolean operations
  - Without type-checking, LLVM's type checking would fail
- Type-checker can
  - Reject the program, require a rewrite
  - Insert type coercion to cast types (C does this)
- Note: We will not give you incorrectly typed test cases
  - Bonus project: add type-checking to SimpleC expression
  - Bonus project: add a Boolean type to the SimpleC

Demo: Type-Checking Expressions

```
(x * 3 != 0) && (y < 10)

(10 - foo) || (bar == 4)

x = 10 < y
```

Implementing If Statements

- Each branch represents a mutually exclusive choice
  - Only one branch (if any) are to be executed
- Branch instructions can jump past code
  - Leaves code unexecuted
- Choice depends on value of conditional expression

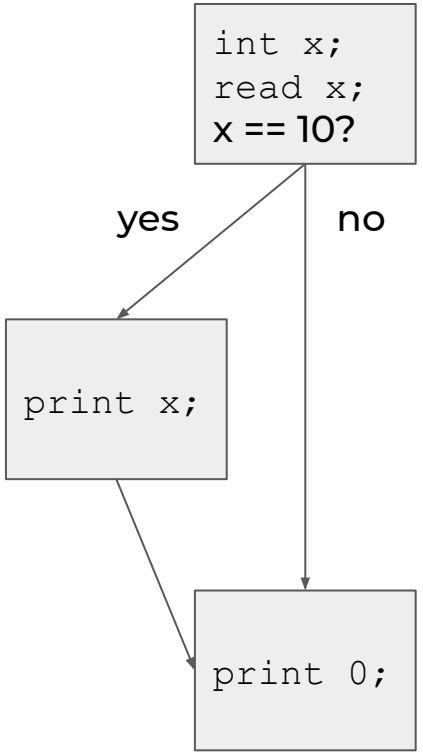
```
int x;
read x;
if (x == 10) {
  print x;
}
print 0;
```

Only executed if condition is met

Flowchart Reveals Branches

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

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



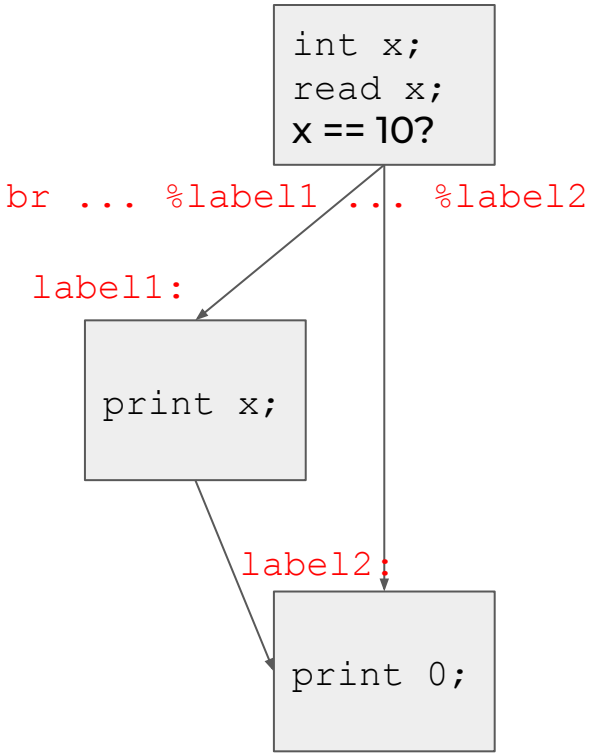
Branching in LLVM

- br instruction takes
  - A condition
  - The label to jump to if the condition is true
  - The label to jump to if the condition is false

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

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

Branch and Labels in Flowchart



Combing Comparisons with Branching

```
    ; "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
```