HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers Lecture 5: Intermediate Representation

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic

Announcements

- Homework 2: X86lite
  - Due Tuesday Sept 24
- Extension School Office Hours started
  - See https://canvas.harvard.edu/courses/63122/pages/office-hours
- Homework 3: LLVMlite
  - will be released next week

Today

- Intermediate representations

Mid-level IR's: Many Varieties

- Intermediate between AST (abstract syntax) and assembly
- May have unstructured jumps, abstract registers or memory locations
- Convenient for translation to high-quality machine code
  - Example: all intermediate values might be named to facilitate optimizations that attempt to minimize stack/register usage
- Many examples:
  - Quadruples: a = b OP c (“three address form”)
  - Triples: OP a b
    - “Name” of result is implicit
    - Useful for instruction selection on X86 via “tiling”
  - SSA: variant of quadruples where each variable is assigned exactly once
    - Easy dataflow analysis for optimization
    - e.g. LLVM: industrial-strength IR, based on SSA
  - Stack-based:
    - Easy to generate
    - e.g., Java Bytecode, UCODE

Growing an IR

- Develop an IR in detail… starting from the very basic.
- Start: a (very) simple intermediate representation for the arithmetic language
  - Very high level
  - No control flow
- Goal: A simple subset of the LLVM IR
  - LLVM = "Low-level Virtual Machine"
  - Used in HW3+
- Add features needed to compile rich source languages

Eliminating Nested Expressions

- Fundamental problem:
  - Compiling complex & nested expression forms to simple operations.

Source

```
((1 + X4) + (3 + (X1 * 5)))
```

AST

```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

IR

?

- Idea: name intermediate values, make order of evaluation explicit.
  - No nested operations.

Translation to Simple Let Language

- Given this:

```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

- Translate to this desired SLL form:

```
let tmp0 = add 1L varX4 in
let tmp1 = mul varX1 5L in
let tmp2 = add 3L tmp1 in
let tmp3 = add tmp0 tmp2 in
  tmp3
```

- Translation makes the order of evaluation explicit
- Names intermediate values
- Note: introduced temporaries are never modified

Building IRs

- Look at files `ir-by-hand.ml` and `ir?.ml`.

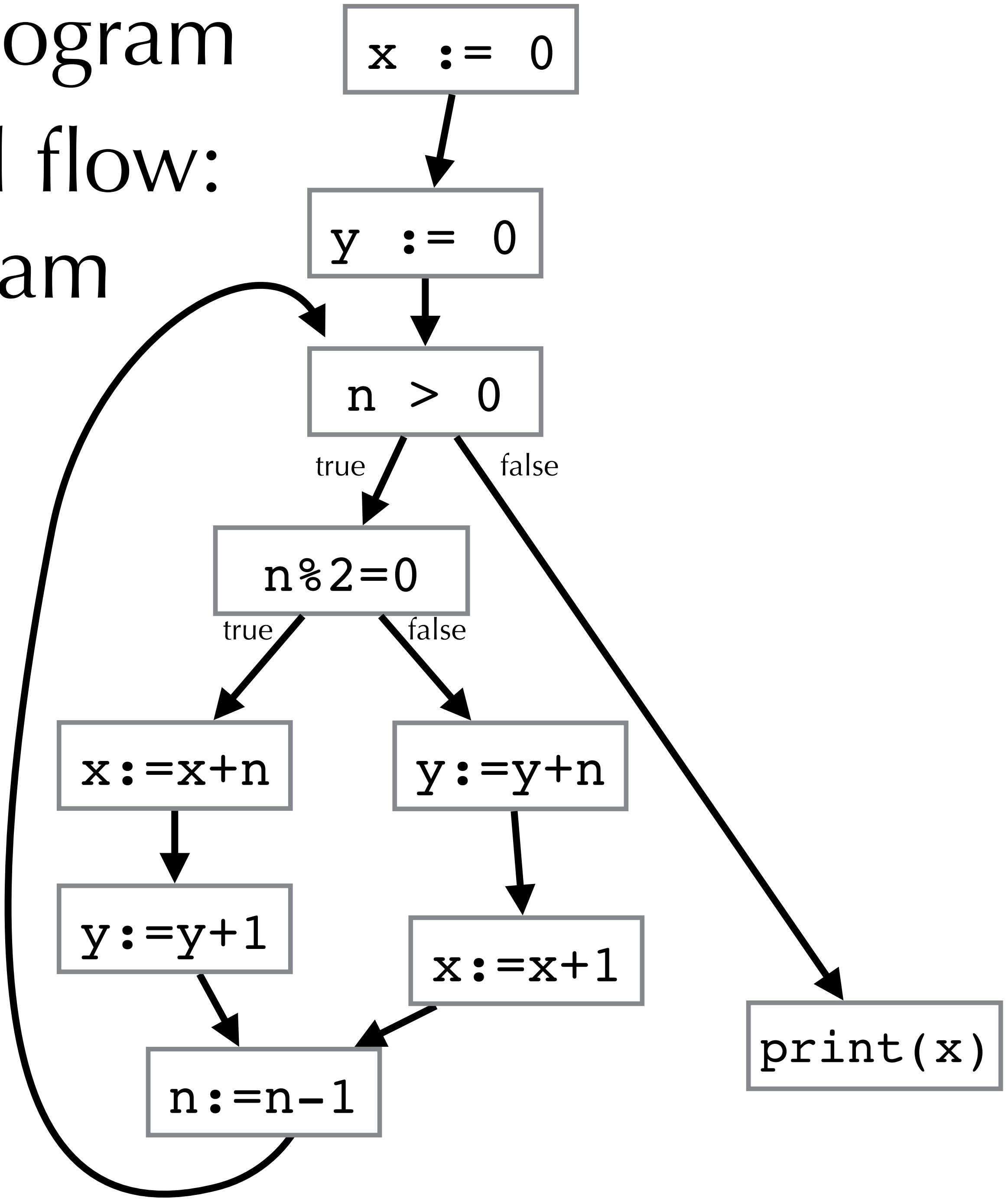
Intermediate Representations

- IR1: Expressions
  - simple arithmetic expressions, immutable global variables
- IR2: Commands
  - global **mutable** variables
  - commands for update and sequencing
- IR3: Local control flow
  - conditional commands & while loops
  - **basic blocks**
- IR4: Procedures (top-level functions)
  - local state
  - call stack

Control-Flow Graphs

- Graphical representation of a program
- Edges in graph represent control flow: how execution traverses a program
- Nodes represent statements

```
x := 0;
y := 0;
while (n > 0) {
   if (n % 2 = 0) {
      x := x + n;
      y := y + 1;
   }
   else {
      y := y + n;
      x := x + 1;
  }
   n := n - 1;
}
print(x);
```



Basic Blocks

- We will require that nodes of a control flow graph are **basic blocks**
  - Sequences of statements such that:
    - Can be entered only at beginning of block
    - Can be exited only at end of block
      - Exit by branching, by unconditional jump to another block, or by returning from function
- Basic blocks simplify representation and analysis

Basic Blocks

- Basic block: single entry, single exit

```
x := 0;
y := 0;
while (n > 0) {
  if (n % 2 = 0) {
    x := x + n;
    y := y + 1;
  }
  else {
    y := y + n;
    x := x + 1;
 }
  n := n - 1;
}
print(x);
```