Lecture 7

CIS 3410/7000: COMPILERS

Announcements

- HW2: X86lite
  - Due Oct 21st

INTERMEDIATE REPRESENTATIONS

Why do something else?

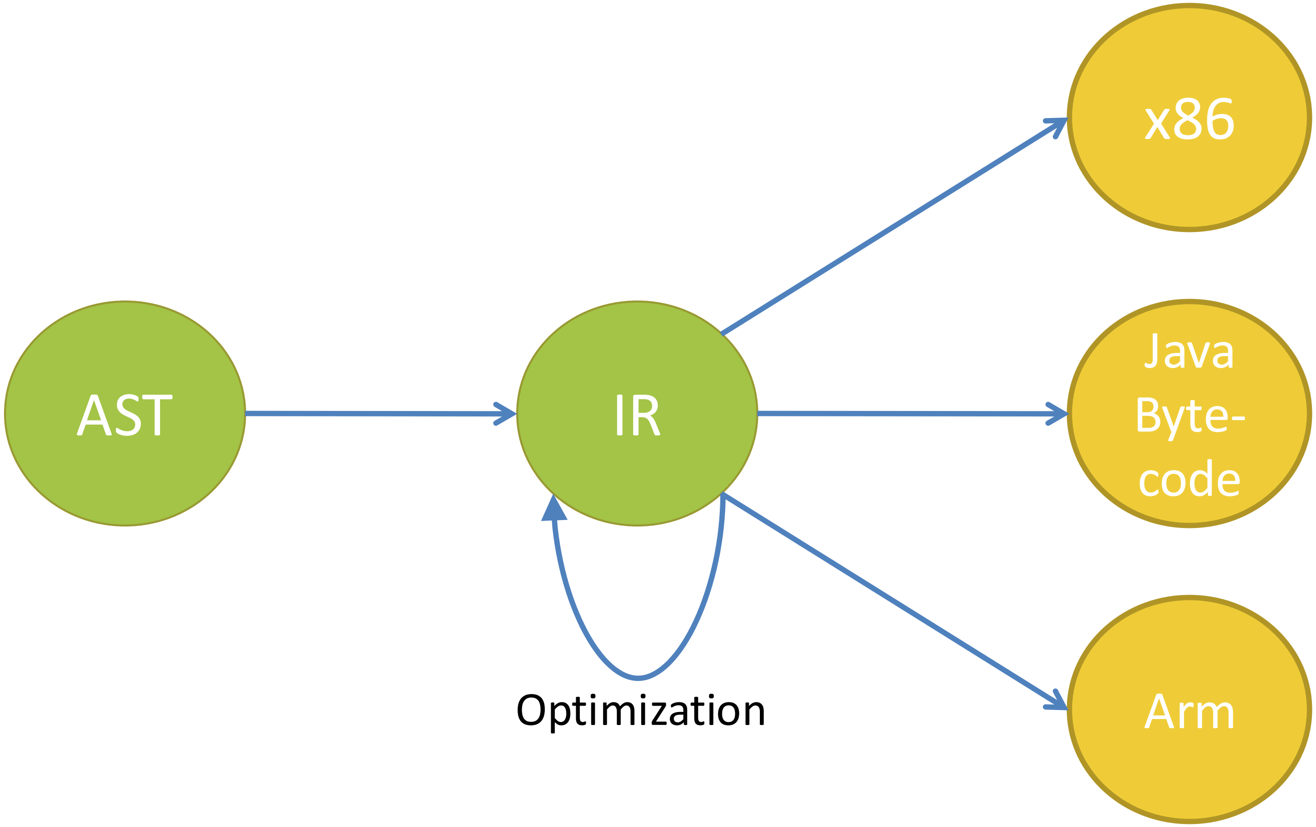
- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.
- Richer source language features are hard to encode
  - Structured data types, objects, first-class functions, etc.
- It's hard to optimize the resulting assembly code.
  - The representation is too concrete – *e.g.*, it has committed to using certain registers and the stack
  - Only a fixed number of registers
  - Some instructions have restrictions on where the operands are located
- Control-flow is not structured:
  - Arbitrary jumps from one code block to another
  - Implicit fall-through makes sequences of code non-modular (*i.e.*, you can't rearrange sequences of code easily)
- Retargeting the compiler to a new architecture is hard.
  - Target assembly code is hard-wired into the translation

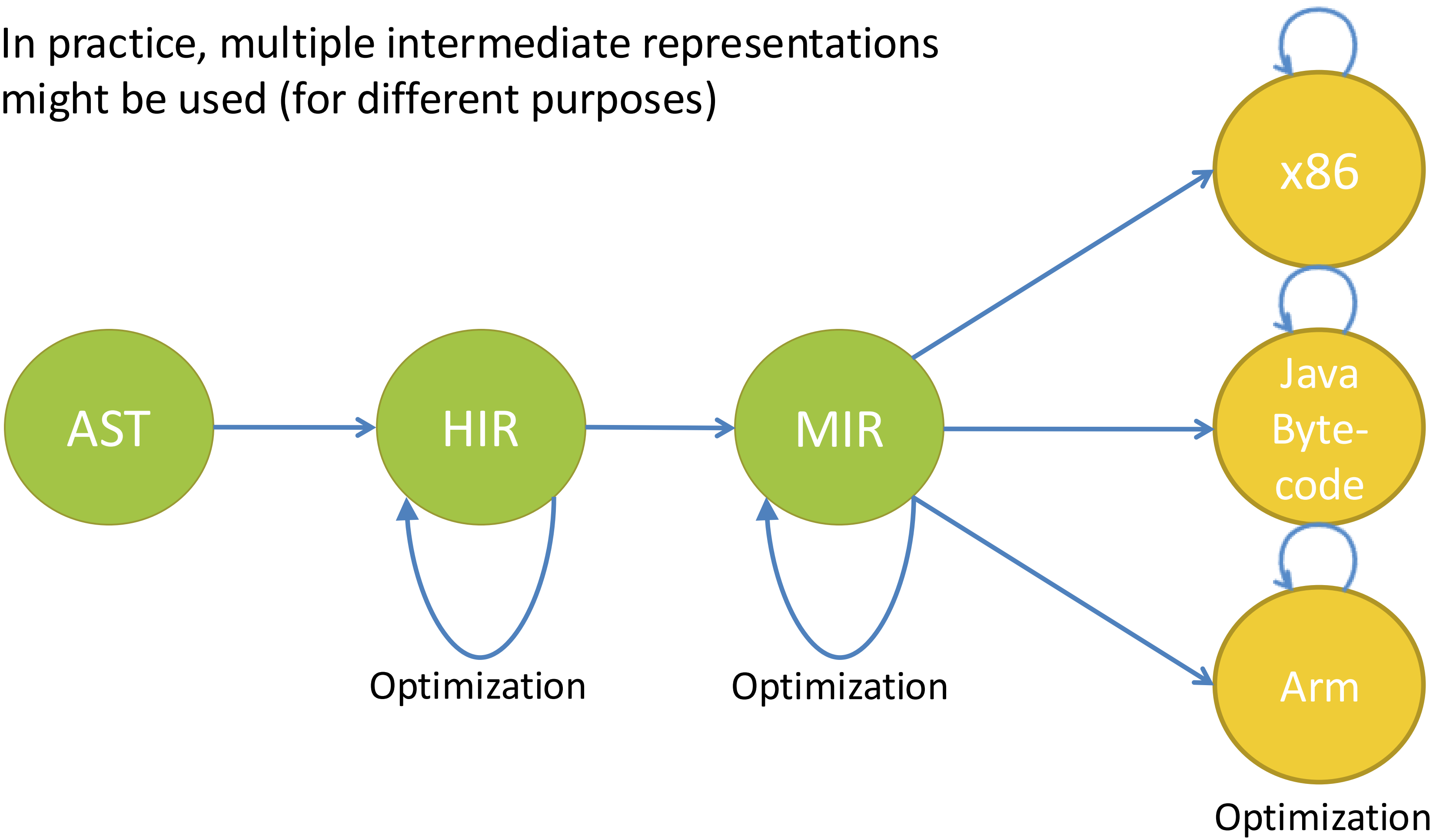
Intermediate Representations (IR's)

- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization.



Multiple IR's

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- In practice, multiple intermediate representations might be used (for different purposes)



What makes a good IR?

- Easy translation target (from the level above)
- Easy to translate (to the level below)
- Narrow interface
  - Fewer constructs means simpler phases/optimizations
- Example: Source language might have “while”, “for”, and “foreach” loops (and maybe more variants)
  - IR might have only “while” loops and sequencing
  - Translation eliminates “for” and “foreach”

```
⟦for(pre; cond; post) {body}⟧
              =
⟦pre; while(cond) {body;post}⟧
```
*

*Here the notation ⟦cmd⟧ denotes the “translation” or “compilation” of the command cmd.

IR's at the extreme

- High-level IR's
  - Abstract syntax + new node types not generated by the parser
    - e.g., Type checking information or disambiguated syntax nodes
  - Typically preserves the high-level language constructs
    - Structured control flow, variable names, methods, functions, etc.
    - May do some simplification (e.g., convert `for` to `while`)
  - Allows high-level optimizations based on program structure
    - e.g., inlining "small" functions, reuse of constants, etc.
  - Useful for semantic analyses like type checking

- Low-level IR's
  - Machine dependent assembly code + extra pseudo-instructions
    - e.g., a pseudo instruction for interfacing with garbage collector or memory allocator (parts of the language runtime system)
    - e.g., (on x86) a `imulq` instruction that doesn't restrict register usage
  - Source structure of the program is lost:
    - Translation to assembly code is straightforward
  - Allows low-level optimizations based on target architecture
    - e.g., register allocation, instruction selection, memory layout, etc.

- What's in between?

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:
  - Triples: OP a b
    - Useful for instruction selection on X86 via "tiling"
  - Quadruples: a = b OP c (RISC-like "three address form")
  - 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

SIMPLE LET-BASED IR

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 SLL

- 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

see: ir-by-hand.ml, ir<X>.ml in lec06.zip

INTERMEDIATE REPRESENTATIONS

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 SLL

- 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

Intermediate Representations

- IR1: Expressions
  - *immutable* global variables
  - simple arithmetic *expressions*

- IR2: Commands
  - *mutable* global variables
  - *commands* for update and sequencing

- IR3: Local control flow
  - *conditional* commands & while *loops*
  - *basic blocks*

- IR4: Procedures (top-level functions)
  - *local variables*
  - *call stack*

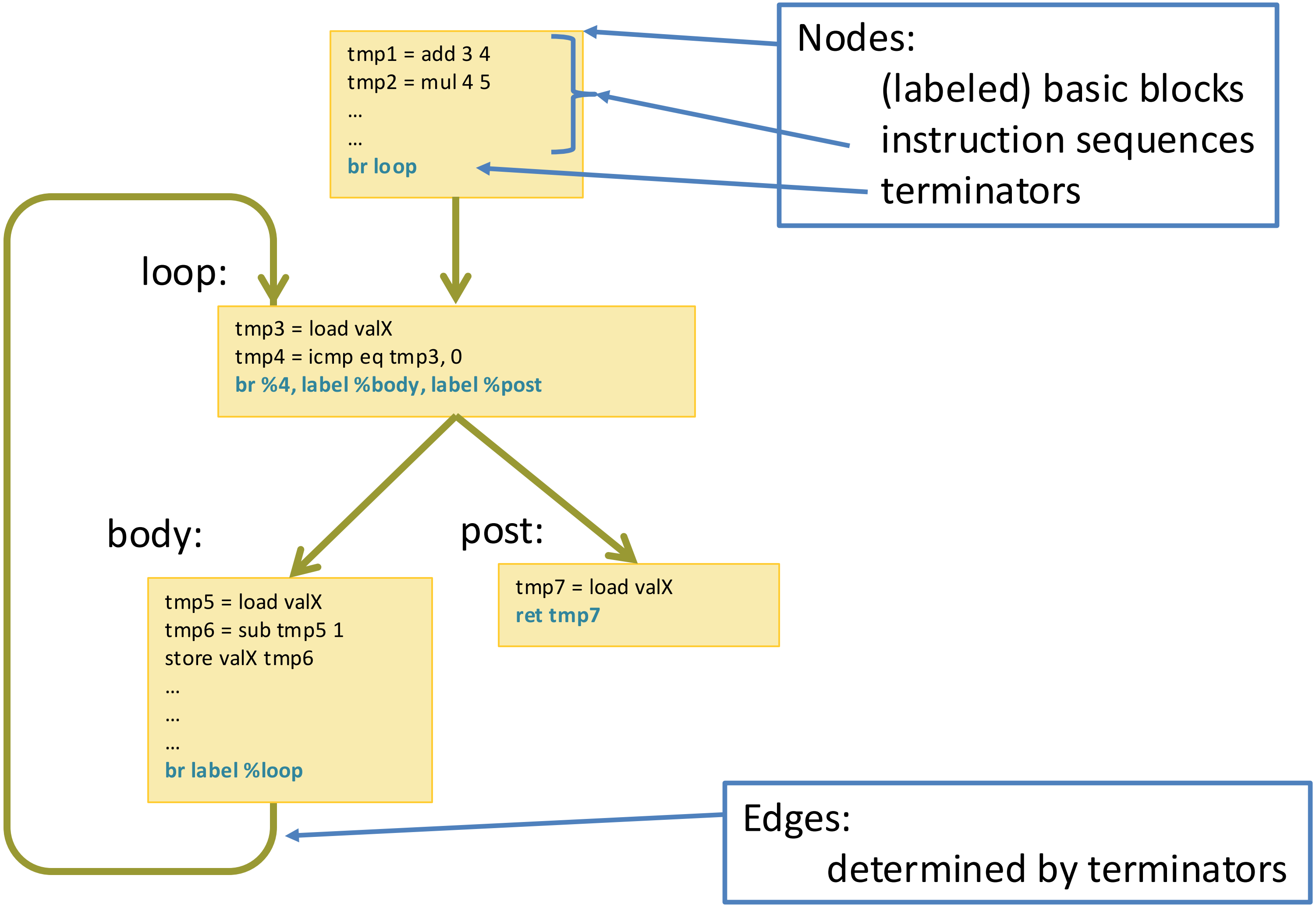
- IR5: “almost” LLVM IR
  - missing *phi-nodes* (explained when we get there)

Basic Blocks

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g., branch or return) the “link”
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target

- Basic blocks can be arranged into a *control-flow graph*
  - Nodes are basic blocks
  - There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.

Control-flow Graphs



See llvm.org

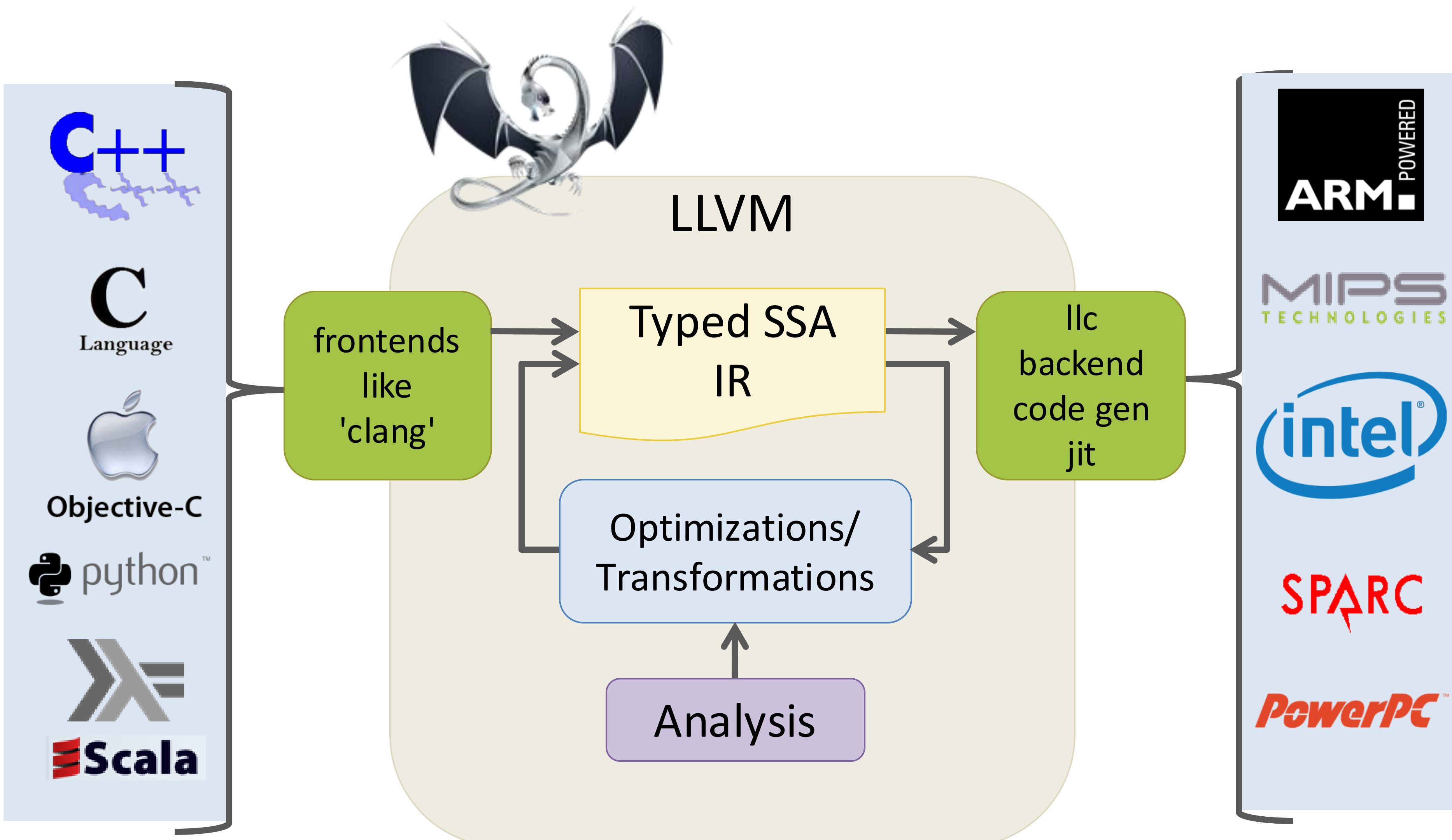
LLVM

Low-Level Virtual Machine (LLVM)

- Open-Source Compiler Infrastructure
  - see llvm.org for full documentation
- Created by Chris Lattner (advised by Vikram Adve) at UIUC
  - LLVM: An infrastructure for Mult-stage Optimization, 2002
  - LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, 2004
- 2005: Adopted by Apple for XCode 3.1
- Front ends:
  - llvm-gcc (drop-in replacement for gcc)
  - Clang: C, objective C, C++ compiler supported by Apple
  - various languages: Swift, ADA, Scala, Haskell, …
- Back ends:
  - x86 / Arm / Power / etc.
- Used in many academic/research projects

LLVM Compiler Infrastructure

[Lattner et al.]



IR3/4/5 vs. LLVM

- “let - in” and
  OCaml-style identifiers:

```
let tmp1 = add 3L 4L in
```

- OCaml-style “let-rec”
  and functions for blocks:

```
let rec entry () =
  let tmp1 = …
and foo () =
  let tmp2 = …
```

- OCaml-style global variables:
  let varX = ref 0L

- Omits let/in and prefixes local identifiers with %:

```
%tmp1 = add i64 3, i64 4
```

- Uses lighter-weight colon notation:

```
entry:
  %tmp1 = …
foo:
  %tmp2 = …
```

- Prefixes globals with @
  define @X = i64 0

Example LLVM Code

- LLVM offers a textual representation of its IR
  - files ending in .ll

factorial64.c

```
#include <stdio.h>
#include <stdint.h>

int64_t factorial(int64_t n) {
  int64_t acc = 1;
  while (n > 0) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

factorial-pretty.ll

```
define @factorial(%n) {
  %1 = alloca
  %acc = alloca
  store %n,  %1
  store 1,  %acc
  br label %start

start:
  %3 = load %1
  %4 = icmp sgt %3, 0
  br %4, label %then, label %else

then:
  %6 = load %acc
  %7 = load %1
  %8 = mul %6, %7
  store %8, %acc
  %9 = load %1
  %10 = sub %9, 1
  store %10, %1
  br label %start

else:
  %12 = load %acc
  ret %12
}
```

Real LLVM

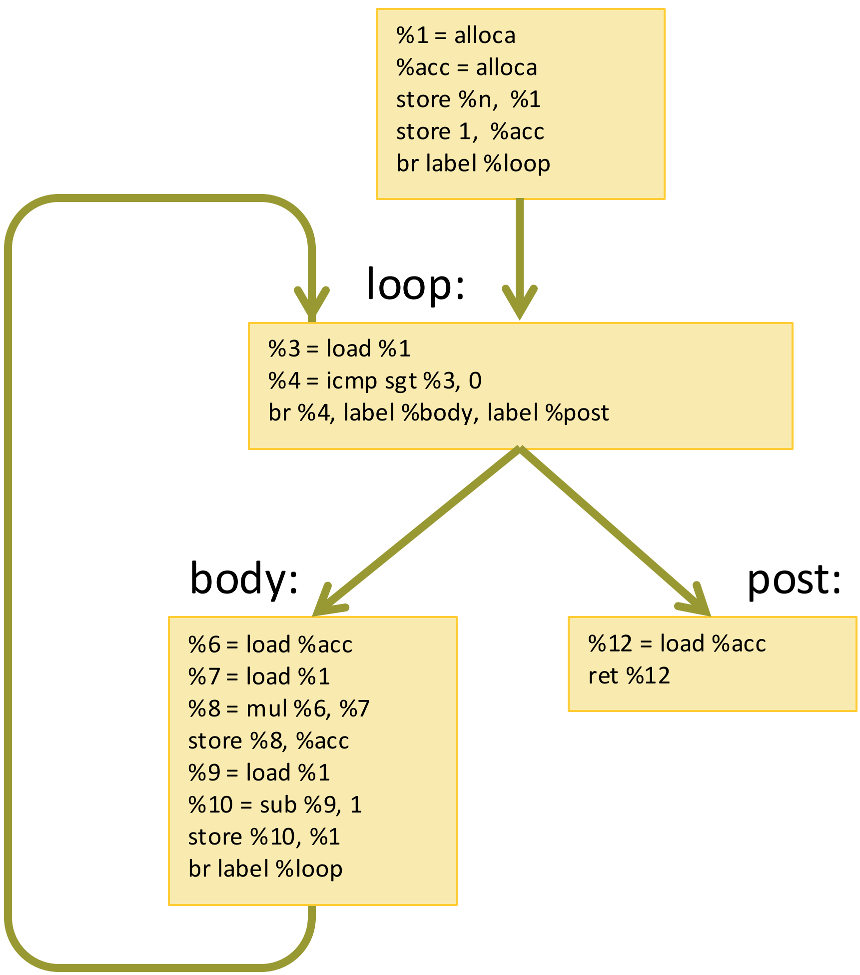
- Decorates values with type information
  i64
  i64*
  i1
- Permits numeric identifiers
- Has alignment annotations
- Keeps track of entry edges for each block:
  preds = %5, %0

factorial.ll

```
; Function Attrs: nounwind ssp
define i64 @factorial(i64 %n) #0 {
  %1 = alloca i64, align 8
  %acc = alloca i64, align 8
  store i64 %n, i64* %1, align 8
  store i64 1, i64* %acc, align 8
  br label %2

; <label>:2                         ; preds = %5, %0
  %3 = load i64* %1, align 8
  %4 = icmp sgt i64 %3, 0
  br i1 %4, label %5, label %11

; <label>:5                         ; preds = %2
  %6 = load i64* %acc, align 8
  %7 = load i64* %1, align 8
  %8 = mul nsw i64 %6, %7
  store i64 %8, i64* %acc, align 8
  %9 = load i64* %1, align 8
  %10 = sub nsw i64 %9, 1
  store i64 %10, i64* %1, align 8
  br label %2

; <label>:11                        ; preds = %2
  %12 = load i64* %acc, align 8
  ret i64 %12
}
```

Example Control-flow Graph

define @factorial(%n) {



}

LL Basic Blocks and Control-Flow Graphs

- LLVM enforces (some of) the basic block invariants syntactically.
- Representation in OCaml:

```
type block = {
    insns : (uid * insn) list;
    term  : (uid * terminator)
}
```

- A *control flow graph* is represented as a list of labeled basic blocks with these invariants:
  - No two blocks have the same label
  - All terminators mention only labels that are defined among the set of basic blocks
  - There is a distinguished, unlabeled, entry block:

```
type cfg = block * (lbl * block) list
```

LL Storage Model: Locals

- Several kinds of storage:
  - Local variables (or temporaries): %uid
  - Global declarations (*e.g.*, for string constants): @gid
  - Abstract locations: references to (stack-allocated) storage created by the alloca instruction
  - Heap-allocated structures created by external calls (*e.g.*, to malloc)

- Local variables:
  - Defined by the instructions of the form %uid = …
  - Must satisfy the ***static single assignment*** invariant
    - *Each %uid appears on the left-hand side of an assignment only once in the entire control flow graph.*
  - The value of a %uid remains unchanged throughout its lifetime
  - Analogous to "let %uid = e in …" in OCaml
- Intended to be an abstract version of machine registers.
- We'll see later how to extend SSA to allow richer use of local variables
  - ***phi nodes***

LL Storage Model: alloca

- alloca instruction allocates stack space and returns a reference to it.
  - The returned reference is stored in local:

    %ptr = alloca type
  - The amount of space allocated is determined by the type

- The contents of the slot are accessed via the load and store instructions:

```
%acc = alloca i64                   ; allocate a storage slot
store i64 3410, i64* %acc           ; store the integer value 3410
%x = load i64, i64* %acc            ; load the value 3410 into %x
```

- Gives an abstract version of stack slots