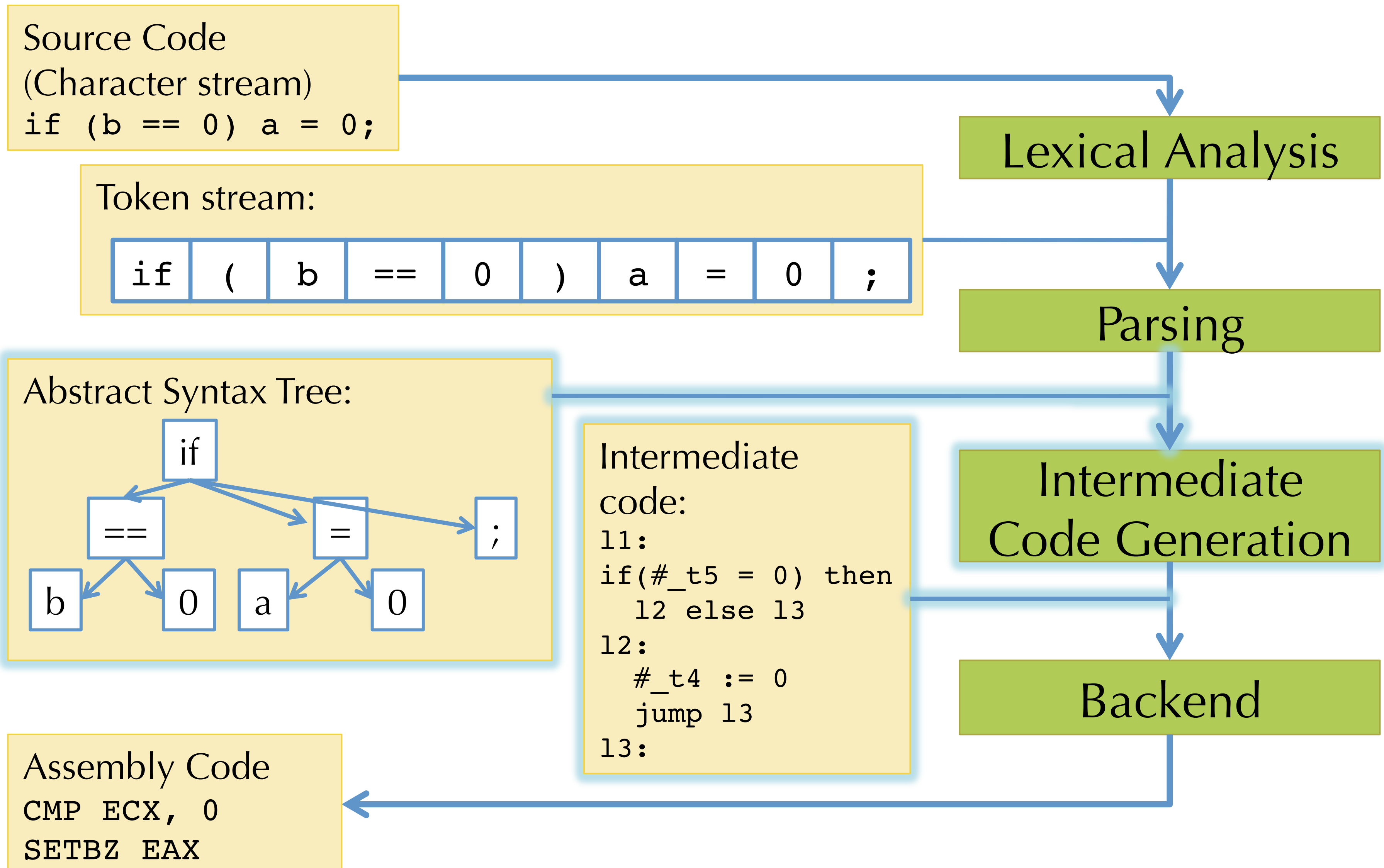
Lecture 10

CIS 341: COMPILERS

Announcements

- Project 3: Compiling Control Flow
  - Due: Monday, February 25th at 11:59pm
- Midterm Exam:
  - Thursday, February 28th
  - In class
  - Examples on the web

Today: Intermediate Representations

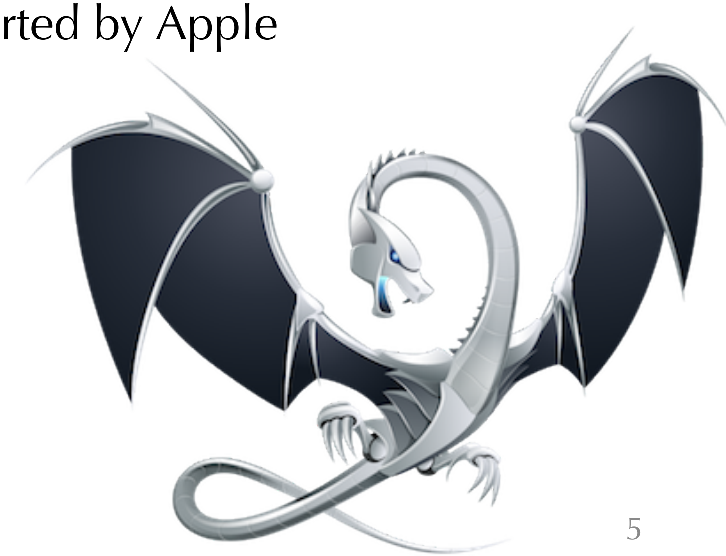


see ll.ml in Project 3

LLVM LITE

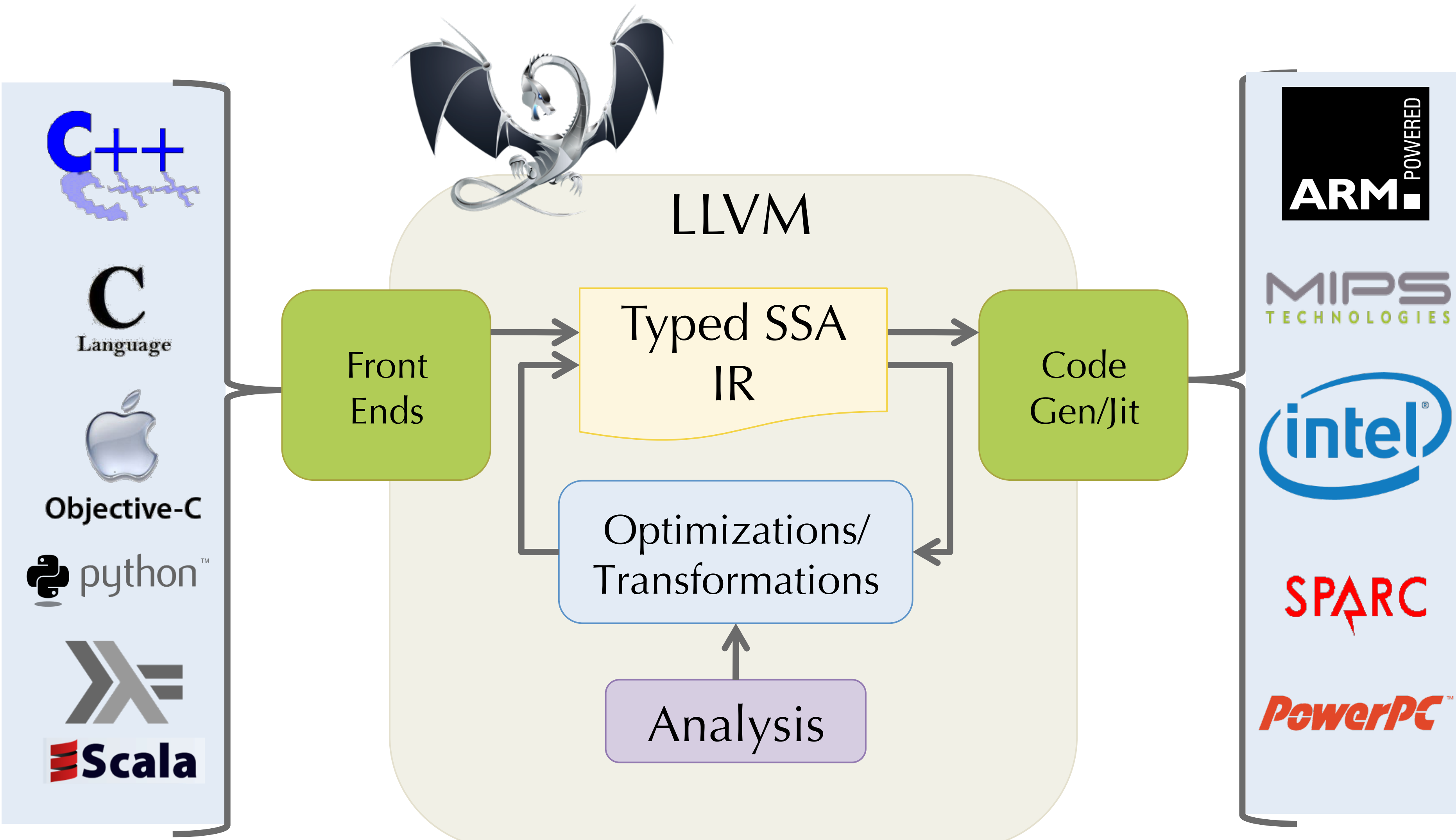
Low-Level Virtual Machine (LLVM)

- Open-Source Compiler Infrastructure
  - see llvm.org for full documntation
- 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: ADA, Scala, Haskell, …
- Back ends:
  - x86 / Arm / Power / etc.
- Used in many academic/research projects
  - Here at Penn: SoftBound, Vellvm



LLVM Compiler Infrastructure

[Lattner et al.]



LL: A Subset of LLVM

```
op     ::= %uid | constant

bop    ::= add | sub | mul | shl | …

cmpop  ::= eq | ne | slt | sle | …

insn ::=
  | %uid = bop op1, op2
  | %uid = alloca
  | %uid = load op1
  | store op1, op2
  | %uid = icmp cmpop op1, op2

terminator ::=
  | ret op
  | br op label %lbl1, label %lbl2
  | br label %lbl
```

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.

LL Basic Blocks and Control-Flow Graphs

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

```
type bblock = {
    label : lbl;
    insns : insn list;
    terminator : terminator
}
```

- A *control flow graph* is represented as a list of 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 entry point label (which labels a block)

```
type prog = {ll_cfg : bblock list; ll_entry : lbl}
```

LL Storage Model: Locals

- Two kinds of storage:
  - Local variables: `%uid`
  - Abstract locations: references to storage created by the `alloca` instruction
- Local variables:
  - Defined by the instructions of the form `%uid = …`
  - Must satisfy the *single static 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.

LL Storage Model: `alloca`

- The alloca instruction allocates a fresh (32-bit) slot and returns a reference to it.
  - The returned reference is stored in local:

```
%ptr = alloca
```

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

```
%acc = alloca          ; allocate a storage slot
store 341, %acc        ; store the integer value 341
%x = load %acc         ; load the value 341 into %x
```

- Gives an abstract version of stack slots
  - Project 3 will show how to compile `alloca`-created storage to stack space.

Example LLVM Code

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

example.c

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

example.ll

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

start:
  %3 = load %1
  %4 = icmp ugt %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, align 4
  ret %12
}
```

Real LLVM

- Decorates values with type information

  `i32`
  `i32*`
  `i1`

- Has alignment annotations

- Keeps track of entry edges for each block: `preds = %start`

```
define i32 @factorial(i32 %n) nounwind uwtable ssp {
entry:
  %1 = alloca i32, align 4
  %acc = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 1, i32* %acc, align 4
  br label %start

start:                                  ; preds = %entry, %else
  %3 = load i32* %1, align 4
  %4 = icmp ugt i32 %3, 0
  br i1 %4, label %then, label %else

then:                                   ; preds = %start
  %6 = load i32* %acc, align 4
  %7 = load i32* %1, align 4
  %8 = mul i32 %6, %7
  store i32 %8, i32* %acc, align 4
  %9 = load i32* %1, align 4
  %10 = sub i32 %9, 1
  store i32 %10, i32* %1, align 4
  br label %start

else:                                   ; preds = %start
  %12 = load i32* %acc, align 4
  ret i32 %12
}
```

SCOPE AND CONTEXTS

Variable Scoping

- Consider the problem of determining whether a programmer-declared variable is in scope.
- See: Project 3 web pages for OAT's scoping rules.
- Issues:
  - Which variables are available at a given point in the program?
  - Shadowing – is it permissible to re-use the same identifier, or is it an error?
- Solution:
  - Contexts

Notation for Scope Checking

- Contexts (using OCaml list notation):

  `G ::= [] | IDENT::G`

- Syntax-directed "functions" that say how to compositionally check the scope
  - One function fore each syntactic category of the grammar.
  - Each function takes an input context (variables that are in scope)
  - May produce an output context (if new variables are introduced)

    $G \vdash$ exp

    $G \vdash$ stmt

    $G \vdash$ vdecl $\Rightarrow G$

    $G \vdash$ vdecl_list $\Rightarrow G$

    $G \vdash$ block $\Rightarrow G$

    $G \vdash$ prog