Lecture 9

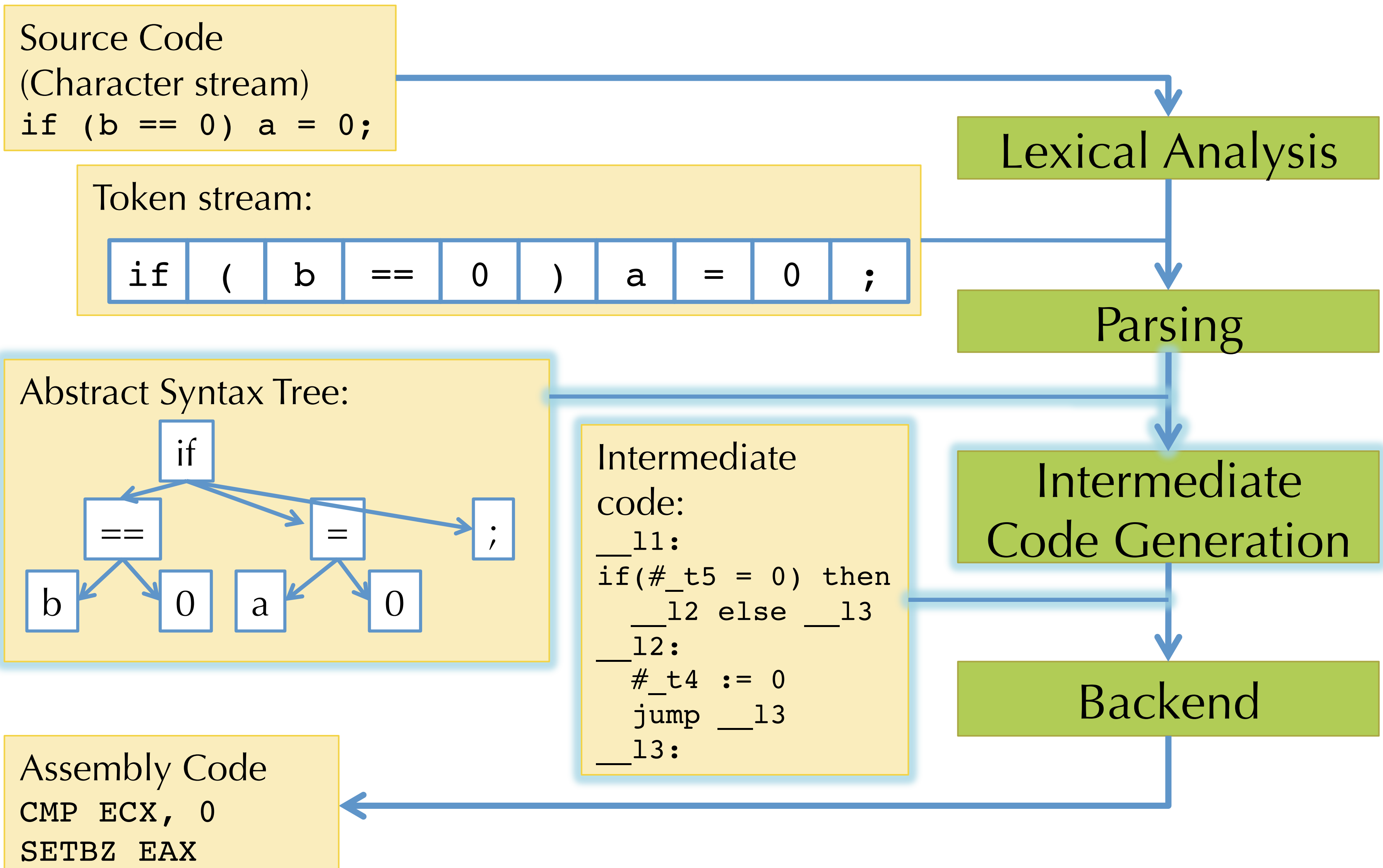
CIS 341: COMPILERS

Announcements

- Project 2: Parsing and Compiling Expressions
  – Due: *Tonight* at 11:59:59
- Project 3: Compiling Control Flow
  – Available soon

INTERMEDIATE REPRESENTATIONS

Today: Intermediate Representations



Directly Translating AST to Assembly

- For simple languages, no need for intermediate representation.
  - e.g. the "boolean" language from earlier, or the language of Project 2

- Main Idea: Maintain invariants
  - e.g. Code emitted for a given expression computes the answer into Eax

- Key Challenges (for Project 2):
  - storing intermediate values needed to compute complex expressions
  - some instructions use specific registers (e.g. shift)
  - logic operations evaluate to exactly 0 or 1

One Simple Strategy

- Compilation is the process of "emitting" instructions into an instruction stream.
- To compile an expression, we recursively compile sub expressions and then process the results.
- Invariants:
  - Compilation of an expression yields its result in Eax
  - Arg (X) is stored in a dedicated register Edx
  - Intermediate values are pushed onto the stack (we can't easily use registers – why?)
  - Stack slot is popped after use (so the space is reclaimed)
- Resulting code is wrapped to comply with cdecl calling conventions:
  - Edx is initialized with the value of X from the stack

- [DEMO] See the compiler.ml for example code.

Why do something else?

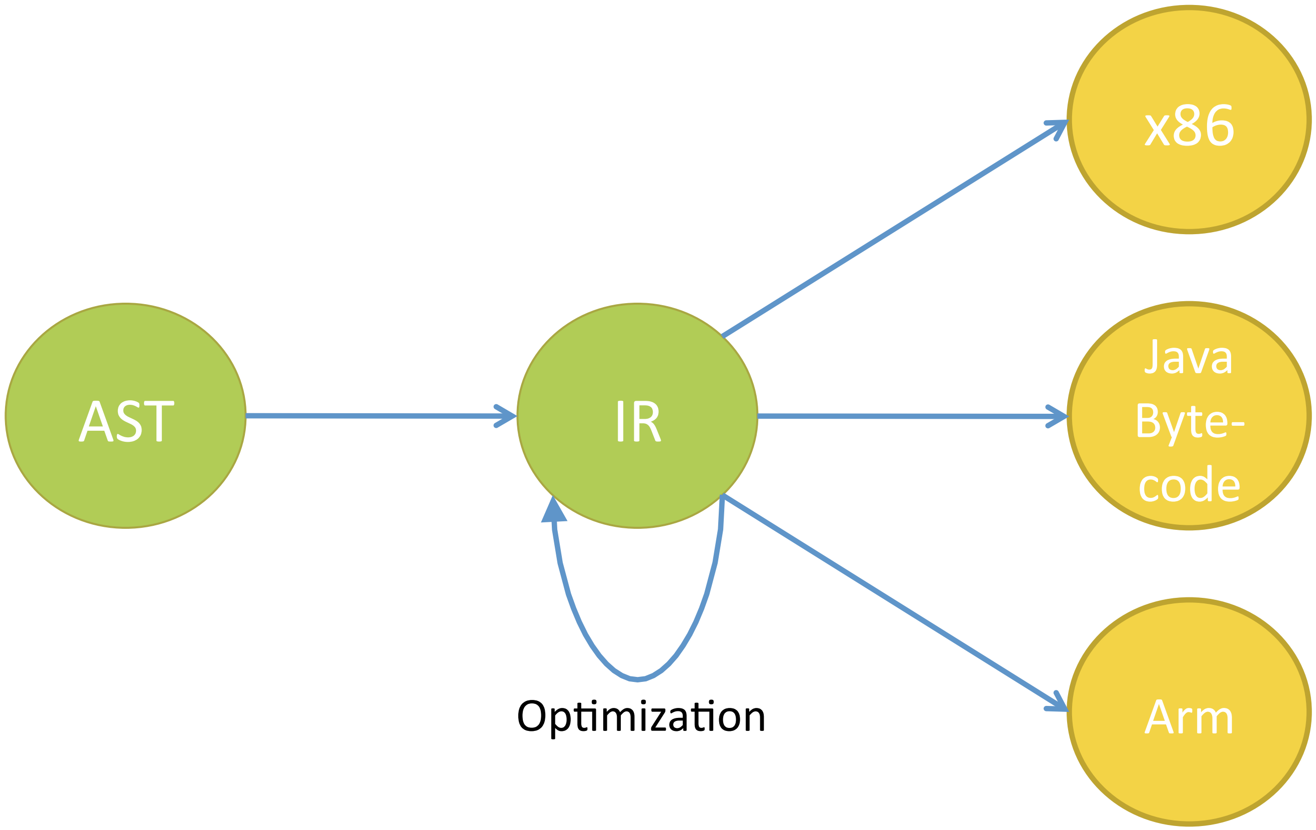
- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.
- It's hard to do optimizations on the resulting assembly code.
  - The representation is too concrete – e.g. it has committed to using certain registers and the stack
- Retargeting the compiler to a new architecture is hard.
  - Target assembly code is hard-wired into the translation

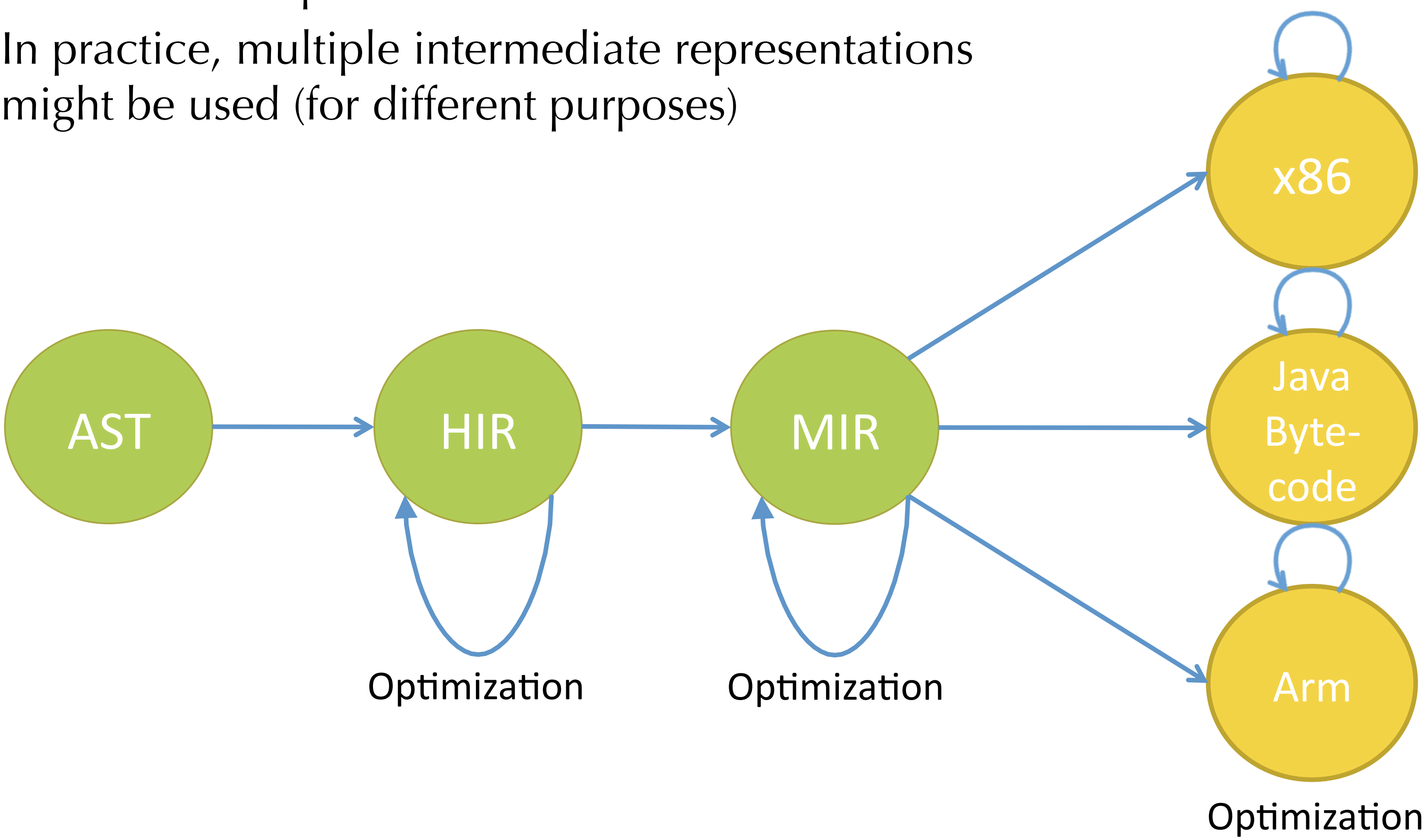
Intermediate Representations (IR's)

- 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 information needed to do 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 level AST might have "while", "for", and "do" loops (and maybe more variants)
  - IR might have only "while" loops and sequencing
  - Translation eliminates "for" and "do"

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

  - Here the notation ⟦`exp`⟧ denotes the "translation" or "compilation" of `exp`

IR's at the extreme

- High-level IR's
  - AST + 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 MUL instruction that doesn't restrict register usage
  - Source structure of the program is lost:
    - Translation to assembly code is trivial
  - Allows low-level optimizations based on target architecture
    - e.g. instruction selection, memory layout, etc.

- What's in between?

Mid-level IR's: Many Varieties

- Intermediate between AST 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 ("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

Two Example IR's

- Two example IR's in more detail… starting from the very basic.

- A (very) simple intermediate representation for the arithmetic language
  - Very high level
  - No control flow
  - See: ssa.ml in lec11.zip

- A simple subset of the LLVM IR
  - LLVM = "Low-level Virtual Machine"
  - Used in Projects 3+

SIMPLE LET-BASED IR

Eliminating Nested Expressions

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

Source
```
((1 + 2) + (3 + (X + 5)))
```

AST
```
Plus(Plus(Int 1, Int2),
     Plus(Int 3, Plus(Arg,
                      Int 5)))
```

IR

?

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

Simple Let Language SLL

```
tmp          // names for temporary values
imm          // 32-bit integer values
exp ::= tmp | imm
op  ::= Add(exp, exp)
      | GetArg
      | ...  // no nested operations
cmd ::= let tmp = op in cmd
      |  return exp
```

Basic idea: restrict the language so that it can only express simple sequences of non-nested expressions.

OCaml-like syntax for ease of reading.

Translation to SLL

- Given this:

```
Plus(Plus(Int 1, Int2),
     Plus(Int 3, Plus(X,
                      Int 5)))
```

- Translate to this desired SLL form:

```
let tmp0 = Add(1, 2) in
let tmp1 = GetArg in
let tmp2 = Add(tmp1, 5) in
let tmp3 = Add(3, tmp2) in
let tmp4 = Add(tmp0, tmp3) in
   return tmp4
```

- Note: translation makes the order of evaluation explicit.

Translation 1: Streams of Instructions

- See the code in sll.ml
  - functions emit_exp and sll_exp

- Compare with
  - emit_exp of compile.ml

- Similar code generation as with the “stack-based”

Continuation Passing Style

$[\![exp]\!]$ `k`

- Idea: parameterize the compilation function by an extra argument k
- k is itself a function, called a *continuation*.
  - The continuation function takes one argument, which is the answer computed by exp
  - The continuation says how to "continue" processing the result.

- Call the translation function with an "initial" continuation that just returns the result.

- Variants:
  - k is a metalevel function that returns a metalevel value ⇒ interpreter
  - k is a metalevel function that returns object level code ⇒ compiler
  - k is an object level function or label ⇒ useful for optimizations

Translation 2: CPS Translation to SLL

- Translation function written in *continuation passing style* (CPS):
  - Note the extra argument 'k' to the ⟦-⟧ translation

```
⟦Cint n⟧ k         =    k   (Imm n)
⟦Plus(l, r)⟧ k     =
        ⟦l⟧(fun ansl ->
             ⟦r⟧(fun ansr ->
                  let tmp = mk_uid () in
                    Let(tmp, Add(ansl, ansr),
                          k (Var tmp)))
```

- Idea: represent "what to do next" with a meta-level function called a continuation – allows for concise expression of compilation algorithms
- Here: the *object level* is the language of sums and the *meta level* is OCaml (the language used for implementation)

Translating Simple SLL to X86

- Translation of SSA to X86 is a bit tricky:
  - Need to find locations for the temporary values (either in registers or on the stack)
  - X86 uses the "two address" version of Add, which is a bit inconvenient

```
let tmp0 = Add(1, 2) in
let tmp1 = GetArg in
let tmp2 = Add(tmp1, 5) in
let tmp3 = Add(3, tmp2) in
let tmp4 = Add(tmp0, tmp3) in
    return tmp4
```

⇒

```
mov $1, %eax
add $2, %eax
mov %edx, %ebx
add $5, %ebx
add $3, %ebx
add %ebx, %eax
```

- In this case, only two non-dedicated registers are needed:
  `tmp0` and `tmp4` ⇒ `%eax`
  `tmp1`, `tmp2`, `tmp3` ⇒ `%ebx`
  Assumes that `%edx` contains the value for `X` (the user-provided arg)

- We'll address efficient automatic register allocation later in the semester…