HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 4:
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
- College office hours
  - Keep eye on OH calendar at https://www.seas.harvard.edu/courses/cs153
- Extension School office hours
  - Poll out seeking your availability
  - Will start next week

Today

- Compiling expressions directly to assembly
- Motivating Intermediate Representations (IRs)
- Simple Let Language

Demo: Compiling Expressions

- See: `compile.c`, `runtime.c`

Directly Translating AST to Assembly

- For simple languages, no need for intermediate representation.
  - e.g. the arithmetic expression language from
- Main Idea: Maintain invariants
  - e.g. Code emitted for a given expression computes the answer into `rax`
- Key Challenges:
  - storing intermediate values needed to compute complex expressions
  - some instructions use specific registers (e.g. shift)

One Simple Strategy

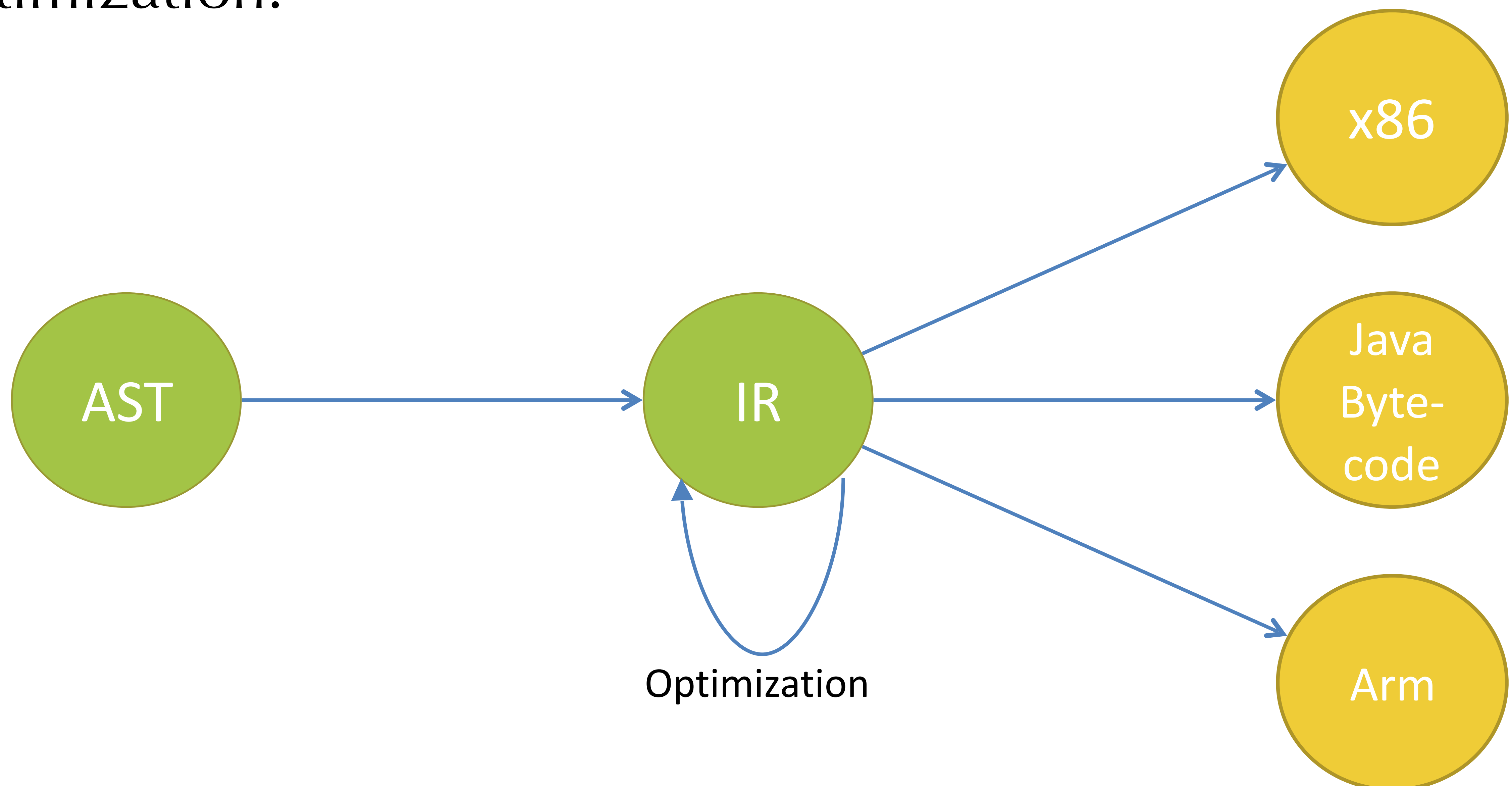
- Compilation is the process of "emitting" instructions into an instruction stream.
- To compile an expression, we recursively compile sub expressions and then process the results.
- Invariants:
  - Compilation of an expression yields its result in `rax`
  - Argument (Xi) is stored in a dedicated operand
  - Intermediate values are pushed onto the stack
  - Stack slot is popped after use (so the space is reclaimed)
- Resulting code is wrapped to comply with cdecl calling conventions
- See the function `compile2` in `compile.ml`

Why intermediate representations?

- These translations are syntax-directed
  - Input syntax uniquely determines output
  - No complex analysis or code transformation is done
  - Works fine for simple languages!
- But…
  - Resulting code quality is poor
  - Richer source language features are hard to encode
    - Structured data types, objects, first-class functions, etc.
  - Hard to optimize the resulting assembly code
    - 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 translation

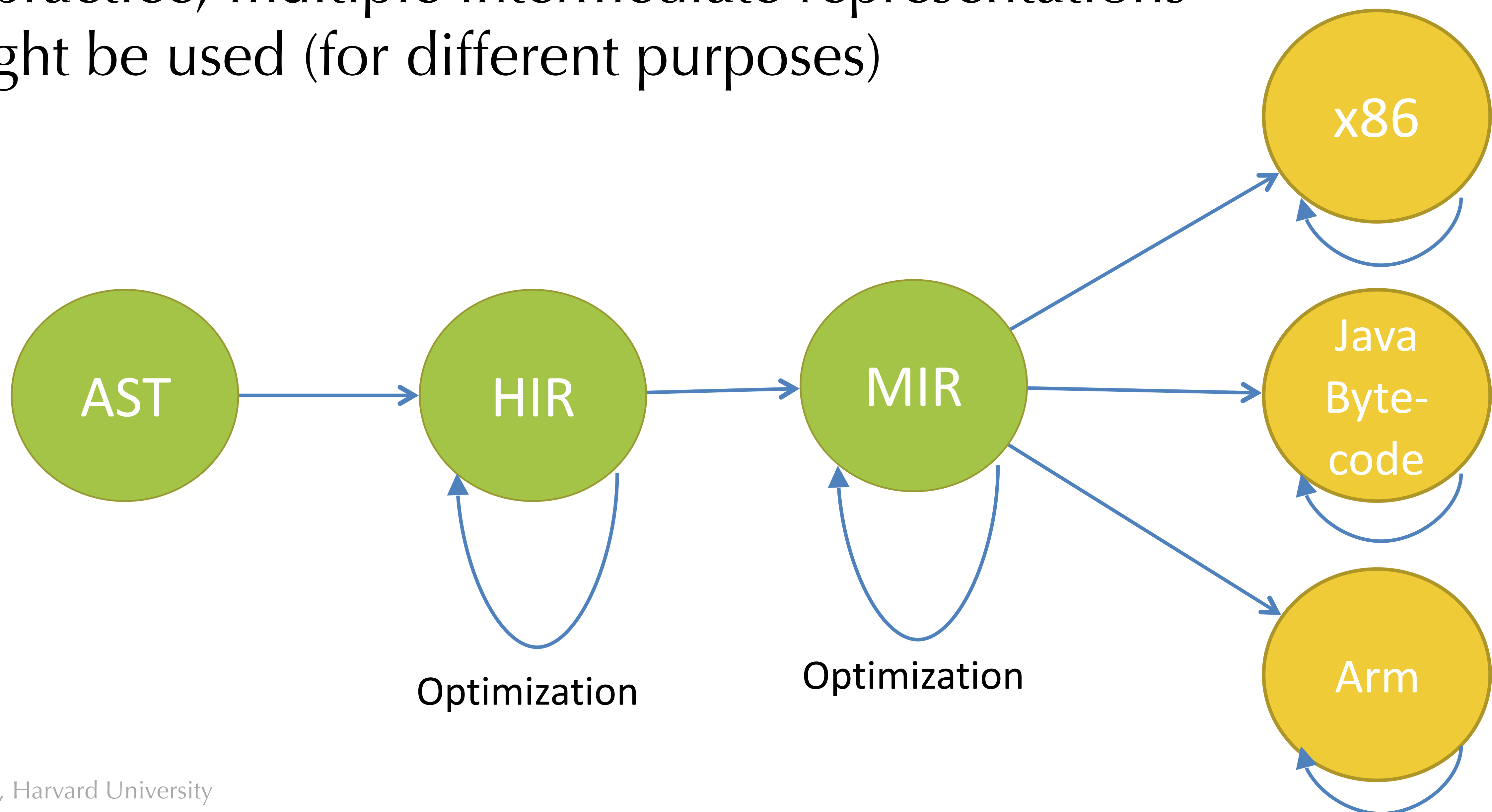
Intermediate Representations (IR's)

- 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)



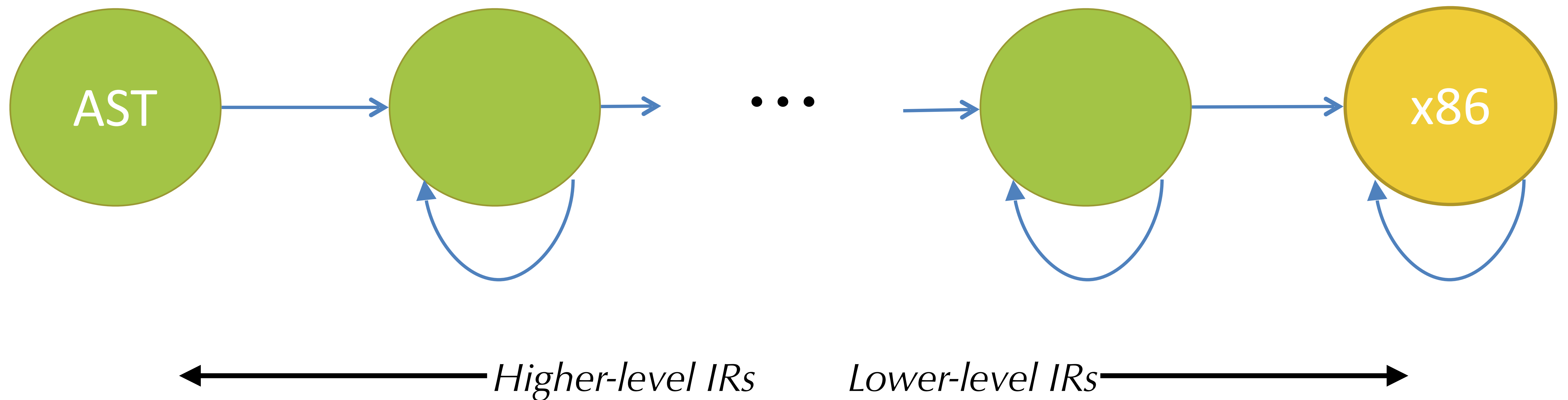
What makes a good IR?

- 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`, `foreach`, `do-while`, `do-until` loops, ...
  - IR might have only `while` loops and sequencing
  - Translation eliminates `for` and `foreach`

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

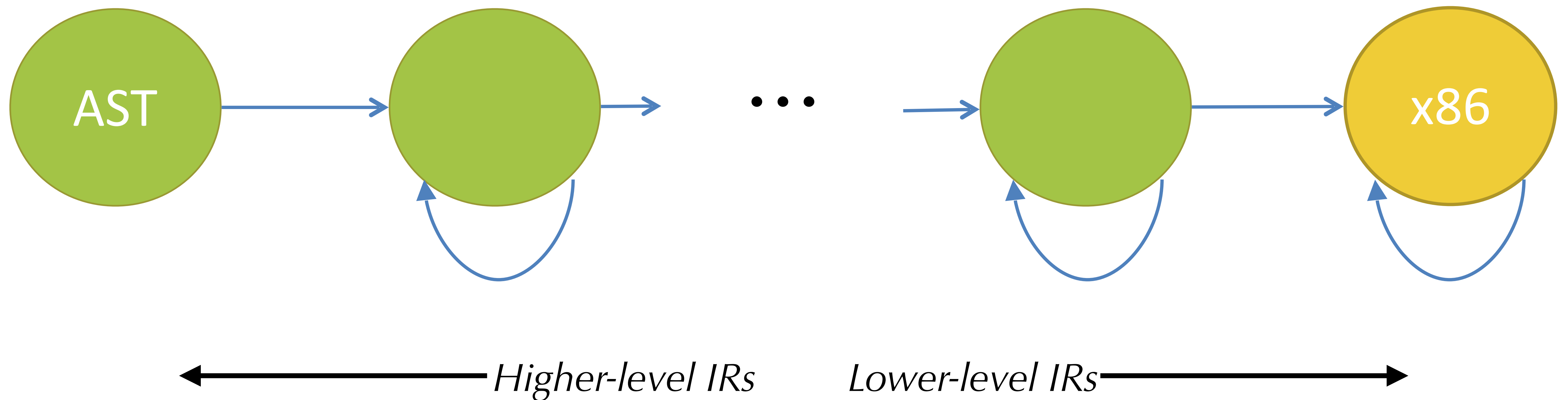
  - Here the notation ⟦cmd⟧ denotes the "translation" or "compilation" of the command cmd.

IR's at the extreme



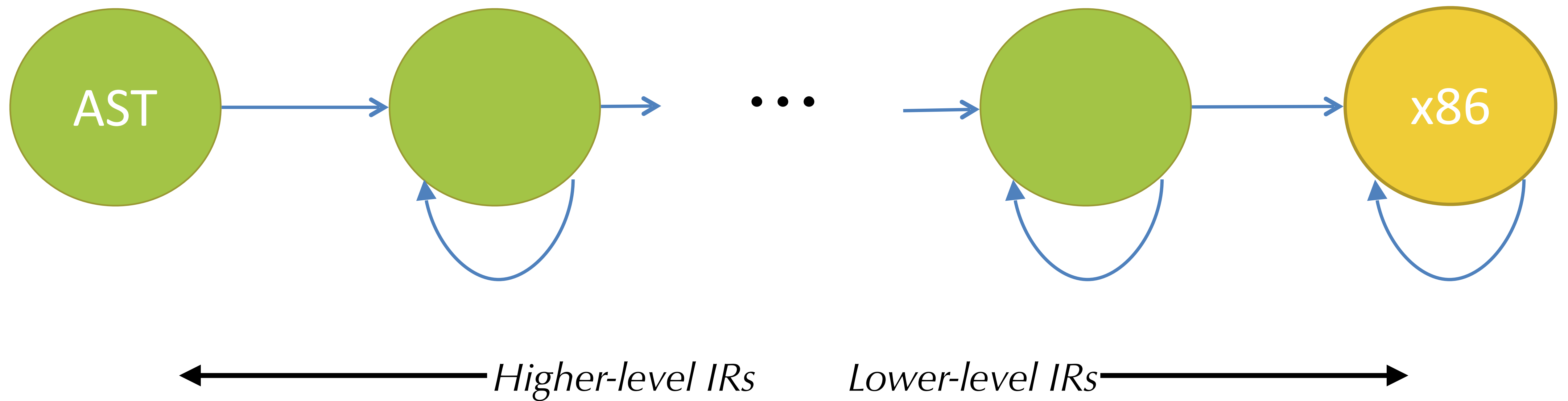
- High-level IR's
  - Abstract syntax + new node types not generated by the parser
    - e.g. Type checking information or disambiguated syntax nodes
  - Typically preserves the high-level language constructs
    - Structured control flow, variable names, methods, functions, etc.
    - May do some simplification (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

IR's at the extreme



- 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.

IR's at the extreme



- What's in between?