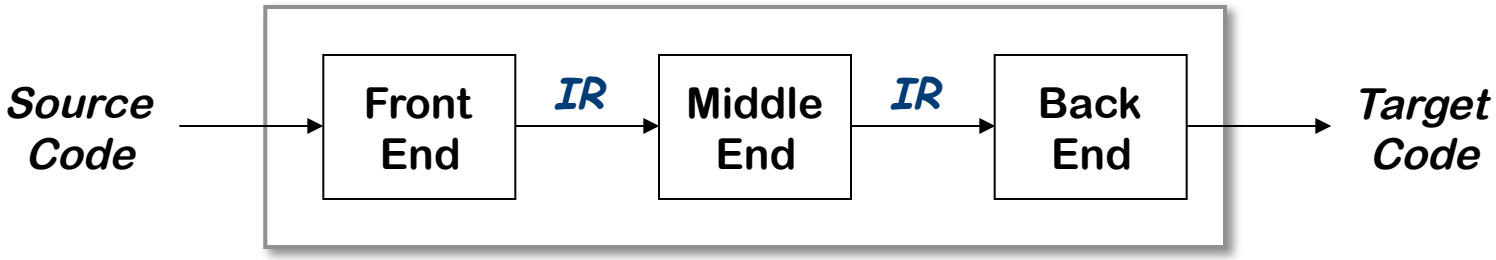
Intermediate Representations
Part I

Where In The Course Are We?

- The rest of the course will focus on issues where the compiler writer needs to choose among alternatives
  - → The choices matter; they affect the quality of compiled code
  - → There may be no "best answer" or "best practice"

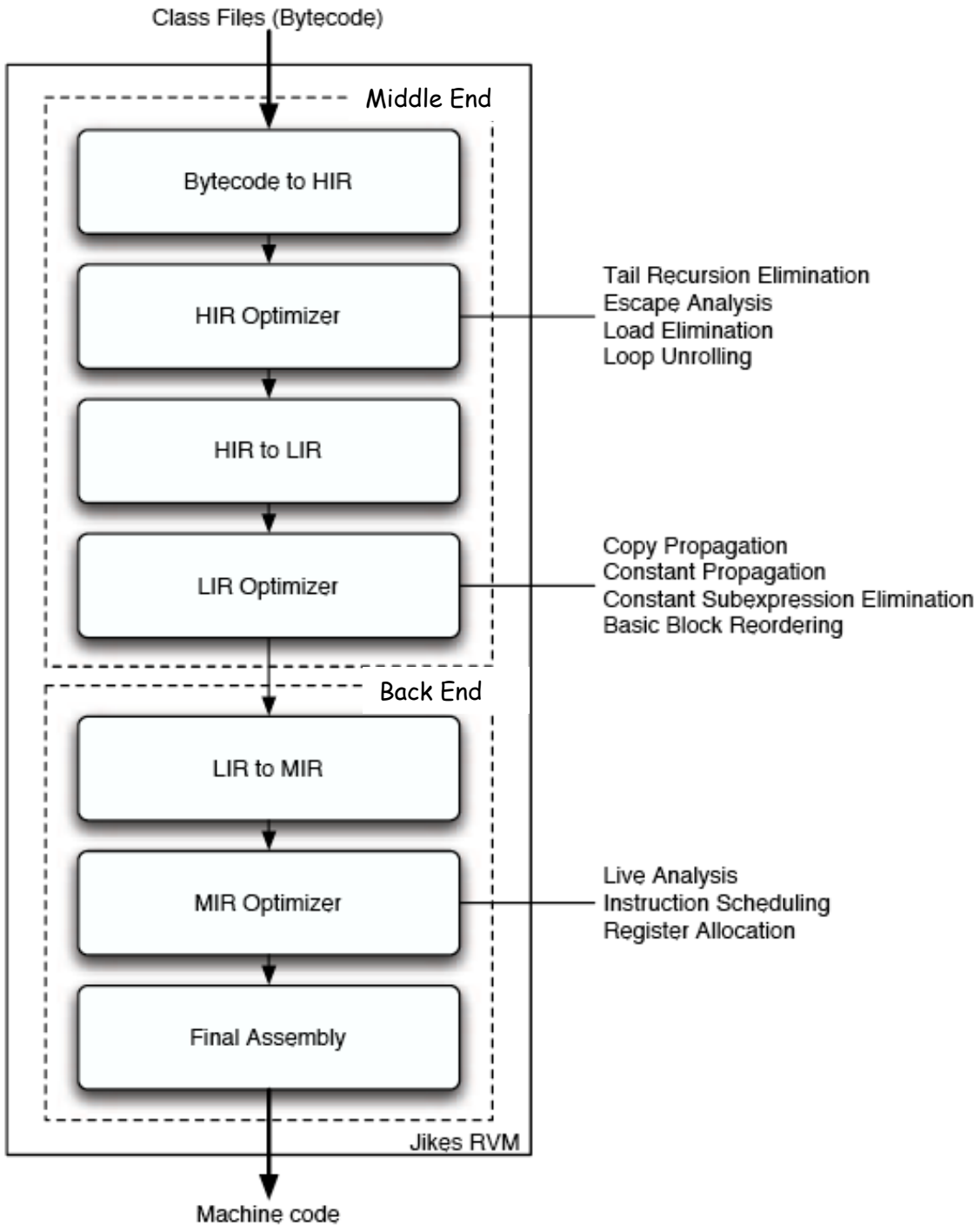
Intermediate Representations



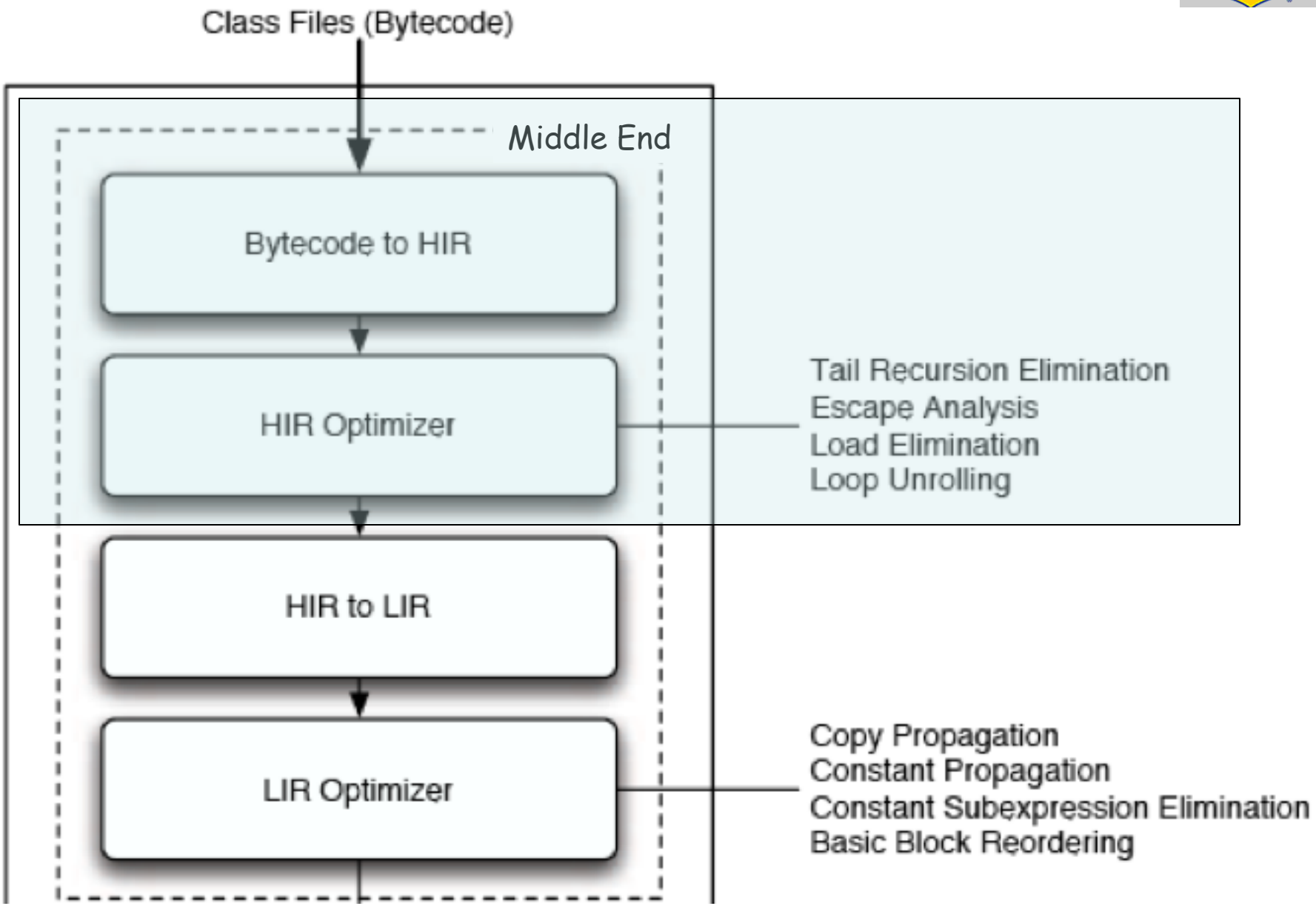
- Front end - produces an intermediate representation (*IR*)
- Middle end - transforms the *IR* into an equivalent *IR* that runs more efficiently
- Back end - transforms the *IR* into native code

- *IR* encodes the compiler's knowledge of the program
- Middle end usually consists of many passes

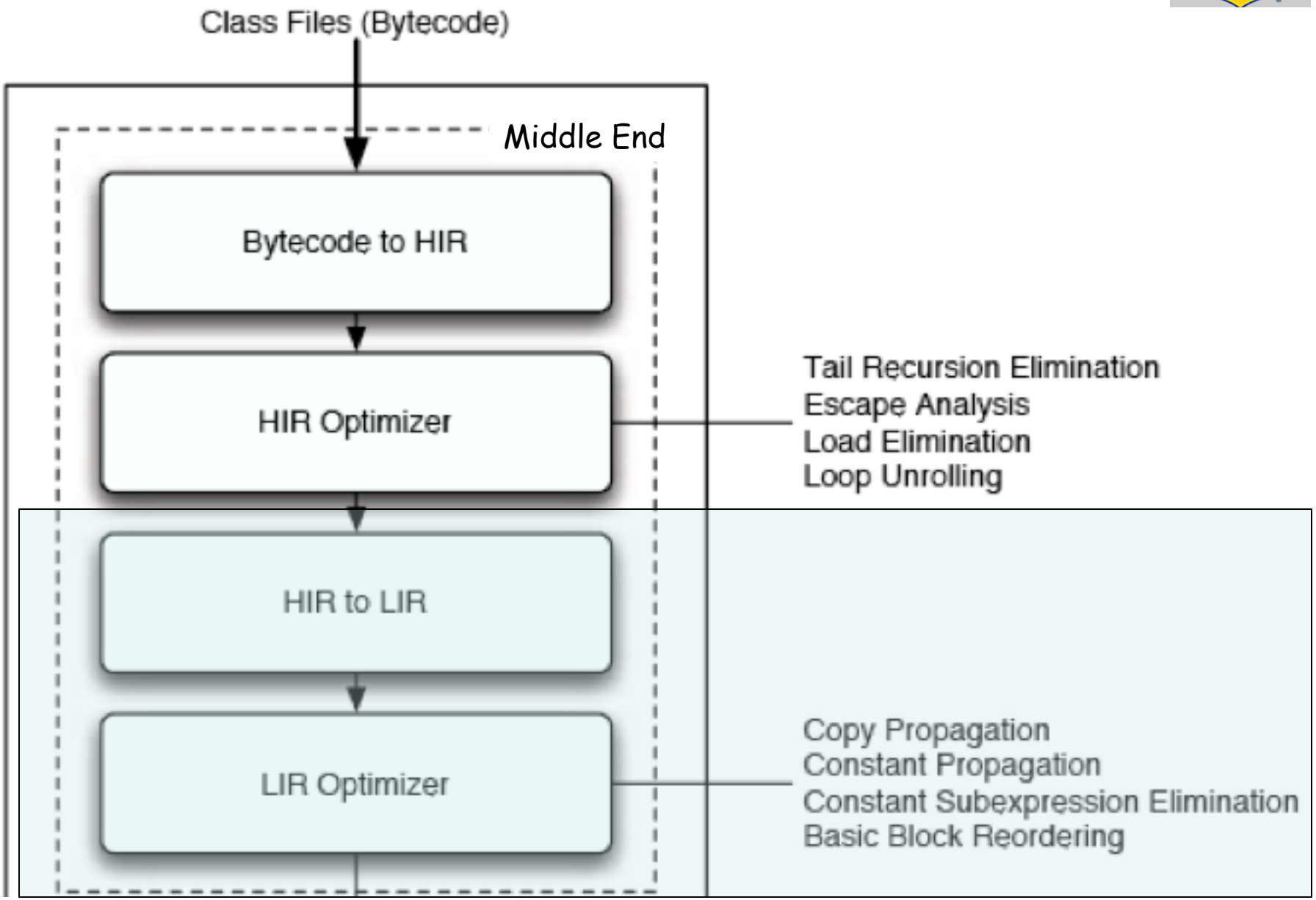
JikesRVM


Class Files (Bytecode)
Middle End
Bytecode to HIR
HIR Optimizer
HIR to LIR
LIR Optimizer
Back End
LIR to MIR
MIR Optimizer
Final Assembly
Jikes RVM
Machine code
Tail Recursion Elimination
Escape Analysis
Load Elimination
Loop Unrolling
Copy Propagation
Constant Propagation
Constant Subexpression Elimination
Basic Block Reordering
Live Analysis
Instruction Scheduling
Register Allocation

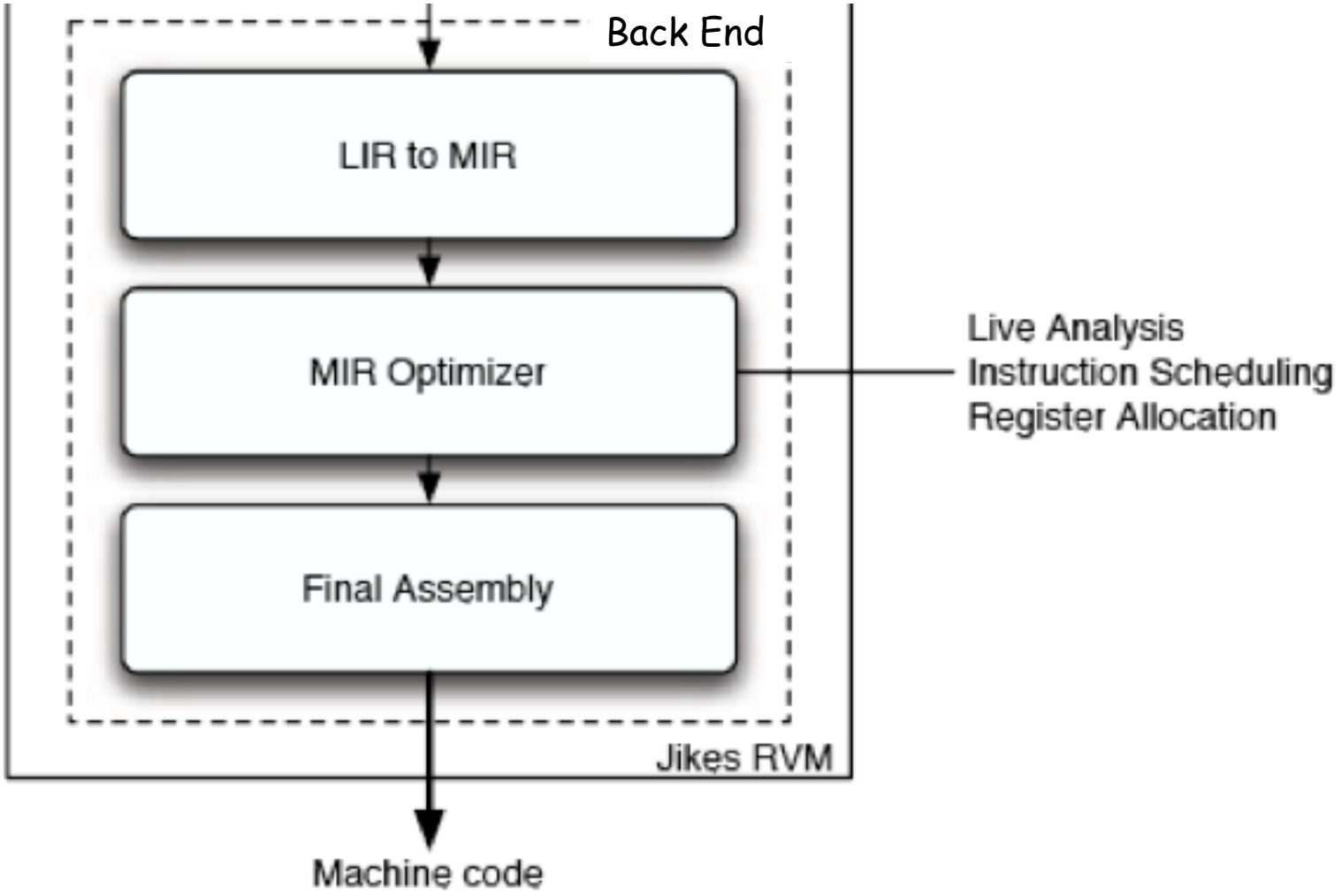
JikesRVM


Class Files (Bytecode)
Middle End
Bytecode to HIR
HIR Optimizer
HIR to LIR
LIR Optimizer
Tail Recursion Elimination
Escape Analysis
Load Elimination
Loop Unrolling
Copy Propagation
Constant Propagation
Constant Subexpression Elimination
Basic Block Reordering

JikesRVM


Class Files (Bytecode)
Middle End
Bytecode to HIR
HIR Optimizer
Tail Recursion Elimination
Escape Analysis
Load Elimination
Loop Unrolling
HIR to LIR
LIR Optimizer
Copy Propagation
Constant Propagation
Constant Subexpression Elimination
Basic Block Reordering

JikesRVM


Back End
LIR to MIR
MIR Optimizer
Final Assembly
Live Analysis
Instruction Scheduling
Register Allocation
Jikes RVM
Machine code

Intermediate Representations

- Decisions in *IR* design affect the speed and efficiency of the compiler

- The importance of different properties varies between compilers
  - → Selecting an appropriate *IR* for a compiler is critical

Some important *IR* properties

- Ease of generation
  - → speed of compilation
- Ease of manipulation
  - → improved passes
- Procedure size
  - → compilation footprint
- Level of abstraction
  - → improved passes

Types of Intermediate Representations

Three major categories

- Structural
- Linear
- Hybrid

Types of Intermediate Representations

Three major categories

- Structural ← Examples: Trees, DAGs
  - → Graphically oriented
  - → Heavily used in source-to-source translators
  - → Tend to be large
- Linear
- Hybrid

Types of Intermediate Representations

Three major categories

- Structural
- Linear

Examples: 3 address code, Stack machine code

  - → Pseudo-code for an abstract machine
  - → Level of abstraction varies
  - → Simple, compact data structures
  - → Easier to rearrange
- Hybrid

Types of Intermediate Representations

Three major categories

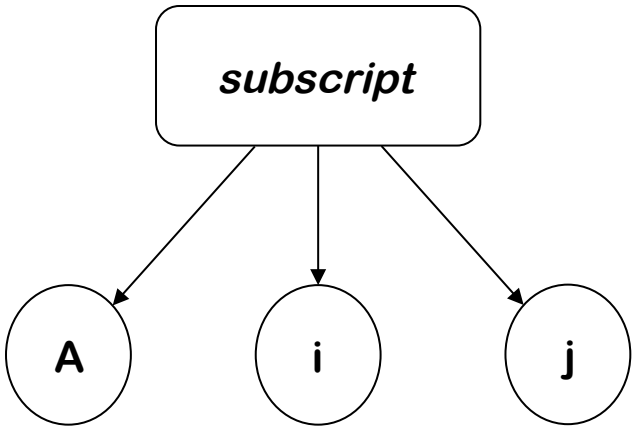
- Structural
- Linear
- Hybrid

Examples:
Control Flow Graph

→ Combination of graphs and linear code

Level of Abstraction

- Two different representations of an array ref:



High level AST:
Good for memory disambiguation

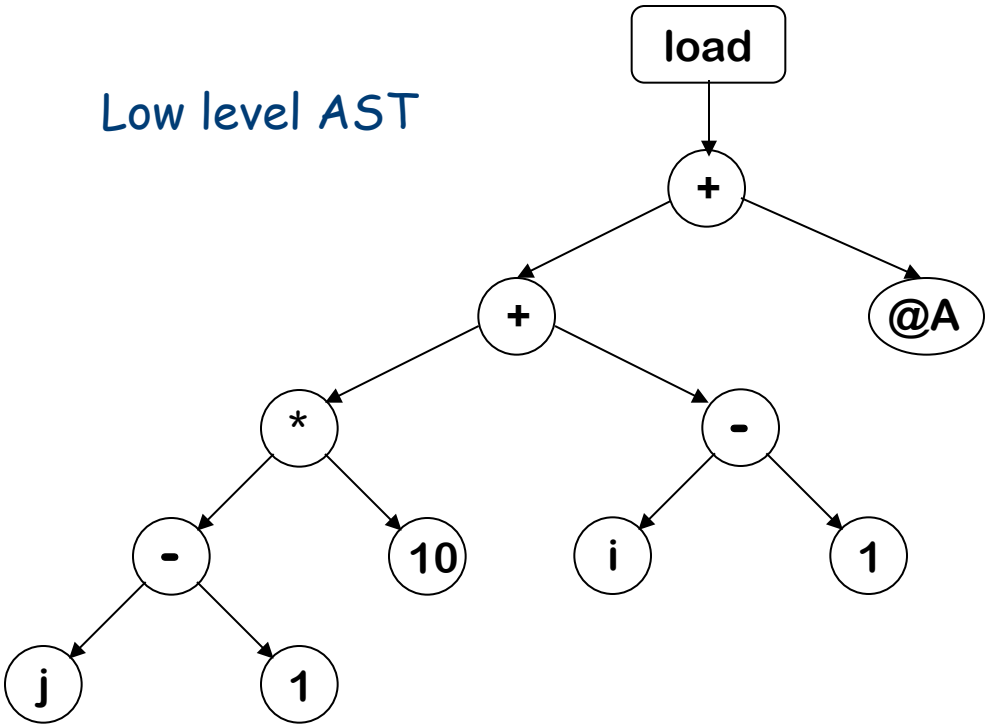
```
loadI  1       => r1
sub    rj, r1  => r2
loadI  10      => r3
mult   r2, r3  => r4
sub    ri, r1  => r5
add    r4, r5  => r6
loadI  @A      => r7
add    r7, r6  => r8
load   r8      => rAij
```

Low level linear code:
Good for address calculation

Level of Abstraction

- Structural *IR*s are usually considered high-level
- Linear *IR*s are usually considered low-level
- Not necessarily true:

Low level AST



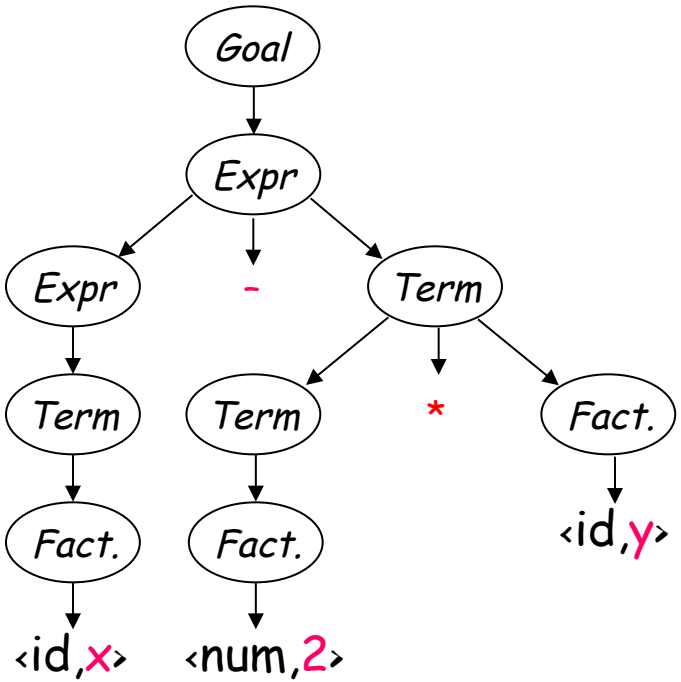

```
loadArray A,i,j
```

High level linear code

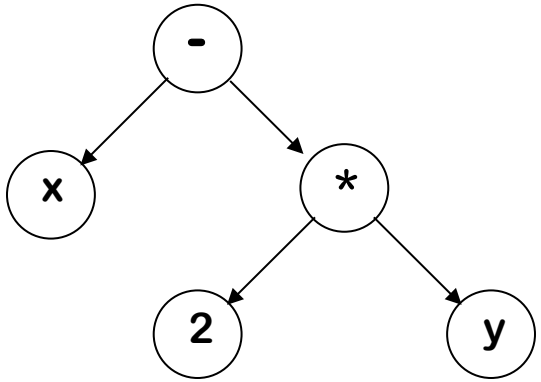
Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed

x - 2 * y



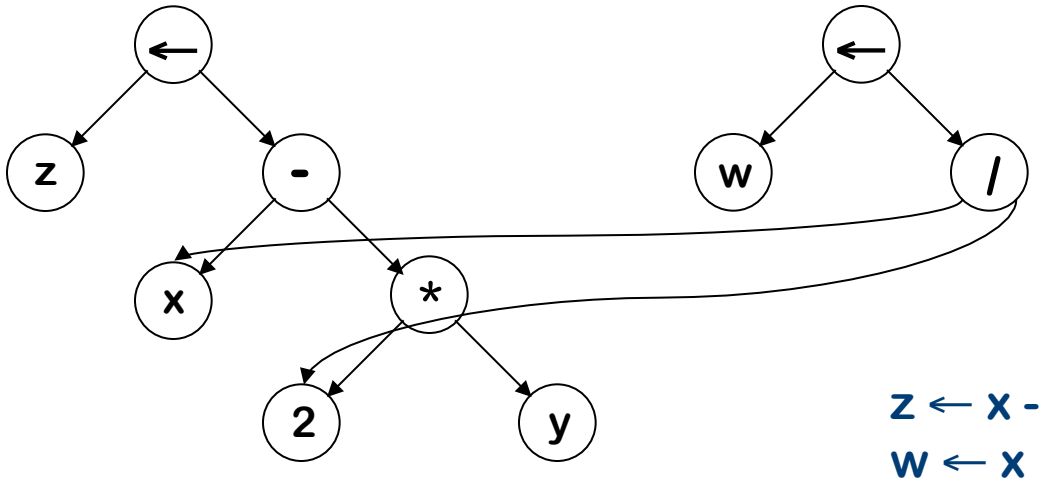
Parse Tree



Abstract Syntax Tree

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



z ← x - 2 * y
w ← x / 2

- Makes sharing explicit
- Encodes redundancy

With two copies of the same expression, the compiler might be able to arrange the code to evaluate it only once.