Lecture 3:

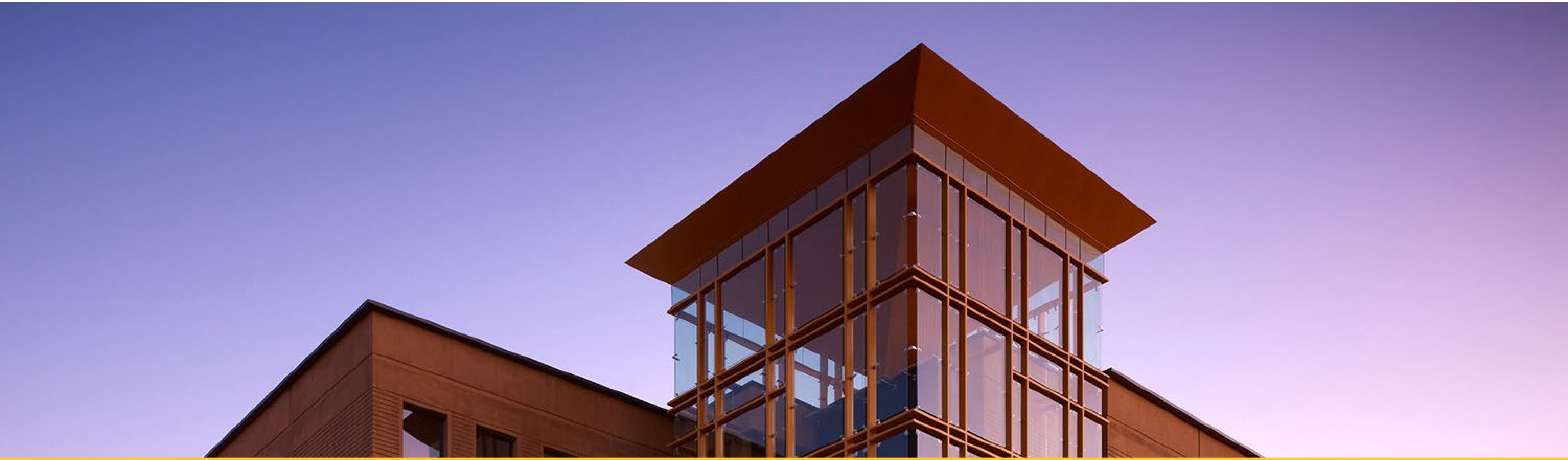
Language and Compiler Basics (I)

Sitao Huang

sitaoh@uci.edu

January 18, 2022

Slide courtesy of Prof. Vikram Adve, UIUC, CS 426: Compiler Construction



Tentative Schedule

- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): ***Language and Compiler Basics***
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7** (2/15, 2/17): Compiler Optimizations for Accelerators
- **Week 8** (2/22, 2/24): Machine Learning Compilers
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*

Languages and Compilers for Hardware Accelerators

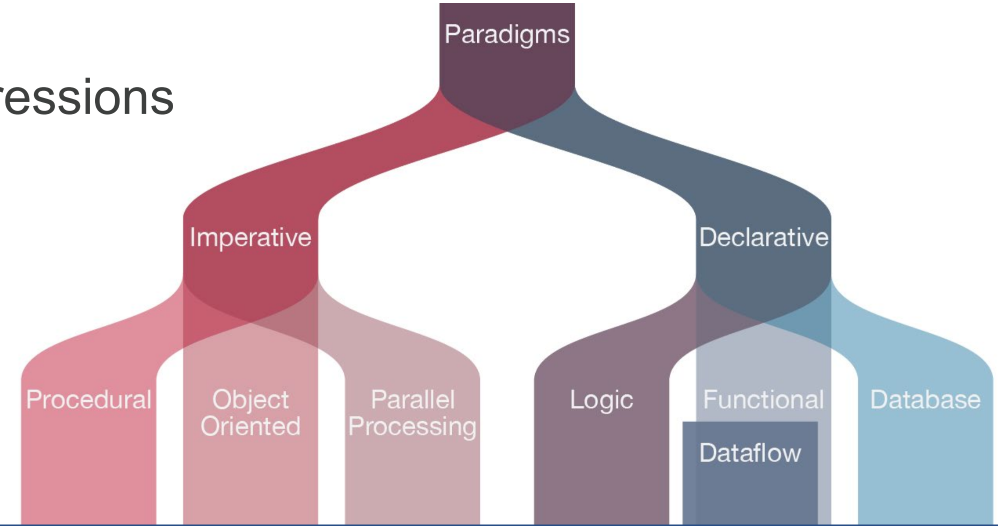
- **Programming languages**: formal language comprising a set of strings, used to implement algorithms
- Provide an abstraction for underlying hardware
- Provide a set of general operators to describe a range of applications
- Primitive elements of programming languages
  - Syntax: rules that define the correct combination of symbols

Lisp example: (from Wikipedia)

```
expression ::= atom | list
atom       ::= number | symbol
number     ::= [+-]?['0'-'9']+
symbol     ::= ['A'-'Z''a'-'z'].*
list       ::= '(' expression* ')'
```

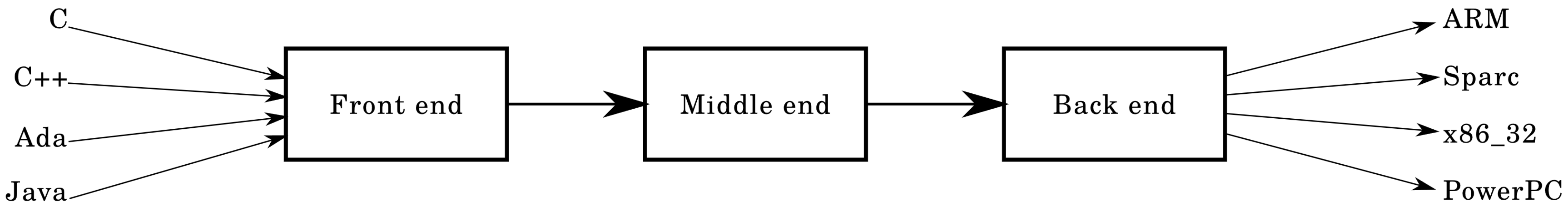
- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace);
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

  - Semantics: meaning of languages
  - Type system: how a language classify values and expressions
  - Standard library and run-time system
- Categories
  - Typed vs untyped languages, static vs dynamic typing
  - Functional programming vs imperative programming
  - ……



Languages and *Compilers* for Hardware Accelerators

- **Compilers**: a special program that processes code in a particular programming language and translates input code into the code in another or the same language.
- Categories
  - Just-In-Time (JIT) compiler, Ahead-of-Time (AOT) compiler
  - Source-to-source compiler
  - ……
- Three-stage compiler structure
  - Front end: translate source code to intermediate representation (IR), manages symbol table
  - Middle end: compiler analysis (e.g., data-flow analysis) and optimization (e.g., loop transformation)
  - Back end: architecture specific optimizations, code generation



Compilers

- **Compilers**: a special program that processes code in a particular programming language and translates input code into the code in another or the same language.
- Examples:
  - C++ to x86 assembly
  - C++ to C
  - Java to JVM bytecode
  - C to C (or any language to itself)
    - Make code faster/smaller, instrumentation, etc.

Use of Compiler Technology

- ***Code generation***: To translate a program in a high-level language to machine code for a particular processor
- ***Optimization***: Improve program performance for a given target machine
- ***Text formatters***: translate TeX to dvi, dvi to postscript, etc.
- ***Interpreters***: "on-the-fly" translation of code, e.g., Java, Perl, csh, Postscript
- ***Automatic parallelization or vectorization***
- ***Debugging aids***: e.g., purify for debugging memory access errors
- ***Performance instrumentation***: e.g., `-pg` option of cc or gcc for profiling
- ***Security***: JavaVM uses compiler analysis to prove safety of Java code
- ***Many more cool uses!*** Hardware design / synthesis, power management, code compression, fast simulation of architectures, transparent fault-tolerance, . . .

Key: *Ability to extract properties of a program (analysis), and optionally transform it (synthesis/transformation)*

Source: Prof. Vikram Adve, UIUC, CS 426: Compiler Construction

A Code Optimization Example

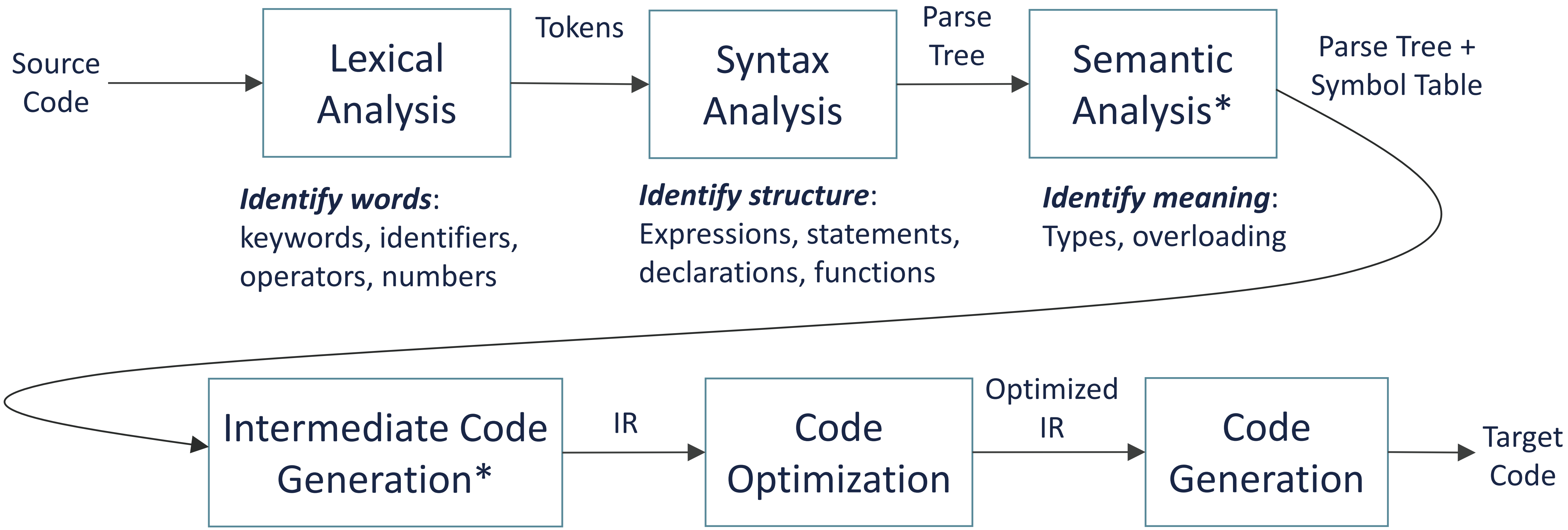
What machine-independent optimizations are applicable to the following C example? When are they safe?

```
int main() {
   ...
   X = ...;
   N = 1; i = 1;
   while (i <= 100) {
     j = i * 4;
     N = j * N;
     Y = X * 2.0;
     A[i] = X * 4.0;
     B[j] = Y * N;
     C[j] = N * Y * C[j];
     i = i + 1;
    }
    printArray(B, 400);
    printArray(C, 400);
 }
```

```
X = ...
N = 1;
j = 4;                 // Induction Variable Substitution (SUBST),
                       // Strength Reduction
Y = X * 2.0;           // Loop-Invariant Code Motion (LICM)
while (j <= 400) {     // Linear Function Test Replacement (LFTR)
                       // Dead Code Elimination (DCE) for i * 4
  N = j * N;
                       // DCE of A, since A not aliased to B or C
  tmp = Y * N;
  B[j] = tmp;
  C[j] = tmp * C[j];   // Common Subexpression Elimination (CSE)
  j = j + 4;           // Induction Variable Substitution,
                       // Strength Reduction
}
printArray(B, 400);
printArray(C, 400);
```

Source: Prof. Vikram Adve, UIUC, CS 426: Compiler Construction

General Structure of a Compiler



Identify words: keywords, identifiers, operators, numbers

Identify structure: Expressions, statements, declarations, functions

Identify meaning: Types, overloading

Generate IR: Synthesize internal representation from parse tree

Transform IR: Modify code to improve performance

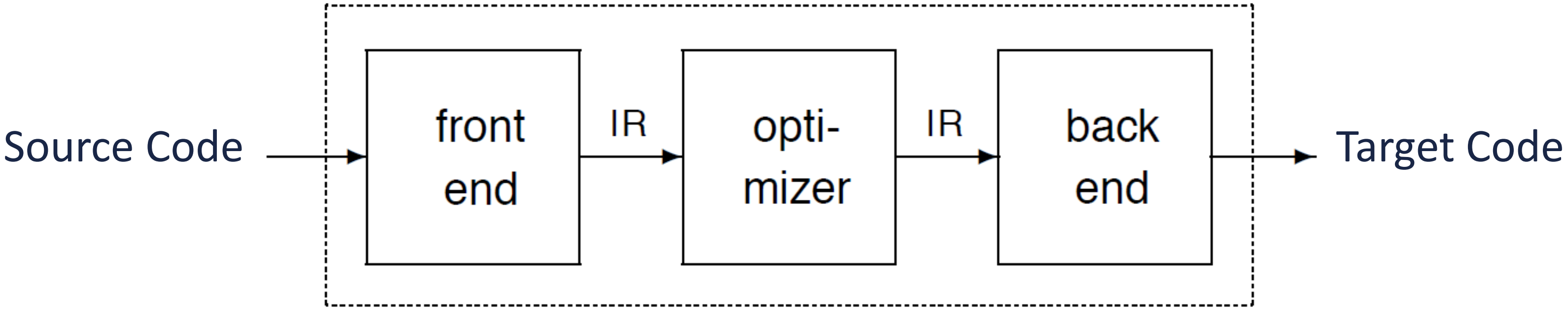
Generate target code: Translate IR to target language

**** Order varies***

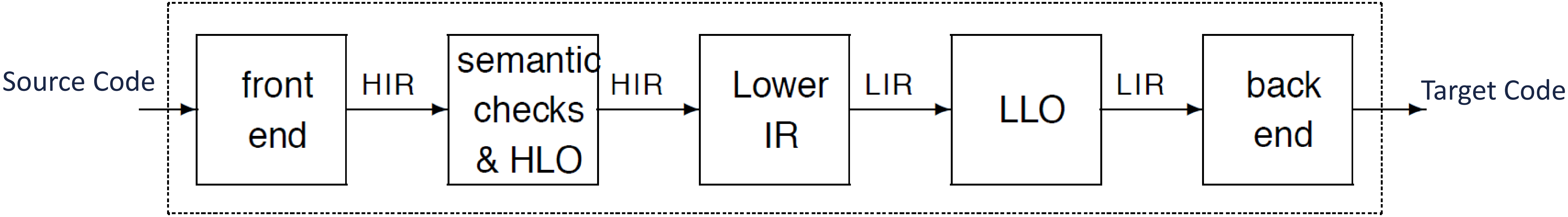
Intermediate Representation (IR)

- Intermediate Representation (IR) encodes all knowledge the compiler has derived about the source program.

Simple compiler structure



More typical compiler structure



Components and Design Goals for an IR

Components of IR

- *Code representation*: actual statements or instructions
- *Symbol table* with links to/from code
- *Analysis information* with mapping to/from code
- *Constants table*: strings, initializers, ...
- *Storage map*: stack frame layout, register assignments

Design Goals for an IR?

- No universally good IR
- The right choice depends strongly on the goals of the compiler

Common Code and Analysis Representations

- Code Representations
  - Usually have ***only one*** at a time
  - Common alternatives:
    - Abstract Syntax Tree (AST)
    - SSA form + CFG
    - 3-address code [+ CFG]
    - Stack code
  - Influences:
    - semantic information
    - types of optimizations
    - ease of transformations
    - speed of code generation
    - size
- Analysis Representations
  - May have ***several*** at a time
  - Common choices:
    - Control Flow Graph (CFG)
    - Symbolic expression DAGs
    - Data dependence graph (DDG)
    - SSA form
    - Points-to graph / Alias sets
    - Call graph
  - Influences:
    - analysis capabilities
    - optimization capabilities

Categories of IRs By Structure

- Graphical IRs
  - Trees, directed graphs, DAGs
  - Node/edge data structures tend to be large
  - Harder to rearrange
  - Examples: AST, CFG, SSA, DDG, Expression DAG, Point-to graph
- Linear IRs
  - Pseudo-code for abstract machine
  - Many possible semantic levels
  - Simple, compact data structures
  - Easier to rearrange
  - Examples: 3-address, 2-address, accumulator, or stack code

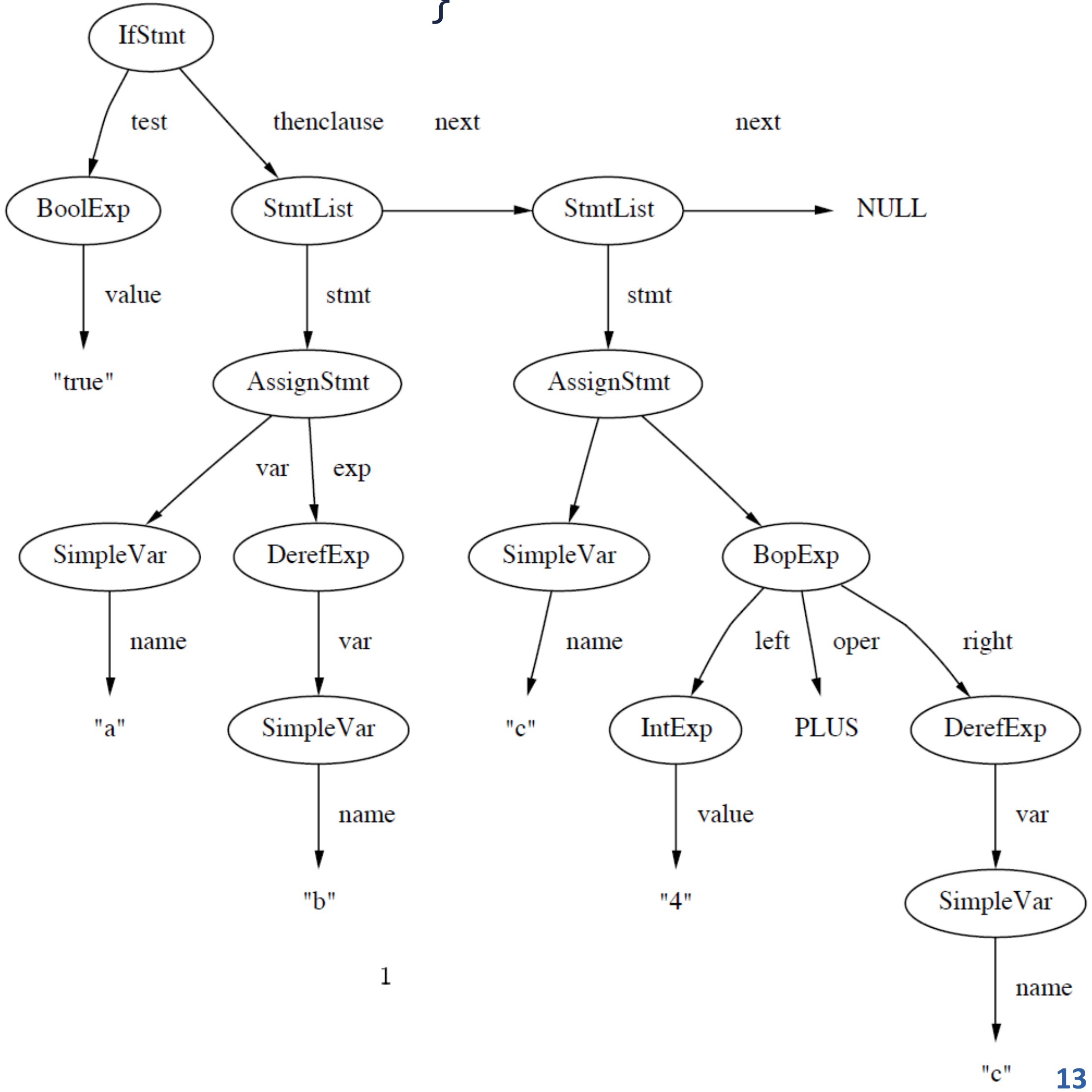
C Instruction	2 address	3 address
`r = x;`	`mov r, x`	`mov r, x`
`r = x + y;`	`mov r, x` `add r, y`	`add r, x, y`

- Hybrid IRs as the Code Representation
  - CFG + 3-address code (SSA or non-SSA)
  - AST (for control flow) + 3-address code (for basic blocks)
  - …

Abstract Syntax Tree (AST)

- Abstract Syntax Tree (AST): tree representation of the abstract syntactic structure of text (source code) written in a formal language
- It retains syntactic structure of the code
- Widely used in *source-source* compilers
- Captures both control flow constructs and straight-line code explicitly
- Traversal and transformations are both relatively expensive
  - Both are pointer-intensive
  - Transformation are memory-allocation-intensive

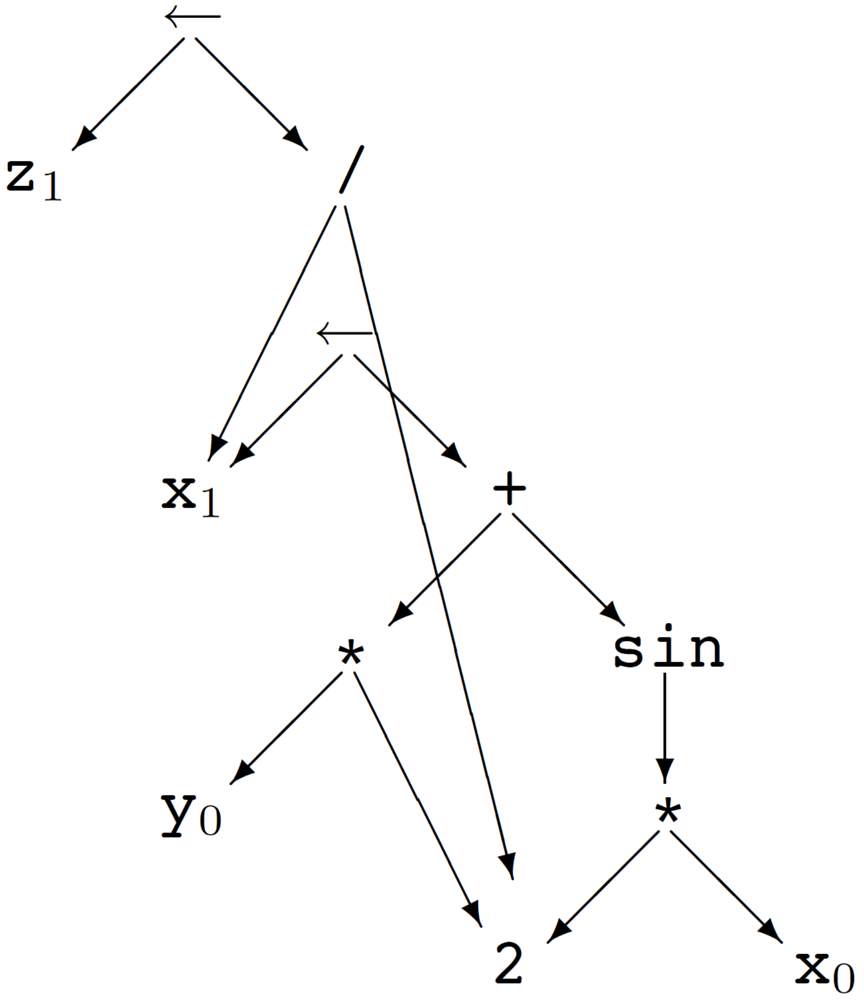
```
if (true) {
     a = b;
     c = 4 + c;
}
```



Directed Acyclic Graph (DAG)

- A Directed Acyclic Graph (DAG) is similar to an AST but with a unique node for each *value*.
- Advantages:
  - Sharing of values is explicit
  - Exposes redundancy (value computed twice)
    - Powerful representation for symbolic expressions
- Disadvantages:
  - Difficult to transform (e.g., delete a statement)
  - Not useful for showing control flow structure
    - Better for *analysis* than *transformation*

```
x ← 2*y + sin(2*x)
z ← x / 2
```



Control Flow Graph (CFG)

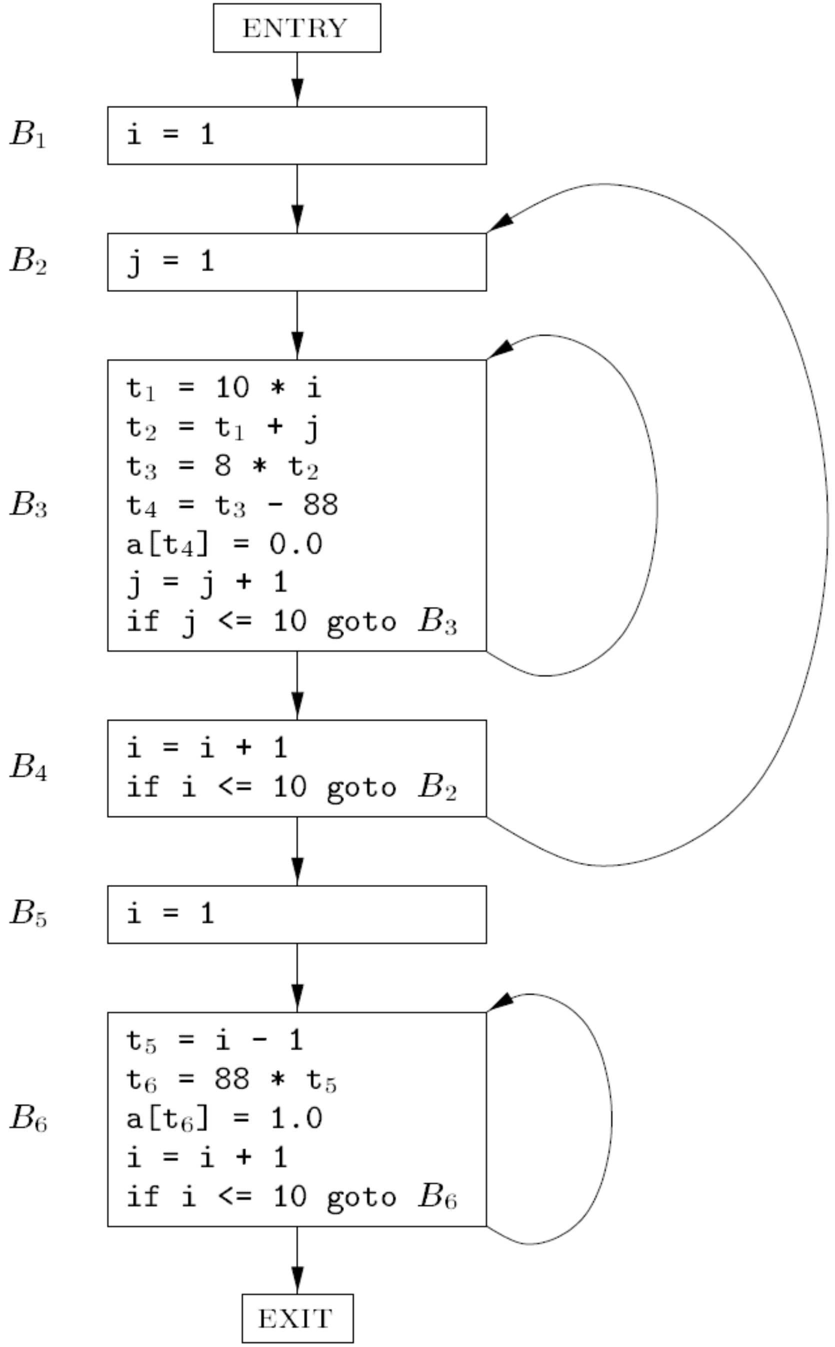
- **Basic Block**: a *maximal* consecutive sequence of statements (or instructions) $S_1 \cdots S_n$ such that:
  - (a) the flow of control must enter the block at S_1, and
  - (b) if S_1 is executed, then $S_2 \cdots S_n$ are all executed in that order (unless one of the statements causes the program to halt)
- **Leader**: the first statement of a basic block
- **CFG**: a directed graph (usually for a single procedure) in which:
  - Each node is a single basic block
  - There is an edge $b_1 \rightarrow b_2$ if control ***may*** flow from last statement of b_1 to first statement of b_2 in *some* execution

***Note**: A CFG is a <u>conservative approximation</u> of the control flow!*

Control Flow Graph (CFG)

- Example:

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```



Dominance in Control Flow Graphs

- **Dominates**: B_1 dominates B_2 *iff* all paths from entry node to B_2 include B_1
- Intuitively, B_1 is always executed before executing B_2 (or B_1 = B_2)

Which assignments dominate (X+Y)?

```
X = 1;
if (...) {
    Y = 4;
}
... = X + Y;
```

Which assignments dominate (X+Y)?

```
X = 1;
if (...) {
    Y = 4;
    ... = X + Y;
}
```

Static Single Assignment (SSA) Form

- Static Single Assignment (SSA) is a property of an IR: each variable is assigned exactly once, and every variable is defined before it is used.
- Informally, a program can be converted into SSA form as follows:
  - Each assignment to a variable is given a unique name
  - All of the uses reached by that assignment are renamed
- Easy for straight-line code:

$$
\begin{array}{lll}
V & \leftarrow & 4 \\
 & \leftarrow & V + 5 \\
V & \leftarrow & 6 \\
 & \leftarrow & V + 7
\end{array}
\qquad\qquad
\begin{array}{lll}
V_0 & \leftarrow & 4 \\
 & \leftarrow & V_0 + 5 \\
V_1 & \leftarrow & 6 \\
 & \leftarrow & V_1 + 7
\end{array}
$$

- What about flow of control?
  - Introduce ϕ-functions

SSA with Control Flow

Two-way branch

```
if (...)
      X = 5;
else
      X = 3;

Y = X;
```

if (...)
$\quad X_0 = 5;$
else
$\quad X_1 = 3;$
$X_2 = \phi(X_0, X_1);$
$Y_0 = X_2;$

While loop

$\quad j = 1;$
S: // *while (j < x)*
$\quad$ if ($j >= X$)
$\quad\quad$ goto **E**;
$\quad j = j+1;$
$\quad$ goto **S**
E:
$\quad N = j;$

$\quad j_5 = 1;$
S: $\quad j_2 = \phi(j_5, j_4);$
$\quad$ if ($j_2 >= X$)
$\quad\quad$ goto **E**;
$\quad j_4 = j_2+1;$
$\quad$ goto **S**
E:
$\quad N = j_2;$

SSA with Control Flow

- ϕ-functions: In a basic block B with N predecessors, $P_1, P_2, \cdots, P_N$,

$$X = \phi(V_1, V_2, \cdots, V_N)$$

  assigns $X = V_j$ if control enters block B from P_j, $1 \leq j \leq N$.

- Properties of ϕ-functions:
  - ϕ is not an executable operation
  - ϕ has exactly as many arguments as the number of incoming BB edges
  - Think about ϕ argument V_i as being evaluated on CFG edge from predecessor P_i to B
- ***SSA form definition:***

  A program is in SSA form if:
  - Each variable is assigned a value in exactly one statement
  - Each use of a variable is *dominated* by the definition

Tradeoffs of SSA Form

Strengths:

- Each use is reached by a single definition (simpler analyses)
- Def-use pairs are explicit: compact dataflow information
- No *write-after-read* and *write-after-write* dependences
- Can be directly transformed during optimizations

Many dataflow optimizations are *much* faster

Weaknesses:

- Space requirement: many variables, many ϕ Functions
- Limited to scalar values; an array is treated as one big scalar
- When target is low-level machine code, limited to “virtual registers” (memory is not in SSA form)
- Copies introduced when converting back to real code

Stack Machine Code

- Used in compilers for stack architectures
- Popular again for bytecode languages, e.g., in JVM
- Advantages:
  - Compact form
  - Introduced names are implicit, not explicit
  - Simple to generate and execute code
- Disadvantages:
  - Does not match current architectures
  - Many spurious dependences due to stack
    - Difficult to reordering transformations
  - Cannot “reuse” expressions easily (must store and re-load)
    - Difficult to express optimized code

Example

$$x - 2 * y - 2 * z$$

Stack machine code:

```
push x
push 2
push y
multiply
push 2
push z
multiply
add
subtract
```

Three Address Code

- Three Address Code: a term used to describe many different representations where each statement is single operator and there are at most three operands
- Advantages:
  - Compact and very uniform
  - Makes intermediates values explicit
  - Suitable for many levels (high, mid, low)
- Disadvantages:
  - Large name space (due to temporaries)
  - Loses syntactic structure of source

Example

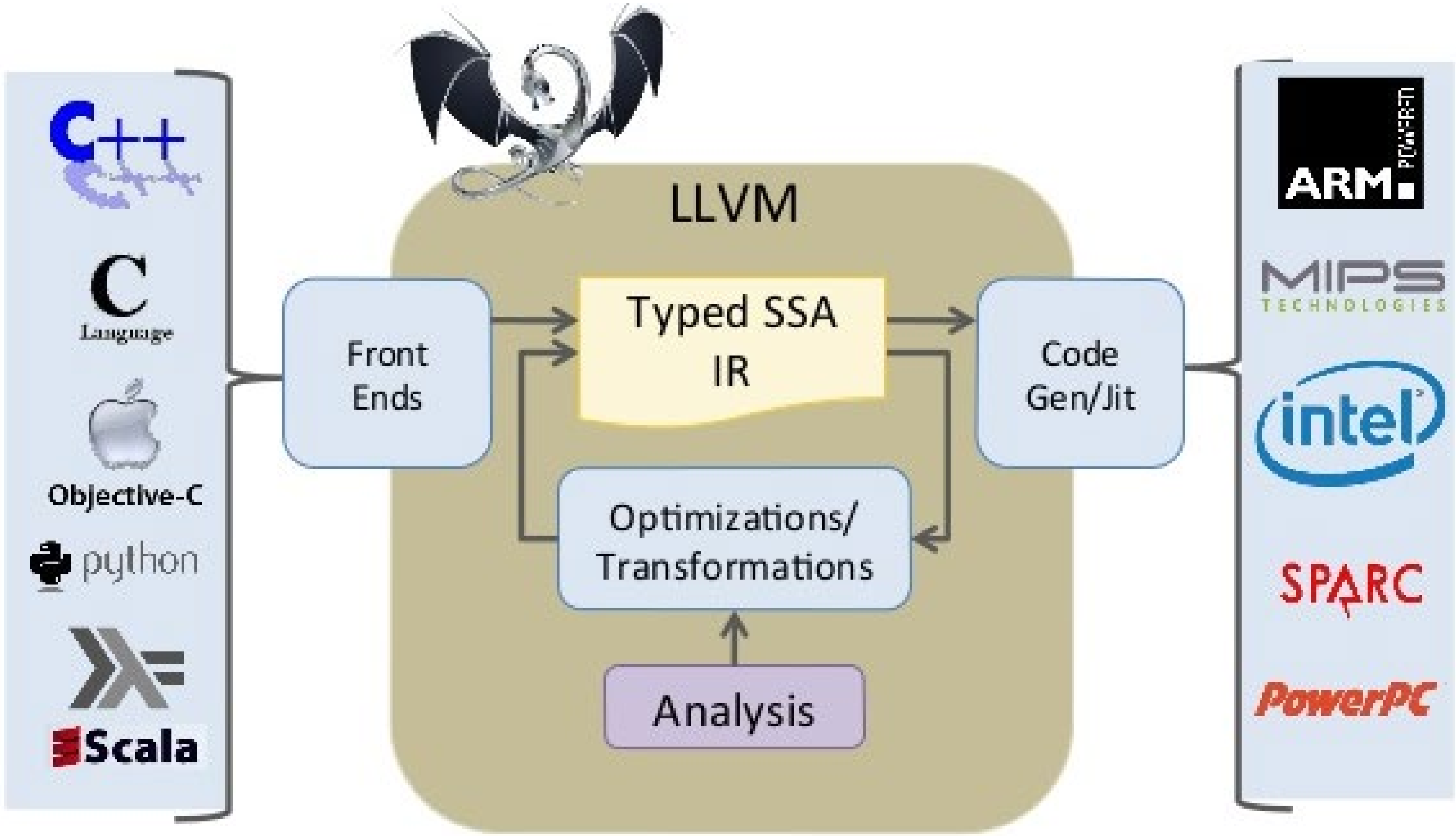
```
if (x > y)
    z = x - 2 * y
```

3-address code:

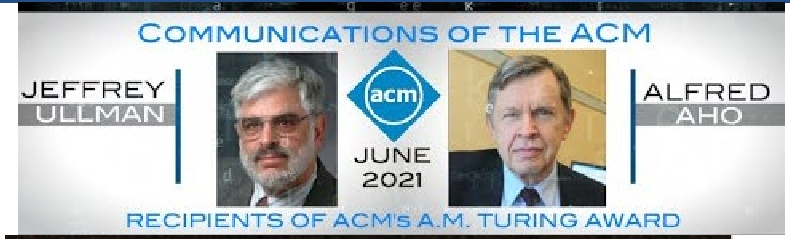
$$
\begin{array}{ll}
 & t_1 \leftarrow \texttt{load } x \\
 & t_2 \leftarrow \texttt{load } y \\
 & t_3 \leftarrow t_1 \; gt \; t_2 \\
 & \texttt{br } t_3 \; L_2 \; L_1 \\
L_1: & t_4 \leftarrow 2 * t_2 \\
 & t_5 \leftarrow t_1 - t_4 \\
 & z \leftarrow \texttt{store } t_5 \\
L_2: & \cdots
\end{array}
$$

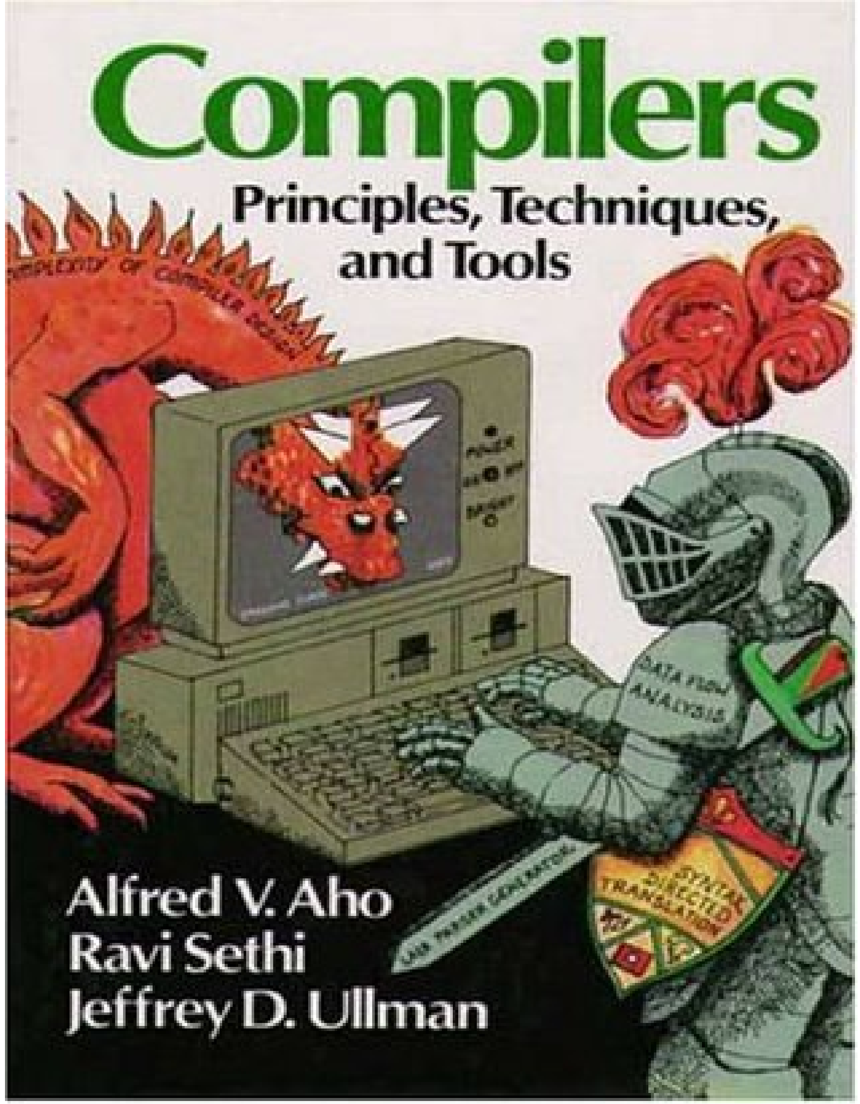
The LLVM Compiler Infrastructure

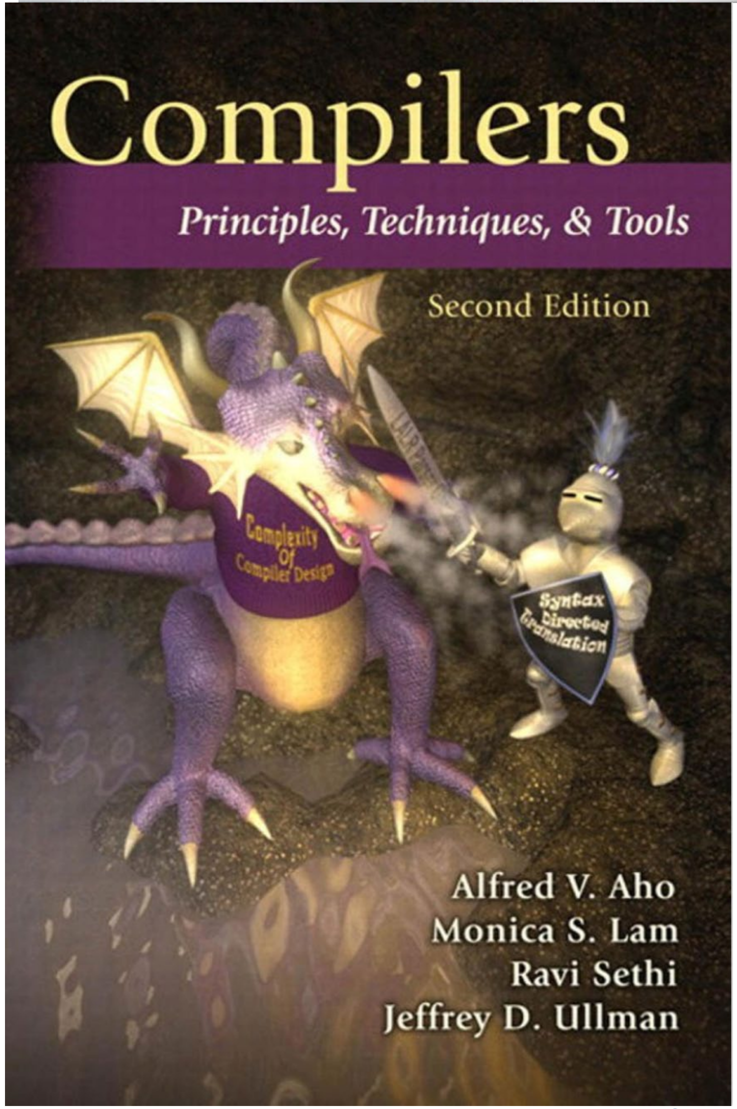
- *The LLVM Project*: a collection of modular and reusable compiler and toolchain technologies
- *LLVM*: the name is not an acronym; originally represents "*Low Level Virtual Machine*"
- Started in 2000 at the *University of Illinois at Urbana-Champaign*, under the direction of *Vikram Adve* and *Chris Lattner*.



Compilers







“Complexity of Compiler Design”, “LALR parser generator”, “Syntax Directed Translation”, “Data Flow Analysis”

UCI EECS

EECS 221:
Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu