CS 4240: Compilers

Lecture 1: Compiler Structure, Intermediate Representations

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

January 7, 2019

Introduction

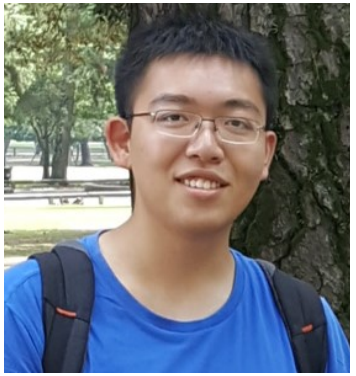
- Scope of CS 4240: Topics in the design of programming language translators, including scanning, parsing, semantic checks, intermediate representations, code optimization, and code generation
- Text: Engineering a Compiler (Cooper and Torczon), 2nd edition
  - Published by Morgan-Kaufmann, an Elsevier imprint
  - Highly recommended. While all course material will be covered in slides, the textbook contains in-depth technical details.
    - Note that all slides are subject to the copyright notice below.
- Piazza site: http://piazza.com/gatech/spring2019/cs4240a
- Canvas site: canvas.gatech.edu
- Additional course logistics will be covered at end of today's lecture

Teaching Staff & Office Hours

- Instructor: Vivek Sarkar
- Email: vsarkar@gatech.edu
- Office Hours
  - Mondays, 2pm - 3pm, KACB 2332

- TA: Youngsuk Kim
- Email: ykim837@gatech.edu
- Office Hours
  - Wednesdays, 10am - 12noon, KACB 2317

- TA: Fangke Ye
- Email: yefangke@gatech.edu
- Office Hours
  - Tuesdays, 3pm - 5pm, KACB 2317



My Research Interests: Programming Languages, Compilers, and Runtime Systems for Parallel Systems

1982-1987: PhD, Stanford University (Advisor: John Hennessy)

–Partitioning and Scheduling Single-Assignment Programs for Multiprocessor Execution

1987-2007: Research Staff Member & Senior Manager, IBM Research

–PTRAN automatic parallelization project (led by Dr. Fran Allen)

–ASTI optimizer for locality and parallelism used in IBM product compilers since 1996

–Jikes Research Virtual Machine (Jalapeno project)

–X10 parallel language (DARPA HPCS program)

1996-1998: Visiting Associate Professor, MIT

–Founding member of MIT RAW project

–Taught UG 6.035 class (compilers)

2007-2017: Professor, Rice University

–Habanero Extreme Scale Research project: programming models, compilers, and runtimes based on C/C++ and Java

–Pliny "big-code" analytics project (DARPA MUSE program)

–Created and taught UG class on Fundamentals of Parallel Programming (COMP 322)

–Instructor for Coursera specialization on parallel, concurrent & distributed programming

–Department Chair during 2013—2016

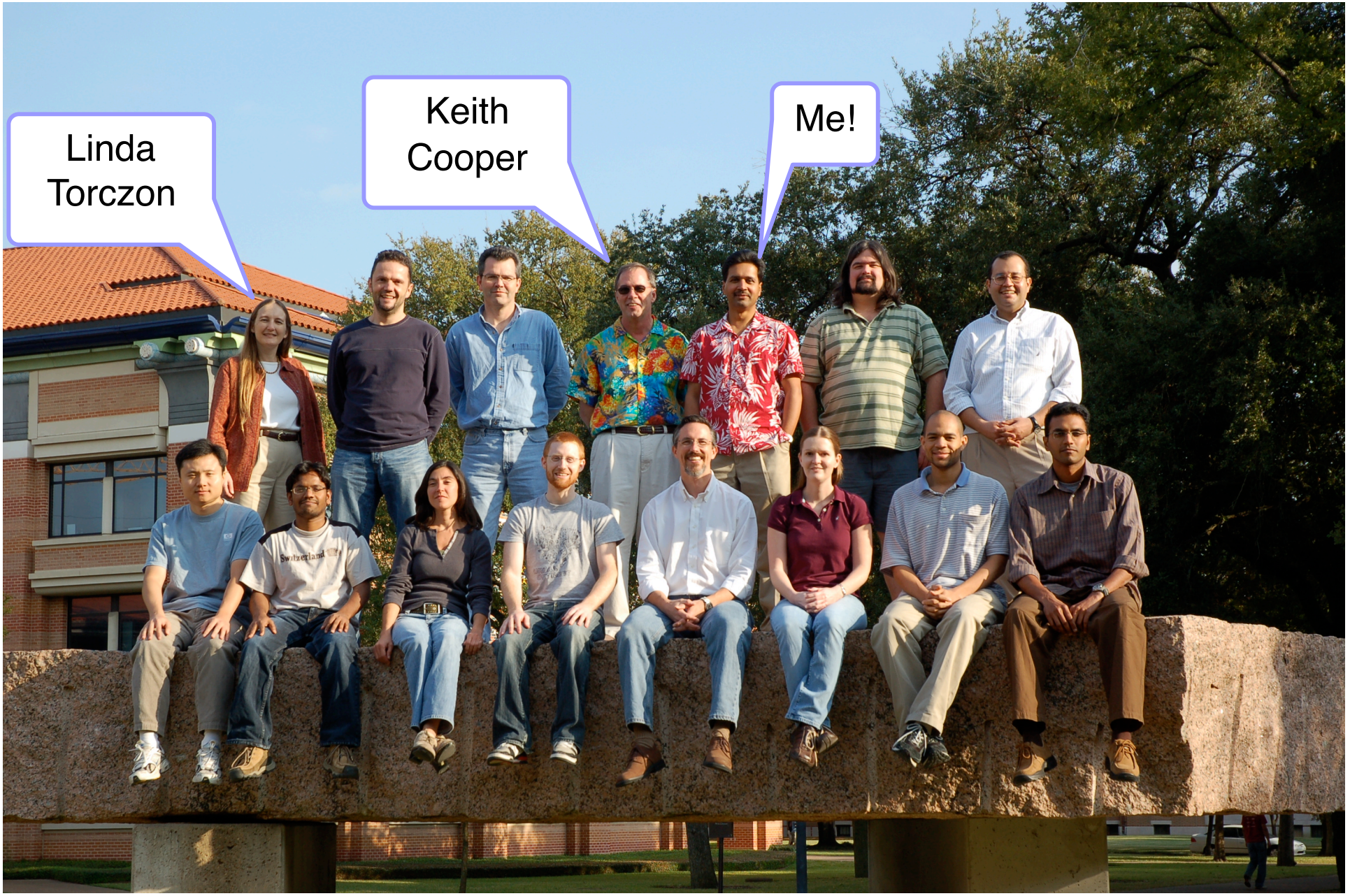
Aug 2017 onwards: Professor, School of Computer Science, Georgia Tech

–Co-Director, Center for Research on Novel Computing Hierarchies (CRNCH)

–PI, DDARING project in DARPA-funded Software Defined Hardware program

–Instructor, CS 4240, Spring 2018 & 2019

Textbook authors were my colleagues at Rice University!



Compilers

- What is a compiler?
  - A program that translates an executable program in one language into an executable program in another language
  - The compiler should improve the program, in some way
- What is an interpreter?
  - A program that reads an executable program and produces the results of executing that program
- C is typically compiled, Scheme is typically interpreted
- Java is compiled to bytecodes (code for the Java VM)
  - which are then interpreted
  - Or a hybrid strategy is used
    - → Just-in-time compilation

Why Study Compilation?

- Compilers are important
  - Responsible for many aspects of system performance
  - Attaining performance has become more difficult over time
    - → In 1980, typical code got 85% or more of peak performance
    - → Today, that number is closer to 5 to 10% of peak
- Compilers are interesting
  - Compilers include many applications of theory to practice
  - Writing a compiler exposes practical algorithmic & engineering issues
- Compilers are everywhere
  - Many practical applications have embedded languages
    - → Commands, macros, formatting tags …
  - Many applications have input formats that look like languages

Intrinsic Merit

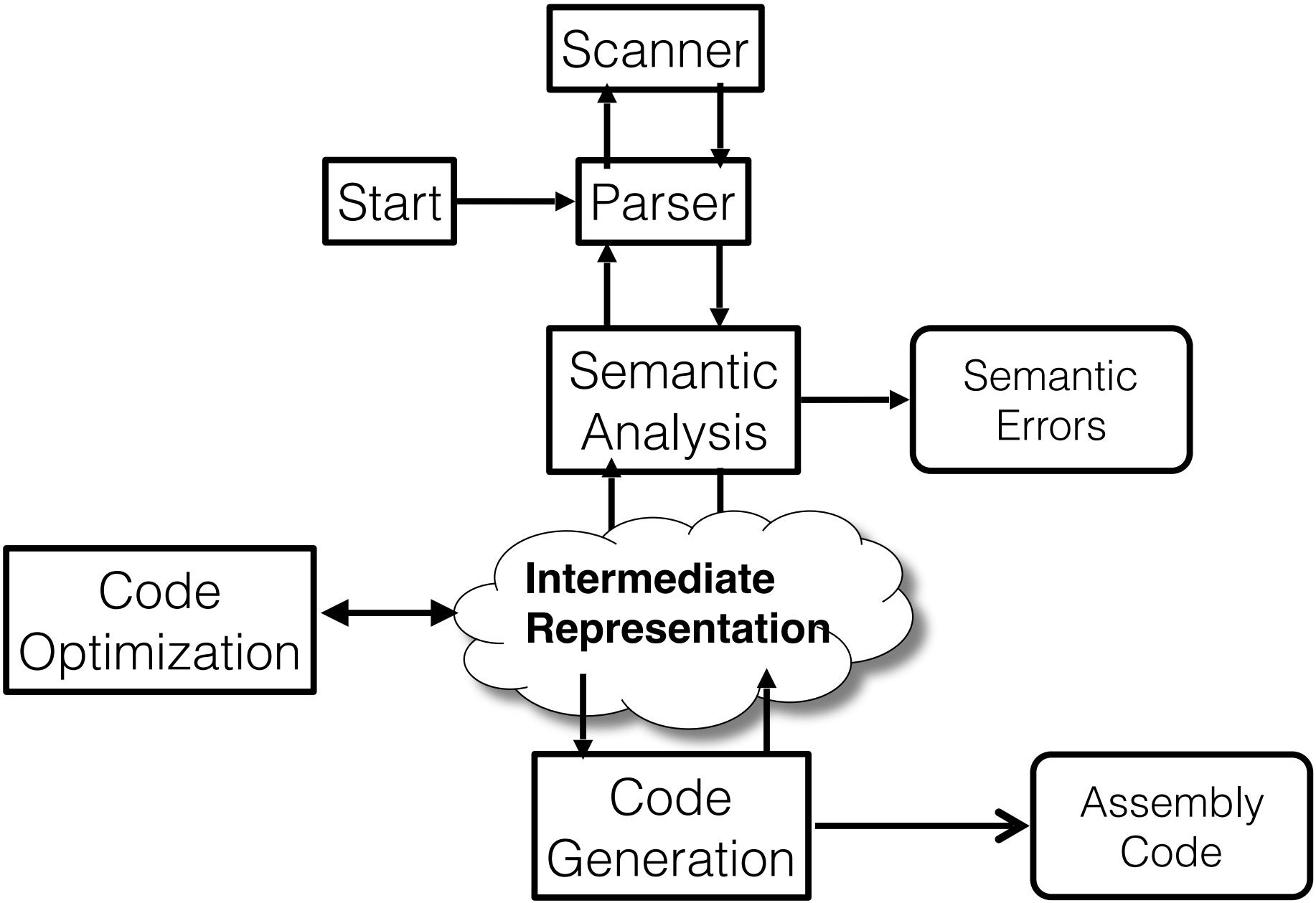
- Compiler construction poses challenging and interesting problems:
  - Compilers must do a lot but also run quickly
  - Compilers have primary responsibility for run-time performance
  - Compilers are responsible for making it acceptable to use the full power of the programming language
  - Computer architects perpetually create new challenges for the compiler by building more complex machines
    - → Compilers must hide that complexity from the programmer
- A successful compiler requires mastery of the many complex interactions between its constituent parts

Intrinsic Interest

- Compiler construction involves ideas from many different parts of computer science

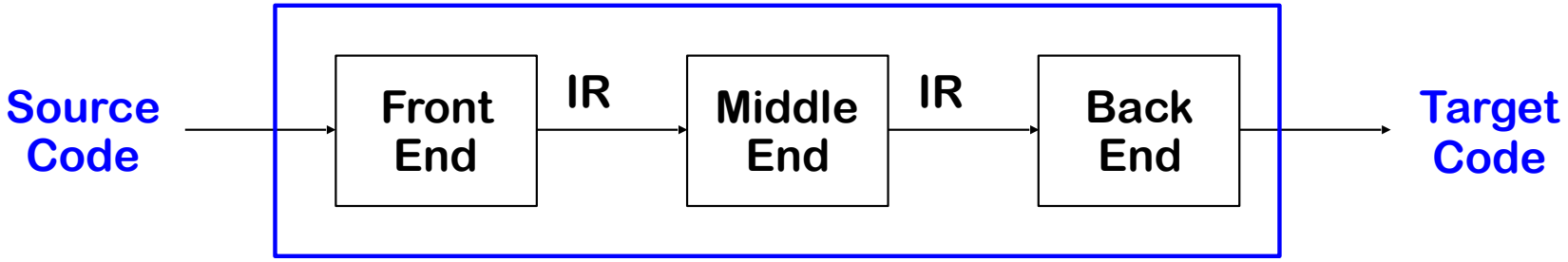
Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms, union-find Dynamic programming
Theory	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Pipeline & hierarchy management Instruction set use

Structure of a Full Compiler



Role of Intermediate Representations in Code Generation

- 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

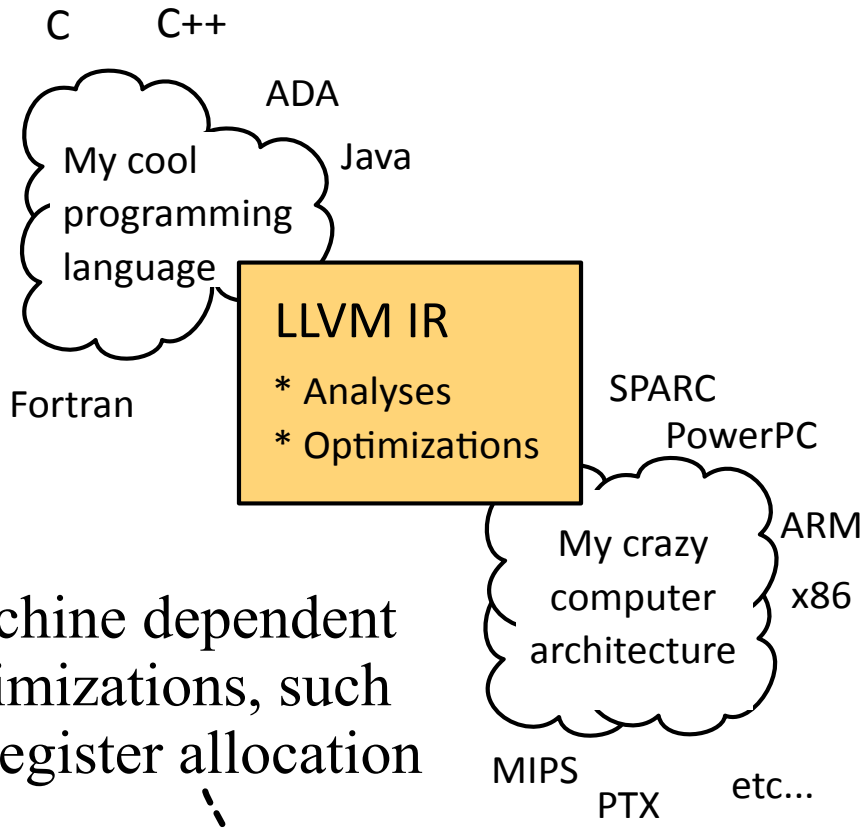


- IR encodes the compiler's knowledge of the program
- Middle end usually consists of several passes

Modern example: CLANG + LLVM

LLVM is a compiler infrastructure designed as a set of reusable libraries with well-defined interfaces♤.

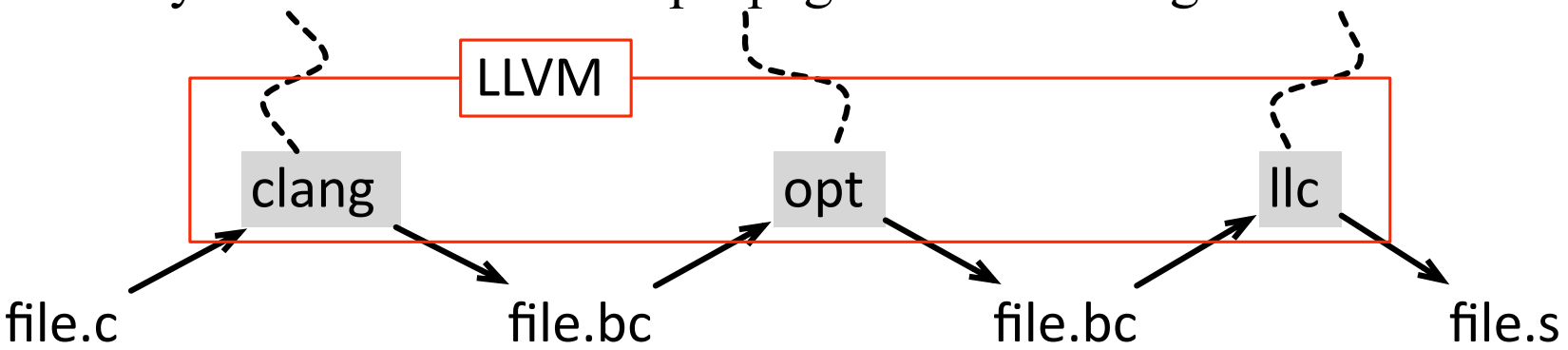
- Implemented in C++
- Several front-ends
- Several back-ends
- First release: 2003
- Open source
- `http://llvm.org/`



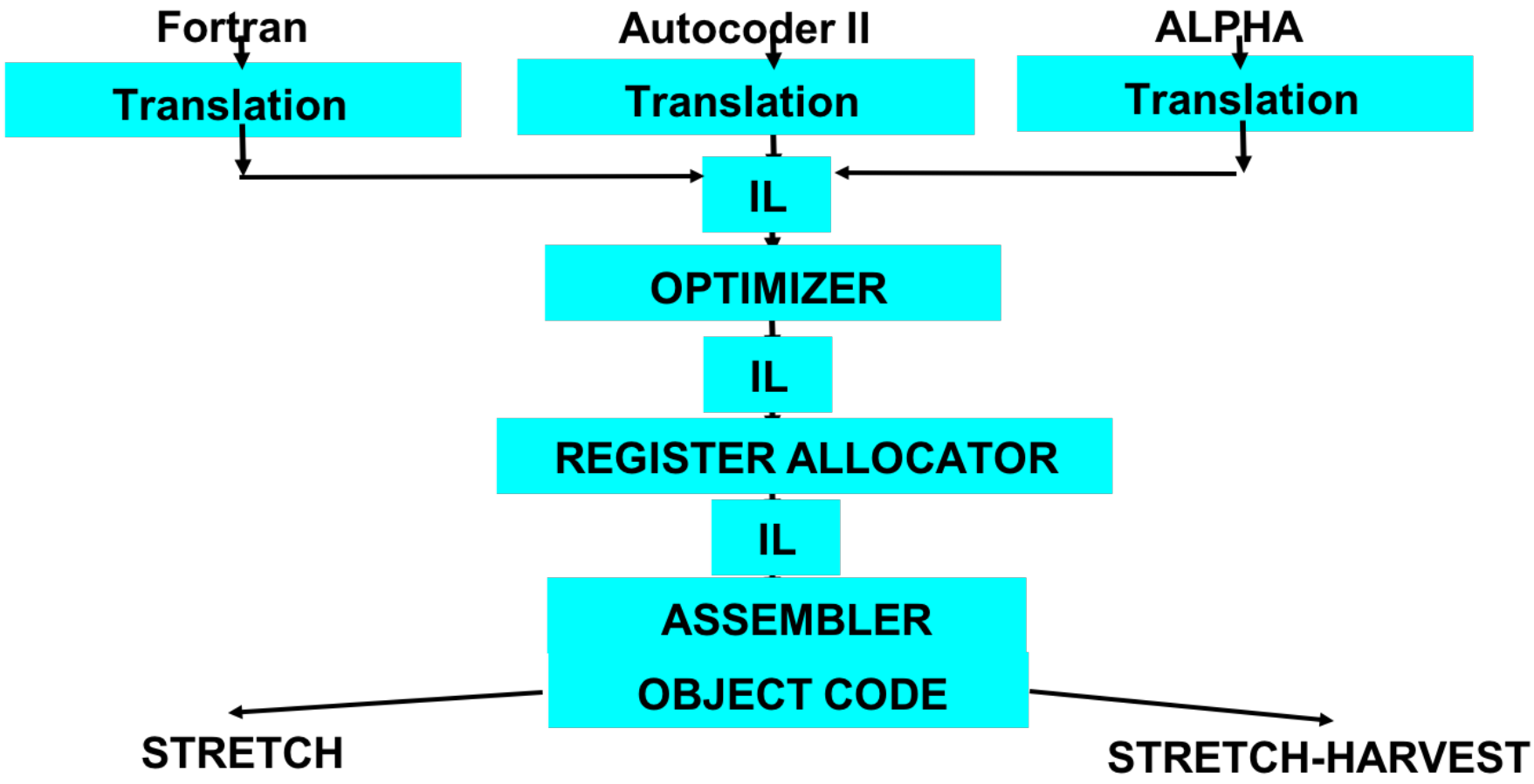
The front-end that parses C into bytecodes

Machine independent optimizations, such as constant propagation

Machine dependent optimizations, such as register allocation



Example from 60 years ago …



Stretch – Harvest Compiler Organization (1958 - 1962)

Source: "Compiling for Parallelism", Fran Allen, Turing Lecture, June 2007

Intermediate Representations

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

- Some important IR properties
  - Ease of generation
  - Ease of manipulation
  - Procedure size
  - Freedom of expression
  - Level of abstraction

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

Types of Intermediate Representations

Three major categories

- Structural
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large

  Example:
  Abstract Syntax Tree (AST)

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

  Examples:
  3 address code
  Stack machine code

- Hybrid
  - Combination of graphs and linear code

  Example:
  Control-flow graph (CFG)

Stack Machine Code

Originally used for stack-based computers, now Java

» Example:

x - 2 * y becomes

```
push x
push 2
push y
multiply
subtract
```

Advantages

» Compact form

Implicit names take up no space, where explicit ones do!

» Introduced names are implicit, not explicit

» Simple to generate and execute code

Useful where code is transmitted
over slow communication links (e.g., to mobile phones!)

Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:

$$x \leftarrow y \ \underline{op} \ z$$

With 1 operator (op) and, at most, 3 names (x, y, & z)

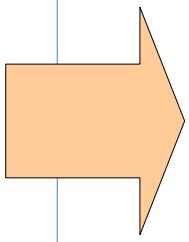
Example:

Advantages:

- Resembles many real machines
- Introduces a new set of temporary variables (virtual registers), e.g., t1, r1, r2, …
- Compact form

Generating 3-address Code: example

```
if (c == 0) {
   while (c < 20) {
      c = c + 2;
   }
}
else
   c = n * n + 2;
```

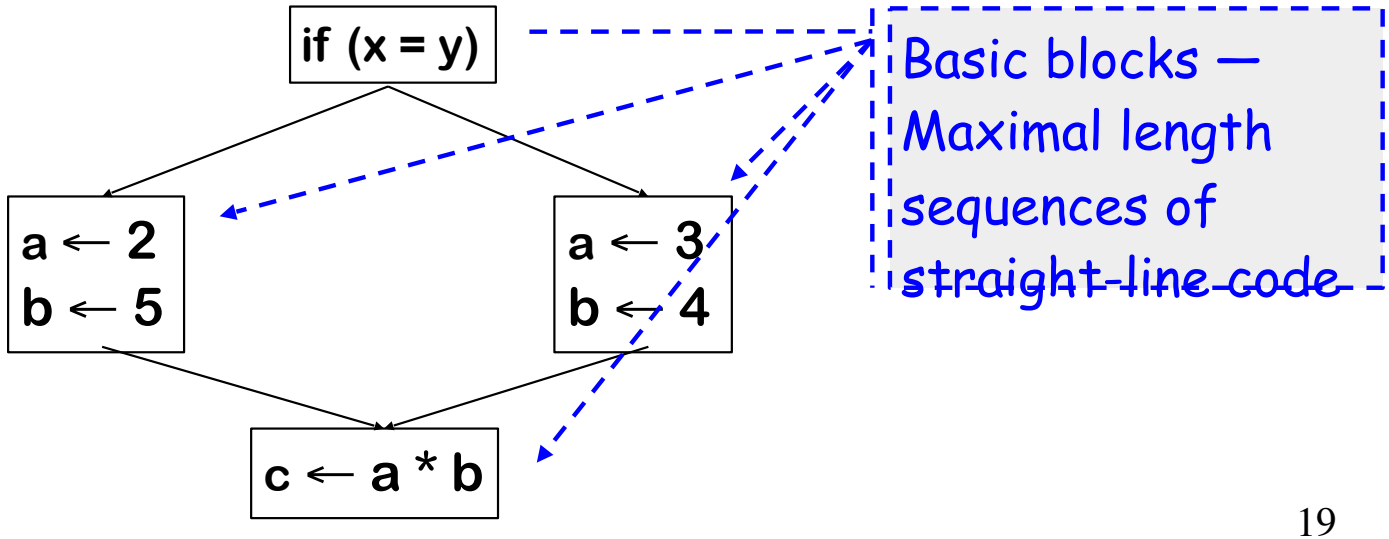


```
1.t1 = c == 0
2.br t1, lab1
3.t2 = n * n
4.c = t2 + 2
5.goto end
6.lab1:
7.t3 = c >= 20
8.br t3, end
9.c = c + 2
10.goto lab1
11.end:
```

Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow
- Potential for exceptions can reduce basic block size in some languages, e.g., NullPointerException in Java
- Example



Tiger-IR to be used in our class projects

- Three address code
- Instruction format
  - op, x, y, z
  - This is equivalent to "x <— y op z",
- For example, the source code expression, 2 * a + (b - 3) can be translated to the following IR instructions which computes the expression in t3
  - mult, t2, a, 2
  - sub, t1, b, 3
  - add, t3, t1, t2
- t1, t2, t3 are temporary variables generated by the compiler when generating the IR

Branch operations in Tiger-IR

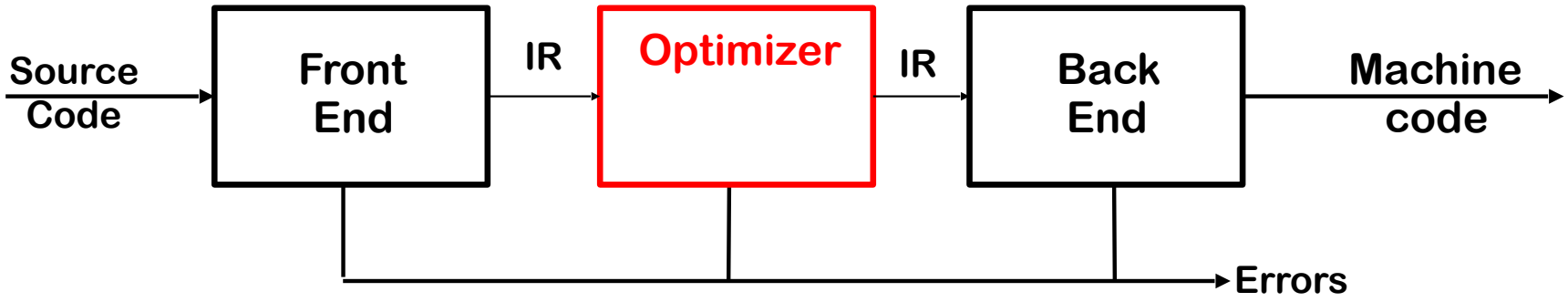
Op	Example source	Example IR
goto	break;	goto, after_loop, ,

Op	Example source	Example IR
breq	if(a <> b) then	breq, a, b, after_if_part
brneq	if(a = b) then	brneq, a, b, after_if_part
brlt	if(a >= b) then	brlt, a, b, after_if_part
brgt	if(a <= b) then	brgt, a, b, after_if_part
brgeq	if(a < b) then	brgeq, a, b, after_if_part
brleq	if(a > b) then	brleq, a, b, after_if_part

Functions in Tiger-IR

- Function starts with a function signature IR statement denoting the return type, name of the function, and function parameters., e.g.,
  - int Foo (int first, float second)
- Function signature is followed by a Data Segment consisting of type declarations for all variables (including arrays and temporaries), e.g.,
  - int-list : t1, t2 , t3, a , b[100]
  - float-list : t4, t3, t6
- Function call instruction is an exception to the 3-address rule
  - call, func_name, param1, param2, …, paramn
- And function calls with return values will have a similar structure:
  - callr, x, func_name, param1, param2, …, paramn

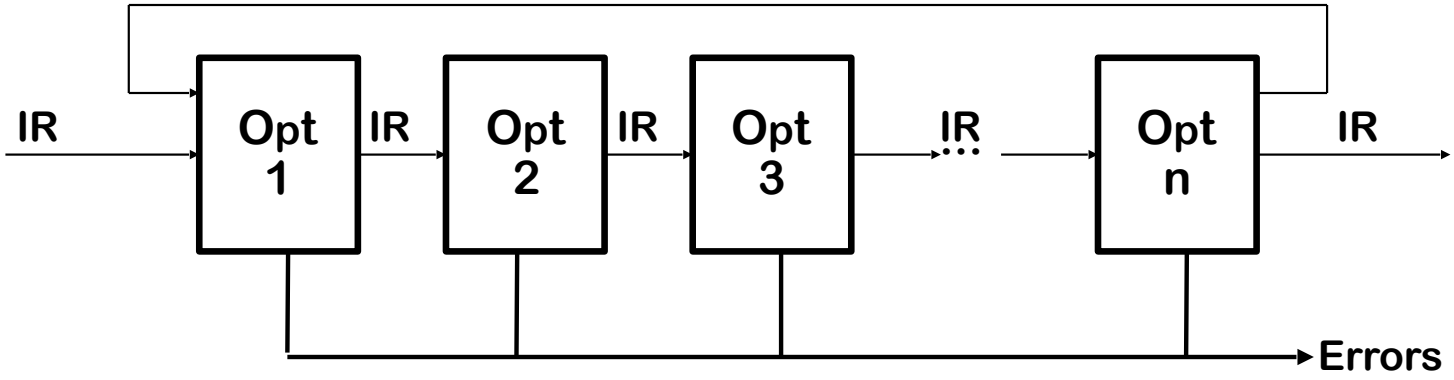
Traditional Three-pass Compiler



Code Improvement (or Optimization)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variables

The Optimizer



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

The Role of the Optimizer

- The compiler can implement a procedure in many ways
- The optimizer tries to find an implementation that is "better"
  - Speed, code size, data space, …

To accomplish this, it

- Analyzes the code to derive knowledge about run-time behavior
  - Data-flow analysis, pointer disambiguation, …
  - General term is "static analysis"
- Uses that knowledge in an attempt to improve the code
  - Literally hundreds of transformations have been proposed

Nothing "optimal" about optimization

- Proofs of optimality are uncommon, and assume restrictive & unrealistic conditions when available

Dead code elimination

DEAD

- Conceptually similar to mark-sweep garbage collection
  - Mark useful operations
  - Everything not marked is useless
- Need an efficient way to find and to mark useful operations
  - Start with critical operations
  - Work back up data flow edges to find their antecedents

Define critical

- I/O statements, linkage code (entry & exit blocks), return values, calls to other procedures
- Global variables that can be visible on program exit

Dead code elimination

Mark

```
1.   for each op i
2.      clear i's mark
3.      if i is critical then
4.         mark i
5.         add i to WorkList

6.   while (Worklist ≠ Ø)
7.      remove i from  WorkList
8.         (i has form "x←y op z" )
9.      for each instruction j that
10.     writes to y or z
11.      if j is not marked then
12.        mark j
13.        add j to WorkList
```

Sweep

```
for each op i
   if i is not marked then
      delete i
```

NOTES:

1) Not all instructions that write to y or z need to be marked. We can only focus on "reaching definitions" (next lecture).

2) Branch instructions need special handling in general. A simple approach is to mark all branch instructions as critical. See textbook for more sophisticated approaches.

Grading (details in syllabus)

- Written Homeworks (15%)
- Programming Projects (30%)
- Midterm exam (20%)
- Final exam (30%)
- Class Participation (5%

Today's in-class Worksheet

- Worksheets can be solved collaboratively
  - All other course work must be done individually or in project groups (see syllabus for details)
- Each student should turn in their own solution, based on collaborative discussions
- Worksheets will not be graded or returned, but solutions will be provided
- Worksheets will contribute to class participation grade
- Worksheets will inform teaching staff of concepts that need to be reviewed/reinforced in future lectures