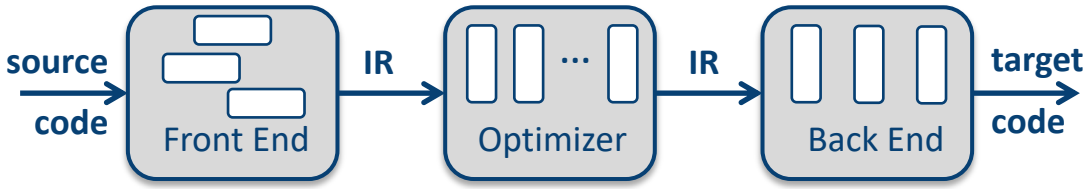
COMP 412
FALL 2018

Introduction & First Content

Comp 412



Copyright 2018, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 1 in EaC2e

Critical Facts

The official syllabus is posted on Esther, Piazza, and the class web site.

COMP 412 — *Introduction to Compiler Construction*

Topics *in the design of programming language translators, including scanning, parsing, semantic elaboration, compile-time & runtime data structures, and code generation.*

Instructors:	Keith Cooper	keith@rice.edu	DH 2065	x6013
	Zoran Budimlić	zoran@rice.edu	DH 3134	x5708
Assistant:	Annepha Hurlock	annepha@rice.edu	DH 3122	x5186

- Office Hours: *will be posted on Piazza*
- Text: *Engineering a Compiler*, 2nd Edition
  - Royalties for sales to COMP 412 go to the Torczon Fellowship
- Class web site will have handouts, lecture notes, …
  - We will not distribute handouts in class; get them from the web
- Discussion site on Piazza
  - You should have received an invitation

Slides are typically posted to the web site before class.

Basis for Grading

Official syllabus is posted on Esther, Piazza, and class web site.

The class will have two exams and three programming assignments

Item		Description	Weight
Midterm Exam		*Week of October 15, likely 10/16*	25 %
Final Exam		*Scheduled by the registrar*	25 %
Lab 1	(8/22)	Scanning & Parsing ILOC	14 %
Lab 2	(9/14)	Local Register Allocator	18 %
Lab 3	(10/26?)	Local Instruction Scheduler	18 %

Closed-book, closed-notes exams

- To pass COMP 412, you must hand in all the labs and take both exams
  - There is partial credit on labs, but your scores will be better if the labs work
- The Lab 1 handout will be available Wednesday, 8/22

Notice: *Any student who needs accommodations for a disability in COMP 412 should contact one of the instructors or contact Alan Russell, Rice's Director of Disability Support Services. Alan's office is on the 1st floor of Allen Center.*

Class-taking Technique for COMP 412

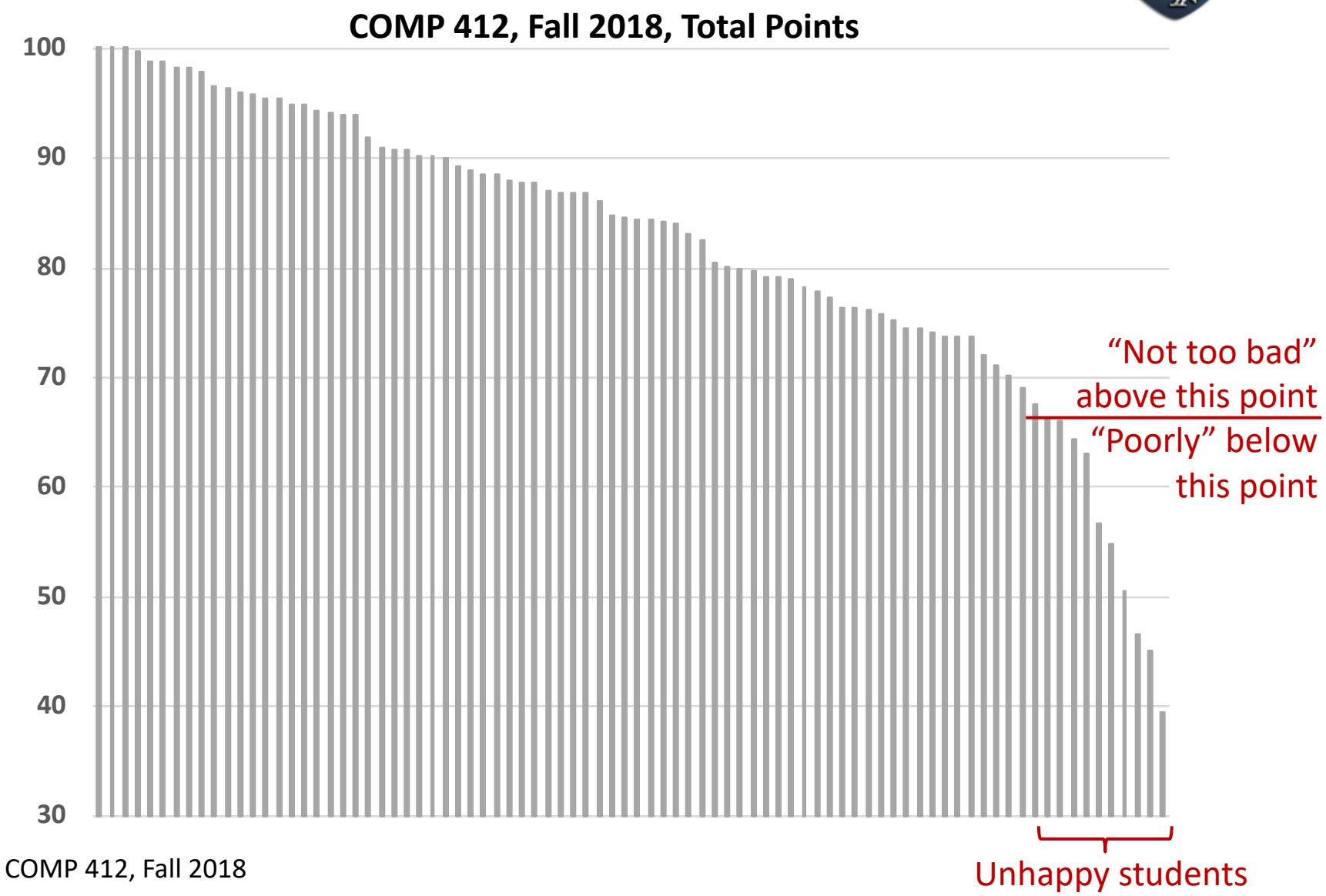
- Attend class
  - The tests will cover both lecture and reading
  - Test questions tend to come from low-attendance classes
- Ask questions
  - Interrupt the lecture if you have a question
  - PowerPoint materials should be online before class
- Read the book
  - Not all material will be covered in class
  - Think through the section review questions
- Start the programming assignments when you get them
  - COMP 412 is not a programming course
    - → Projects are graded on functionality, documentation, and lab reports, not style
    - → Correctness is critical; these are compiler components
- Work problems on your own
  - Good practice for the tests

For example, Chapter 1 provides an overview of how a compiler works. Go read it. The lectures & tests will assume that you have done so.

That's why they exist

Results matter
Efficiency matters

Last Year's Grades



COMP 412, Fall 2018

Class-taking Technique for COMP 412

What causes students to do poorly in 412 or to drop the course?

- Failure to start the labs on time
  - Lab 1 has a short time period (2 weeks)
  - Labs 2 and 3 are difficult, for all of *correctness*, *effectiveness*, & *efficiency*
  - Start them as soon as they are available
  - You will want time to do extensive testing and tuning
    → ***A week for effectiveness & efficiency***
  - Most drops occur during immediately after the register allocator lab when students realize that we were serious about starting early
  - Start early; ask questions; read Piazza
- Failure to attend class, read the book, and take the exams seriously
  - We cover material in class that is not entirely in the book and we test over material in the book that is not mentioned in class
  - We know what is in the book & in the lectures
  - Lecture can be tedious; we will do our best to keep you awake
  - Feel free to post lecture questions to Piazza

Choice of Programming Language

In COMP 412, you may use any programming language available on the CLEAR systems

- In the context of COMP 412, **PERL** is not a programming language
- Most students work in Java, Python, C, or C++
- You will want to reuse the front end from Lab 1 in Labs 2 & 3, so we recommend that you use the same language in all the assignments

The philosophy behind this freedom is simple

- You should work in a language and a toolset where you are comfortable and competent
- Make sure that you have a good set of tools
- Make sure that the same version of the language is available on **CLEAR**.

It is a **BAD** strategic decision to decide to learn a new language in the course of doing a 412 lab (*don't make the lab harder by your language choice*)

Do not blame the language for your lab's performance. ("Java is just slow.")
Look at the performance charts at the back of the lab handout.

Compilers

- What is a **compiler**?
  - A program that translates an *executable* program in one language into an *executable* program, usually 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 and C++ are typically compiled

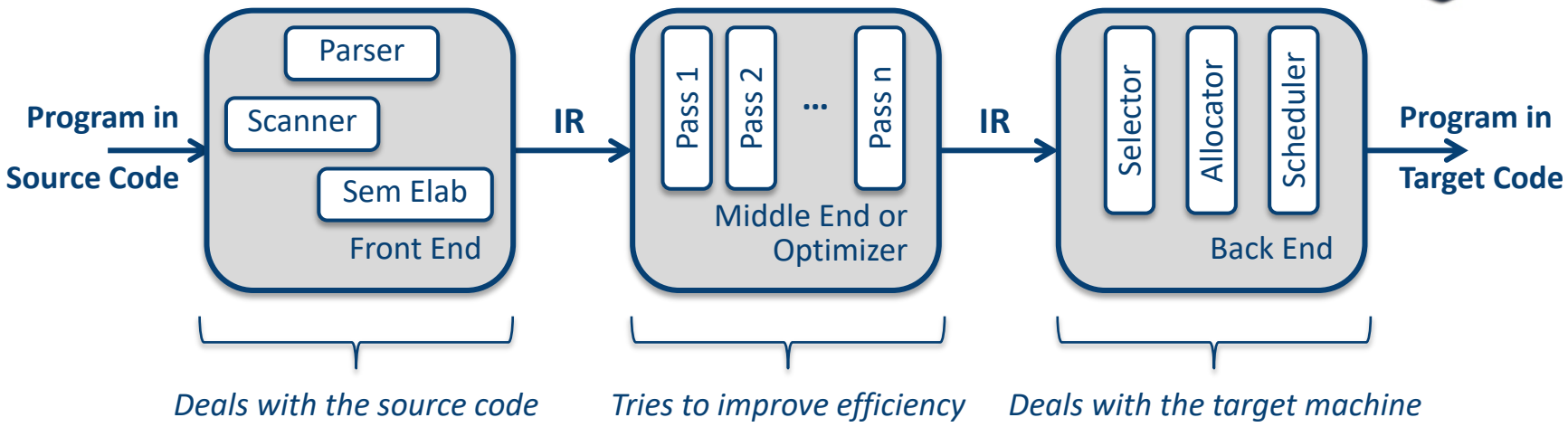
  Python & Scheme are typically interpreted
- Java is complicated
  - compiled to bytecode (*code for the Java VM*)
  - which are then interpreted
  - or a hybrid strategy is used

    → Just-in-time compilation

Common misstatement: *x* is an interpreted language, or *x is* a compiled language

Compiler Structure

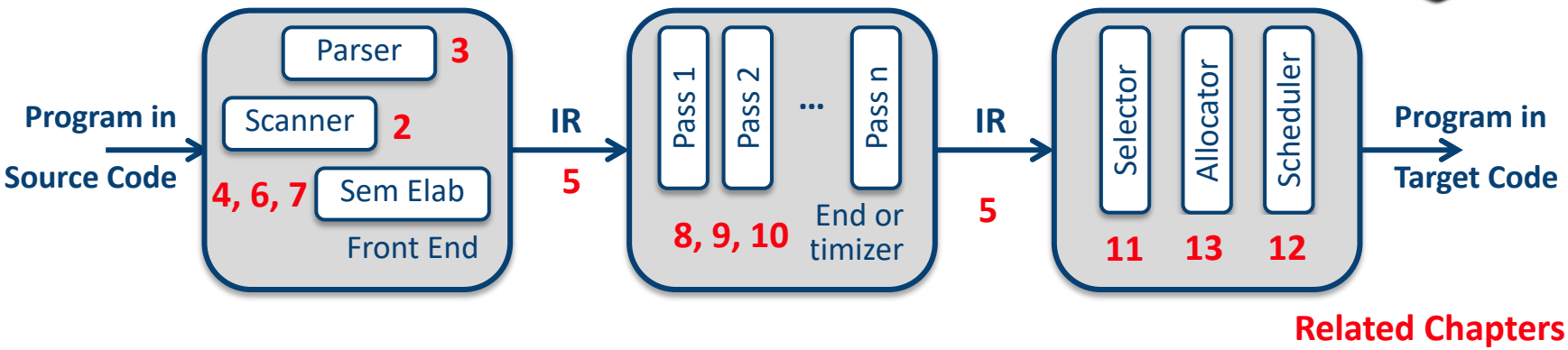
Compilers are among the most complex programs that humans write.



How does a compiler work?

- Front end analyzes program in source language & builds some internal representation for the program ("**IR**")
- Optimizer analyzes & rewrites the **IR** to improve the final code
  - The connection between the **IR** and the final code may be subtle
- Back end translates the **IR** into the target language
  - Target language is usually the instruction set of some target processor

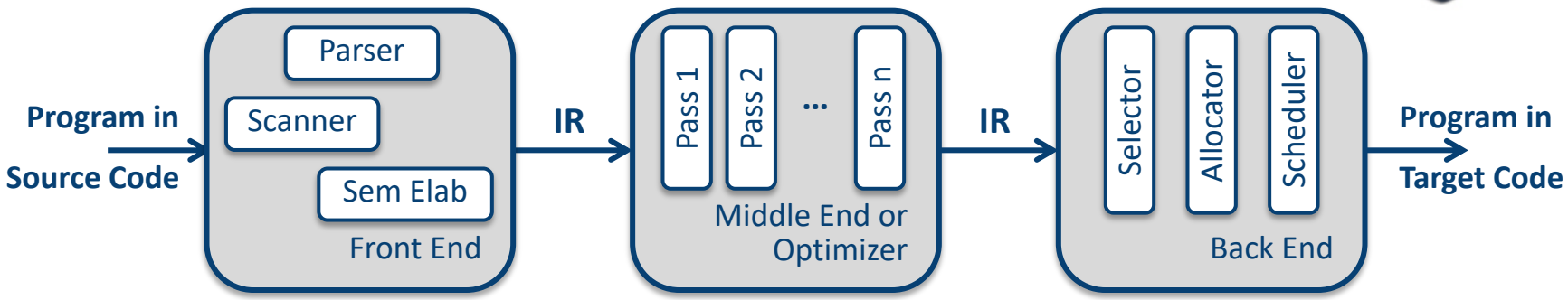
Compiler Structure



How does this structure relate to the syllabus?

- Lecture will correspond (roughly) to a linear walk through the chapters
  - Will skip much of 8, 9, & 10 (See COMP 512)
- Programming Assignments will skip around
  - Lab 1 from chapters 2 and 3
  - Lab 2 from chapter 13
  - Lab 3 from chapter 12

Compiler Structure



Scanner: Translate stream of characters into stream of classified words

Parser: Determine if stream of words is a sentence in the source langauge

Semantic Elaboration: Check deeper meaning & build appropriate **IR**(s)

Optimizer: Rewrite the **IR** form of the code to "improve" it in some measure

Instruction Selector: Rewrite the **IR** into target language operations

Register Allocator: Rewrite the code to fit the finite register set of the target

Instruction Scheduler: Reorder the operations so that they run faster

We will come back to this drawing again, & again, & again, & again, …

The Big Question

Pedantic Professor's Answer

Why study compilers?

- Compilers are *interesting*
  - Large complicated software systems that must efficiently tackle hard algorithmic problems — approximate solutions to NP complete problems
  - Application of theory to practice
  - Wonderful mix of high-level theory and low-level implementation detail
- Compilers are *fundamental*
  - Primary responsibility for application performance
    - → Performance becomes more difficult as processors become more complex
  - The alternative (assembly language) is much less attractive
- Compilers (& interpreters) are *everywhere*
  - Many applications have embedded languages
    - → XML, HTML, macros, commands, Visual Basic in Excel, …
  - Many applications have input formats that look like languages
- Compiler-related knowledge appears in interview questions

The Big Question

Direct Benefit to Most Students

Why study compilers?

In other **COMP** courses, you are taught to use a variety of abstractions, ranging from object orientation to hash maps to closures to …

- Each of these abstractions has a price
- You need to understand that price before you implement
  - Abstraction is critical to successful construction of interesting programs, but you must understand the costs and make intelligent decisions about when to replace an abstraction with a more efficient & concrete implementation
  - Careful choice of abstractions & where to use them can be the difference between a fast system & a slow (or infeasible) one
- Examples:
  - Use of virtual function calls in performance-critical kernels
  - Use of scripting languages, such as **PHP**, for back-end server applications
  - Use of hash maps over enumerated types as array indices

The Big Question

Direct Benefit to Most Students

Why study compilers?

In many applications, performance matters.

Students (and many software engineers) often lack a clear understanding of how to approach performance problems.

- That is one reason for the performance component of lab 2 & lab 3

One useful strategy to improve application performance

- Design at the appropriate level of abstraction
- If performance is an issue
  - Measure where the application spends time
  - In those places, replace the abstract implementation with a semantically equivalent implementation that is faster and more concrete
- Repeat until you are happy with the results

Any time that you find yourself using a hashmap over a compact set (e.g., lab 1's token types or lab 2's registers), you should ask yourself why …

Simple Examples

All data collected on a quiescent, multiuser Intel T9600 @ 2.8 GHz using code compiled with gcc 4.1 –O3

Which loop is faster?

```
for (x=0; x<n; x++)
      for (y=0; y<n; y++)
            A[x][y] = 0;

for (y=0; y<n; y++)
      for (x=0; x<n; x++)
            A[x][y] = 0;

p = & A[0][0];
t = n * n;
for (x=0; x<t; x++)
            *p++ = 0;
```

All three loops have distinct performance

0.51 seconds on 10,000 x 10,000 array

~ 5x

1.65 seconds on 10,000 x 10,000 array

~ 15x

0.11 seconds on 10,000 x 10,000 array

A good compiler should know these tradeoffs on each target and generate the best code.

Few real compilers do.

Conventional wisdom suggests using

```
bzero((void *) &A[0][0],(size_t) n * n * sizeof(int));
```

0.52 seconds on 10,000 x 10,000 array

The Big Question

Understanding how compilers work can help in many other areas

Example: Multi-level, recurrent neural networks

- Ankit Patel (Rice ECE, Baylor Med) studies neural networks
- He is trying to understand how they represent knowledge
  - Are the internal models used by a RNN efficient?
  - Are the internal models used by an RNN understandable by humans?
- One promising point of investigation is understanding formal languages
  - If we train an **RNN** to recognize a simple language, does its model look anything like what a human would derive? And how efficient is that model?
  - If we train an **RNN** to recognize a simple language, does it create abstractions? (*e.g.*, words and parts of speech rather than characters)
- Material covered in 412 is critical to this kind of inquiry

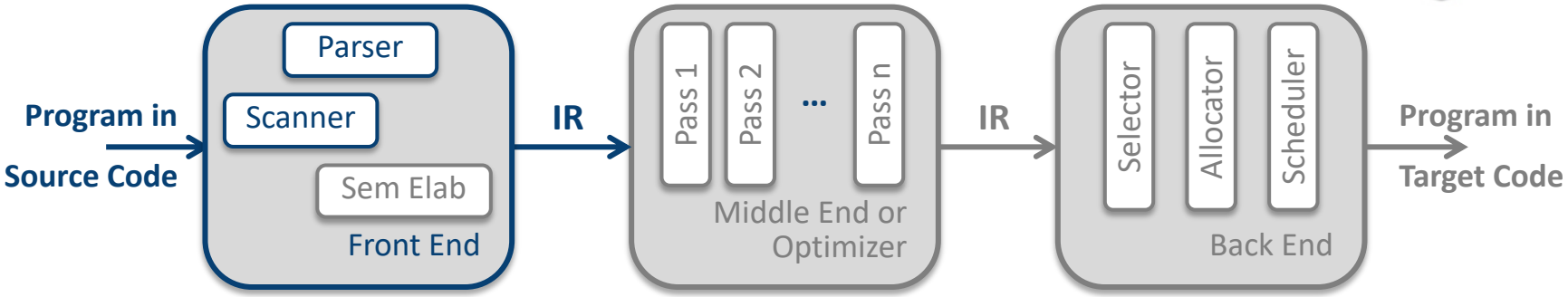
The Big Question

Understanding how compilers work can help in many other areas

More Examples:

- If you are building a server that must grow to billions of transactions per hour, use an efficient, compiled language (*e.g.*, not **PHP**)
- If you are building a commercial operating system, use a base language and runtime that checks for string overflow (*e.g.*, not **ANSI C**)
- If you are writing long-running code, use a language with a managed runtime (*i.e.*, automatic storage reclamation) *and* use leak-detecting tools
- If you need to meet real-time deadlines, use **malloc()** and **free()**, *along with* leak-detecting tools

Next Class



Next class, we will look at how to build a simple front end (scanner & parser) for lab 1. Lab 1 will be available on Wednesday.

- Start reading Chapter 2 (through § 2.3)
- Make sure that you have a working account on **CLEAR**
- Go back to your notes from **ELEC 220** or look at the **ILOC Virtual Machine** notes on the **COMP 412** lectures page and review how a processor works. In lab 1, you will (essentially) scan and parse assembly code. In labs 2 and 3, you will manipulate it, and need to have an intuitive understanding of how processors execute instruction streams.

Extra Slides

(*a longer, more detailed example*)

Simple Examples

Example from R^n Programming Environment, Rice, circa 1984

Abstraction has its price (& that price is often higher than expected)

- In the 1980's, we built the R^n Programming Environment
  - Bitmap displays and mice were new & poorly supported
  - SUN Workstation (& others) had no window systems
  - Predated the Mac, Windows, and so on.

- We built our own window system
  - It had to represent rectangles on the screen
    - → Window is a pair of points, menu item, mouse location, …
  - Mouse tracking was difficult (10 MHz Motorola 68010)
    - → Each mouse movement generated an interrupt & a pair of <x,y> coordinates
    - → At each movement, had to repaint old cursor location, save the new cursor location, xor the cursor onto that location, and paint the resulting small patch to screen

We hit serious performance problems due to the point abstraction

In 1984, it was much easier to compile good (*i.e.*, efficient) code because processors & memory systems were simpler than they are today.

Simple Examples

Example from Rⁿ Programming Environment, Rice, circa 1984

The Rⁿ point abstraction (*old example, modern compilers*)

```
struct point {    /* Point on the plane of windows */
  int x; int y;
}

void Padd(struct point p, struct point q, struct point * r) {
  r->x = p.x + q.x;
  r->y = p.y + q.y;
}

int main( int argc, char *argv[] )  {
  struct point p1, p2, p3;

  p1.x = 1; p1.y = 1;
  p2.x = 2; p2.y = 2;

  Padd(p1, p2, &p3);

  printf("Result is <%d,%d>.\n", p3.x, p3.y);
}
```

Code for Intel Core 2 Duo

Simple Examples (point add)

gcc 4.1, -S option

```
_main:  (some boilerplate code ellided for brevity's sake)
L5:
        popl    %ebx
        movl    $1, -16(%ebp)
        movl    $1, -12(%ebp)
        movl    $2, -24(%ebp)
        movl    $2, -20(%ebp)
        leal    -32(%ebp), %eax
        movl    %eax, 16(%esp)
        movl    -24(%ebp), %eax
        movl    -20(%ebp), %edx
        movl    %eax, 8(%esp)
        movl    %edx, 12(%esp)
        movl    -16(%ebp), %eax
        movl    -12(%ebp), %edx
        movl    %eax, (%esp)
        movl    %edx, 4(%esp)
        call    _PAdd
        movl    -28(%ebp), %eax
        movl    -32(%ebp), %edx
        movl    %eax, 8(%esp)
        movl    %edx, 4(%esp)
        leal    LC0-"L00000000001$pb"(%ebx), %eax
        movl    %eax, (%esp)
        call    L_printf$stub
        addl    $68, %esp
        popl    %ebx
        leave
```

Assignments to p1 and p2

Setup for call to PAdd

Setup for call to printf

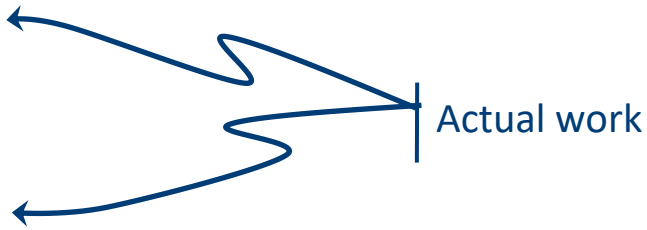
Address calculation for format string in printf call

Simple Examples (point add)

gcc 4.1, -S option

Code for PAdd

```
_PAdd:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %edx
        movl    16(%ebp), %eax
        addl    %eax, %edx
        movl    24(%ebp), %eax
        movl    %edx, (%eax)
        movl    12(%ebp), %edx
        movl    20(%ebp), %eax
        addl    %eax, %edx
        movl    24(%ebp), %eax
        movl    %edx, 4(%eax)
        leave
        ret
```



The code does a lot of work to execute two add instructions.

- factor of 10 in overhead
- and a window system does a lot of point adds

Code optimization (careful compile-time reasoning & transformation) can make matters better.

N.B.: We had this problem in the early 1980s, with full optimization. The same code, compiled with gcc for a modern Intel processor, hits the same problem. The difficulty lies in the translation of the point abstraction and C's rules for parameter passing, not in the specific compiler technology or microprocessor model.

Simple Examples (point add)

gcc 4.1, -O3 option

```
_main:  (some boilerplate code elided for brevity's sake)
L5:
        popl    %ebx
        subl    $20, %esp
        movl    $3, 8(%esp)
        movl    $3, 4(%esp)
        leal    LC0-"L00000000001$pb"(%ebx), %eax
        movl    %eax, (%esp)
        call    L_printf$stub
        addl    $20, %esp
        popl    %ebx
        leave
        ret
```

It moved PAdd inline and folded the known constant values of p1 and p2.

With the right information, a good compiler can work wonders.

- It kept the implementation of PAdd around because it could not tell if it was needed by a call in another file

  → Consequence of separate compilation

What if the compiler did not know the values of p1 and p2?

This particular problem is inherently interprocedural — that is, the compiler must analyze and optimize multiple procedures at the same time to find the inefficiency and improve the code. Inline substitution converts the interprocedural problem into a single-procedure issue. -O3 uses inline substitution.

Simple Examples (point add)

gcc 4.1, -O3 option

```
_main:  (some boilerplate code ellided for brevity's sake)
L5:
        popl    %ebx
        subl    $20, %esp
        movl    _one-"L00000000001$pb"(%ebx), %eax
        addl    _two-"L00000000001$pb"(%ebx), %eax
        movl    %eax, 8(%esp)
        movl    %eax, 4(%esp)
        leal    LC0-"L00000000001$pb"(%ebx), %eax
        movl    %eax, (%esp)
        call    L_printf$stub
        addl    $20, %esp
        popl    %ebx
        leave
        ret
```

The optimizer inlined PAdd

The optimizer recognized that
p1.x = p1.y and p2.x = p2.y
so
p1.x + p2.x = p1.y + p2.y.

If I make PAdd static (and, therefore, hidden), it deletes the code for PAdd

This code is from a more general version of "main".

→ We put 1 and 2 into global variables names "one" and "two".

→ gcc inlined PAdd and subjected the arguments to local optimization.

→ Because it did not know the values, gcc could not eliminate the adds. It did, however, recognize that the second one was redundant.

⇾ gcc did well on this example.

Simple Examples (point add)

In truth, I lied to simplify the example.

- To save space, the graduate student who wrote the code declared x and y as **short int**, not **int**
  - Each instance of x or y was converted from short to int on the way into the PAdd() call and back to int once inside PAdd().
  - The return values were converted from **short int** to **int** inside PAdd() and back to short in the calling routine.
- The code for each call was much, much longer than the contents of PAdd(). Roughly fifty operations to perform two additions.
- The number of calls was an order of magnitude larger than the number of active points, so the four-byte-per-point savings from short was dwarfed by the code space required for shorts.

Eventually, we converted the call to a macro. That made a significant difference in the cost and allowed the hardware/software combination to keep up with mouse movement.