CENG513 Compiler Design and Construction
Introduction

Note by Işıl ÖZ:
Our slides are adapted from Cooper and Torczon's slides that are prepared for COMP 412 at Rice.

Course Overview

- Instructor: Işıl ÖZ, isiloz@iyte.edu.tr

- Lectures: Friday 09:45-12:30, D-4
- Textbook: Engineering a Compiler (Cooper and Torczon)
- Practical examples with LLVM framework
- Slides/Code Samples: On MS Teams before the lecture, code: kg300y4
- Grading
  - 20% Midterm - Take home
  - 40% Final - Take home
  - 20% Prog. Assignments
  - 20% Paper Reproduce and Presentation

Course Content (Tentative)

- Scanning
- Parsing
- Intermediate Representations
- Code Optimizations
- Instruction Selection
- Instruction Scheduling
- Register Allocation

Compilers

- What is a compiler?
  - A program that translates a program written in one language into 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* executing that program

- C is typically compiled, Python is typically interp
- 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 Compiler Construction?

- 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
    - Compiler has become a prime determiner of performance
- Compilers are interesting
  - Compilers include many applications of theory to practice
  - Writing a compiler exposes 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

Reducing the Price of Abstraction

Computer Science is the art of creating virtual objects and making them useful.

- We invent abstractions and uses them
- We invent ways to make them efficient
- Programming is the way we realize these inventions

Well-written compilers make abstraction affordable

- Cost of executing code should reflect the underlying work rather than the way the programmer chose to write it
- Change in expression should bring small performance change
- Cannot expect compiler to devise better algorithms
  - Don't expect bubblesort to become quicksort

Simple Examples

All data collected with gcc 4.1, -O3, running on a queiscent, multiuser Intel T9600 @ 2.8 GHz

Which is faster?

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      A[i][j] = 0;

for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      A[j][i] = 0;

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

All three loops have distinct performance.

0.51 sec on 10,000 x 10,000 array

1.65 sec on 10,000 x 10,000 array

0.11 sec 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.

Simple Examples

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

Abstraction has its price

```
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);
}
```

Simple Example (point add)

gcc 4.1, -S option

```
_main:  (some code skipped 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
        ret
```

Code for Intel Core 2 Duo

Assignments to p1 and p2

Setup for call to PAdd

Setup for call to printf

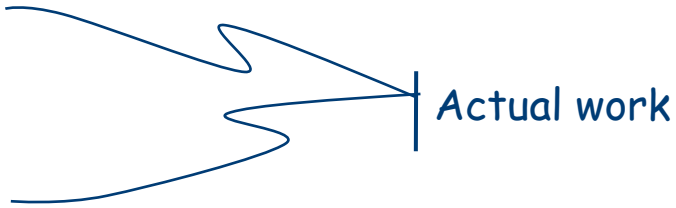
Address calculation for format string in printf call

Simple Example (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.

→ And a window system does a lot of point adds

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

Simple Example (point add)

gcc 4.1, -S –O3 option

```
_main:  (some code skipped 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 inlined PAdd and folded the known constant values of p1 and p2.

With the right information, a good compiler can work wonders.
→ It kept the body of PAdd because it could not tell if it was dead

What if it could not recognize the values of p1 and p2?

Simple Example (point add)

gcc 4.1, -S -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
```

I put 1 and 2 in global variables named "one" and "two".

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.

This code shows the more general version.
It inlined PAdd and subjected the arguments to local optimization.
It still had to perform the adds, but it recognized that the second one was redundant.
→ Gcc did a good job on this example.

Why Compilers?

Compiler construction poses challenging and interesting problems:

- Compilers must process large inputs, perform complex algorithms, 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

How Compilers?

Compiler construction involves ideas from many different parts of computer science

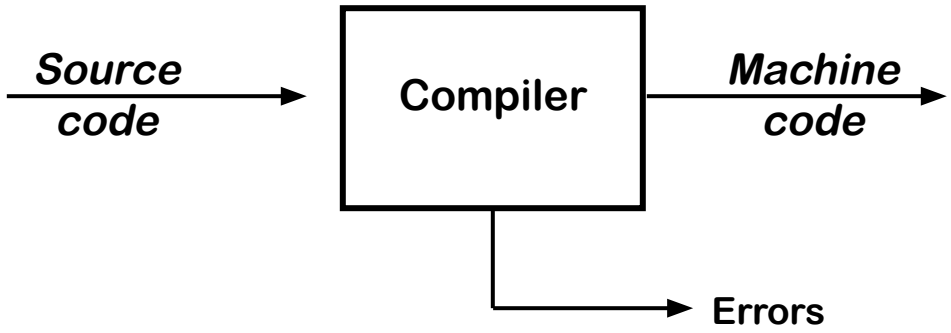
Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms, union-find Dynamic programming
Theory	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Pipeline & hierarchy management Instruction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

Why Does This Matter Today?

In the last years, most processors have gone multicore

- The era of clock-speed improvements is drawing to an end
  - Faster clock speeds mean higher power (n^2 *effect)*
  - Smaller wires mean higher resistance for on-chip wires
- For the near term, performance improvement will come from placing multiple copies of the processor (*core*) on a single die
  - Classic programs, written in old languages, are not well suited to capitalize on this kind of multiprocessor parallelism
    - → Parallel languages, some kinds of OO systems, functional languages
  - Parallel programs require sophisticated compilers

High-level View of a Compiler

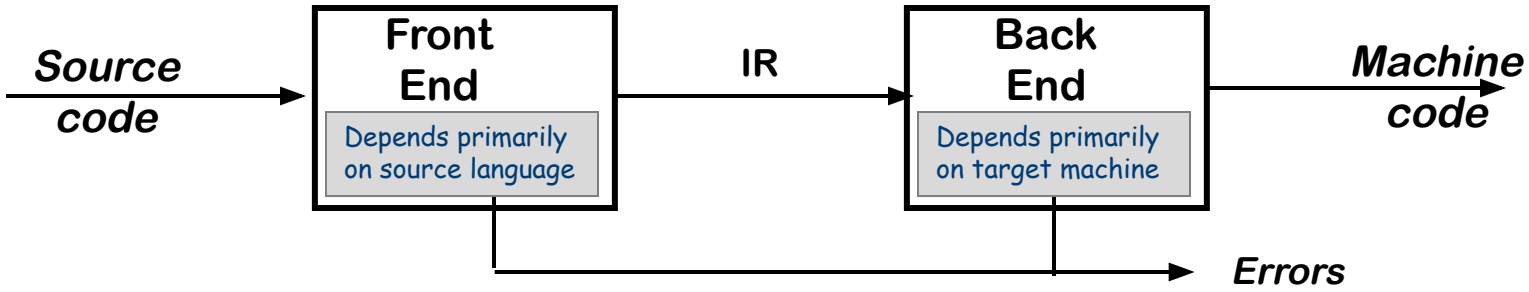


Implications

- Must recognize legal (and illegal) programs
- Must generate correct code
- Must manage storage of all variables (and code)
- Must agree with OS & linker on format for object code

Big step up from assembly language—use higher level notations

Traditional Two-Phase Compiler



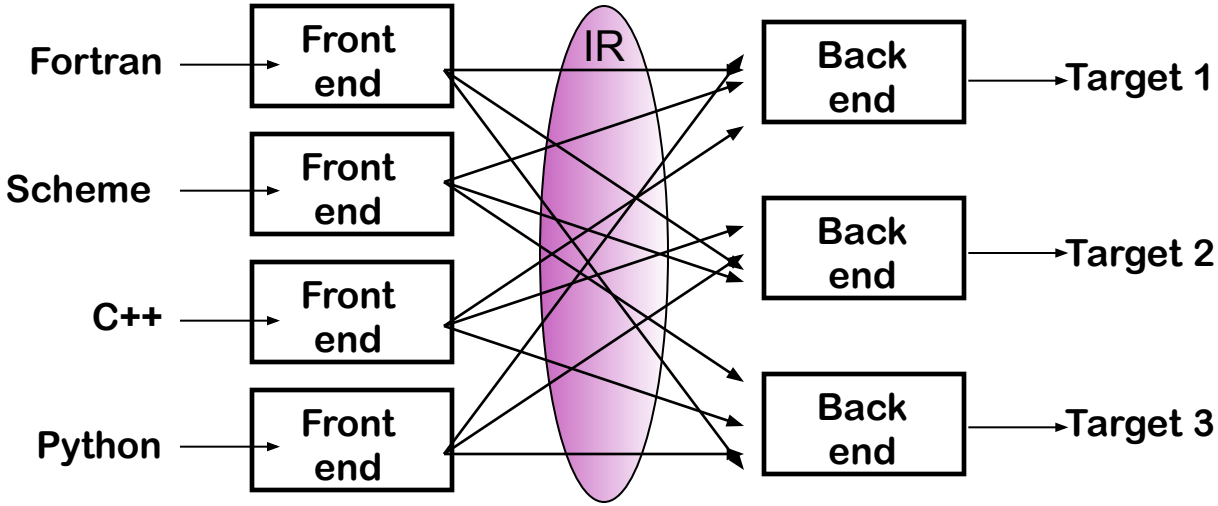
Implications

- Use an intermediate representation (IR)
- Front end maps legal source code into IR
- Back end maps IR into target machine code
- Admits multiple front ends & multiple passes (*better code*)

Classic principle from software engineering: Separation of concerns

Typically, front end is O(n) or O(n log n), while back end is NP-Complete

A Common Fallacy



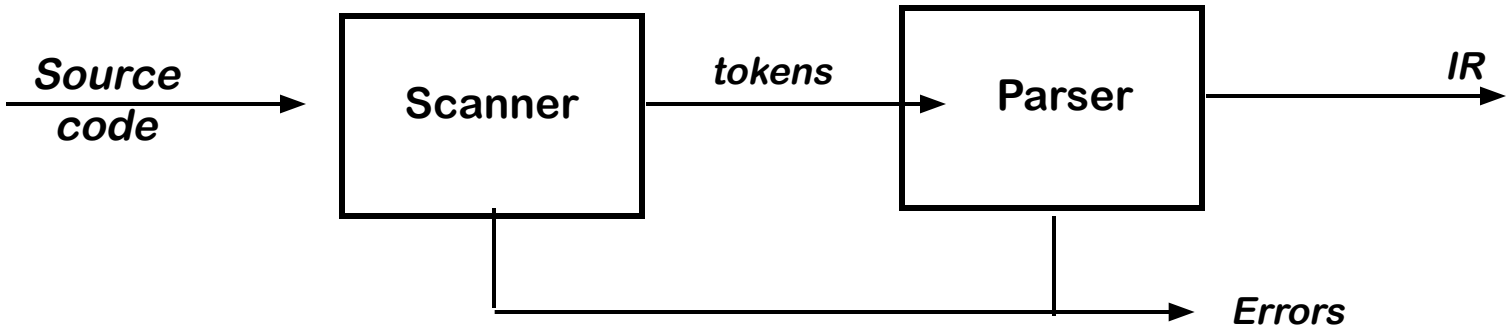
Can we build *n* x *m* compilers with *n*+*m* components?

- Must encode all language specific knowledge in each front end
- Must encode all features in a single IR
- Must encode all target specific knowledge in each back end

Successful in systems with assembly level (or lower) IRs

e.g., gcc's rtl or llvm ir

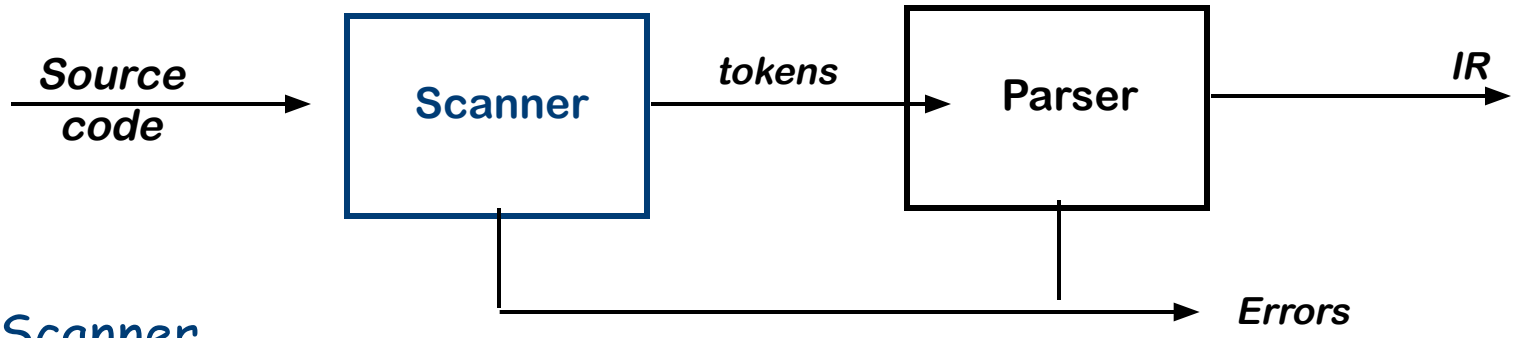
The Front End



Responsibilities

- Recognize legal (& illegal) programs
- Report errors in a useful way
- Produce IR & preliminary storage map
- Shape the code for the rest of the compiler
- Much of front end construction can be automated

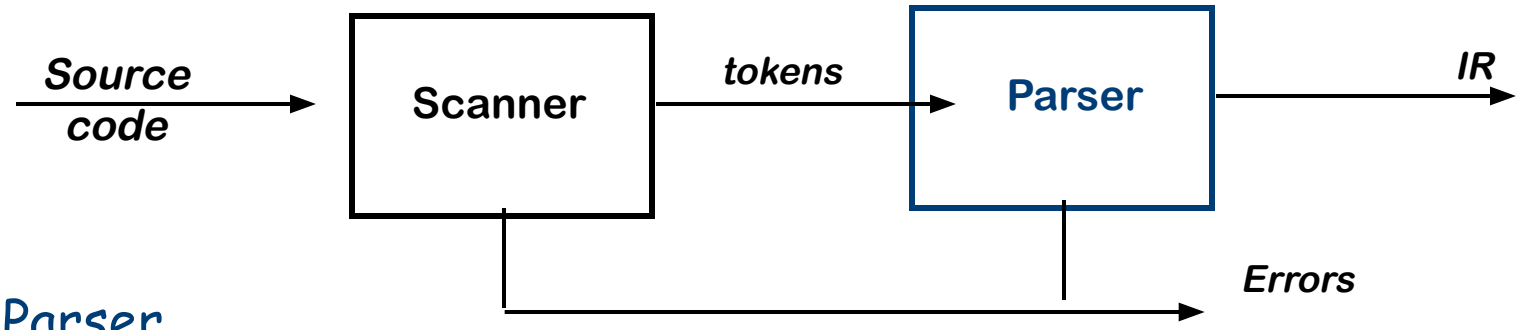
The Front End



Scanner

- Maps character stream into words—the basic unit of syntax
- Produces pairs — a word & its part of speech
  x = x + y ; *becomes* <id,x> = <id,x> + <id,y> ;
  — *word ≅ lexeme, part of speech ≅ token type, pair ≅ a token*
- Typical tokens include *number*, *identifier*, +, -, new, *while*, *if*
- Speed is important

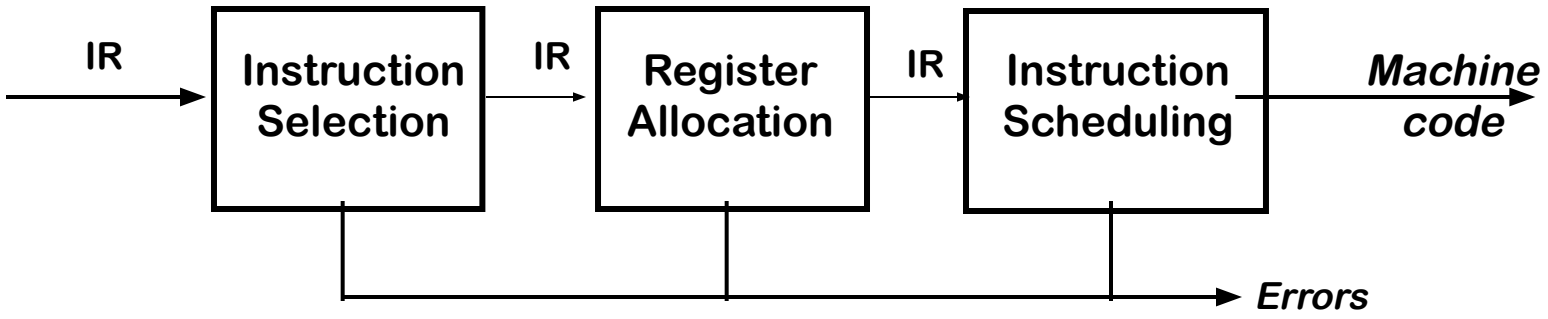
The Front End



Parser

- Recognizes context-free syntax & reports errors
- Guides context-sensitive ("semantic") analysis (*type checking*)
- Builds IR for source program

The Back End

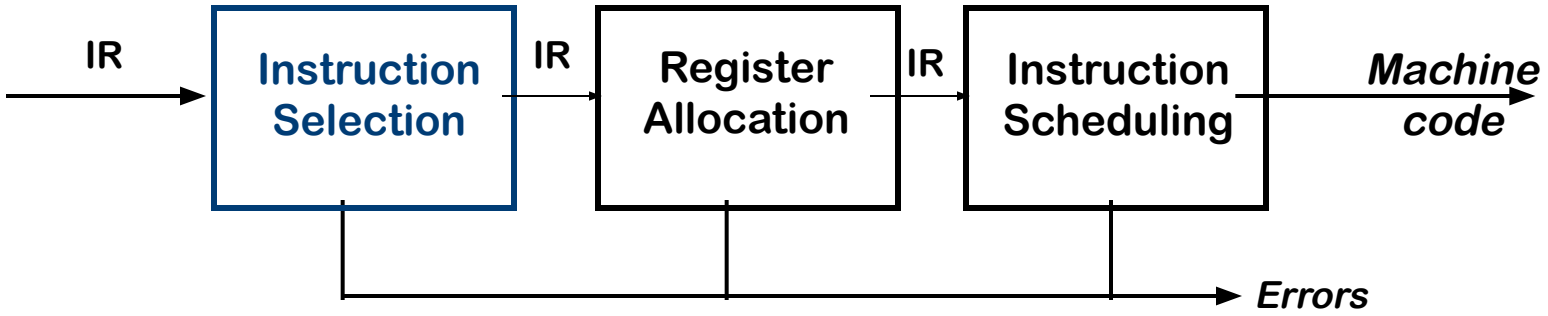


Responsibilities

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces

Automation has been *less* successful in the back end

The Back End



Instruction Selection

- Produce fast, compact code
- Take advantage of target features such as addressing modes
- Usually viewed as a pattern matching problem
  - *ad hoc* methods, pattern matching, dynamic programming
  - Form of the IR influences choice of technique

Standard goal has become "locally optimal" code.

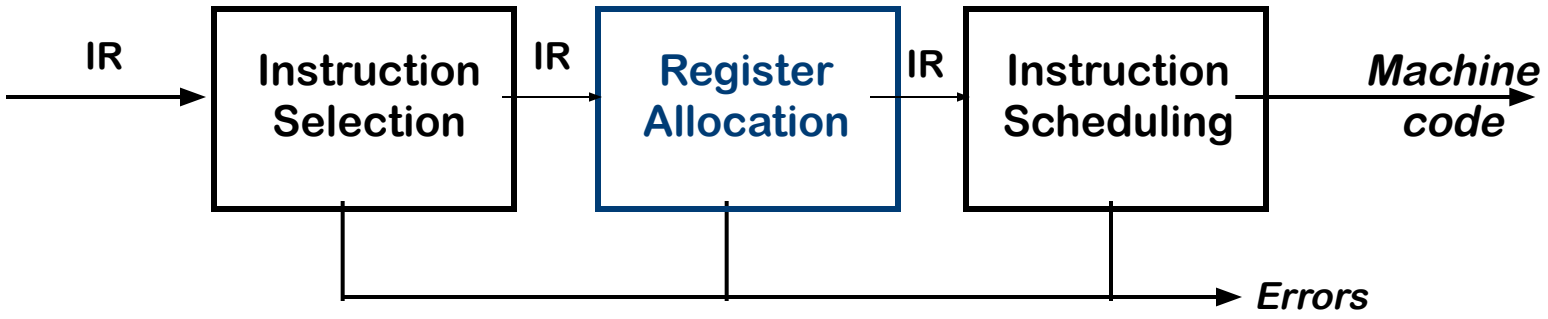
Instruction Selection $w \leftarrow w \times 2 \times x \times y \times z$

```
loadAI  r_arp, @w ⇒ r_w      // load 'w'
loadI   2         ⇒ r_2      // constant 2 into r_2
loadAI  r_arp, @x ⇒ r_x      // load 'x'
loadAI  r_arp, @y ⇒ r_y      // load 'y'
loadAI  r_arp, @z ⇒ r_z      // load 'z'
mult    r_w, r_2  ⇒ r_w      // r_w ← w × 2
mult    r_w, r_x  ⇒ r_w      // r_w ← (w × 2) × x
mult    r_w, r_y  ⇒ r_w      // r_w ← (w × 2 × x) × y
mult    r_w, r_z  ⇒ r_w      // r_w ← (w × 2 × x × y) × z
storeAI r_w       ⇒ r_arp, 0 // write r_w back to 'w'
```

The book gives low-level examples in a notation ILOC (intermediate language for an optimizing compiler)
You can think it as the assembly language for a simple RISC machine.

r_{arp} : a register that contains the start of data storage for the current procedure, also known as the activation record pointer

The Back End



Register Allocation

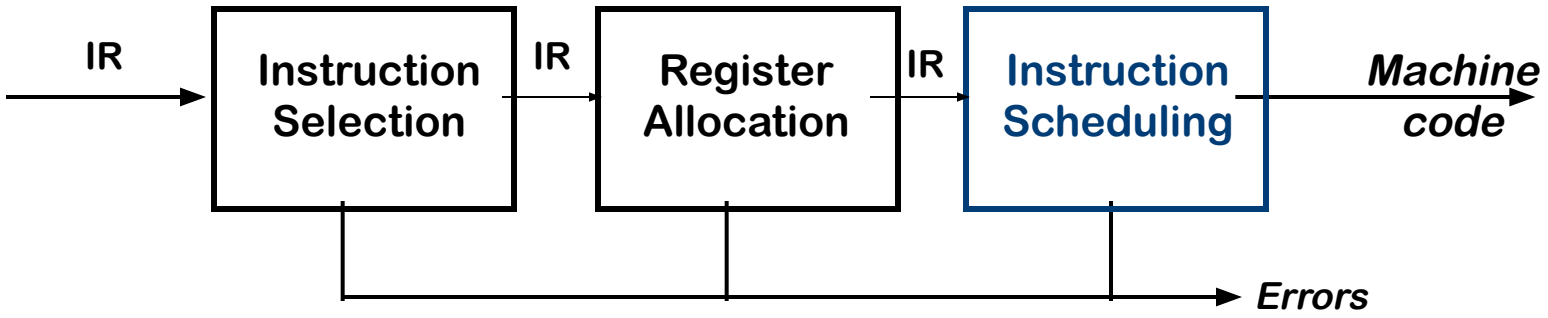
- Have each value in a register when it is used
- Manage a limited set of resources
- Can change instruction choices & insert LOADs & STOREs
- Optimal allocation is NP-Complete in most settings

Compilers approximate solutions to NP-Complete problems

Register Allocation for $w \leftarrow w \times 2 \times x \times y \times z$

```
loadAI   r_arp, @w ⇒ r1        // load 'w'
add      r1, r1    ⇒ r1        // r1 ← w × 2
loadAI   r_arp, @x ⇒ r2        // load 'x'
mult     r1, r2    ⇒ r1        // r1 ← (w × 2) × x
loadAI   r_arp, @y ⇒ r2        // load 'y'
mult     r1, r2    ⇒ r1        // r1 ← (w × 2 × x) × y
loadAI   r_arp, @z ⇒ r2        // load 'z'
mult     r1,r2     ⇒ r1        // r1 ← (w × 2 × x × y) × z
storeAI  r1        ⇒ r_arp, @w // write r_w back to 'w'
```

The Back End



Instruction Scheduling

- Avoid hardware stalls and interlocks
- Use all functional units productively
- Can increase lifetime of variables (changing the allocation)

Optimal scheduling is NP-Complete in nearly all cases

Heuristic techniques are well developed

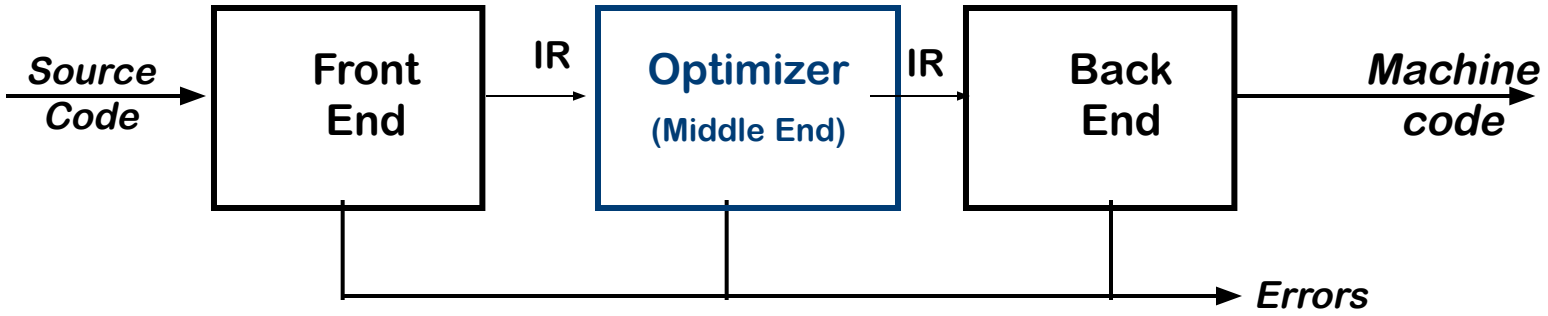
Instruction Scheduling $w \leftarrow w \times 2 \times x \times y \times z$

Start	End					
1	3	loadAI	r_{arp}, @w	$\Rightarrow$ r_1	// load 'w'	
4	4	add	r_1, r_1	$\Rightarrow$ r_1	// $r_1 \leftarrow w \times 2$	
5	7	loadAI	r_{arp}, @x	$\Rightarrow$ r_2	// load 'x'	
8	9	mult	r_1, r_2	$\Rightarrow$ r_1	// $r_1 \leftarrow (w \times 2) \times x$	
10	12	loadAI	r_{arp}, @y	$\Rightarrow$ r_2	// load 'y'	
13	14	mult	r_1, r_2	$\Rightarrow$ r_1	// $r_1 \leftarrow (w \times 2 \times x) \times y$	
15	17	loadAI	r_{arp}, @z	$\Rightarrow$ r_2	// load 'z'	
18	19	mult	r_1, r_2	$\Rightarrow$ r_1	// $r_1 \leftarrow (w \times 2 \times x \times y) \times z$	
20	22	storeAI	r_1	$\Rightarrow$ r_{arp}, @w	// write r_w back to 'w'	

Instruction Scheduling $w \leftarrow w \times 2 \times x \times y \times z$

Start	End			
1	3	loadAI	r_{arp}, @w $\Rightarrow$ r_1	// load 'w'
2	4	loadAI	r_{arp}, @x $\Rightarrow$ r_2	// load 'x'
3	5	loadAI	r_{arp}, @y $\Rightarrow$ r_3	// load 'y'
4	4	add	r_1, r_1 $\Rightarrow$ r_1	// $r_1 \leftarrow w \times 2$
5	6	mult	r_1, r_2 $\Rightarrow$ r_1	// $r_1 \leftarrow (w \times 2) \times x$
6	8	loadAI	r_{arp}, @z $\Rightarrow$ r_2	// load 'z'
7	8	mult	r_1, r_3 $\Rightarrow$ r_1	// $r_1 \leftarrow (w \times 2 \times x) \times y$
9	10	mult	r_1, r_2 $\Rightarrow$ r_1	// $r_1 \leftarrow (w \times 2 \times x \times y) \times z$
11	13	storeAI	r_1 $\Rightarrow$ r_{arp}, @w	// write r_w back to 'w'

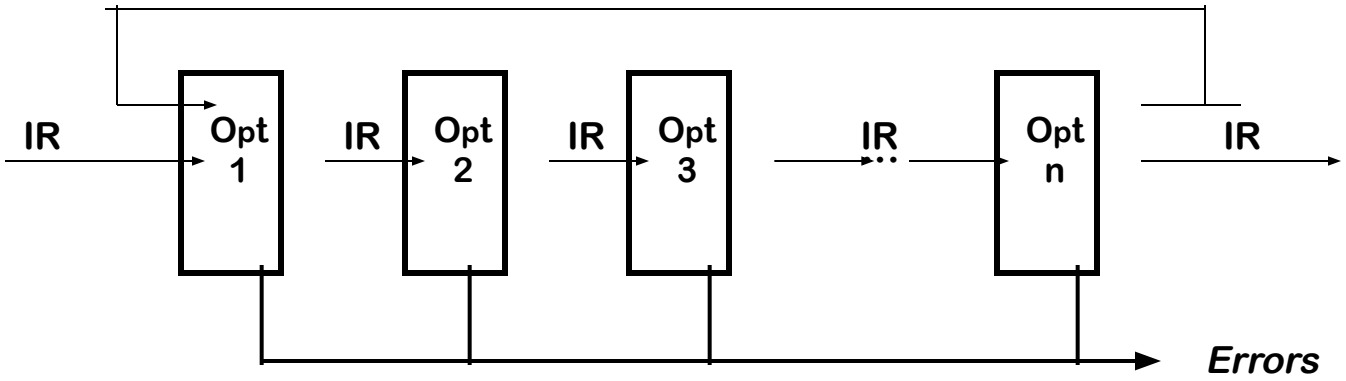
Traditional Three-Phase Compiler



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 (or Middle End)



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

Optimization Example

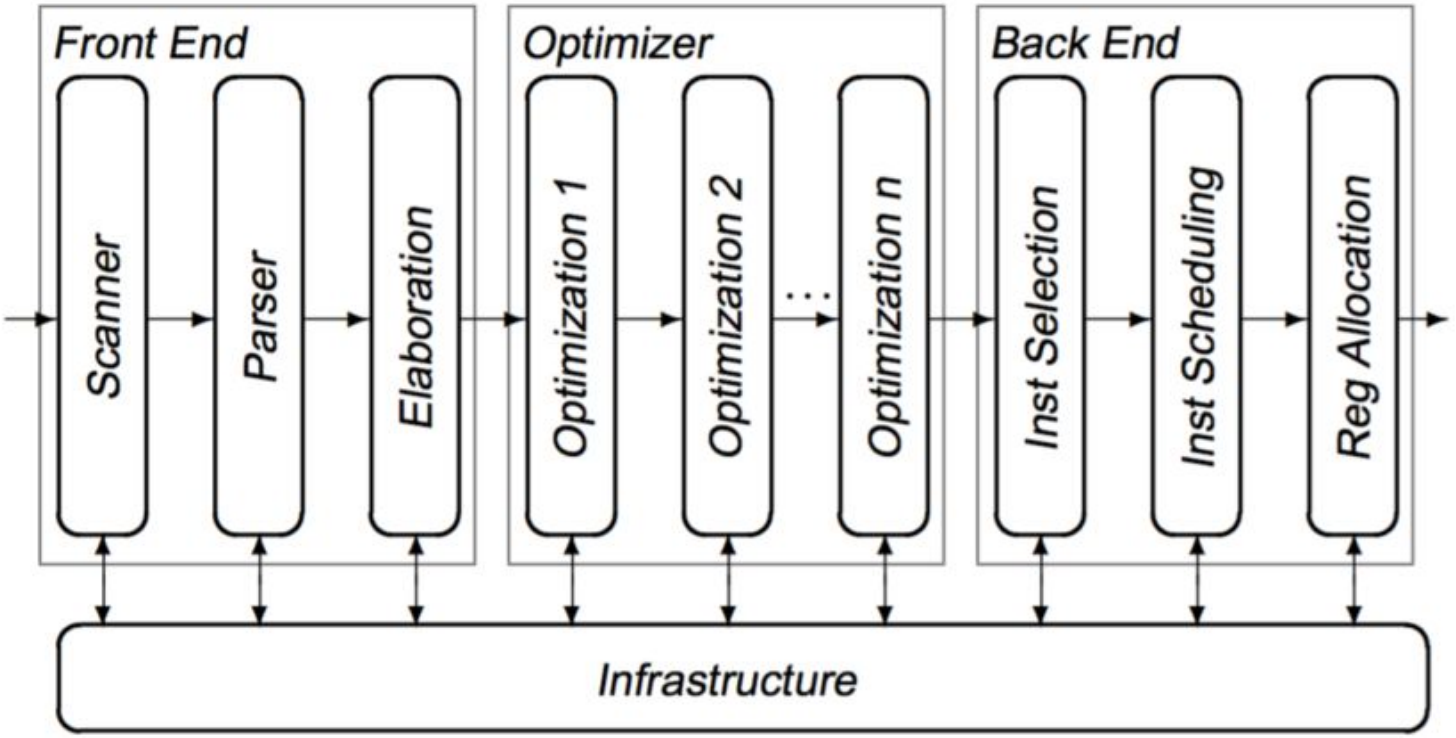
```
x ← ···
y ← ···
w ← 1
for i = 1 to n
  read z
  w ← w × 2 × x × y × z
  end
```

Surrounding Context

```
x ← ···
y ← ···
w ← 1
t ← 2 × x × y
for i = 1 to n
  read z
  w ← w × z × t
  end
```

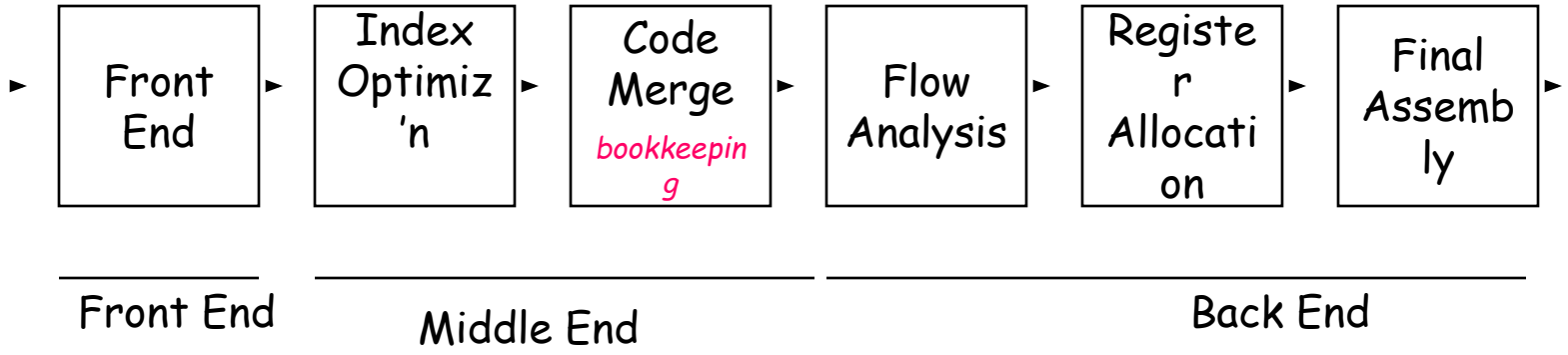
Improved Code

Traditional Three-part Compiler



Classic Compilers

1957: The FORTRAN Automatic Coding System



- Six passes in a fixed order
- Generated good code
  - Assumed unlimited index registers
  - Code motion out of loops, with ifs and gotos
  - Did flow analysis & register allocation