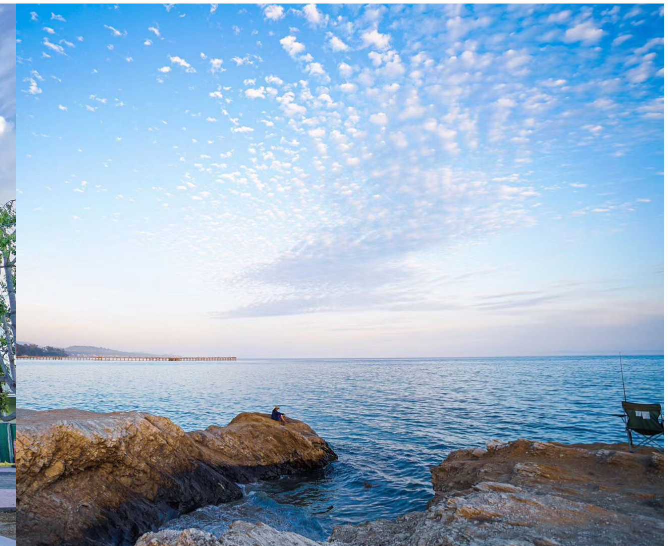
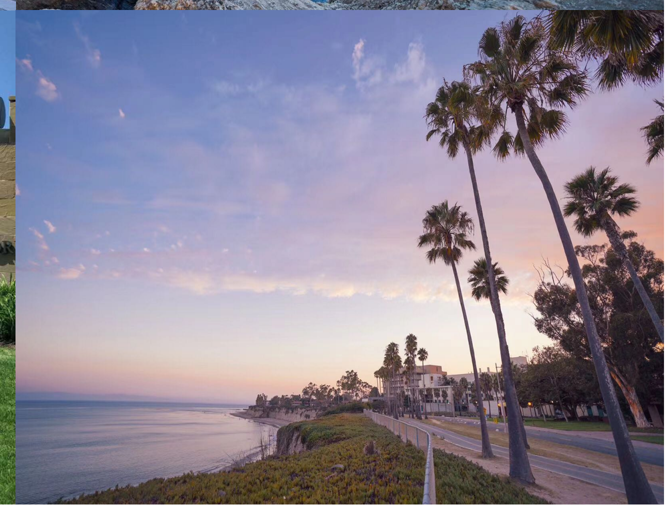
Introduction to CS 293S Advanced Compiler Technology --- Code Optimizations for Scalar and Parallel Programs

Yufei Ding

UC SANTA BARBARA





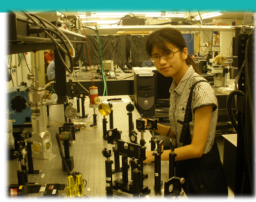
Biography

Transfer to CS,
Working with my Ph.D.
advisor Xipeng Shen

2012

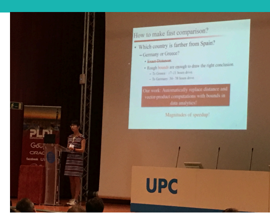
Join CS at UCSB as an
Assistant Professor, as of
Nov, 2017

2017



2005-2012

B.S. and M.S. in
Condensed Matter Physics
+ Laser Optics



2012-2017

My research resides in the
intersection of Compiler
Technology and (Big) Data
Analytics

2017- 2020

Explore Quantum Computing +
Computer Systems
Recently, also system support for
Brain-computer Interfaces

About Me

- Yufei Ding, Assist. Prof in CS
    - addressed by Prof. Ding
- Married, two kids

Research Interest

- Making computing more intelligent and efficient through software systems (**compiler**, runtime, library, tools, etc.)

- And many other interesting research problems (e.g., machine learning, quantum computing, brain-computer interface)

Course Management

- No textbooks.
- Slides define the scope of the course
  - Slides would be posted online after the class

Teaching Philosophy

- Make students think…
- and think critically.

Grading Policy

- 30% assignment (3 homework),
- 20% paper review (2 paper, paper list will be sent out later)
- 20% paper presentation (50 mins)
- 30% project and a final project presentation.
  - 1-3 students in a group.

- No cheating.
- No late submission accepted.

Communication

- See course webpage for other important policy and details. http://www.cs.ucsb.edu/~yufdeiding/cs293s

- Email to yufeiding@cs.ucsb.edu.

Introduction to Code Optimization

- Simple definition: Enhance the quality of a program.

- What is the metric for quality of a program?
- Why is it important to enhance the quality?

Introduction to Code Optimization

- Simple definition: Enhance the quality of a program.
- What is the quality of a program?
  - Speed, energy, power, code size, memory footprint, reliability, security, resilience, readability, extensibility, etc.

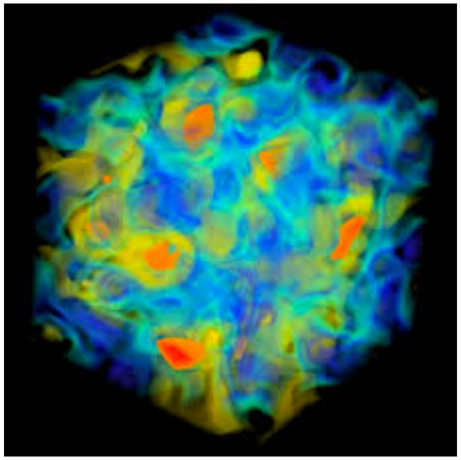
Importance of Code Optimization

- Modern humanity development is based on computing
- Code quality determines the quality of computing and hence the quality of humanity development

Importance of Code Optimization

- Scientifically: scope and precision of scientific simulation, reliability for critical missions
- Economically: “1% performance improvement saves Google millions of dollars” —Google
- Health, defense, …

QCD simulation



Cluster, 1996

Google datacenter

Introduction to Code Optimization

» Simple definition: Enhance the quality of a program.

» Who makes the enhancement?

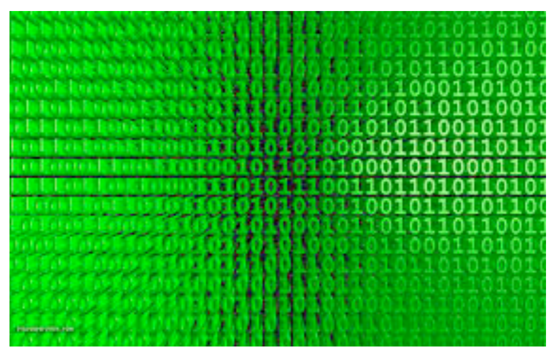
» How to do it?

Introduction to Code Optimization

- Simple definition: Enhance the quality of a program.
- Who makes enhancement?
  - **Compiler**, runtime, programmer
- How to do it? *The core of this course.*
  - Program analysis to understand programs
  - Program transformation to materialize the enhancement

Overview of Compiler

Compiler



Compilers

What is a compiler?

- A program that **translates** a program in one language into a program in another language
- It should improve the program, in some way

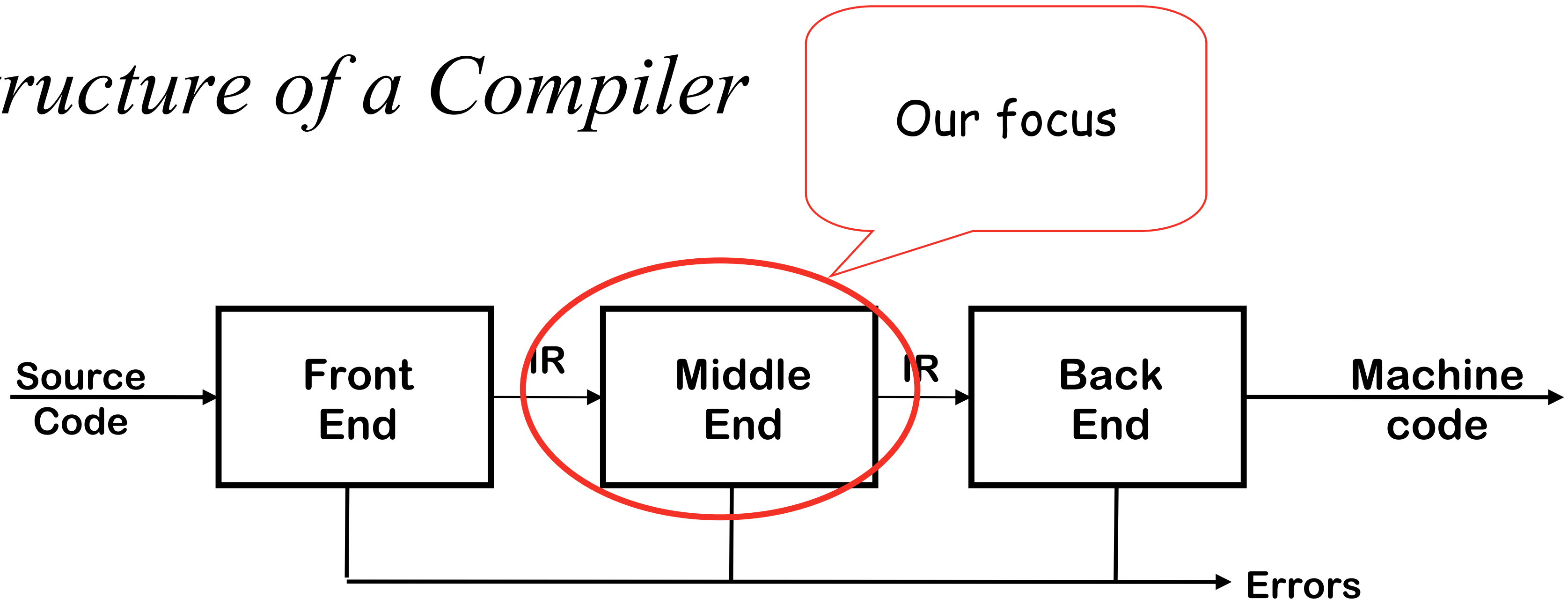
What is an interpreter?

- A program that reads a program and produces the results of **executing** that program

Compilers

- 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

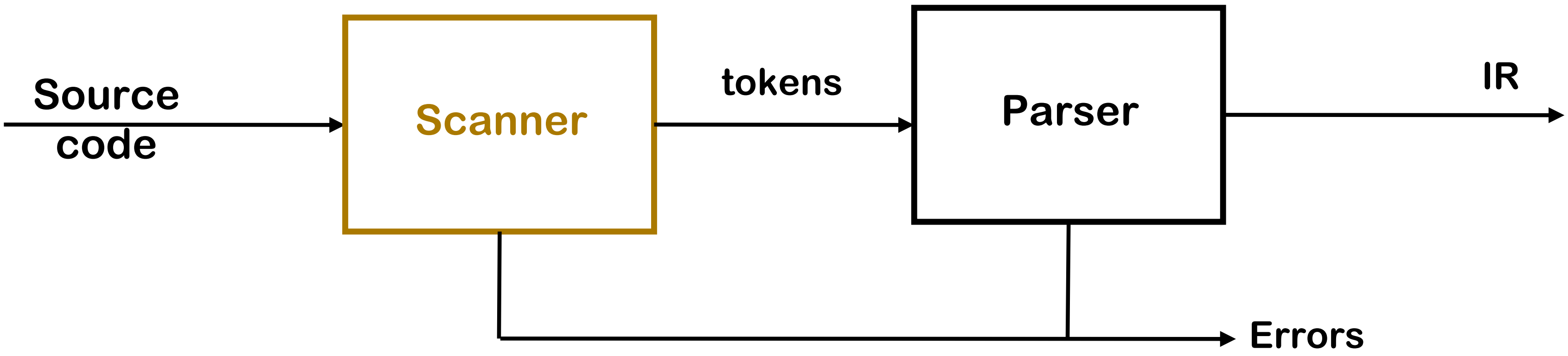
Structure of a Compiler



- 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 and backends
- Middle end with multiple passes for different optimizations

Typically, front end is O(n) or O(n log n), while back end is NPC

The Front End



Scanner

- Break the inputs into individual pieces
- Decide the functionality of each piece

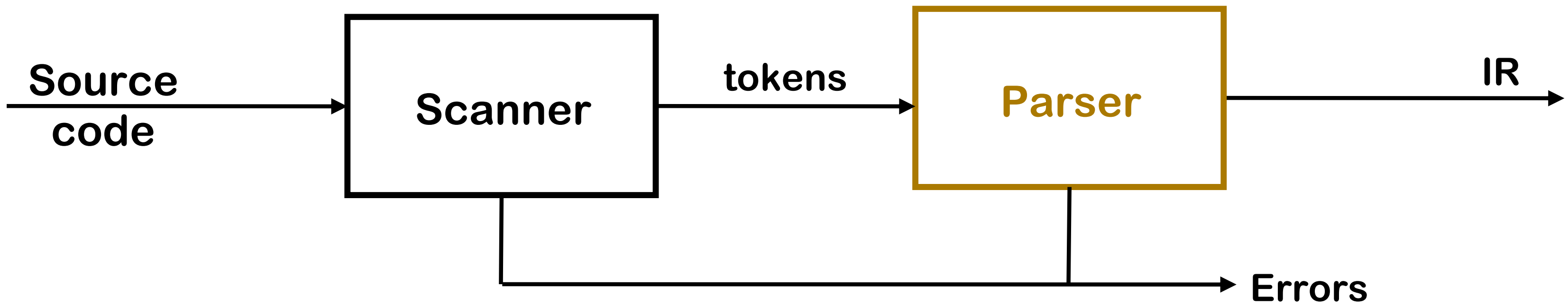
$$x = x + 2 ;$$

becomes <id,x> = <id,x> + <number,2>

- Reports errors
- Analogy:

Dogs are animals. => noun verb noun

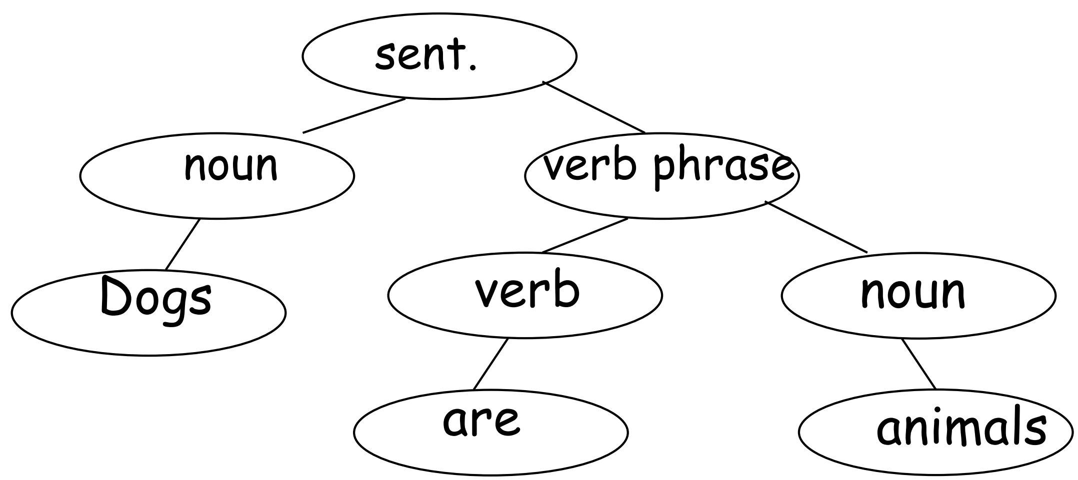
The Front End



Parser

- Organize the pieces back based on some predefined production rules
- Reports errors
- Analogy:

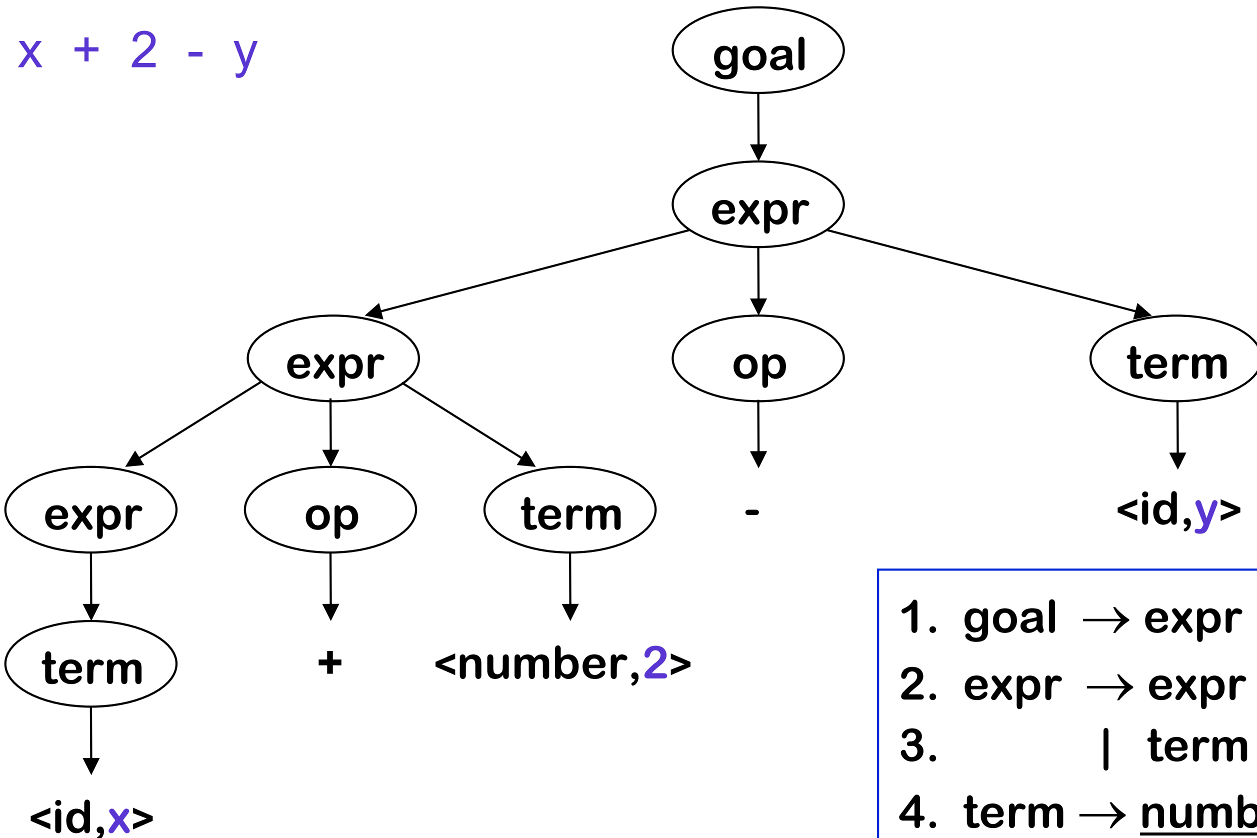
Dogs are animals. ==>



The Front End

A parser can be represented by a tree (parse tree or syntax tree)

x + 2 - y



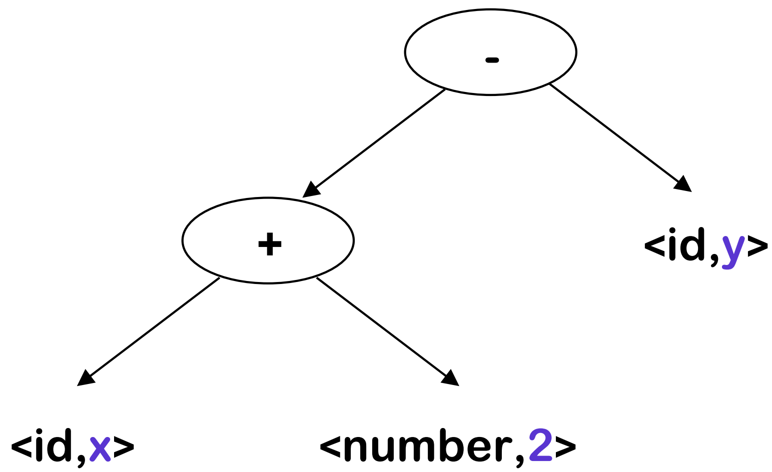
This contains a lot of unneeded Information.

1. goal $\rightarrow$ expr
2. expr $\rightarrow$ expr op term
3. | term
4. term $\rightarrow$ <u>number</u>
5. | <u>id</u>
6. op $\rightarrow$ +
7. | -

The Front End

Compilers often use an abstract syntax tree (AST)

x+2-y

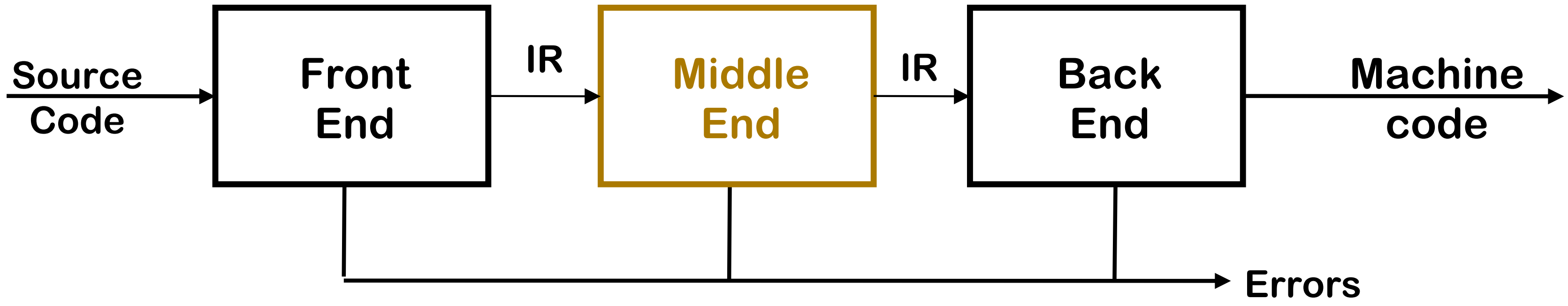


The AST summarizes grammatical structure, without including detail about the derivation

This is much more concise.

AST is one form of intermediate representation (IR)

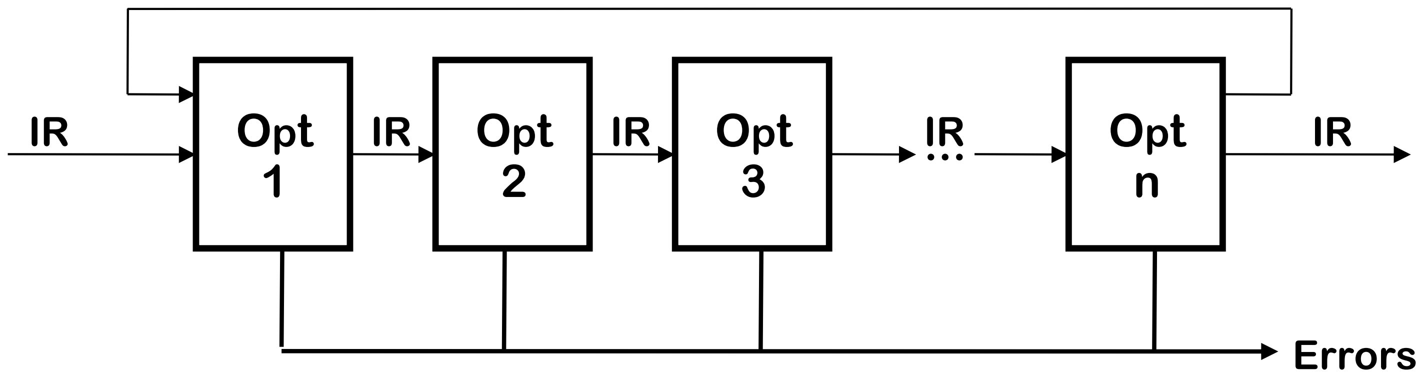
Traditional Three-Pass Compiler



Code Improvement (or <u>Optimization</u>)

- Analyzes IR and rewrites (or <u>transforms</u>) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Definition of "meaning" varies

The Optimizer (or Middle End)

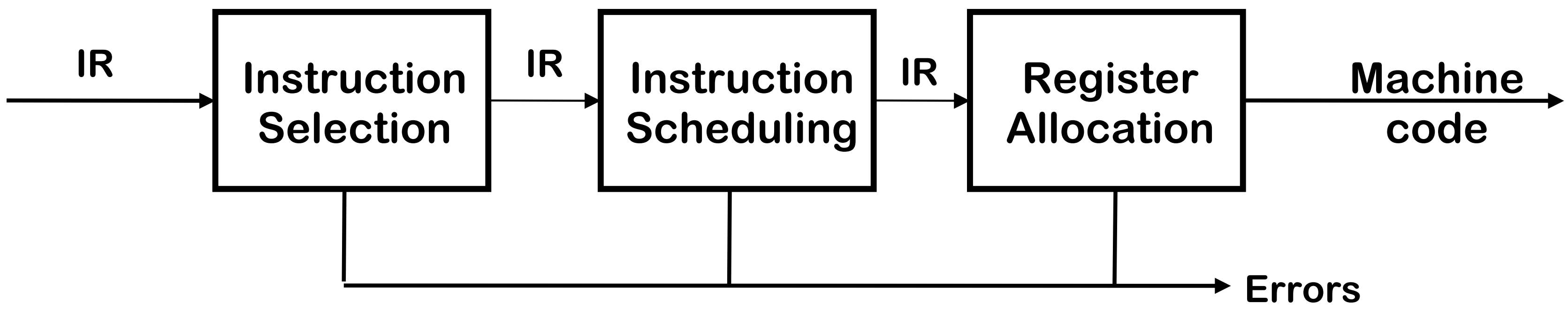


Modern optimizers are structured as a series of passes

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

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 much less successful in the back end

Classification of Compilers

Time of compilation

- Offline compilation
  - e.g., GCC
- Just-In-Time compilation (JIT)
  - e.g. JIT in Java Virtual Machine
  - e.g., Javascript compiler (V8) in Chrome

Unit of compilation

- Function (or Method)
  - e.g., GCC
- Trace
  - e.g., Old Javascript JIT in Mozilla

Considerations of Optimization

- Profitability
- Safety
- Risk

Examples

```
Void f1 (int *px, int *py){
        …
        *px += *py;
        *px += *py;
        …
}
```

```
Void f2 (int *px, int *py){
        …
        *px += 2 * *py;
        …
}
```

- Which example is more efficient? Why?
- Can an optimizing compiler automatically do the transformation, if one is better than the other?

Examples

```
int f();

int func1() {
        return f() + f() + f() + f();
}
```

```
int f();

int func2() {
        return 4*f();
}
```

- Which example is more efficient? Why?
- Can an optimizing compiler automatically do the transformation, if one is better than the other?

Examples

```
int f();

int func1() {
        return f() + f() + f() + f();
}
```

```
int f();

int func2() {
        return 4*f();
}
```

```
int counter = 0;
int f() {
        return counter++;
}
```

Sources of Inefficiencies

from code development

- Programmer
- Source-language abstraction
  - e.g., A[i, j], A[i, j+1], function call

from translation

- Context-oblivious translation
  - a=0;
  - b = b*a;
- Side effects of transformations
  - e.g., compiler introduced load/store to temporary variables

Components in Program Optimization

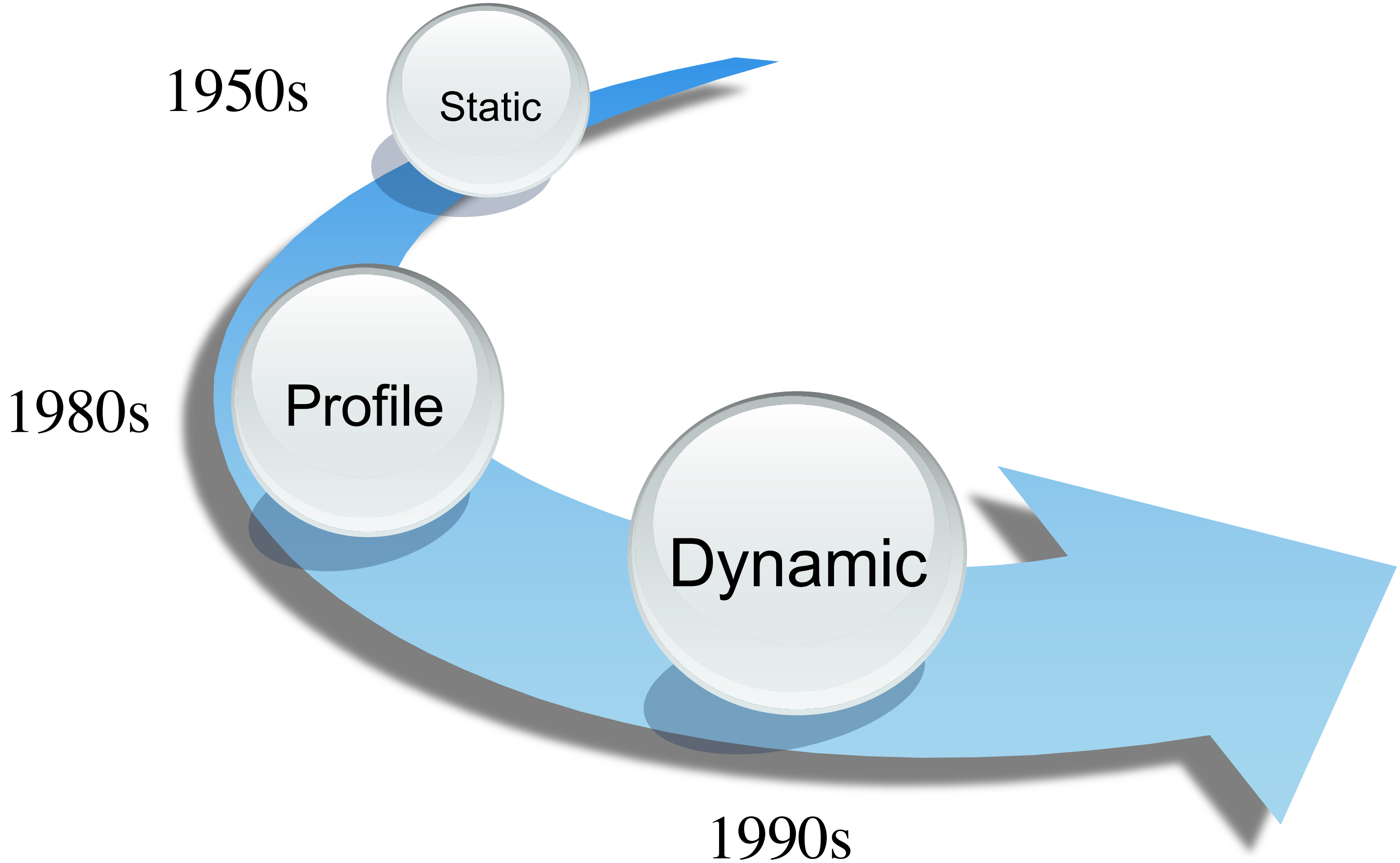
Program Analysis

- Understanding the program
  - Relations among statements (or control flows)
  - Relations among data (or data flows)
  - Invariants in both

Program Transformations

- Enhancing the program
  - Reducing the overhead of abstraction
    - E.g. array-address calculation
  - Taking advantage of special cases
    - E.g. constant propagation
  - Matching processor resources
    - E.g. minimizing memory accesses
    - E.g. parallelization

Three Paradigms of Optimizations



Static Code Analysis

- Example of function inlining:

```
for (i=0; i< n;
i++){
    b = i;
    foo ( );
}
void foo ()
{
   a += b;
   ...
}
```

function inlining →

```
for (i=0; i< n; i++){
    b = i;
    a += b;
     ...
}
```

Q: Should we inline every function call?

Static Code Analysis

- Example of function inlining:

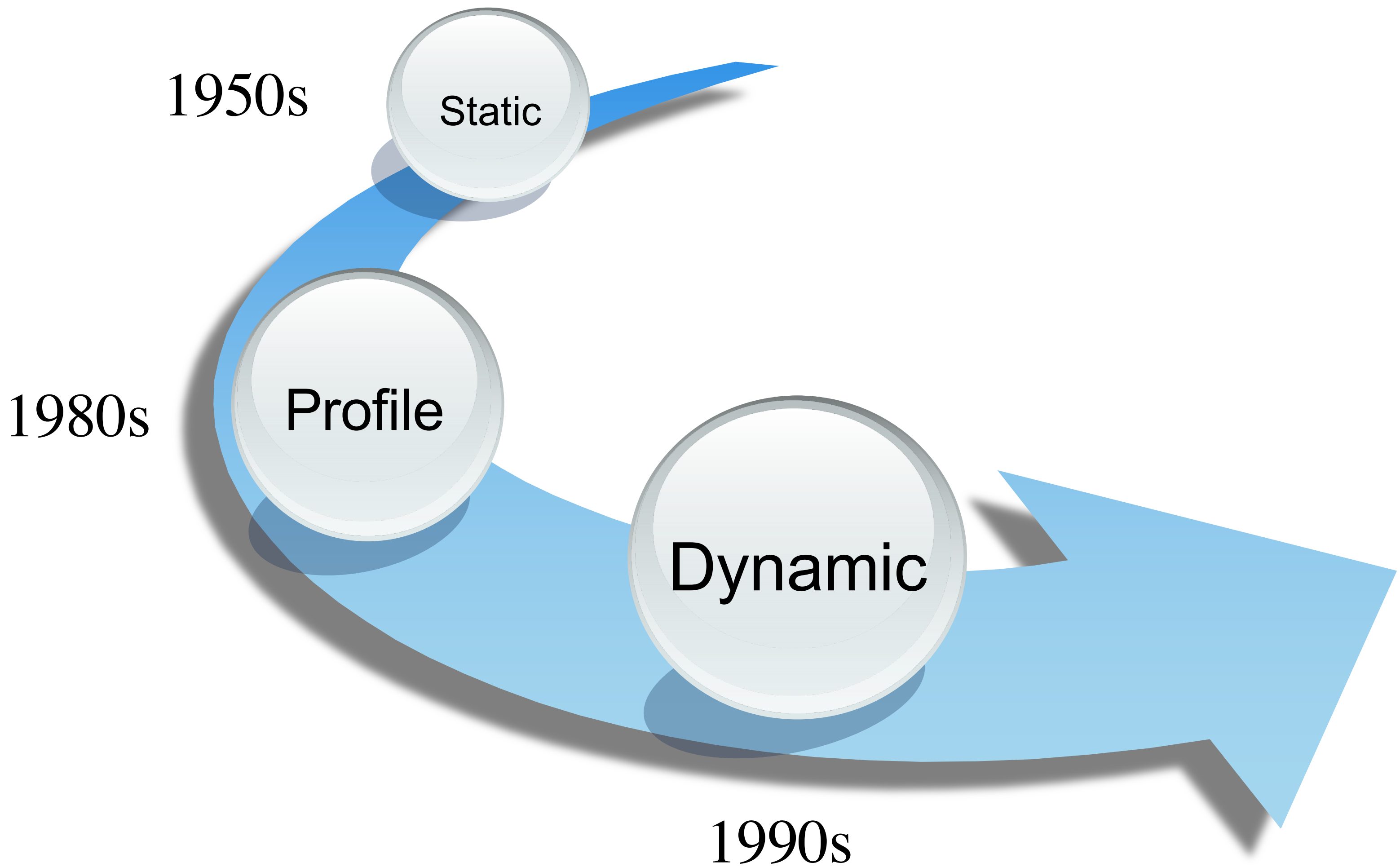
```
for (i=0; i< n;
i++){
    b = i;
    foo ( );
}
void foo ()
{
   a += b;
   ...
}
```

function inlining →

```
for (i=0; i< n; i++){
    b = i;
    a += b;
     ...
}
```

Q: Should we inline every function call?
- code size, recursive calls, register (cache) performance, …

Three Paradigms of Optimizations



Profiling-Based Optimizations

- Instrumentation: Insert some recording statements into the program
- Run the program on some inputs
- Recompile the program according to the observations.

```
for (i=0; i< n; i++){
    foo ( );
}
```

```
_prof_record(n);
for (i=0; i< n; i++){
    foo ( );
}
```

The instrumented codes will be removed when releasing the software. They are costly.

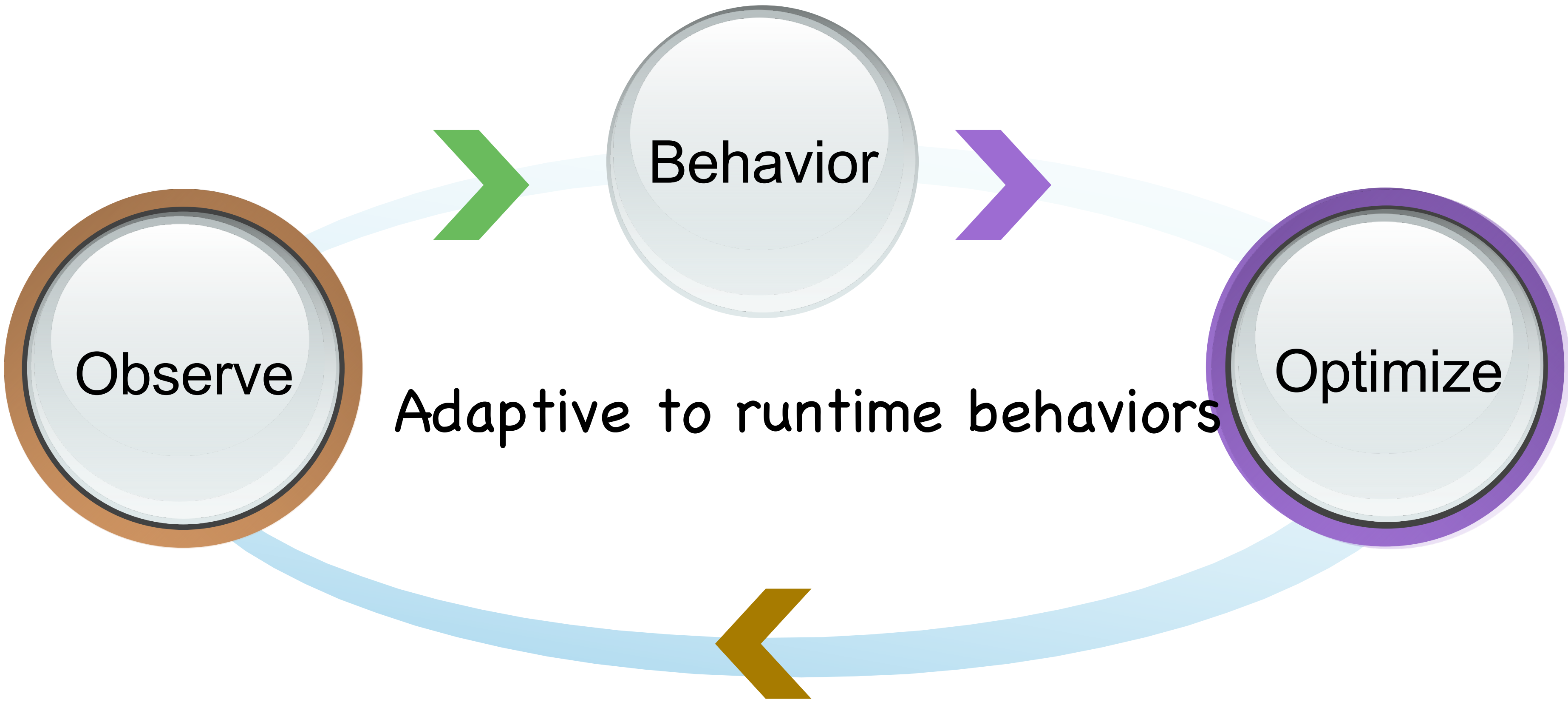
Cons

- Limited by the training runs, hard to adapt to new program inputs.

```
if (xoption > 0)
   n*=1000;
_prof_record(n);
for (i=0; i< n; i++){
     foo ( );
}
```

In all training runs, xoption < 0.

Dynamic Optimizations



Adaptive to runtime behaviors

Widely used in Java, C#, etc

Dynamic Optimizations

- Observe and optimize a program during runtime
- Example: Java Virtual Machine.

```
if (xoption > 0)
  n*=1000;
for (i=0; i< n; i++){
    foo ( );
}
```

During runtime, JVM keeps observing the stack to determine the hotness of a method.

If a method is found to be hot, inline it.

Limitations

- Delay and inaccuracy for being reactive and learning from recent history

```
if (xoption > 0)
   n*=1000;
for (i=0; i< n; i++){
     foo ( );
}
```