PURDUE
UNIVERSITY®

# Introduction to Compilers

Compilers: Principles And Practice

Tiark Rompf

# What is a compiler?

You've all certainly used one . . .

- Translate one programming language to another

## What is a compiler?

You've all certainly used one . . .

- Translate one programming language to another
- Usually: human-friendly to machine friendly

## What is a compiler?

You've all certainly used one . . .

- Translate one programming language to another
- Usually: human-friendly to machine friendly
- Hopefully, use target machine efficiently

# What is a compiler?

You've all certainly used one . . .

- Translate one programming language to another
- Usually: human-friendly to machine friendly
- Hopefully, use target machine efficiently

# What is a compiler?

You've all certainly used one . . .

- Translate one programming language to another
- Usually: human-friendly to machine friendly
- Hopefully, use target machine efficiently

Key:

- A compiler is just another program!

In this class you will learn how to write compilers.

# Why take this class?

Intellectual:

- If you want to understand software, you need to understand compilers
- Touches on all aspects of CS (algorithms, hardware, systems, ...)

# Why take this class?

Intellectual:

- If you want to understand software, you need to understand compilers
- Touches on all aspects of CS (algorithms, hardware, systems, ...)

Pragmatic:

- Hiring managers and PhD admissions know this
- All big companies have compiler teams
  *(Google: V8, Dart, Android, Facebook: HHVM, Infer, Apple: Swift, LLVM, JavaScript Core, Microsoft: .Net, Oracle: Java, ...)*
- **LOTS** of exciting compiler challenges in scaling AI / Deep Learning
  *(GPUs, TensorFlow, PyTorch, ...)*

# Some notable CS 352 alumni

- Filip Pizlo: BS 2003
  now manager of JSC at Apple

- Michael Armbrust: BS 2006
  PhD at Berkeley, now tech lead of Spark SQL at DataBricks

- Ben Titzer: BS 2002
  PhD at UCLA, now tech lead of V8 at Google

# CS 352 Logistics

Presence:

- Lectures: Tuesday, Thursday 12:00pm – 1:15pm
- PSO sessions: Wednesday 1:30pm – 3:20pm, Friday 3:30pm – 5:20pm

Online:

- Website: https://tiarkrompf.github.io/cs352/
- Piazza: https://piazza.com/purdue/fall2017/cs352/

Grading:

- 30% midterm, 30% final, 40% projects
- Project 1 is on the website. Due Monday night!

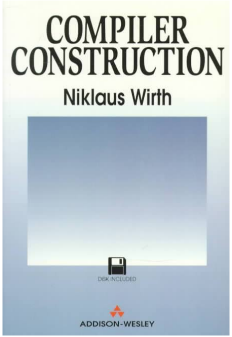
# CS 352 Teaching Assistants

- Grégory Essertel
  (co-teacher)

- James Decker

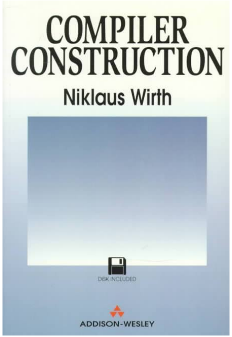
- Md Nasim

# Compiler Books




Niklaus Wirth: Compiler Construction (1996,2014)

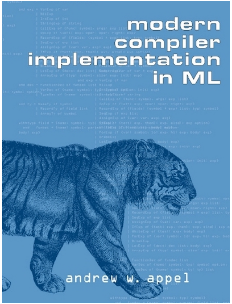
# Compiler Books




Niklaus Wirth: Compiler Construction (1996,2014)

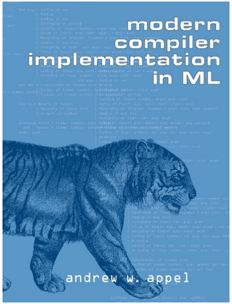
*"To keep both the resulting compiler reasonably simple . . . we postulate an architecture according to our own choice."*

# Compiler Books




Andrew Appel: Modern Compiler Implementation in ML/Java/C

# Compiler Books




Andrew Appel: Modern Compiler Implementation in ML/Java/C

> *"A student who implements all the phases described in Chapters 1-11 of the book will have a working compiler." (Chapter 12)*

# CS 352 Structure

Part 1:

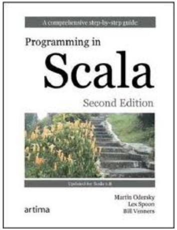
- Start with tiny language that we know how to map to machine code
- Gradually add features
- Simple, but not too simple

Part 2:

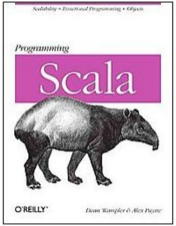
- Analysis and optimizations
- Runtime systems (VMs, garbage collection)
- Dynamic language features

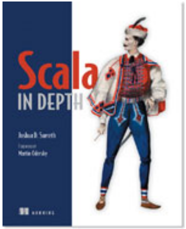
Projects implemented in Scala, languages of study will be Scala subsets.

# Scala








The first part of “Scala for the Impatient” is available for free download.

# Representing Programs

Compilers operate on programs as data

- Unstructured list of characters: source code

  ```
  "1+2*3"
  ```

- Trees or graphs: internal representation

  ```
  (+ 1 (* 2 3))
  ```

- List of machine instructions

  ```
  movq  $2, %rax
  imulq $3, %rax
  addq  $1, %rax
  ```

# Simple Arithmetic

Abstract syntax (BNF notation):

```
<exp> ::= <num>
       |  <exp> + <exp>
       |  <exp> - <exp>
       |  <exp> * <exp>
       |  <exp> / <exp>
```

BNF = Backus-Naur Form

# Programs as Data

Abstract syntax as Scala data structure:

```scala
abstract class Exp
case class Lit(x: Int) extends Exp
case class Plus(x: Exp, y: Exp) extends Exp
case class Minus(x: Exp, y: Exp) extends Exp
case class Times(x: Exp, y: Exp) extends Exp
case class Div(x: Exp, y: Exp) extends Exp
```

# Writing an Interpreter

```scala
type Val = Int

def eval(e: Exp): Val = e match {
  case Lit(x)       => x
  case Plus(x,y)    => eval(x) + eval(y)
  case Minus(x,y) => eval(x) - eval(y)
  // (more cases ...)
}
```

# Writing an Interpreter

```
type Val = Int

def eval(e: Exp): Val = e match {
  case Lit(x)      => x
  case Plus(x,y)  => eval(x) + eval(y)
  case Minus(x,y) => eval(x) - eval(y)
  // (more cases ...)
}

eval(Plus(Lit(3),Lit(4))) // => 7
```

# Our First Compiler

```scala
type Code = String

def trans(e: Exp): Code = e match {
  case Lit(x)      => s"$x"
  case Plus(x,y)   => s"(${trans(x)} + ${trans(y)})"
  case Minus(x,y) => s"(${trans(x)} - ${trans(y)})"
  // (more cases ...)
}
```

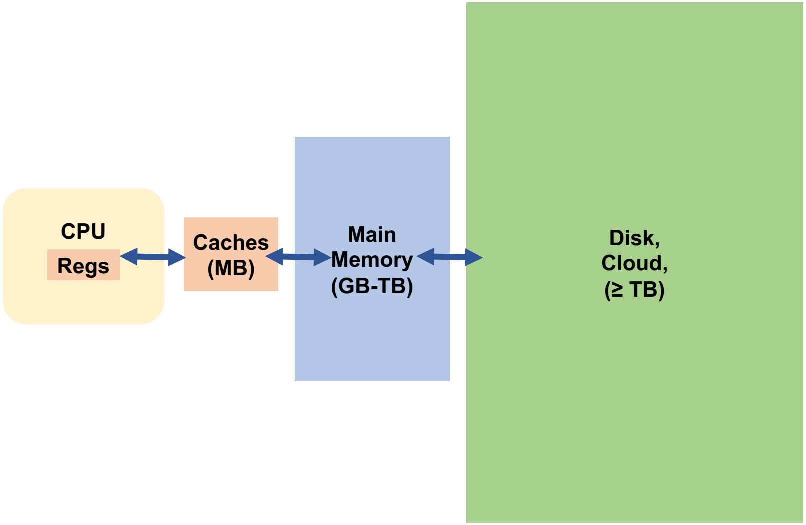
# Our First Compiler

```scala
type Code = String

def trans(e: Exp): Code = e match {
  case Lit(x)      => s"$x"
  case Plus(x,y)   => s"(${trans(x)} + ${trans(y)})"
  case Minus(x,y)  => s"(${trans(x)} - ${trans(y)})"
  // (more cases ...)
}

eval(Plus(Lit(3),Lit(4))) // => "(3+4)"
```

# Our First Compiler

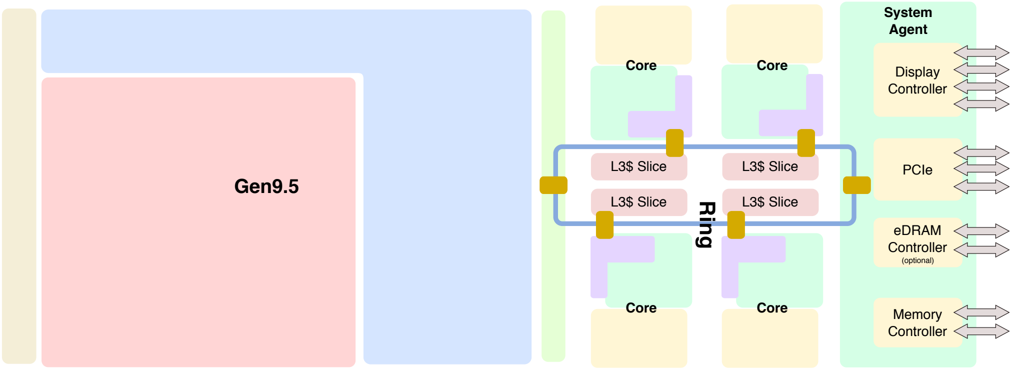
```
type Code = String

def trans(e: Exp): Code = e match {
  case Lit(x)     => s"$x"
  case Plus(x,y)  => s"(${trans(x)} + ${trans(y)})"
  case Minus(x,y) => s"(${trans(x)} - ${trans(y)})"
  // (more cases ...)
}

eval(Plus(Lit(3),Lit(4))) // => "(3+4)"

eval(Plus(Lit(1),Plus(Lit(2),Lit(3)))) // => "(1+(2+3))"
```

# Architecture Refresher

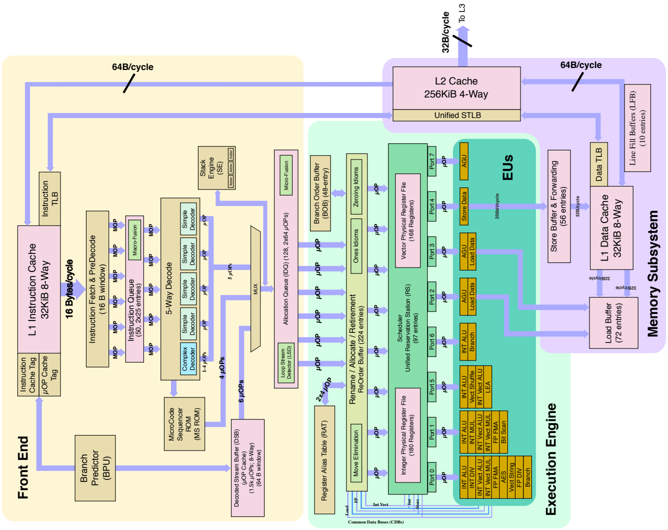


# Architecture Refresher (Intel Skylake)

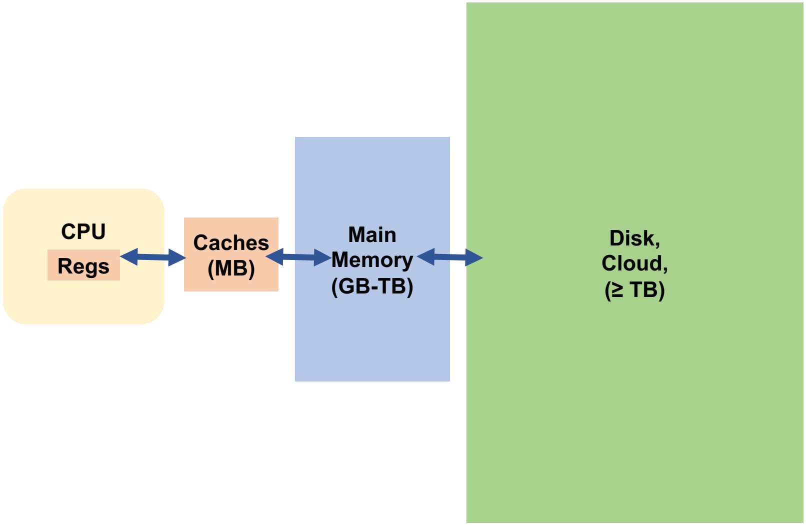
https://en.wikichip.org/wiki/intel/microarchitectures/skylake



# Architecture Refresher (Intel Skylake)



# Memory Hierarchy



# An Interpreter with Explicit Memory

```
var memory = new Array[Int](MEM_SIZE)
var used = 0

def eval(e: Exp): Unit = e match {
  case Lit(x)      => memory(used) = x; used += 1
  case Plus(x,y)  => eval(x)
                     ???


  ...
}
```

# An Interpreter with Explicit Memory

```
var memory = new Array[Int](MEM_SIZE)
var used = 0

def eval(e: Exp): Unit = e match {
  case Lit(x)      => memory(used) = x; used += 1
  case Plus(x,y)  => eval(x)
                     val u = used
                     eval(y)
                     memory(used) = memory(u-1) + memory(used-1)
                     used += 1
  ...
}
```

# A Stack-Based Interpreter

```
var memory = new Array[Int](MEM_SIZE)

def eval(e: Exp, sp: Int): Unit = e match {
  case Lit(x)     => memory(sp) = x
  case Plus(x,y)  => eval(x,sp)
                     ???
  ...
}
```

# A Stack-Based Interpreter

```
var memory = new Array[Int](MEM_SIZE)

def eval(e: Exp, sp: Int): Unit = e match {
  case Lit(x)      => memory(sp) = x
  case Plus(x,y)   => eval(x,sp); eval(y,sp+1)
                      memory(sp) += memory(sp+1)
  ...
}
```

# A Stack-Based Compiler

```
def trans(e: Exp, sp: Int): Unit = e match {
  case Lit(x)      => println(s"memory($sp) = $x")
  case Plus(x,y)  => trans(x,sp); trans(y,sp+1)
                     println(s"memory($sp) += memory(${sp+1})")
  ...
}
```

# A Stack-Based Compiler

```
def trans(e: Exp, sp: Int): Unit = e match {
  case Lit(x)     => println(s"memory($sp) = $x")
  case Plus(x,y)  => trans(x,sp); trans(y,sp+1)
                     println(s"memory($sp) += memory(${sp+1})")
  ...
}

trans(Plus(Lit(1),Plus(Lit(2),Lit(3))),0) // 1+(2+3)
```

# A Stack-Based Compiler

```
def trans(e: Exp, sp: Int): Unit = e match {
  case Lit(x)      => println(s"memory($sp) = $x")
  case Plus(x,y)   => trans(x,sp); trans(y,sp+1)
                      println(s"memory($sp) += memory(${sp+1})")
  ...
}

trans(Plus(Lit(1),Plus(Lit(2),Lit(3))),0) // 1+(2+3)

memory(0) = 1
memory(1) = 2
memory(2) = 3
memory(1) += memory(2)
memory(0) += memory(1)
```

# A Stack-Based Compiler Targeting x86-64 Registers

```
val regs = Seq("%rbx", "%rcx", "%rdi", "%rsi", "%r8", "%r9")
def trans(e: Exp, sp: Int): Unit = e match {
  case Lit(x)      => println(s"movq $$$x, ${regs(sp)}")
  case Plus(x,y)   => trans(x,sp); trans(y,sp+1)
                      println(s"addq ${regs(sp+1)}, ${regs(sp)}")
  ...
}

trans(Plus(Lit(1),Plus(Lit(2),Lit(3))),0) // 1+(2+3)

movq $1, %rbx
movq $2, %rcx
movq $3, %rdi
addq %rdi, %rcx
addq %rcx, %rbx
```

# Parsing

We have seen how to translate ASTs to Assembly code

How can we translate source code to ASTs?

# Source Code as Stream of Characters

Reading a single-digit number:

```
val in: Reader[Char] // peek, hasNext(), next()

def isDigit(c: Char) = '0' <= c && c <= '9'

def getNum(): Int = {
  if (in.hasNext(isDigit)) (in.next() - '0')
  else expected("Number")
}

def parseTerm: Exp = Lit(getNum)
```

# Parsing Sequences of Operations

```
val in: Reader[Char] // peek, hasNext(), next()

def parseTerm: Exp = Lit(getNum)

def parseExpression: Exp = {
  var res = parseTerm
  while (in.hasNext(isOperator)) {
    in.next() match {
      case '+' => res = Plus(res, parseTerm)
      case '-' => res = Minus(res, parseTerm)
    }
  }
  res
}
```

## Operator Precedence

We can successfully parse expressions like “1+2+3” into

```
Plus(Plus(Lit(1),Lit(2)),Lit(3))
```

or the equivalent of “(1+2)+3”.

But what about “1+2*3“?

With the current logic, this will parse as “(1+2)*3“, which is probably not what we want.

...

See next lecture!

# Where are we?

In just one lecture, we have built an end-to-end compiler, from simple arithmetic expressions to native x86-64 code.

In Project 1 (due in one week), you will complete the bits that were missing on the slides.

Over the next lectures, we will add language features such as variables, control flow, functions, etc.

We will keep the pace high, and have a fully functional compiler for a quite substantial language in no time.