CS107 Compilers

Guannan Wei
guannan.wei@tufts.edu

Jan 15, 2026

Spring 2026

Tufts University

What is a Compiler?

You have certainly used one ...

- A program that translates one programming language to another
- Usually, from human-friendly language to machine-friendly language
- Hopefully, generating code that uses the target machine efficiently

Why take this class?

This course: the theory and practice of implementing programming languages

Why take this class?

This course: the theory and practice of implementing programming languages

Intellectual:

- If you want to understand software, you need to understand compilers
  - Understand the implementation of programming languages
  - Make you a better programmer
- Touches on all aspects of CS (algorithms, data structures, hardware, systems, etc.)

Why take this class?

This course: the theory and practice of implementing programming languages

Pragmatic:

- Hiring managers and PhD admissions know this
- All big companies have compiler teams
  - Google: Chrome V8, MLIR, Go, Kotlin, Dart
  - Amazon: Rust, Lean
  - Apple: Swift, LLVM, JavaScriptCore
  - Meta: Hack, HHVM, Infer
  - Microsoft: TypeScript, C#, F#, Visual Studio
  - ...
- Infrastructure of modern/future AI/quantum computing:
  - Tensorflow/XLA, JAX, PyTorch, CUDA, Triton, AWS Neuron, ...
  - Quantinuum, IonQ, ...

Logistics

Topics:

- Parsing, type checking, interpretation and compilation
- Intermediate representations, CPS transformation, closure conversion, SSA
- Analysis and optimizations, function inlining, register allocation
- Runtime representation, garbage collection
- ...

Logistics

Topics:

- Parsing, type checking, interpretation and compilation
- Intermediate representations, CPS transformation, closure conversion, SSA
- Analysis and optimizations, function inlining, register allocation
- Runtime representation, garbage collection
- ...

7 Hands-on Projects:

- Start with a tiny language and compiling to machine code
- Gradually add interesting features (variables, control flow, arrays, functions, etc.)
- You will build parsers, intermediate representations, optimizations, code generators, and runtime systems

Logistics

- Lecture: Tuesday and Thursday 4:30-5:45 PM, JCC 140
- 7 programming projects
- Midterm and final exams
- Piazza: https://piazza.com/tufts/spring2026/cs107
  - We will use Piazza for announcements, questions, and discussions
- Canvas:
  - Submitting projects and grading
- No required textbook

Grading Policy

- Projects: 30%
  - Extra credit up to 5%
- Midterm exam: 30%
- Final exam: 40%
- Piazza participation: extra credit up to 10%
  - Recognition for active participation and instructor-endorsed answers
- You need to achieve a minimum of 25% in each of the three components (projects, midterm, final) for a passing grade.

Grading Policy

- Projects: 30%
  - Extra credit up to 5%
- Midterm exam: 30%
- Final exam: 40%
- Piazza participation: extra credit up to 10%
  - Recognition for active participation and instructor-endorsed answers
- You need to achieve a minimum of 25% in each of the three components (projects, midterm, final) for a passing grade.

This is not a easy course! Be prepared to put in significant effort.

AI Policy and Academic Integrity

- You may use AI tools (e.g., ChatGPT, GitHub Copilot) to help your learning
- You should complete assignments on your own
  - No copy of code or collaboration with others
  - If you use AI tools, you must disclose it in what ways you use it in your submission
  - Do not submit anything you don't understand or can't explain
- Discussion about general concepts is allowed
  - Help your peers on Piazza (will be recognized)
- You are responsible for following the university and SOE's academic integrity policy, and violations will be reported

A Few Languages

- We will use **Scala 3** to write compilers for a small subset of Scala
  - **Meta language** (language we use to write the compiler): Scala 3
  - **Object/source language** (language we compile): a small subset of Scala
  - **Target language** (language we compile to): x86-64 assembly

A Few Languages

- We will use **Scala 3** to write compilers for a small subset of Scala
  - **Meta language** (language we use to write the compiler): Scala 3
  - **Object/source language** (language we compile): a small subset of Scala
  - **Target language** (language we compile to): x86-64 assembly
- Why Scala?
  - Expressive high-level language with functional and object-oriented features
  - Rich type system supporting algebraic data types, pattern matching, generics, etc
  - "**the only academic-designed language of the 21st century to achieve widespread mainstream adoption**" – citation from the ACM Programming Languages Achievement Award 2025

Learning Scala

- If you have taken CS105, it should be easy to pick up Scala 3
- Official Scala 3 Book (Online): https://docs.scala-lang.org/scala3/book/introduction.html
  - Highly recommended go through at least the first few chapters

Representing Programs

Compilers operate on *programs as data*:

- Source code: unstructured sequence of characters

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

Representing Programs

Compilers operate on *programs as data*:

- Source code: unstructured sequence of characters

  `"1 + 2 * 3"`

- Intermediate representation: trees or graphs (data structures in compiler)

  `(+ 1 (* 2 3))`

```
    +
   / \
  1   *
     / \
    2   3
```

Representing Programs

Compilers operate on *programs as data*:

- Source code: unstructured sequence of characters

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

- Intermediate representation: trees or graphs (data structures in compiler)

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

- Output: sequence of machine instructions

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

```
  +
 / \
1   *
   / \
  2   3
```

Grammar

Concrete syntax of source code expressed as a context-free grammar (BNF notation):

```
n ∈ ℤ                                  (integers)
<exp> ::= n                            (literal)
        | <exp> + <exp>                (addition)
        | <exp> * <exp>                (multiplication)
        | <exp> - <exp>                (subtraction)
        | <exp> / <exp>                (division)
```

- Grammar describes the valid *form* of expressions in our language

Program as Data

Abstract syntax: representing the program as data structure (e.g., tree):

```
enum Exp:
  case Lit(n: Int)
  case Add(e1: Exp, e2: Exp)
  case Sub(e1: Exp, e2: Exp)
  case Mul(e1: Exp, e2: Exp)
  case Div(e1: Exp, e2: Exp)
```

Note: `enum` is Scala's way of defining **algebraic data types**. Sometimes we will also use `trait` / `abstract class` + `case class` for the same purpose.

Program as Data

Abstract syntax: representing the program as data structure (e.g., tree):

```
enum Exp:
  case Lit(n: Int)
  case Add(e1: Exp, e2: Exp)
  case Sub(e1: Exp, e2: Exp)
  case Mul(e1: Exp, e2: Exp)
  case Div(e1: Exp, e2: Exp)
```

Note: `enum` is Scala's way of defining **algebraic data types**. Sometimes we will also use `trait` / `abstract class` + `case class` for the same purpose.

Example:

```
val expr = Add(Lit(1), Mul(Lit(2), Lit(3))) // 1 + (2 * 3)
```

Writing an Interpreter

An **interpreter** evaluates the expression directly:

```
type Val = Int

def eval(e: Exp): Val =
  e match
    case Lit(n)       => n
    case Add(e1, e2) => eval(e1) + eval(e2)
    case Sub(e1, e2) => eval(e1) - eval(e2)
    // more cases ...
```

Writing an Interpreter

An **interpreter** evaluates the expression directly:

```
type Val = Int

def eval(e: Exp): Val =
  e match
    case Lit(n)      => n
    case Add(e1, e2) => eval(e1) + eval(e2)
    case Sub(e1, e2) => eval(e1) - eval(e2)
    // more cases ...
```

Example

```
val expr = Add(Lit(1), Mul(Lit(2), Lit(3))) // 1 + (2 * 3)
eval(expr) // 7
```

Our first compiler

From interpreters to compilers:

```scala
type Code = String

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

Our first compiler

From interpreters to compilers:

```
type Code = String

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

Note: `s" ... "` is Scala's string interpolation syntax, `${...}` inserts the result of the expression into the string

```
    case Add(x, y) =>
      val c1 = trans(x)
      val c2 = trans(y)
      s"($c1 + $c2)"
```

Our first compiler

From interpreters to compilers:

```
type Code = String

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

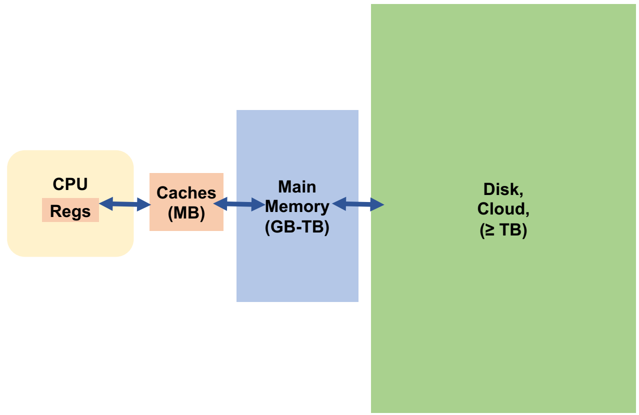
Example

```
val expr = Add(Lit(1), Mul(Lit(2), Lit(3)))
trans(expr) // "(1 + (2 * 3))"
```

Essentially printing the AST back to a string!

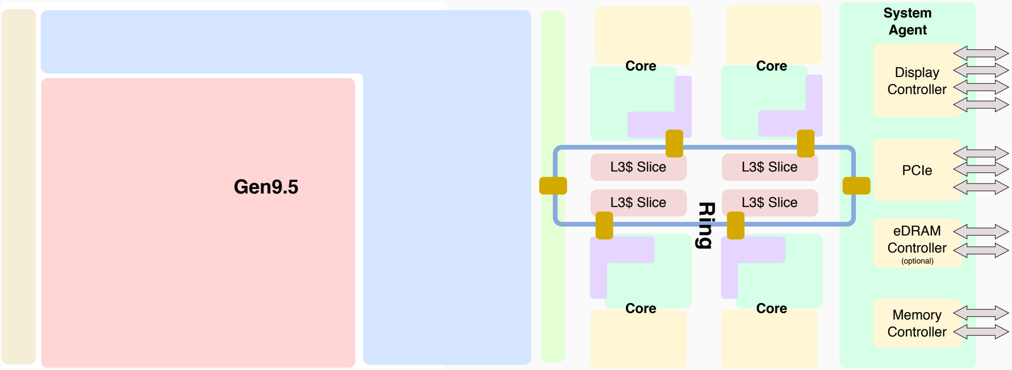
Architecture Refresher

- We need to use the native hardware efficiently
- CPU specifies a set of instructions it can execute
- Memory hierarchy: registers, L1/L2/L3 caches, main memory, disk, ...



Architecture Refresher (Intel Skylake)

- A 4-core Intel Skylake CPU



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

Assembly Refresher

- We use AT&T syntax for x86-64 assembly (default for GNU assembler)
- General-purpose registers: `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rsi`, `%rdi`, `%r8`, `%r9`, ...
- Operand order `op src, dst`

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

Interpreter with Explicit Memory

In our meta-language (Scala), allocate a memory array to store intermediate results:

```
val memory = new Array[Int](MEM_SIZE)
var used = 0 // the current index that can be used
def eval(e: Exp): Unit =
  e match
    case Lit(x)    => memory(used) = x; used += 1
    case Add(x, y) =>
      eval(x)
      ???
    ...
```

Interpreter with Explicit Memory

In our meta-language (Scala), allocate a memory array to store intermediate results:

```scala
val memory = new Array[Int](MEM_SIZE)
var used = 0 // the current index that can be used
def eval(e: Exp): Unit =
  e match
    case Lit(x)    => memory(used) = x; used += 1
    case Add(x, y) =>
      eval(x)
      val u = used
      eval(y)
      memory(used) = memory(u-1) + memory(used-1)
      used += 1
    ...
```

Interpreter with Explicit Memory

In our meta-language (Scala), allocate a memory array to store intermediate results:

```
val memory = new Array[Int](MEM_SIZE)
var used = 0 // the current index that can be used
def eval(e: Exp): Unit =
  e match
    case Lit(x)    => memory(used) = x; used += 1
    case Add(x, y) =>
      eval(x)
      val u = used
      eval(y)
      memory(used) = memory(u-1) + memory(used-1)
      used += 1
    ...
```

Contract: `eval` puts the result of evaluating `e` into `memory(used)`

A Stack-Based Interpreter

- Why not just tell the `eval` function where to store the result?

```
val memory = new Array[Int](MEM_SIZE)
def eval(e: Exp, sp: Int): Unit =
  e match
    case Lit(x)    => memory(sp) = x
    case Add(x, y) =>
      eval(x, sp)
      ???
    ...
```

A Stack-Based Interpreter

- sp (stack pointer) indicates the position in memory to store the result

```
val memory = new Array[Int](MEM_SIZE)
def eval(e: Exp, sp: Int): Unit =
  e match
    case Lit(x)     => memory(sp) = x
    case Add(x, y) =>
      eval(x, sp)
      eval(y, sp+1)
      memory(sp) += memory(sp+1)
    ...
```

A Stack-Based Compiler

- Our second compiler: just print out the operations performed by the interpreter!

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

Example: `trans(Add(Lit(1), Add(Lit(2), Lit(3))), 0) // 1+(2+3)`

A Stack-Based Compiler

- Our second compiler: just print out the operations performed by the interpreter!

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

Example: `trans(Add(Lit(1), Add(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

- Our third compiler: use a sequence of registers as a stack

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

Example: `trans(Add(Lit(1), Add(Lit(2), Lit(3))), 0) // 1+(2+3)`

A Stack-Based Compiler Targeting x86-64 Registers

- Our third compiler: use a sequence of registers as a stack

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

Example: `trans(Add(Lit(1), Add(Lit(2), Lit(3))), 0) // 1+(2+3)`

```
%rbx = $1
%rcx = $2
%rdi = $3
%rcx += %rdi
%rbx += %rcx
```

A Stack-Based Compiler Targeting x86-64 Registers

- Further tweak syntax to generate valid x86-64 assembly code:

```
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 Add(x, y) =>
    trans(x, sp)
    trans(y, sp+1)
    println(s"addq ${regs(sp+1)}, ${regs(sp)}")
  ...
```

Example: `trans(Add(Lit(1), Add(Lit(2), Lit(3))), 0) // 1+(2+3)`

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

Parsing

Parsing

We have seen how to translate an abstract syntax tree (AST) to assembly code.

How can we translate source code to ASTs?

```
1+2*3 -> Add(Lit(1), Mul(Lit(2), Lit(3)))
```

Source Code as Stream of Characters

Reading a single-digit number:

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

def isDigit(c: Char): Boolean = '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] // implements peek(), hasNext(), next()

def parseTerm: Exp = Lit(getNum)

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

Operator Precedence

We can successfully parse expressions like `1+2+3` into

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

or the equivalent of `(1+2)+3`.

Operator Precedence

We can successfully parse expressions like `1+2+3` into

```
Add(Add(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?

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, Jan 22)**, 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.