CS107: Review

Guannan Wei
guannan.wei@tufts.edu

March 4, 2026

Spring 2026

Tufts University

Logistics

- Project 3 grade/feedback has been released.
- Project 5 will be released after midterm.
- Midterm scope: everything covered in the lecture and project up to closure conversion (including week 7-1).
- Time: 75 minutes.

Review

At this point, you have built a reasonably useful compiler for MiniScala.

- the language supports primitive data types, (higher-order) functions, mutable variables, arrays, while loops, conditionals, ...
- the language expressive enough (Turing-complete) to write interesting programs,
- the compiler is realistic enough (parsing, type checking, conversion to CPS, code generation to x86-64 assembly).

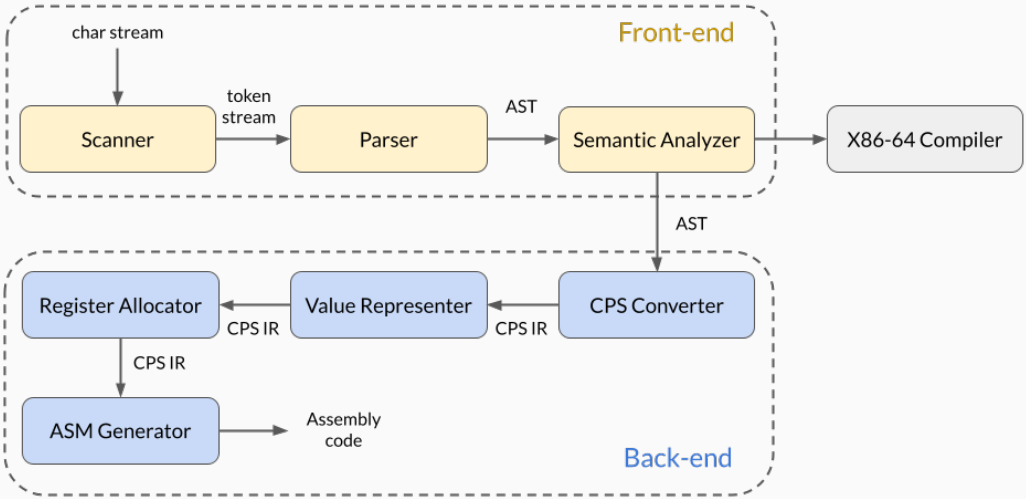
Review

At this point, you have built a reasonably useful compiler for MiniScala.

- the language supports primitive data types, (higher-order) functions, mutable variables, arrays, while loops, conditionals, ...
- the language expressive enough (Turing-complete) to write interesting programs,
- the compiler is realistic enough (parsing, type checking, conversion to CPS, code generation to x86-64 assembly).

You should be proud of what you have accomplished so far!

Review


char stream
Front-end
token stream
AST
Scanner
Parser
Semantic Analyzer
X86-64 Compiler
AST
Register Allocator
Value Representer
CPS Converter
CPS IR
CPS IR
CPS IR
ASM Generator
Assembly code
Back-end

Review

Compiler is just a piece of program that translates source code from one language to another.

Programs are data: Compiler takes other programs as input, operates on different representations (AST, IR, etc.), and produces assembly code as output.

Review: Syntax

Concepts:

- Concrete syntax
  - regular expressions, context-free grammars, BNF
- Representation: abstract syntax tree (AST)
- Tokenization/scanner, parsing

Review: Syntax

Concepts:

- Concrete syntax
  - regular expressions, context-free grammars, BNF
- Representation: abstract syntax tree (AST)
- Tokenization/scanner, parsing

Given the grammar,

- generate syntactically valid program
- write down the corresponding tokens/AST given the program
- identify ill-formed programs
- implement the parser implementation

Type system

Specification: typing judgment and inference rules.

Typing judgments $\Gamma \vdash e : T$ assert that in the environment Γ, the expression e is of type T.

Inference rules specify how we can form typing judgments:

$$\frac{\mathit{condition1} \qquad \mathit{condition2} \qquad \ldots}{\mathit{conclusion}} \text{ NAME OF THE RULE}$$

If all conditions can be proven **true** then the conclusion is **true**.

Type system

Specification: typing judgment and inference rules.

Given a type system,

- prove a program is well-typed by building a proof tree using the inference rules
- identify ill-typed programs

Implementation: type checking and inference

```
def typeCheck(exp: Exp, pt: Type)(env: Env): Exp = {
  val nexp = typeInfer(exp, pt)(env)
  val rtp = typeConforms(nexp.tp, pt)(env)
  nexp.withType(rtp)
}

def typeInfer(exp: Exp, pt: Type)(env: Env): Exp = ...
```

Given a (simple) type system,

- implement the type checker and inference algorithm

Type System

What does type soundness mean?

Type System

What does type soundness mean?

"well-typed programs do not go wrong" – Robin Milner.

Type System

What does type soundness mean?

"well-typed programs do not go wrong" – Robin Milner.

How do you know if a type system is sound or unsound?

Intermediate Representation

Concepts:

- What is an IR and why do we need an IR?
- Continuation-passing style (CPS)
- CFG/RFL
- Static Single Assignment (SSA)

Intermediate Representation

Concepts:

- What is an IR and why do we need an IR?
- Continuation-passing style (CPS)
- CFG/RFL
- Static Single Assignment (SSA)

You need to be able to

- manually converting a program to CPS form
- manually converting a program to CFG/RFL/SSA form
- understand the connection between CPS and SSA
- implement the conversion to CPS form

Interpretation and Code Generation

We have seen various form of interpreters (e.g. value interpreter, environment-passing, stack-based, etc.).

Given the underlying assembly/architecture semantics,

- generate correct assembly code for various higher-level language constructs (e.g., function calls, loops, conditionals, ...)
- understand calling conventions
- value representation (tagging, closure conversion, ...)

Questions?