LL Parsing & Semantics

CSE 401/M501

Adapted from Spring 2021

Announcements

- Parser + AST due **TONIGHT**!
- Homework 3 (LL grammars) due Wednesday
- Next section: midterm review
    - Bring your conceptual questions and past midterm questions!

25 14:30-15:20 Lecture CSE2 G10 *Symbol tables and representation of types* 18:00-19:00 OH (Robert) CSE2 153 and Zoom	26 16:00-17:00 OH (Larry) Zoom	27 14:30-15:20 Lecture CSE2 G10 *Type checking / semantics wrapup; start x86-64* x86-64 slides 17:00-18:00 OH (Apollo) CSE2 153 and Zoom	28 Section *Interpreters; more about LL parsing* WE ARE HERE 15:30-16:30 OH (Jack) CSE2 151 and Zoom 20:00-21:00 OH (Morel) Zoom 23:00 Project: parser+AST due	29 14:30-15:20 Lecture CSE2 G10 *x86-64 (everything you forgot from 351)*

May

Monday	Tuesday	Wednesday	Thursday	Friday
02 14:30-15:20 Lecture CSE2 G10 *Code shape I - basics* 18:00-19:00 OH (Robert) CSE2 153 and Zoom	03 16:00-17:00 OH (Larry) Zoom	04 14:30-15:20 Lecture CSE2 G10 *Code shape II - objects and dynamic dispatch* 17:00-18:00 OH (Apollo) CSE2 153 and Zoom 23:00 hw3 due (LL grammars & parsing)	05 Section *Midterm review* 15:30-16:30 OH (Jack) CSE2 151 and Zoom 20:00-21:00 OH (Morel) Zoom	06 14:30-15:20 Midterm exam

Agenda

- Semantics & Type Checking
  - Review: Semantics vs. Type Checking
  - Type Checking for MiniJava
- LL parsing worksheet

Semantics & Type Checking

Semantics, Dynamic and Static

semantics: precise meaning of program syntax

what interpretation or code generation implements

dynamic semantics: systematic rules to define runtime behavior

static semantics: systematic rules to define *statically correct* behavior

what type checking implements

Static Semantics of MiniJava

Every language has its own idea of “statically correct,”
but in MiniJava, statically correct code must...

1. *never* add, subtract, multiply, or print non-integers
2. *never* call a non-existent method
3. *never* access a non-existent field

n. … and so on (see the assignment page for more)

How do type checks relate to these conditions?

Type Checking for MiniJava

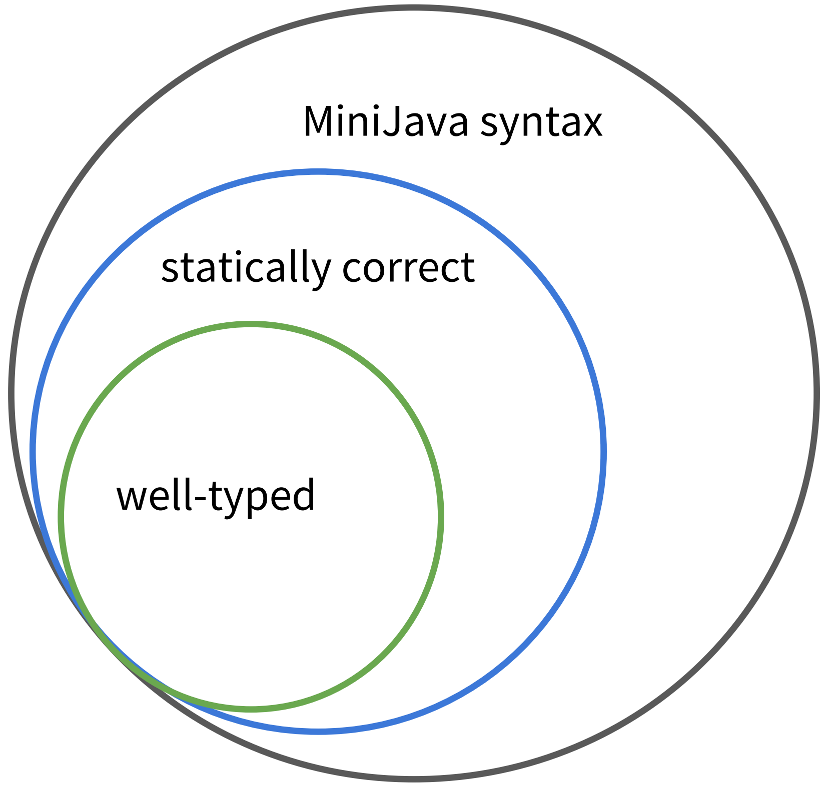
The type checker's goal is to verify that a source program is statically correct.

We can't check that directly, but we can build a checkable type system so that:

well-typed $\Longrightarrow$ ***statically correct***

Note: type checking depends on context – an implementation will depend on keeping track of types across different contexts (a scoped symbol table)

Type Checking for MiniJava



Examples

Suppose the following declarations are in effect:
Global scope: `class Foo { int f; int m(boolean b); }`
Local scope: `Foo this (implicit); int x; boolean y;`

In these scopes, which MiniJava expressions have type `int`? Why (not)?

`56`	`x+(new Foo()).f`	`x+this.m()`
`2+x`	`x+y`	`x+z.m(y)`
`this.f`	`(new Bar()).f`	`x+this.m(true)`

Scopes and Symbol Tables

Accurately tracking scope information, via symbol tables, is critical to type checking.

Some guiding observations from today:

- All classes in MiniJava will need symbol tables
  - When looking for a symbol, start in method table, then enclosing class, then global
- To generate symbol tables, it will make your life easier to go layer-by-layer
  - Global information needed everywhere! Makes sense to do that first
  - Easier to check a method body once global information is already computed

- Implementation tip:
  - Add pointers in your AST nodes to relevant type/symbol table information

The Take-Away

Static semantics is usually about what code must **not** do.

- ∴ ruling out ill-behaved traces is a useful mental model
- ∴ implementing and debugging a type checker is all about **edge cases**
- ∴ need to consider all names in scope, with their type (signatures)

Problem 2: Static Semantics & Type Checking

Problem 1: LL parsing