Semantics & Type Checking

CSE 401 Section 7

Jack Eggleston, Aaron Johnston, & Nate Yazdani

Announcements

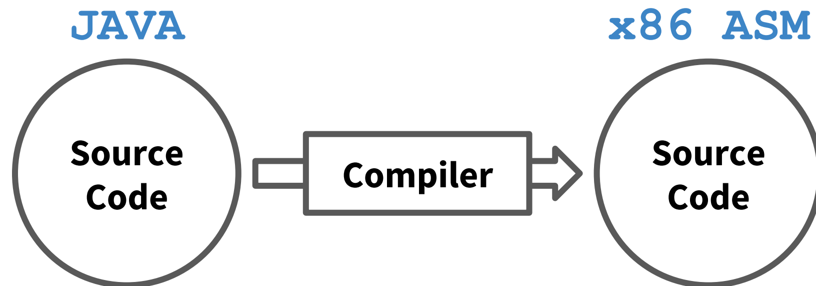
- Midterm grades have been released
  - If you have any questions, feel free to drop by office hours
  - If it really looks like we goofed, submit a regrade request

Announcements

- Midterm grades have been released
    - If you have any questions, feel free to drop by office hours
    - If it really looks like we goofed, submit a regrade request
- Semantics Project Part due November 15th (1 week away!)
    - If you haven’t already, start early! There are plenty of weird edge cases to think about

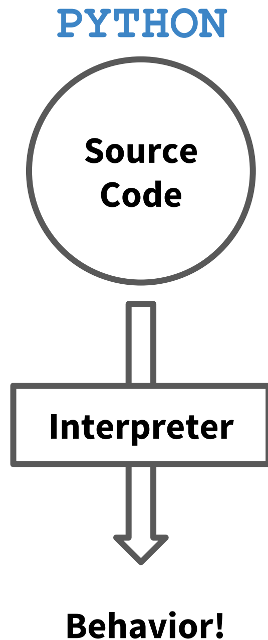
Interpreters vs. Compilers

- Compilers
  - Translate between different languages
  - e.g. MiniJava ⇒ x86 ASM
  - e.g. Java ⇒ Java Byte Code



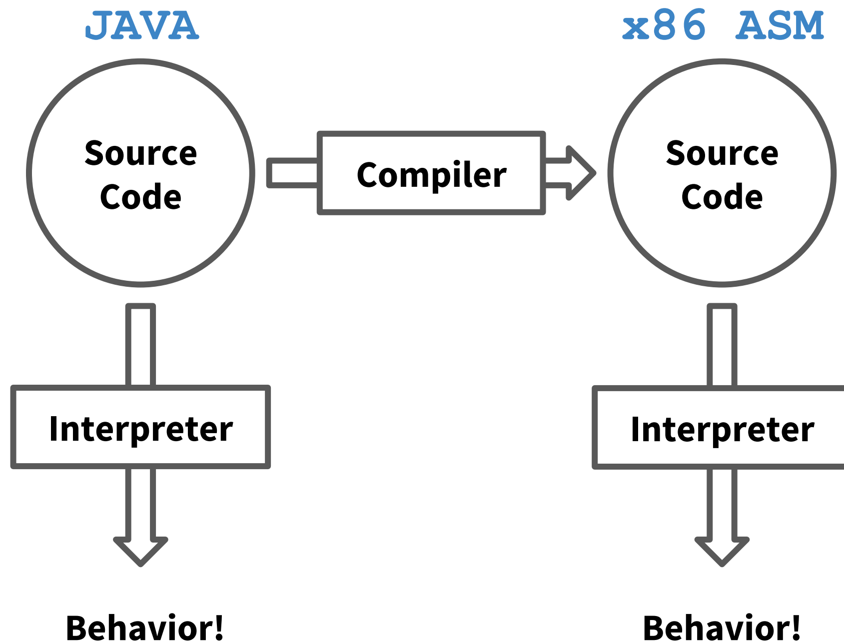
Interpreters vs. Compilers

- Compilers
  - Translate between different languages
  - e.g. MiniJava ⇒ x86 ASM
  - e.g. Java ⇒ Java Byte Code
- Interpreters
  - Take action upon a piece of code as it is read



Interpreters vs. Compilers

- Compilers
  - Translate between different languages
  - e.g. MiniJava ⇒ x86 ASM
  - e.g. Java ⇒ Java Byte Code
- Interpreters
  - Take action upon a piece of code as it is read

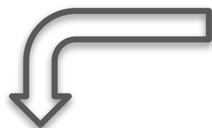


Interpreter Demo

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 computational behavior

static semantics: systematic rules to define *well-behaved* computation

what type checking implements

Generally helpful to think of “well-behaved” as “plays nice with other code.”

Static Semantics of MiniJava

Every language has its own idea of "well-behaved,"
but in MiniJava, well-behaved 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 the well-behavior of a source program:

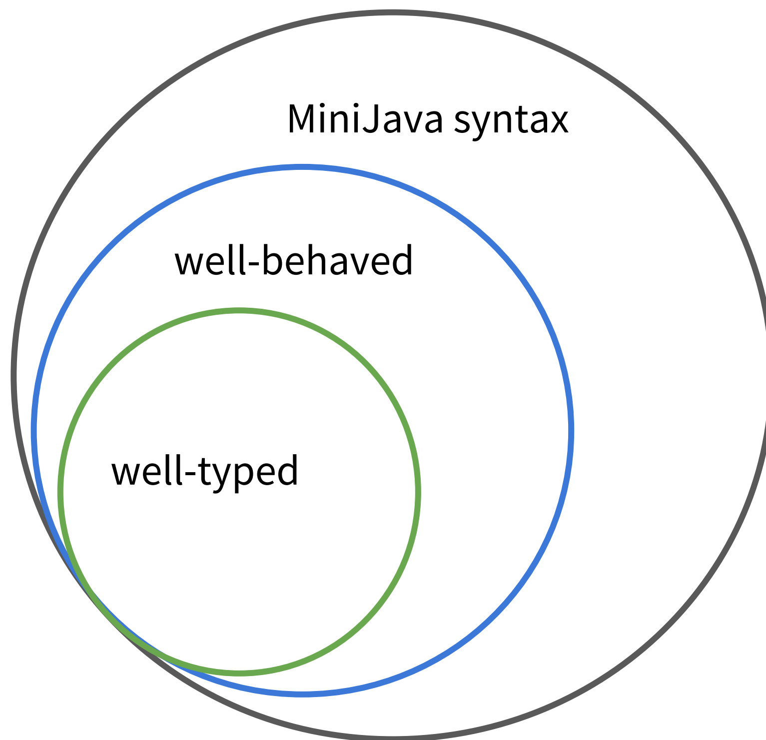
well-typed $\Longrightarrow$ ***well-behaved***

A ***type*** classifies not just values but also expressions w.r.t. a scope.

↳Analogously for type signature and methods/classes.

The hallmark of type checking is *compositionality*, facilitated by scoped symbol tables.

Type Checking for MiniJava



Examples

Global scope: `class Foo { int f; public int m(boolean b); }`

Local scope: `Foo this; 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:

- Symbol information in MiniJava has *layers* of dependence. (What are they?)
- It may make your life easier if you type-check layer by layer.

Implementation tip:

- It might be handy to stash a link to a class's/method's symbol table in its AST node

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)

Beware infinite loops!