

LL Parsing & Semantics

CSE 401/M501

Adapted from Spring 2021

Announcements

- Parser + AST due **TONIGHT!**
- Homework 3 (LL grammars) due Monday 10/27 @ 11:59pm
 - Only one late day, smaller assignment
 - Solutions released Wednesday to review for midterm
- Next section: midterm review
 - Bring your conceptual questions and past midterm questions!

Agenda

- **LL parsing worksheet**
- **Semantics & Type Checking**
 - **Review: Semantics and Type Checking**
 - **Type Checking for MiniJava**

Problem 1: LL parsing

Canonical LL(1) Problems and their Solutions

FIRST Conflict:

Both productions of A have α in their FIRST sets

0. $A ::= \alpha\beta \mid \alpha\gamma$

Solution:

Factor out the prefix (α)

0. $A ::= \alpha \text{ Tail}$

1. $\text{Tail} ::= \beta \mid \gamma$

Left Recursion:

Special FIRST conflict: β in FIRST for both productions

0. $A ::= A\alpha \mid \beta$

Solution:

Create recursive tail from suffix of recursive production

1. $\text{Tail} ::= \alpha \text{ Tail}$

Append Tail to non-recursive productions

0. $A ::= \beta \text{ Tail}$

1. $\text{Tail} ::= \alpha \text{ Tail}$

Add empty string (ϵ) as a rhs for the tail production

0. $A ::= \beta \text{ Tail}$

1. $\text{Tail} ::= \alpha \text{ Tail} \mid \epsilon$

FIRST FOLLOW Conflict:

B is nullable, α in FIRST & FOLLOW

0. $A ::= B\alpha$

1. $B ::= \alpha \mid \epsilon$

Solution:

Substitute B into A

0. $A ::= \alpha\alpha \mid \alpha$

Factor out the prefix (α)

0. $A ::= \alpha \text{ Tail}$

1. $\text{Tail} ::= \alpha \mid \epsilon$

Indirect Left Recursion:

Recursively alternates between A & B

0. $A ::= B\beta$

1. $B ::= A \mid \alpha$

Solution:

Substitute B into A

0. $A ::= A\beta \mid \alpha\beta$

Solve like normal Left Recursion

0. $A ::= \alpha\beta \text{ Tail}$

1. $\text{Tail} ::= \beta \text{ Tail} \mid \epsilon$

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

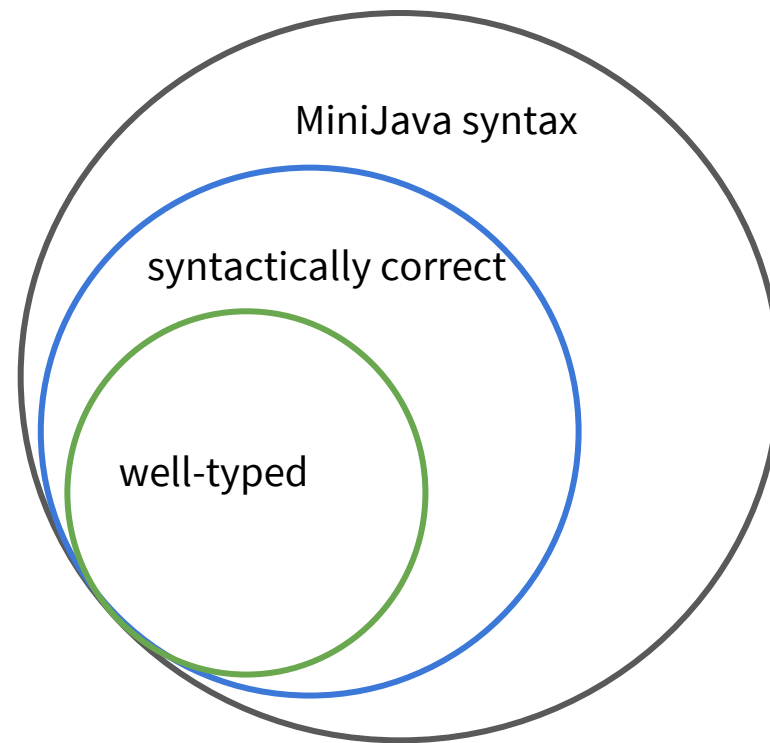
The type checker's goal is to verify that a source program is statically correct.

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

well-typed* \Rightarrow *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 Java expressions have type `int`? Why (not)?

`56`

yes

`x+(new Foo()).f`

yes

`x+this.m()`

no ☹

`2+x`

yes

`x+y`

no ☹

`x+z.m(y)`

no ☹

`this.f`

yes

`(new Bar()).f`

no ☹

`x+this.m(true)`

yes

Scopes and Symbol Tables

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

Some guiding observations from today:

- All classes and methods 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 Takeaway

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