

Software Language Engineering: Semantic analysis/ type checking/ static analysis

Tijs van der Storm



Centrum Wiskunde & Informatica



university of
 groningen

Recap

- Grammar -> Parser -> Parse Tree -> AST
- Name resolution: recover referential structure
- Today
 - Static analysis
 - Type checking

Errors

- People make mistakes...
- Parsing is the first check: syntactic correctness
- But there are errors not captured by grammars
- => Names & types
- But also: non-determinism, deadlock, reachability, dead code, etc.

Static checking

- Static checking phase acts as a **contract**
- Further language processors can **assume** that the program is semantically well-formed:
 - all variables declared
 - all expressions have a correct type
 - etc....

Again two perspectives

- Error checking helps users of the language
- But it also simplifies back-end engineering

Names

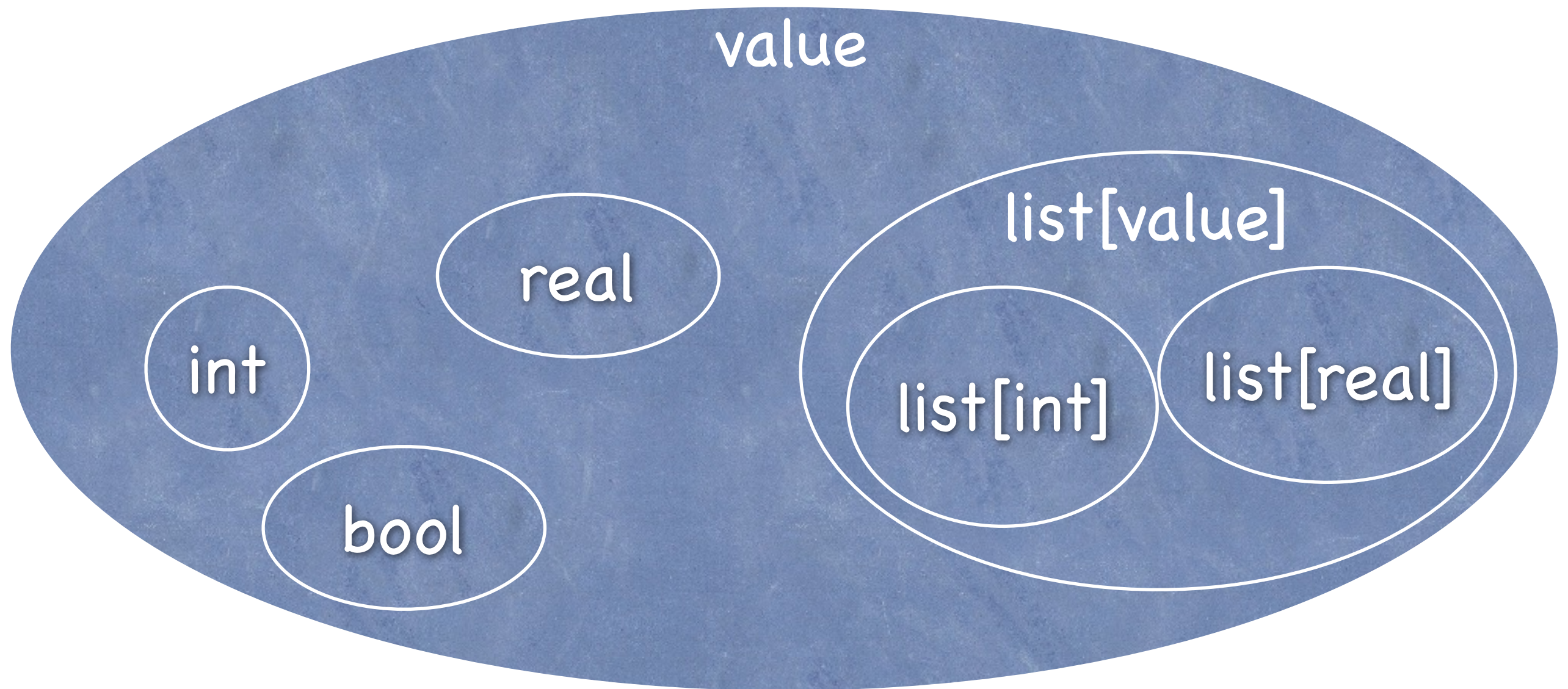
- no such package
- name/file mismatch
- cyclic inheritance
- undefined class
- unimported class
- duplicate method
- shadowing error
- scoping error

```
NameErrors.java X
1 package mypackage;
2
3 public class Something { }
4
5 class Bla extends Bla { }
6
7 class Foo extends Bar {
8     List<Integer> aMethod() { }
9
10    int aMethod() { }
11
12    int anotherMethod() {
13        int x = 3;
14        {
15            int y = 4;
16            int x = 2345;
17        }
18        return y;
19    }
20 }
```

Types

- **Types** partition runtime **values**
- Primitive types: int, bool, str, etc.
- Composite types: algebraic datatypes, records, classes, dictionaries, lists etc. (aka “type constructors”)
- A **type system** assigns types to **expressions**
 - $a + b : \text{int}$
 - $f(x) \{ \text{return } x + 1 \} : \text{int} \rightarrow \text{int}$
 - etc.

Relations between types

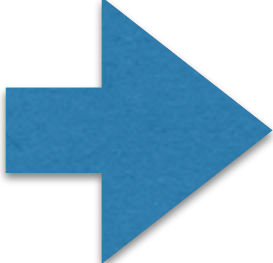


Sub typing

Type judgments

Premise 
$$\frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash_S x : \tau} \quad [\text{Var}]$$

$$\frac{\Gamma \vdash_S e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash_S e_1 : \tau}{\Gamma \vdash_S e_0 e_1 : \tau'} \quad [\text{App}]$$

Conclusion 
$$\frac{\Gamma, x : \tau \vdash_S e : \tau'}{\Gamma \vdash_S \lambda x . e : \tau \rightarrow \tau'} \quad [\text{Abs}]$$

$$\frac{\Gamma \vdash_S e_0 : \tau \quad \Gamma, x : \bar{\Gamma}(\tau) \vdash_S e_1 : \tau'}{\Gamma \vdash_S \text{let } x = e_0 \text{ in } e_1 : \tau'} \quad [\text{Let}]$$



$$\frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash_S x : \tau}$$

[Var]

$$\frac{\Gamma \vdash_S e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash_S e_1 : \tau}{\Gamma \vdash_S e_0 e_1 : \tau'}$$

[App]

$$\frac{\Gamma, x : \tau \vdash_S e : \tau'}{\Gamma \vdash_S \lambda x . e : \tau \rightarrow \tau'}$$

[Abs]

$$\frac{\Gamma \vdash_S e_0 : \tau \quad \Gamma, x : \bar{\Gamma}(\tau) \vdash_S e_1 : \tau'}{\Gamma \vdash_S \text{let } x = e_0 \text{ in } e_1 : \tau'}$$

[Let]

Logical system
not directly
executable

Especially since this
specifies type
inference

Type system vs type checking

- A type system defines a **proof system** to assign types to expressions
 - Used in **soundness** proofs
 - “Well-typed programs don’t go wrong.”
- Type checking presupposes an **algorithm** to (efficiently) assign types to expressions
- **Type inference** analogous, but w/o type annotations

Abstract Interpretation

- Interpretation = evaluation of expressions to **values**
 - $\text{eval}: \text{Expr} \rightarrow \text{Value}$
- **Abstract** interpretation \approx evaluation of expressions to **abstract** “values”
 - $\text{signEval}: \text{Expr} \rightarrow \{+, -, 0, \text{unknown}\}$
 - $\text{typeEval}: \text{Expr} \rightarrow \text{Type}$
 - ...

```
data Expr
  = add(Expr lhs, Expr rhs)
  | let(str x, Expr v, Expr b)
  | var(str x)
  | lit(value n)
  ;
```

The environment

```
alias Env = map[str, value];
```

```
value eval(Expr e, Env env) {
  ...
}
```

The interpreter

```
data Expr
  = add(Expr lhs, Expr rhs)
  | let(str x, Expr v, Expr b)
  | var(str x)
  | lit(value n)
  ;
```

```
alias Env = map[str, value];
```

```
value eval(Expr e, Env env);
```

```
data Expr
  = add(Expr lhs, Expr rhs)
  | let(str x, Expr v, Expr b)
  | var(str x)
  | lit(value n)
  ;
```

```
data Type
  = tint()
  | tstr()
  | tunknown()
  ;
```

```
alias TEnv = map[str, Type];
```

```
Type typeOf(Expr e, TEnv env);
```

```
data Expr
  = add(Expr lhs, Expr rhs)
  | let(str x, Expr v, Expr b)
  | var(str x)
  | lit(value n)
;
```

```
Type typeOf(lit(str _), _) = tstr();
```

```
Type typeOf(lit(int _), _) = tint();
```

```
Type typeOf(add(lhs, rhs), env) = tint();
```

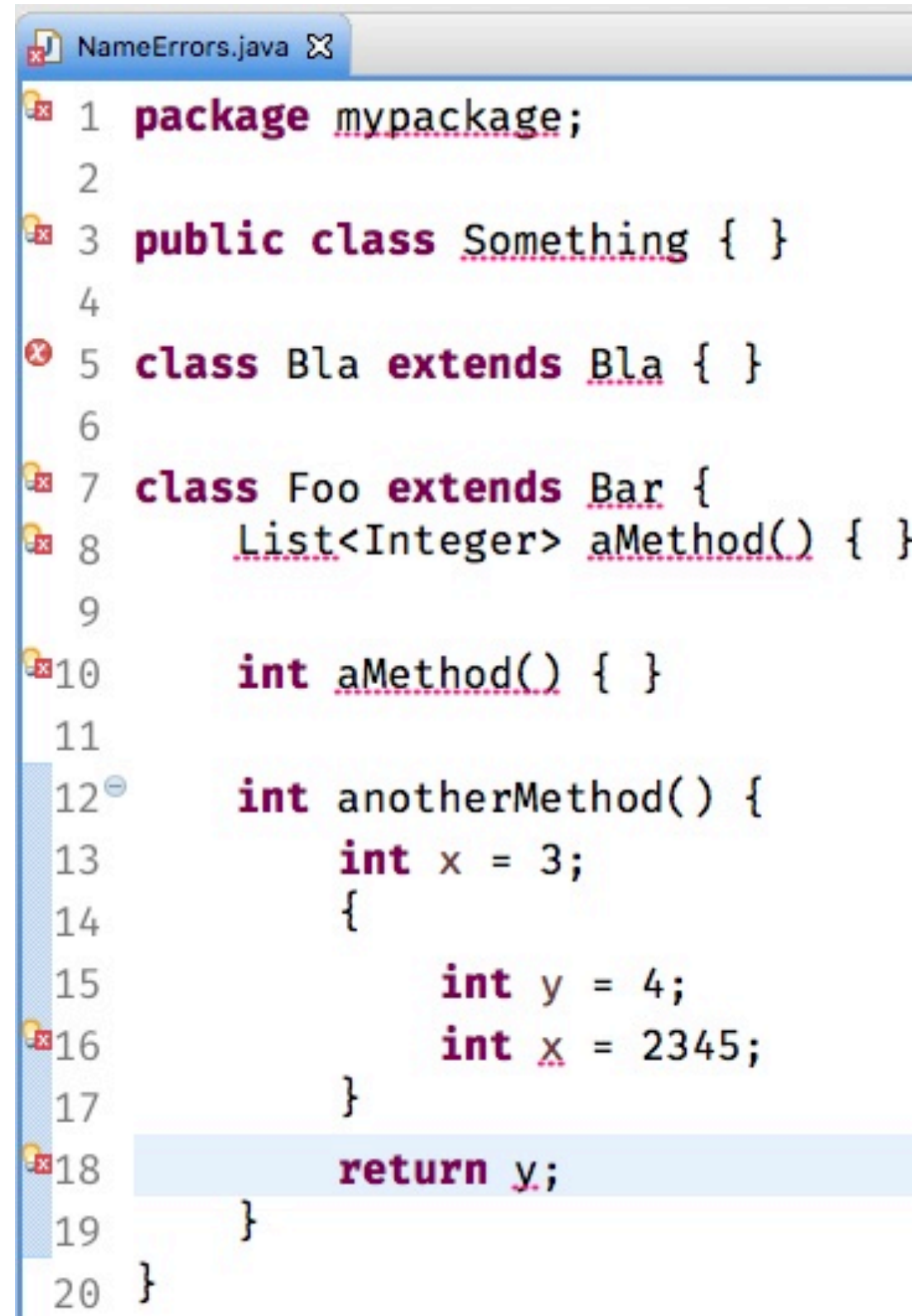
```
Type typeOf(let(x, v, b), env) = typeOf(b, env2)
  when env2 := env + (x: typeOf(v));
```

```
Type typeOf(var(x), env) = env[x]
  when x in env;
```

```
default Type typeOf(_, _) = tunknown();
```

Error messages

- Just assigning types to expressions not enough
- Diagnostics:
 - what is wrong?
 - where?
 - why?
- Further: quick fixes



The screenshot shows a Java IDE window titled "NameErrors.java". The code contains several errors marked with red 'X' icons in the left margin:

```
1 package mypackage;
2
3 public class Something { }
4
5 class Bla extends Bla { }
6
7 class Foo extends Bar {
8     List<Integer> aMethod() { }
9
10    int aMethod() { }
11
12    int anotherMethod() {
13        int x = 3;
14        {
15            int y = 4;
16            int x = 2345;
17        }
18    return y;
19    }
20 }
```

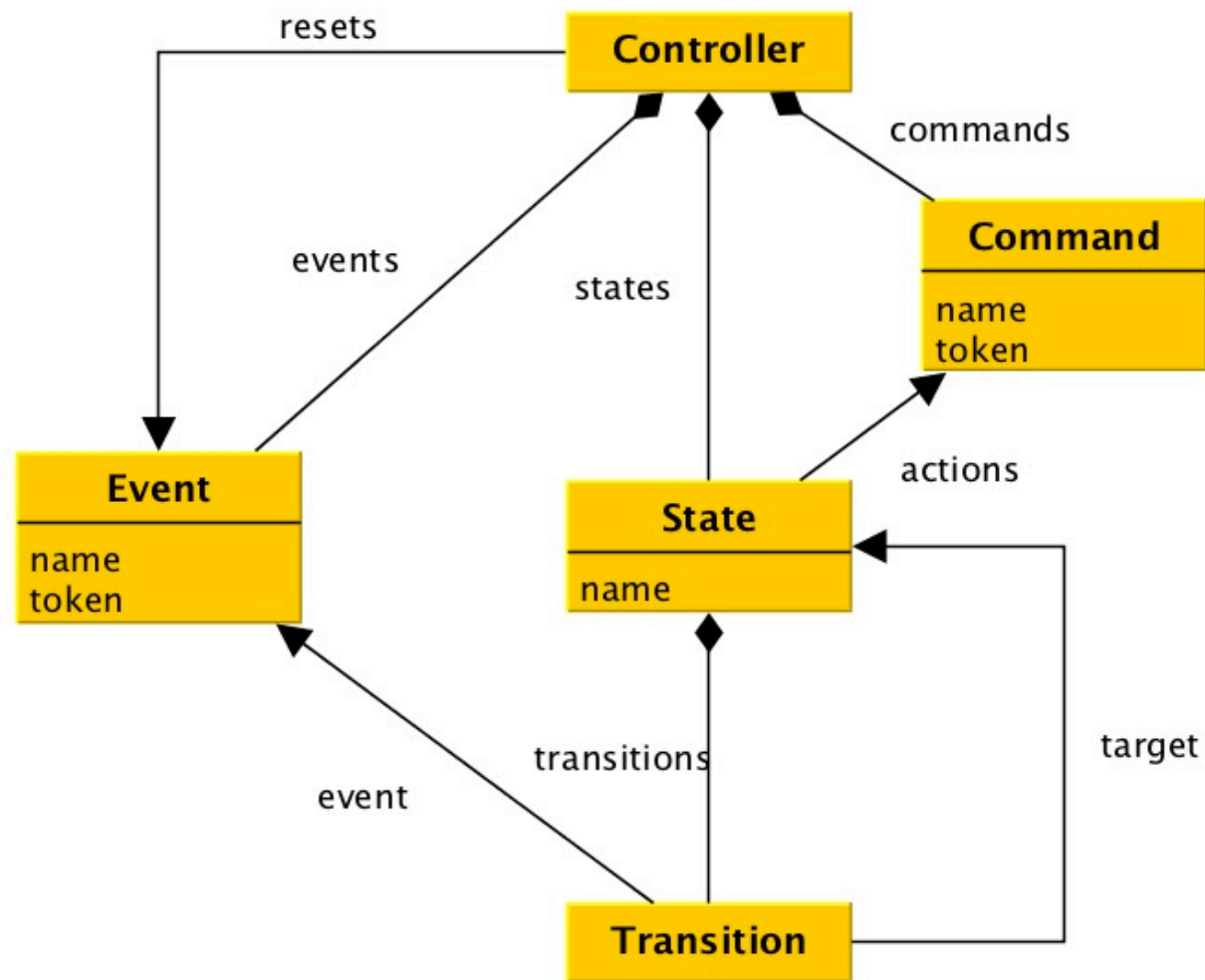
The errors are:

- Line 5: `class Bla extends Bla` - Self-referencing class.
- Line 7: `class Foo extends Bar` - `Bar` is not found.
- Line 8: `List<Integer> aMethod()` - `List` is not found.
- Line 10: `int aMethod()` - Method signature conflict with line 8.
- Line 12: `int anotherMethod()` - Method signature conflict with line 10.
- Line 16: `int x = 2345;` - Variable `x` is already declared in the same scope.
- Line 18: `return y;` - Variable `y` is not found.

In Rascal...

- Error messages defined in module “Message”
 - error, warning, info
- A (type) checker generally produces a `set[Message]`
- Optionally a `map[loc,str]` for hover information
- Both can be interpreted by the IDE

Metamodeling



```
data Controller
    = controller(list[Event] events,
                 list[str] resets,
                 list[Command] commands,
                 list[State] states);
```

```
data State
    = state(str name,
            list[str] actions,
            list[Transition] transitions);
```

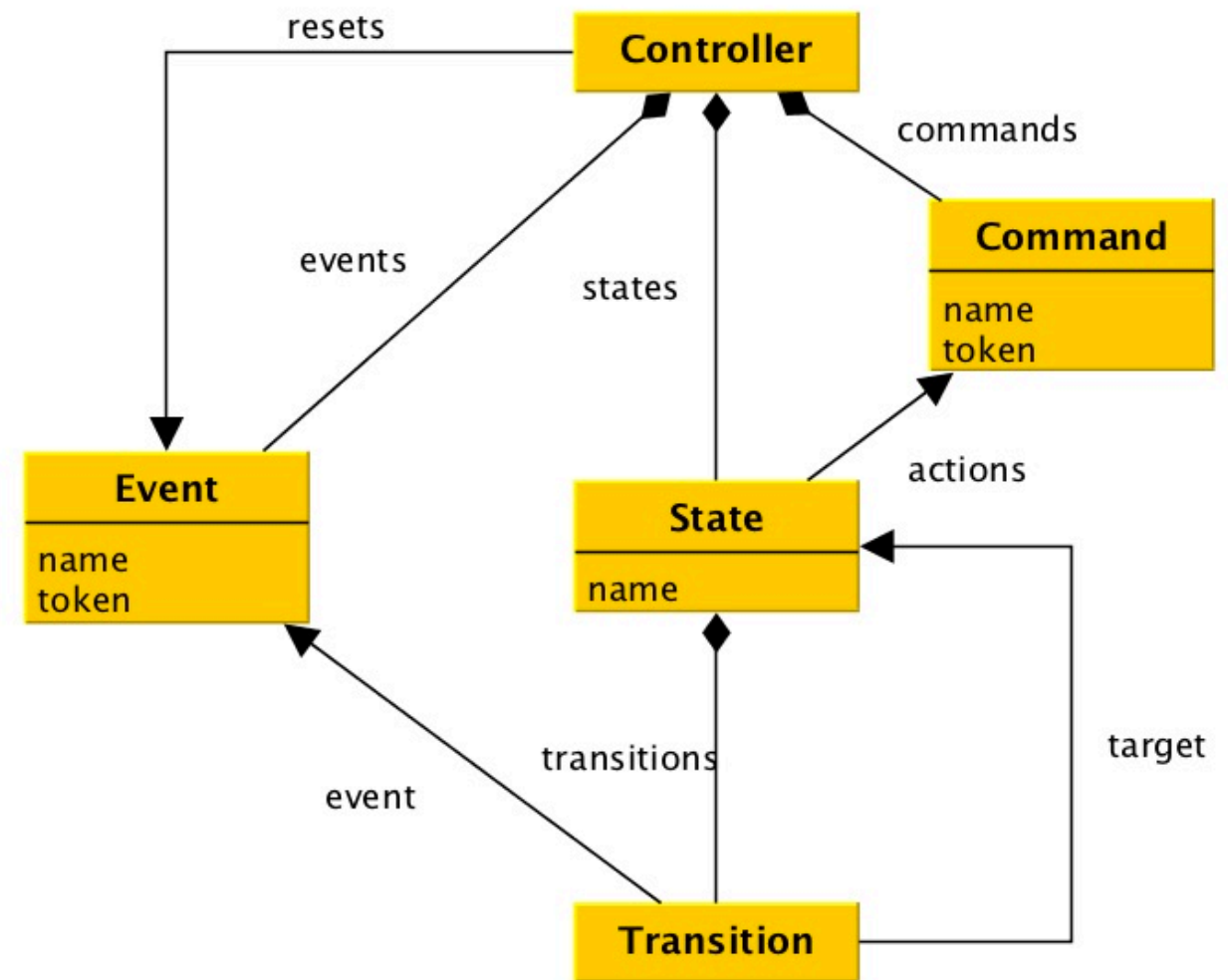
```
data Command
    = command(str name, str token);
```

```
data Event
    = event(str name, str token);
```

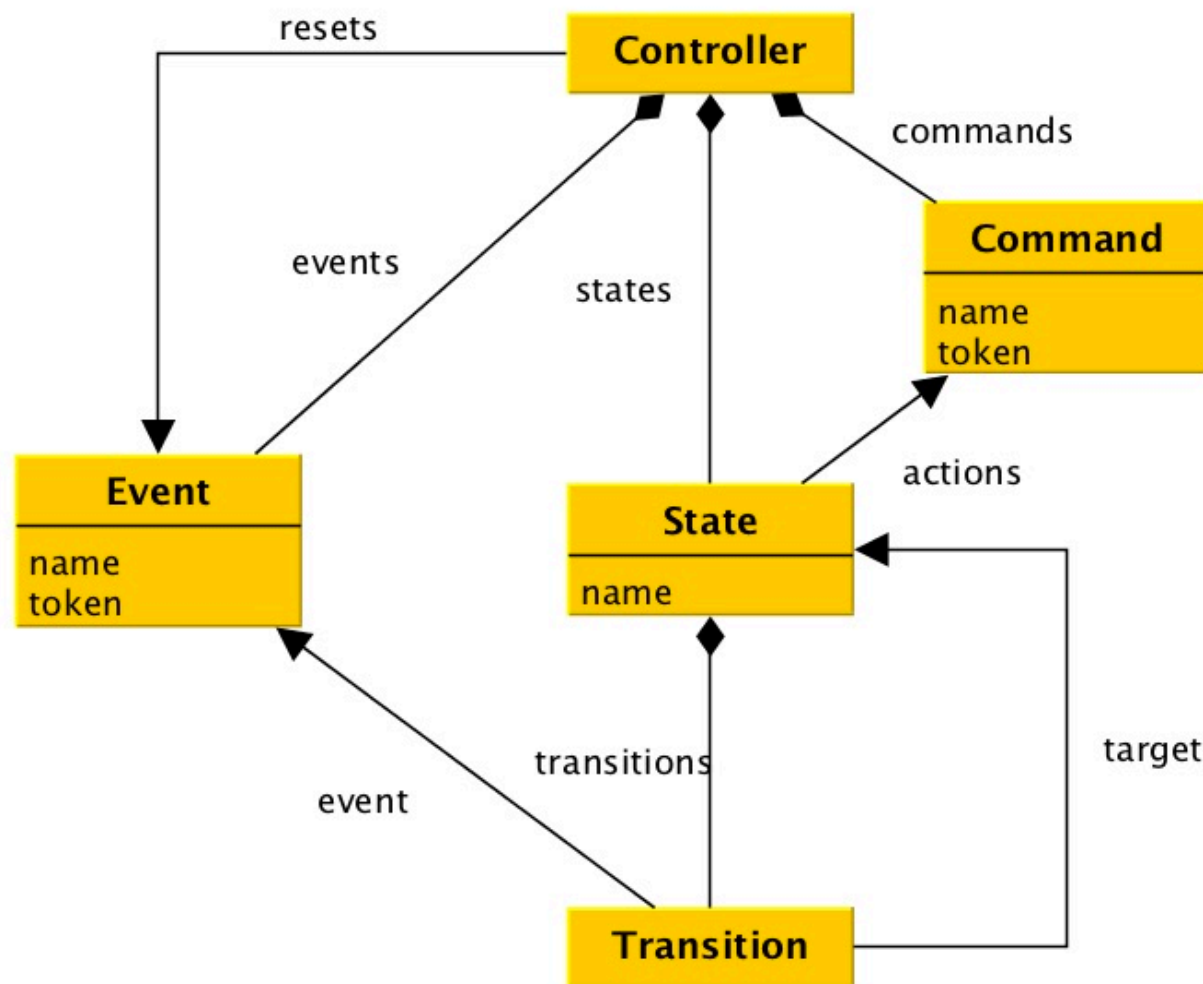
```
data Transition
    = transition(str event, str state);
```

Model validation

Is some model
a valid instance
of a meta model?



Side constraints



- $\forall s \in \text{states}, t \in s.\text{transitions}: t.\text{event} \neq \text{null} \wedge t.\text{target} \neq \text{null}$
- $\forall c \in \text{commands}: c.\text{name} \neq "" \wedge c.\text{token} \neq ""$
- etc.

Errors in QL

- Reference to undefined question
- Condition is not boolean
- Invalid operand types to operator
- duplicate question with different type (!)

```
form taxOfficeExample {  
  "Did you sell a house in 2010?"  
    hasSoldHouse: boolean  
  
  "Did you buy a house in 2010?"  
    hasBoughtHouse: boolean  
  
  "Did you enter a loan?"  
    hasMaintLoan: boolean  
  
  if (hasSoldHouse) {  
    "What was the selling price?"  
      sellingPrice: integer  
    "Private debts for the sold house:"  
      privateDebt: integer  
    "Value residue:"  
      valueResidue: integer =  
        sellingPrice - privateDebt  
  }  
}
```

Warnings in QL

- Same label for different questions
- Different label for occurrences of same question

```
form taxOfficeExample {  
  "Did you sell a house in 2010?"  
  hasSoldHouse: boolean  
  
  "Did you buy a house in 2010?"  
  hasBoughtHouse: boolean  
  
  "Did you enter a loan?"  
  hasMaintLoan: boolean  
  
  if (hasSoldHouse) {  
    "What was the selling price?"  
    sellingPrice: integer  
    "Private debts for the sold house:"  
    privateDebt: integer  
    "Value residue:"  
    valueResidue: integer =  
      sellingPrice - privateDebt  
  }  
}
```

Errors vs warnings

- Error = prevents compilation
- Warning = can still compile, but probably wrong

Next up

- Live coding name checking in the state machine language

