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

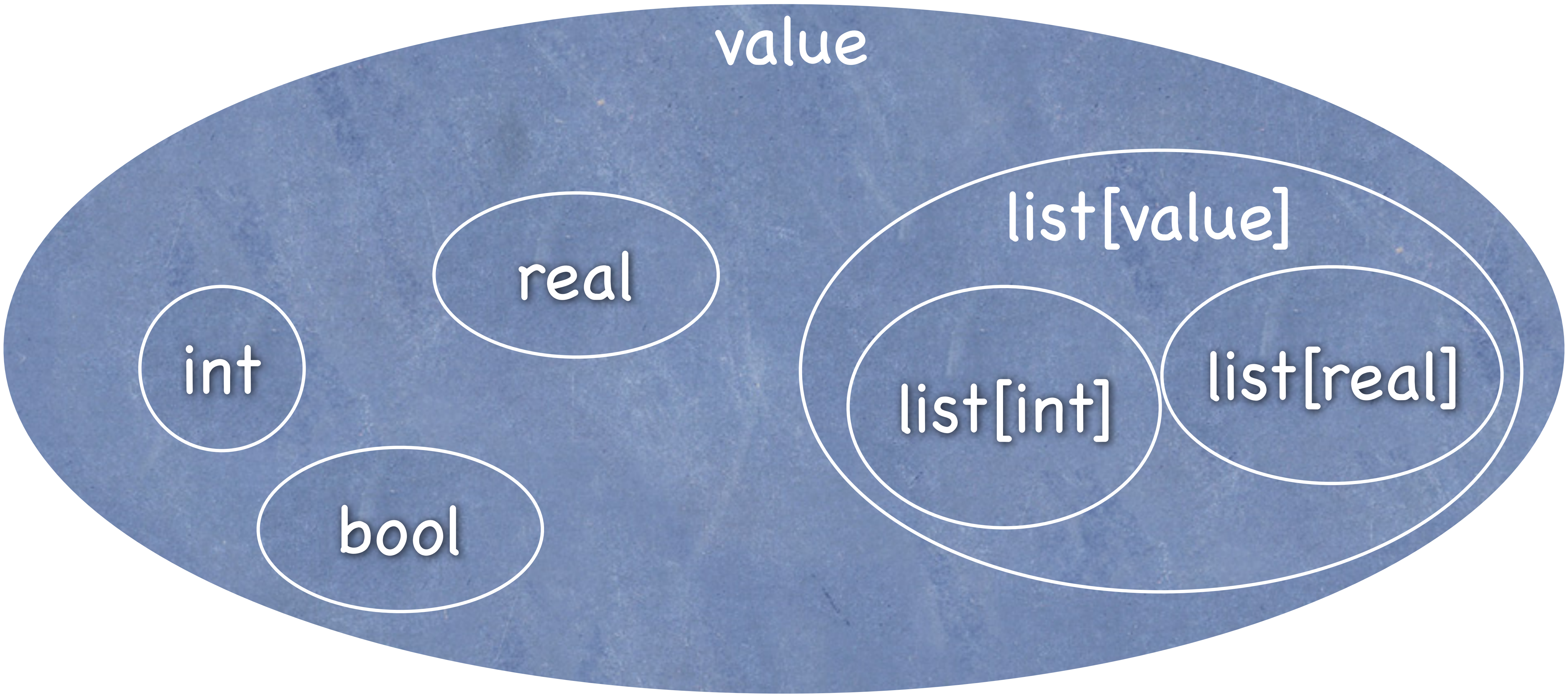
NameErrors.java

```java
package mypackage;

public class Something { }

class Bla extends Bla { }

class Foo extends Bar {
    List<Integer> aMethod() { }

    int aMethod() { }

    int anotherMethod() {
        int x = 3;
        {
            int y = 4;
            int x = 2345;
        }
        return y;
    }
}
```

Types

- **Types** partition run-time **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: int
  - f(x) { return x + 1} : int -> int
  - etc.

Relations between types



Sub typing

Type judgments

Premise

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

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

Conclusion

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

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

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

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

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

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

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**
  - eval: Expr → Value
- **Abstract** interpretation ≈ evaluation of expressions to **abstract** "values"
  - signEval: Expr → {+,-, 0, unknown}
  - typeEval: Expr → Type
  - …

```
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) {
  …
}
```

The environment

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

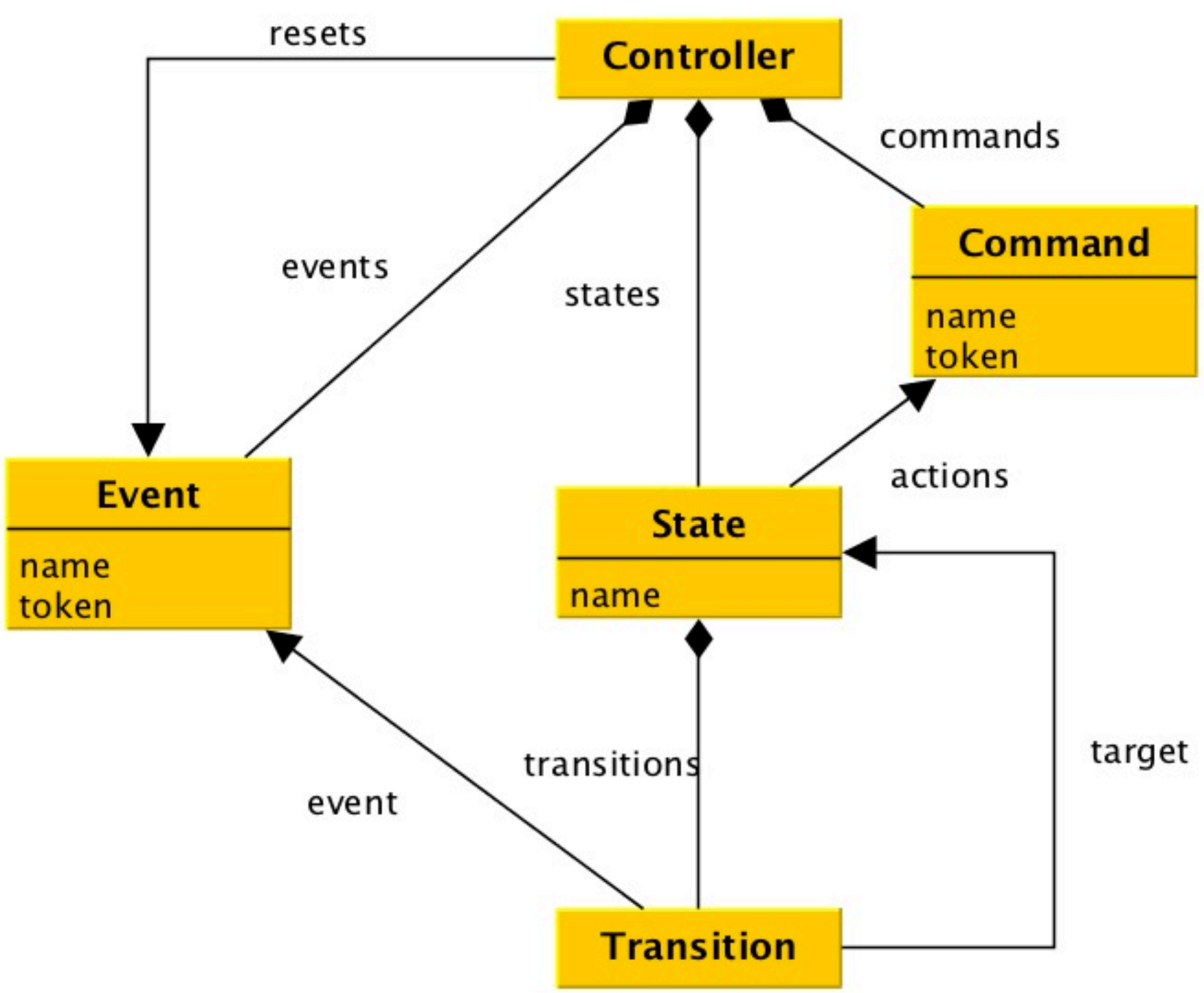
NameErrors.java

```java
package mypackage;

public class Something { }

class Bla extends Bla { }

class Foo extends Bar {
    List<Integer> aMethod() { }

    int aMethod() { }

    int anotherMethod() {
        int x = 3;
        {
            int y = 4;
            int x = 2345;
        }
        return y;
    }
}
```

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

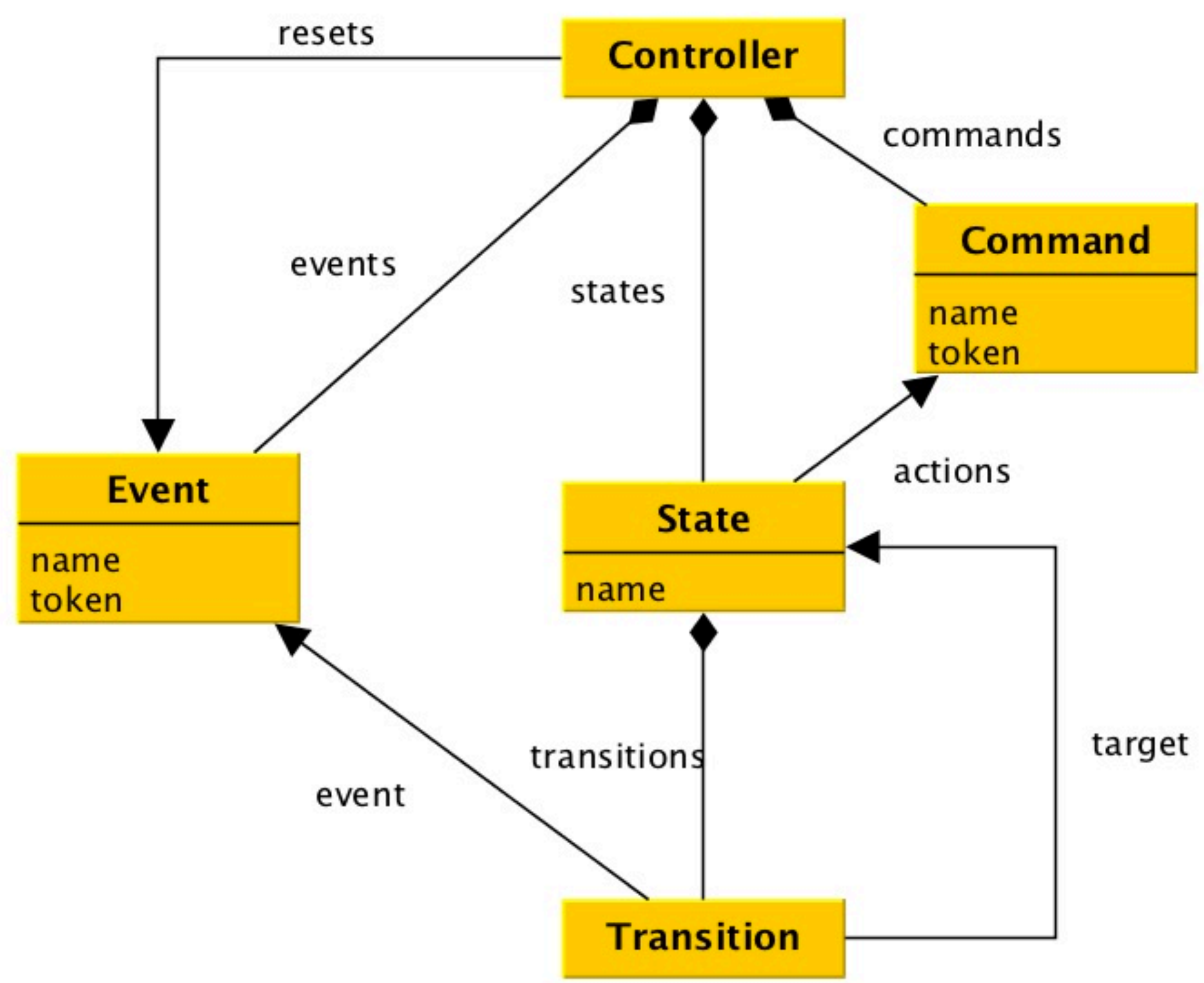
Metamodeling



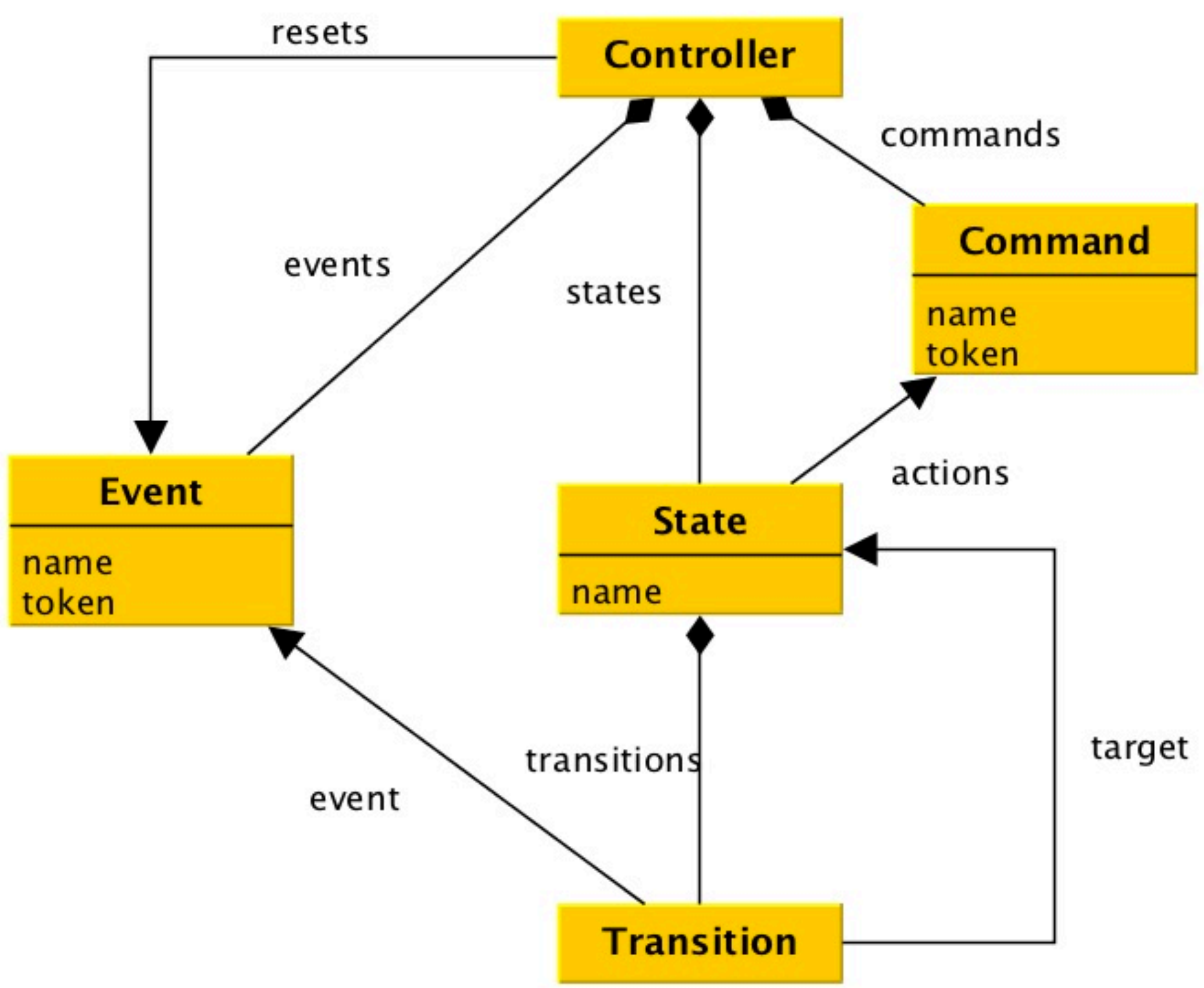

```
data Controller
  = controller(list[Event] events,
               list[str] resets,
               list[Command] commands,
               list[State] states);

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 $\in$ states, t $\in$ s.transitions: t.event $\neq$ null $\wedge$ t.target $\neq$ null
- $\forall$ c $\in$ commands: c.name $\neq$ “” $\wedge$ c.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