Hack your DSL with Rascal

Tijs van der Storm / Pablo Inostroza Valdera
@tvdstorm storm@cwi.nl / @metalinguist pvaldera@cwi.nl

CWI
Centrum Wiskunde & Informatica

University of Amsterdam

Outline

- Part 1
  - Introduction Rascal + Case (QL)
  - Coding: adding a statement to the QL language
- Part 2
  - Advanced topics: analysis & transformation
  - Coding: dependency analysis

What is Rascal?

- A meta programming language
- A language workbench
- A single language and environment to address “all” DSL engineering concerns

Rascal in a nutshell

- A **meta programming language** for **source code analysis and transformation**
- Java like syntax, but functional language
  - Immutable data, higher-order, algebraic data types etc.
- Powerful primitives for parsing, pattern matching, comprehensions, relation calculus, tree traversal
- Integration with Eclipse IDE

QL, a DSL for questionnaires

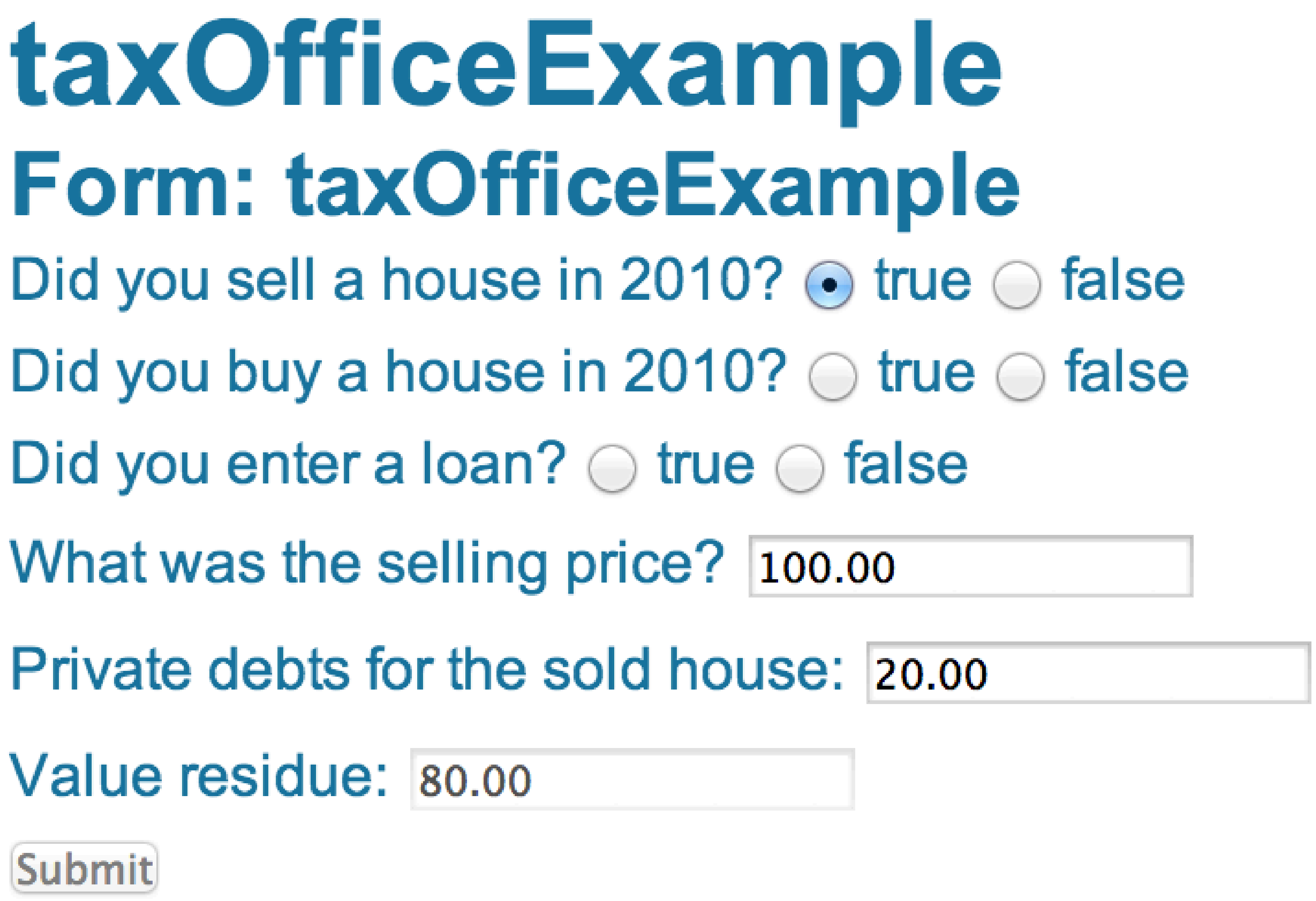
```
form taxOfficeExample {
  "Did you sell a house in 2010?"
    hasSoldHouse: boolean
  "Did you buy a house in 2010?"
    hasBoughtHouse: boolean
  "Did you enter a loan?"
    hasMaintLoan: boolean

  if (hasSoldHouse) {
    "What was the selling price?"
      sellingPrice: money
    "Private debts for the sold house:"
      privateDebt: money
    "Value residue:" valueResidue: money
      = sellingPrice - privateDebt
  }
}
```

Forms

Labeled questions

Conditions

Computed questions



taxOfficeExample

Form: taxOfficeExample

Did you sell a house in 2010? ● true ○ false

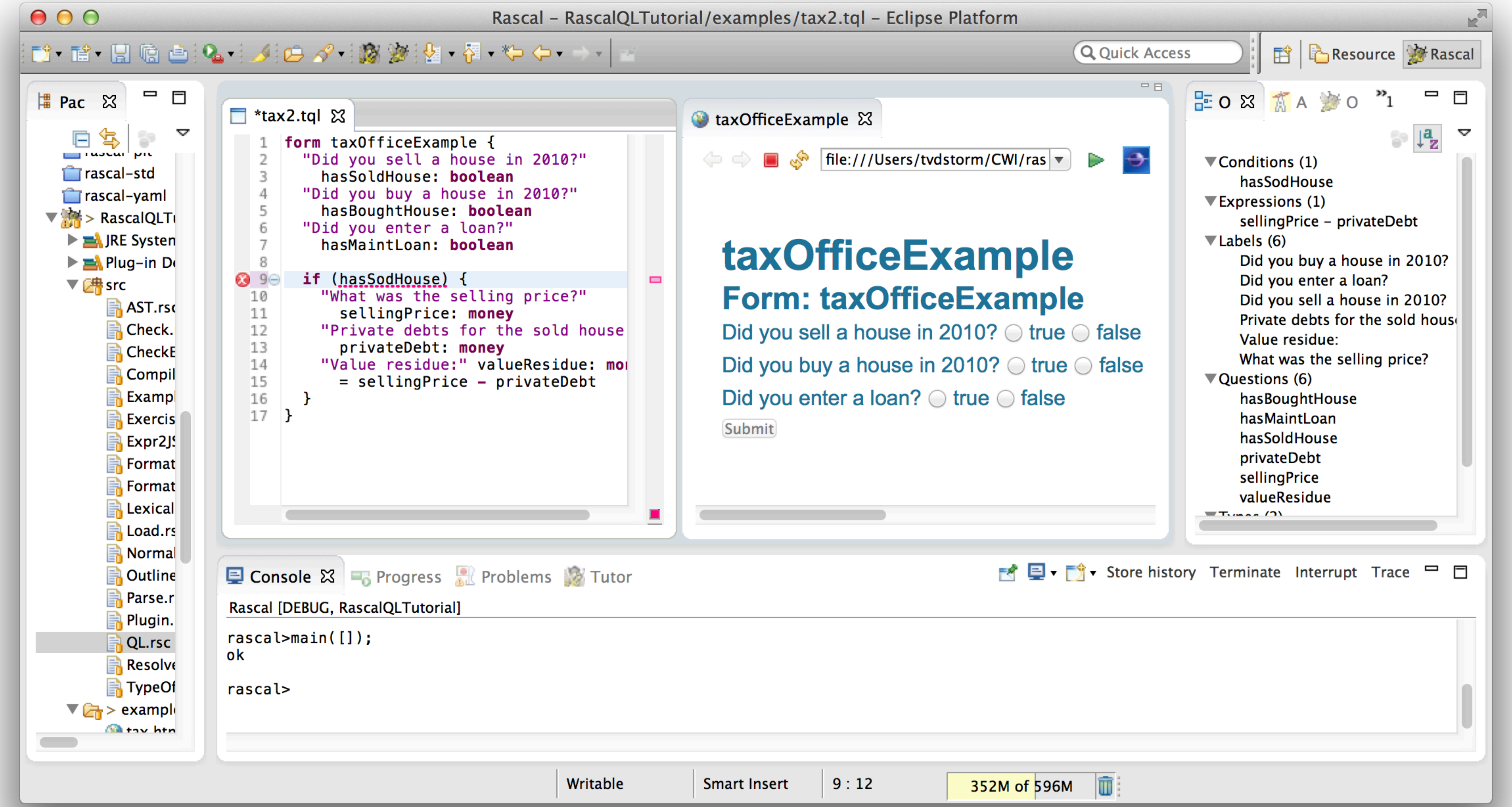
Did you buy a house in 2010? ○ true ○ false

Did you enter a loan? ○ true ○ false

What was the selling price? 100.00

Private debts for the sold house: 20.00

Value residue: 80.00



```
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 (hasSodHouse) {
    "What was the selling price?"
      sellingPrice: money
    "Private debts for the sold house
      privateDebt: money
    "Value residue:" valueResidue: mo
      = sellingPrice - privateDebt
  }
}
```

taxOfficeExample

Form: taxOfficeExample

Did you sell a house in 2010? ○ true ○ false

Did you buy a house in 2010? ○ true ○ false

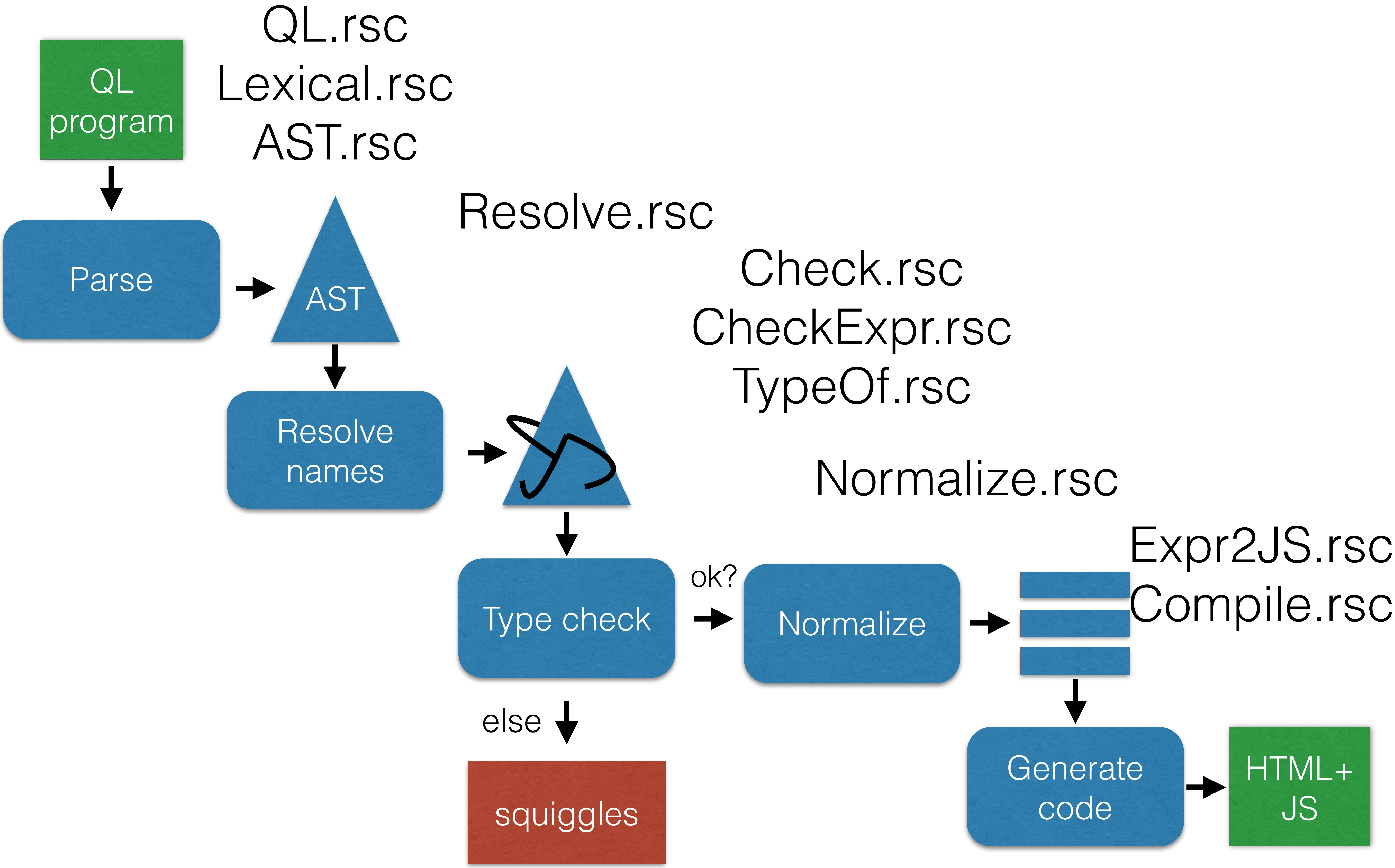
Did you enter a loan? ○ true ○ false

Submit

```
▼Conditions (1)
    hasSodHouse
▼Expressions (1)
    sellingPrice - privateDebt
▼Labels (6)
    Did you buy a house in 2010?
    Did you enter a loan?
    Did you sell a house in 2010?
    Private debts for the sold hous
    Value residue:
    What was the selling price?
▼Questions (6)
    hasBoughtHouse
    hasMaintLoan
    hasSoldHouse
    privateDebt
    sellingPrice
    valueResidue
```

Rascal [DEBUG, RascalQLTutorial]

```
rascal>main([]);
ok

rascal>
```



QL program

QL.rsc
Lexical.rsc
AST.rsc

Parse

AST

Resolve.rsc

Resolve names

Check.rsc
CheckExpr.rsc
TypeOf.rsc

Type check

ok?

else

squiggles

Normalize.rsc

Normalize

Expr2JS.rsc
Compile.rsc

Generate code

HTML+ JS

Parsing:
Context-free grammars

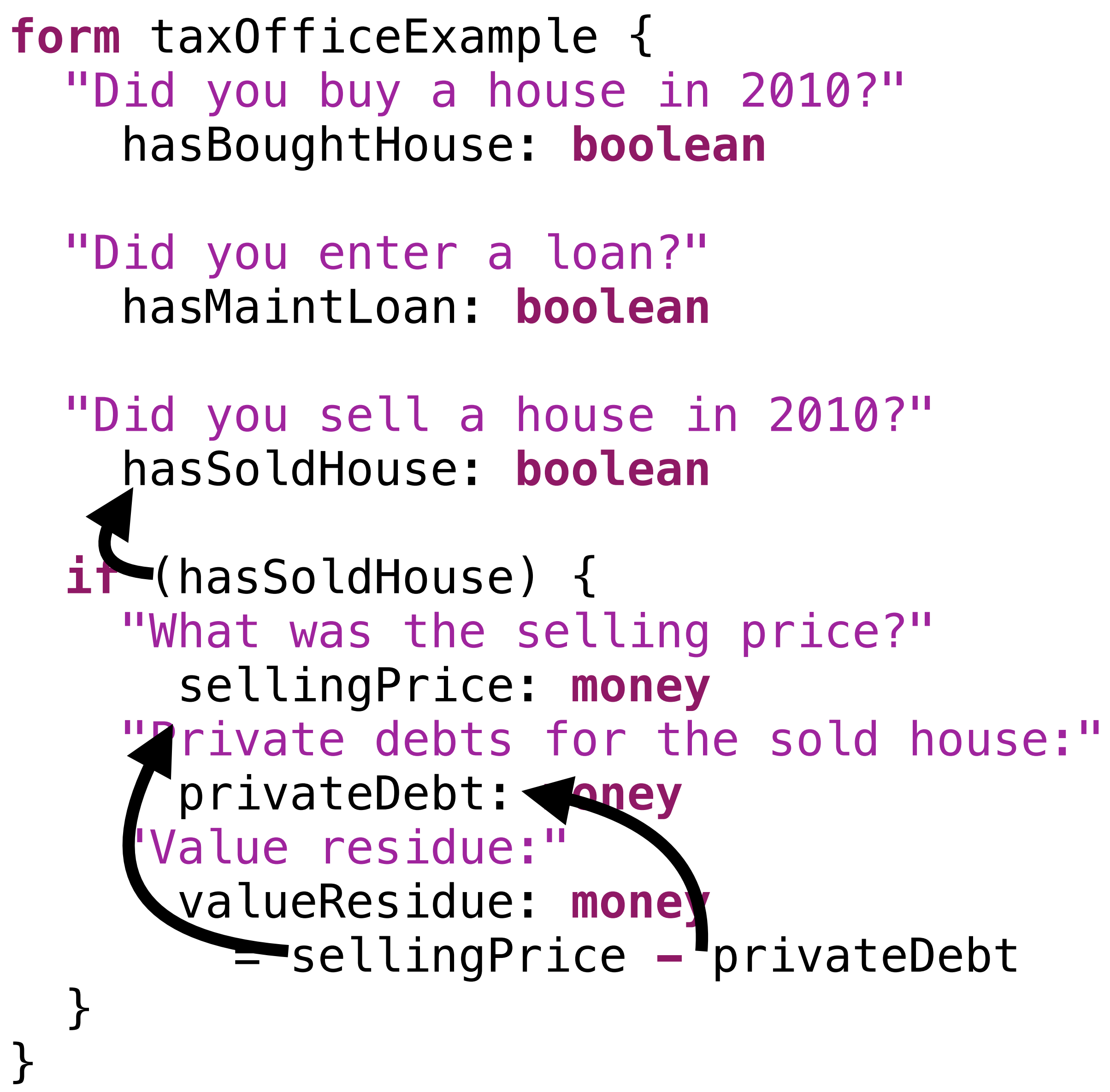
```
start syntax Form
  = form: "form" Id name "{" Question* questions "}"
  ;

syntax Question
  = question: Label label Id name ":" QType type
  | computed: Label label Id name ":" QType type "=" Expr expr
  | ifThen: "if" "(" Expr cond ")" Question () !>> "else"
  | @Foldable group: "{" Question* questions "}"
  ;
```

Abstract syntax:
Algebraic data types

```
data Form
  = form(str name, list[Question] body);

data Question
  = question(str label, Id name, QType tipe)
  | computed(str label, Id name, QType tipe, Expr expr)
  | ifThen(Expr cond, Question body)
  | group(list[Question] questions)
  ;
```

Name resolution: locations & relations

```
alias Names
  = rel[loc use, loc def];

names = resolve(ast);
```



```
form taxOfficeExample {
  "Did you buy a house in 2010?"
    hasBoughtHouse: boolean

  "Did you enter a loan?"
    hasMaintLoan: boolean

  "Did you sell a house in 2010?"
    hasSoldHouse: boolean

  if (hasSoldHouse) {
    "What was the selling price?"
      sellingPrice: money
    "Private debts for the sold house:"
      privateDebt: money
    "Value residue:"
      valueResidue: money
        = sellingPrice - privateDebt
  }
}
```

Type checking: pattern-based functions

```
set[Message] tc(ifThen(c, q), Info i)
  = tci(c, i) + tc(q, i);

set[Message] tc(ifThenElse(c, q1, q2), Info i)
  = tci(c, i) + tc(q1, i) + tc(q2, i);

set[Message] tc(group(qs), Info i)
  = ( {} | it + tc(q, i) |  q <- qs );

set[Message] tc(computed(l, n, _, e), Info i)
  = tcq(l, n, i) + tc(e ,i);

set[Message] tc(question(l, n, _), Info i)
  = tcq(l, n, i);
```

Normalization

```
form taxOfficeExample {
  "Did you buy a house in 2010?"
    hasBoughtHouse: boolean

  "Did you enter a loan?"
    hasMaintLoan: boolean

  "Did you sell a house in 2010?"
    hasSoldHouse: boolean

  if (hasSoldHouse) {
    "What was the selling price?"
      sellingPrice: money
    "Private debts for the sold house:"
      privateDebt: money
    "Value residue:"
      valueResidue: money
        = sellingPrice - privateDebt
  }
}
```

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

Normalization: pattern-based functions

```
list[Question] normalize(form(_, qs))
  = normalize(group(qs), \true());

list[Question] normalize(group(qs), Expr e)
  = ( [] | it + normalize(q, e) | q <- qs );

list[Question] normalize(ifThen(c, q), Expr e)
  = normalize(q, and(e, c));

list[Question] normalize(ifThenElse(c, q1, q2), Expr e)
  = normalize(q1, and(e, c))
  + normalize(q2, and(e, not(c)))
  ;

default list[Question] normalize(Question q, Expr e)
  = [ifThen(e, q)];
```

Code generation: string templates

```
str question2widget(str l, Id v, QType t, str parent, str e)
  = "var <v.name> = new QLrt.SimpleFormElementWidget({
    '  name: \"<v.name>\",
    '  label: <l>,
    '  valueWidget: new QLrt.<type2widget(t)>(<e>)
    '}).appendTo(<parent>);";

str exp2lazyValue(Expr e)
  = "new QLrt.LazyValue(
    '  function () { return [<ps>]; },
    '  function (<ps>) { return <expr2js(e)>; }
    ')"
  when str ps := expParams(e);
```

Hacking!

- Installation: JDK7, Eclipse, Rascal
- Clone project:
  - `https://github.com/cwi-swat/rascal-ql-tutorial-prototype`
- Right-click project, “Start console”
- Check out **exercises::Part1**
- Warm-up: **FizzBuzz**

Exercise 1: add `unless`

- Concrete syntax (QL.rsc)
- Abstract syntax (AST.rsc)
- Type checker (Check.rsc)
- Normalizer (Normalize.rsc)
- No need to change the compiler!

```
syntax Question
  = …
  | ifThen: "if" "(" Expr cond ")" Question () !>> "else"
  | unless: "unless" "(" Expr cond ")" Question
  ;


data Question
  = …
  | ifThen(Expr cond, Question body)
  | unless(Expr cond, Question body)
  ;


set[Message] tc(ifThen(c, q), Info i) = tci(c, i) + tc(q, i);

set[Message] tc(unless(c, q), Info i) = tci(c, i) + tc(q, i);


list[Question] normalize(ifThen(c, q), Expr e)
  = normalize(q, and(e, c));

list[Question] normalize(unless(c, q), Expr e)
  = normalize(q, and(e, not(c)));
```

Hack your DSL with Rascal

Tijs van der Storm / Pablo Inostroza Valdera
@tvdstorm storm@cwi.nl / @metalinguist pvaldera@cwi.nl

CWI Centrum Wiskunde & Informatica

University of Amsterdam

Recap part 1

- Basic features of Rascal: REPL, modules, grammars, ADTs, functions, basic pattern-matching, IDE support.
- Hacking the DSL: adding a new construct
- Evolving language implementation components

Part 2

- Transformation and analysis
  - Desugaring using `visit` (unless)
  - Dependency analysis

Visit

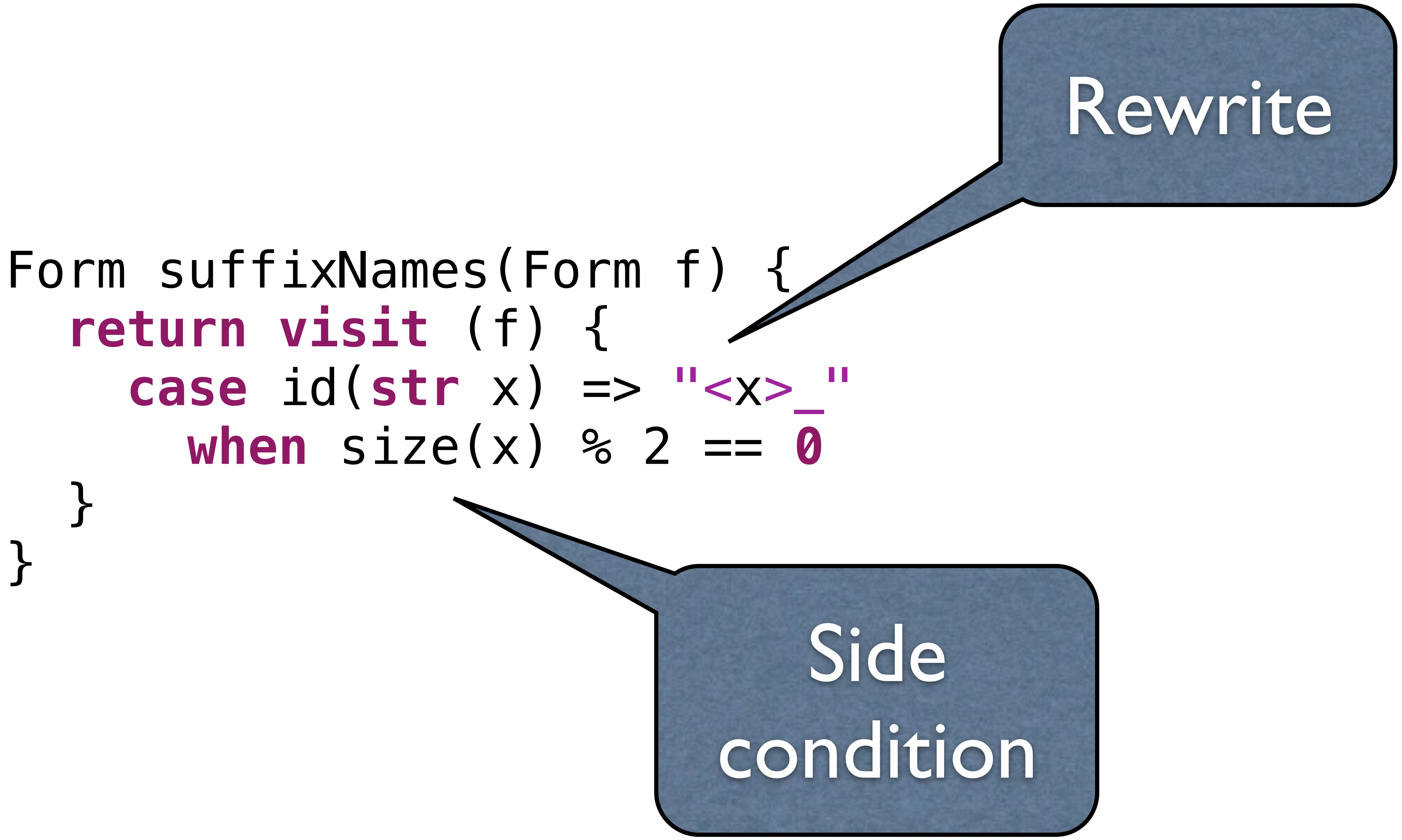
- Similar to case-based match construct
- Visits all nodes of data structure
- Specify cases of interest only
  - “Structure shy”
- Bottom-up, top-down, innermost, outermost strategies

Print all question names

do something when visiting a question

```
visit (ast) {
  case question(_, name, _): println(name);
}
```

Rewriting



```
Form suffixNames(Form f) {
  return visit (f) {
    case id(str x) => "<x>_"
      when size(x) % 2 == 0
  }
}
```

Rewrite

Side condition

Pattern matching

type-based

```
int x := 3;
```

structural

```
add(x, y) := add(var(id(”a”)), var(id( “b”)));
```

anti-matching

```
id(“c") !:= id("a");
```

list matching

```
[*x, 1, *y] := [5, 6, 1, 1, 1, 3, 4];
```

set matching

```
{1, *x} := {4, 5, 6, 1, 2, 3};
```

deep matching

```
/Question q := ast;
/ifThen(x, /computed(_, _, _, x))) := ast;
```

generation

```
3 <- {1,2,3}
int x <- {1,2,3}
```

Comprehensions

list

```
[ i | i <- [1..100], i % 2 == 0 ];
```

map

```
( i: i*i | i <- [1..10] );
```

set & relation

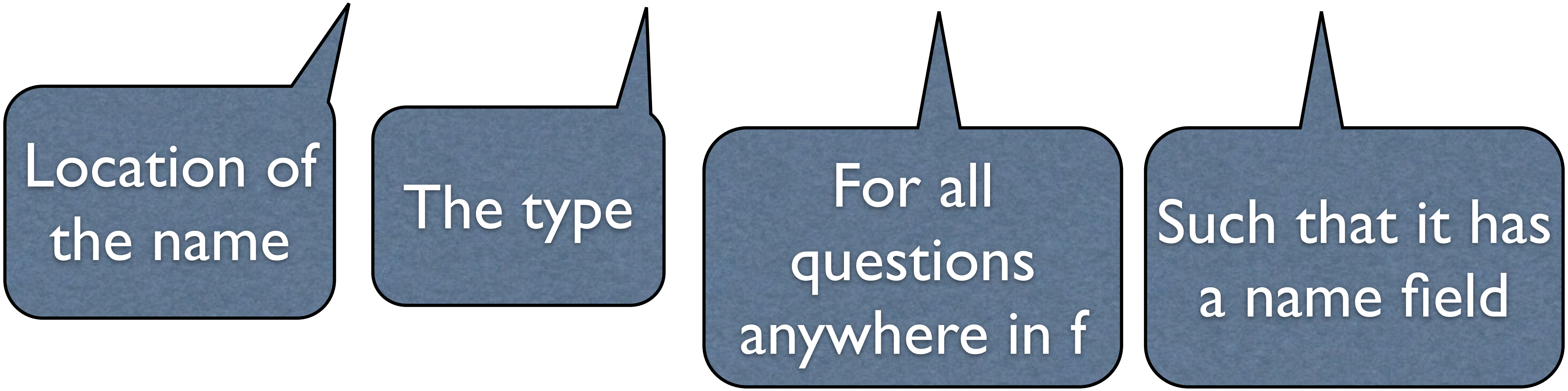
```
{ <i, i*i> | i <- [1..10] };
```

Reducing

```
( 0 | it + i | i <- [1..10] );
```

An example from QL

```
rel[loc name, QType tipe] typeEnv(Form f)
  = { <q.name@location, q.tipe> | /Question q := f, q has name };
```



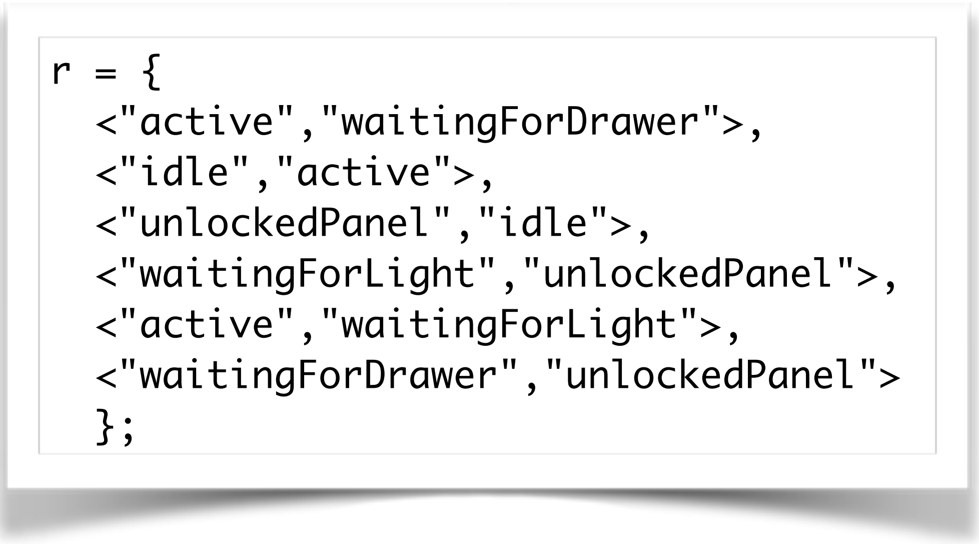
Location of the name

The type

For all questions anywhere in f

Such that it has a name field

Relational calculus



```
r = {
  <"active","waitingForDrawer">,
  <"idle","active">,
  <"unlockedPanel","idle">,
  <"waitingForLight","unlockedPanel">,
  <"active","waitingForLight">,
  <"waitingForDrawer","unlockedPanel">
  };
```

projection

```
r<0>;
```

invert

```
r<1,0>;
```

right image

```
r["active"];
```

transitive closure

```
r+;
```

transitive reflexive closure

```
r*;
```

relation composition

```
r o r
```

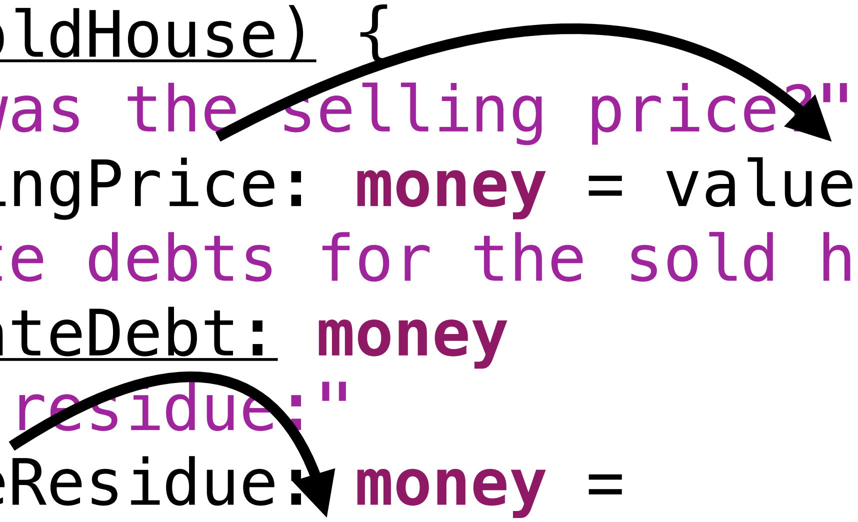
Cycles in questionnaires

```
form taxOfficeExample {
  "Did you buy a house in 2010?"
    hasBoughtHouse: boolean
  "Did you enter a loan?"
    hasMaintLoan: boolean

  if (hasSoldHouse) {
    "What was the selling price?"
      sellingPrice: money = valueResidue
    "Private debts for the sold house:"
      privateDebt: money
    "Value residue:"
      valueResidue: money =
        ((sellingPrice - privateDebt) * 2)
  }
  if (privateDebt > 0) {
    "Did you sell a house in 2010?"
      hasSoldHouse: boolean
  }
}
```

Dependencies.rsc

```
Deps controlDeps(Form f) {
  set[Node] definedIn(Question q) = { … };

  set[Node] usedIn(Expr e)  = { … };

  g = {};
  visit (f) {
    case ifThen(c, q):
      g += {<d, u> | d <- definedIn(q), u <- usedIn(c) };

    case IfThenElse(c, q1, q2):
      g += {<d, u> | d <- definedIn(q), u <- usedIn(c) }
        +  {<d, u> | d <- definedIn(q), u <- usedIn(c) };
  }

  return g;
}
```

Data cycles



```
form taxOfficeExample {
  "Did you buy a house in 2010?"
    hasBoughtHouse: boolean
  "Did you enter a loan?"
    hasMaintLoan: boolean

  if (hasSoldHouse) {
    "What was the selling price?"
      sellingPrice: money = valueResidue
    "Private debts for the sold house:"
      privateDebt: money
    "Value residue:"
      valueResidue: money =
        ((sellingPrice - privateDebt) * 2)
  }
  if (privateDebt > 0) {
    "Did you sell a house in 2010?"
       hasSoldHouse: boolean
  }
}
```

Exercises Part 2

- (Warm-up) Explicit desugaring of unless using visit
- Extracting **data** dependencies

Desugaring unless

```
Form desugar(Form f) {
  return visit(f) {
    case unless(e, q) => ifThen(not(e), q)
  };
}
```

Data dependencies

```
Deps dataDeps(Form f) {
  g = {};
  visit (f) {
    case computed(_, x, _, e):
      g += { <nodeFor(x), nodeFor(y)> | /Id y := e };
  }
  return g;
}
```

Wrap up

- A single, integrated language for meta programming
- Programming: explicit over implicit
- Powerful features to address DSL engineering concerns: grammars, traversal, relations, matching etc.
- Integrated with Eclipse IDE

Algebraic simplification

repeat
until stable

```
innermost visit (exp) {
    case add(0, x) => x
    case add(x, 0) => x
    case sub(x, 0) => x
    case sub(0, x) => neg(x)
    case neg(neg(x)) => x
    case mul(0, x) => 0
    case mul(x, 0) => 0
    case mul(1, x) => x
    case mul(x, 1) => x
    case mul(neg(x), neg(y)) => mul(x, y)
    case mul(neg(x), y) => neg(mul(x, y))
    case mul(x, neg(y)) => neg(mul(x, y))
    case div(x, 0) : throw "div by zero";
    case div(0, x) => 0
    case div(x, 1) => x
  }
```