A type system for name-preserving transformations between syntax representation formats in Rascal

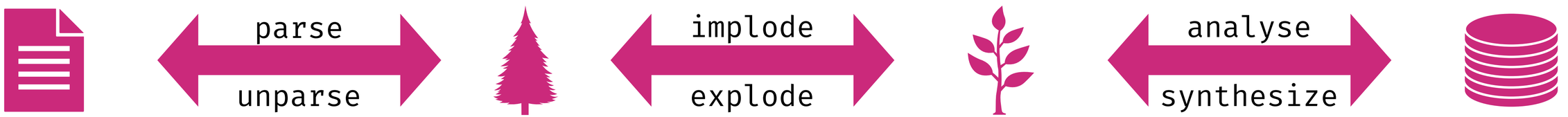
name polymorphism for “implode” and “explode”

Jurgen Vinju

Syntax Roles

```
layout X  = …
keyword X = …
lexical X = …
```

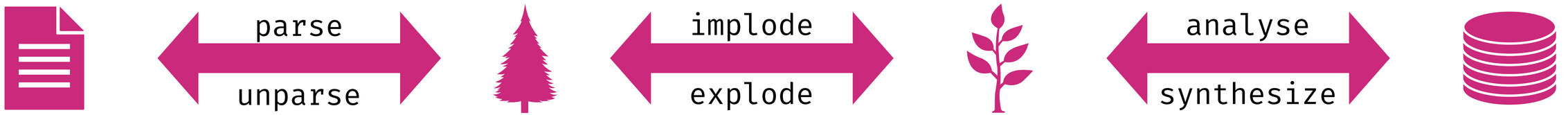
- syntax definitions
  - `syntax Statement = “if” Exp “then” Statement “else” Statement;`
  - `syntax Statement = \if: “if” Exp cond “then” Statement \true “else” Statement \false;`
- data definitions
  - `data Statement = if(Exp cond, Statement \true, Statement \false)`



str Statement <: Tree Statement <: node rel[loc,loc]

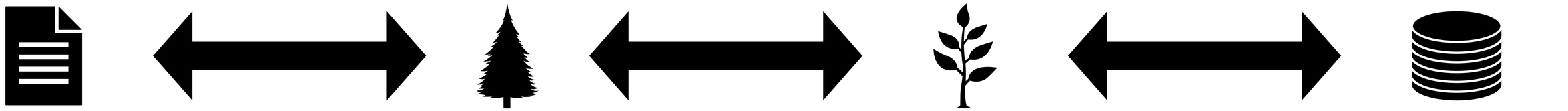
Mappings between syntax roles

- The two essential transformations can not be typed:
  - `Statement implode(Statement e); // ? type name “Statement” is ambiguous`
  - `Statement explode(Statement e); // ? type name “Statement” is ambiguous`
- With generic forms, &T and &U are statically unrelated (unconstrained)
  - `&T implode(type[&T <: node] targetType, &U <: Tree sourceType);`
  - `&T explode(type[&T <: Tree] targetType, &U <: node sourceType);`
- And, explode and implode require full grammars for the target representation



Distinguishing syntax roles

- We need to be able to:
  - **distinguish** between types of the same name, but a different role
  - **express constrained function signatures** that retain names but change roles
- New function signatures:
  - `data[Statement] implode(syntax[Statement] s);`
  - `syntax[Statement] explode(data[Statement] s);`



str

Tree

syntax[Statement]

node

data[Statement]

rel[loc,loc]

Constraining generic functions

- Generic functions guarantee **“name preservation”**:
  - `data[&T] implode(syntax[&U] t); // (&T).name == (&U).name`
  - `syntax[&T] explode(data[&U] t); // (&T).name == (&U).name`
- Implicit types:
  - if `data[X]` exists then `syntax[X]` exists as well, and vice versa.
  - Explode and implode may implicitly and dynamically derive one grammar from the other.
- One extra generic necessity: “**data** *&T*(loc src = |unknown:///|);”
  - for `explode` to retrieve the missing characters from the input file
  - for `implode` to store the actual locations of every parse tree node

Type rules

- T in **data** ==> T = **data**[T] // lift implicit name space to explicit name space, e.g. right after type-parameter binding
- T in **syntax** ==> T = **syntax**[T]
- **syntax**[**syntax**[T]] = **syntax**[T] // idempotence
- **syntax**[**data**[T]] = **syntax**[T] // modify role from data to syntax
- **data**[**data**[T]] = **data**[T] // idempotence
- **data**[**syntax**[T]] = **data**[T] // modify role from syntax to data
- **syntax**[&T] = **syntax**[&T] // open modifiers remain open
- **data**[&T] = **data**[&T]
- **otherwise** data[T] or syntax[T] ==> static error like `**data**[**int**]`
- examples:
  - **data**[&T] implode(**syntax**[&T] x) { ... }
  - **syntax**[&T] **removeAllSrcs**(**syntax**[&T] input) = **visit**(input) { Tree x => unset(x, “src”) };

Regular symbols

- What is syntax[list[A]] and what is data[A*]?
  - can we modify syntax types over the algebra of regular symbols?
  - yes. this is necessary to!
- `int size(&T* list)` - is this a **syntax** list or a **lexical** list?
  - the role matters: **syntax** has layout separators, **lexical** does not
- extending the modifier semantics over the regular symbols
  - **syntax**[A*] = {A L}*
  - **lexical**[A*] = A*
- Finally allows to write generic functions for syntactic lists, optionals, sequences, etc. without type ambiguity
- Fixes pattern matching on syntax types problems
- Encodes the relation between abstract and concrete lists.

Explode

enabling concrete syntax for external parsers

- Explode takes an AST that satisfies the AST contract
- and generates a parse tree, lifting it to the `Tree` data-type
- which takes the role of a “separator syntax tree” (SLE, Aarssen and Van der Storm)
- Productions are simply the constructor types
- Layout nodes before a rule starts, between every constituent, and after the last
- Pattern matching “just works”
- Substitution retains most lot of whitespace and comments

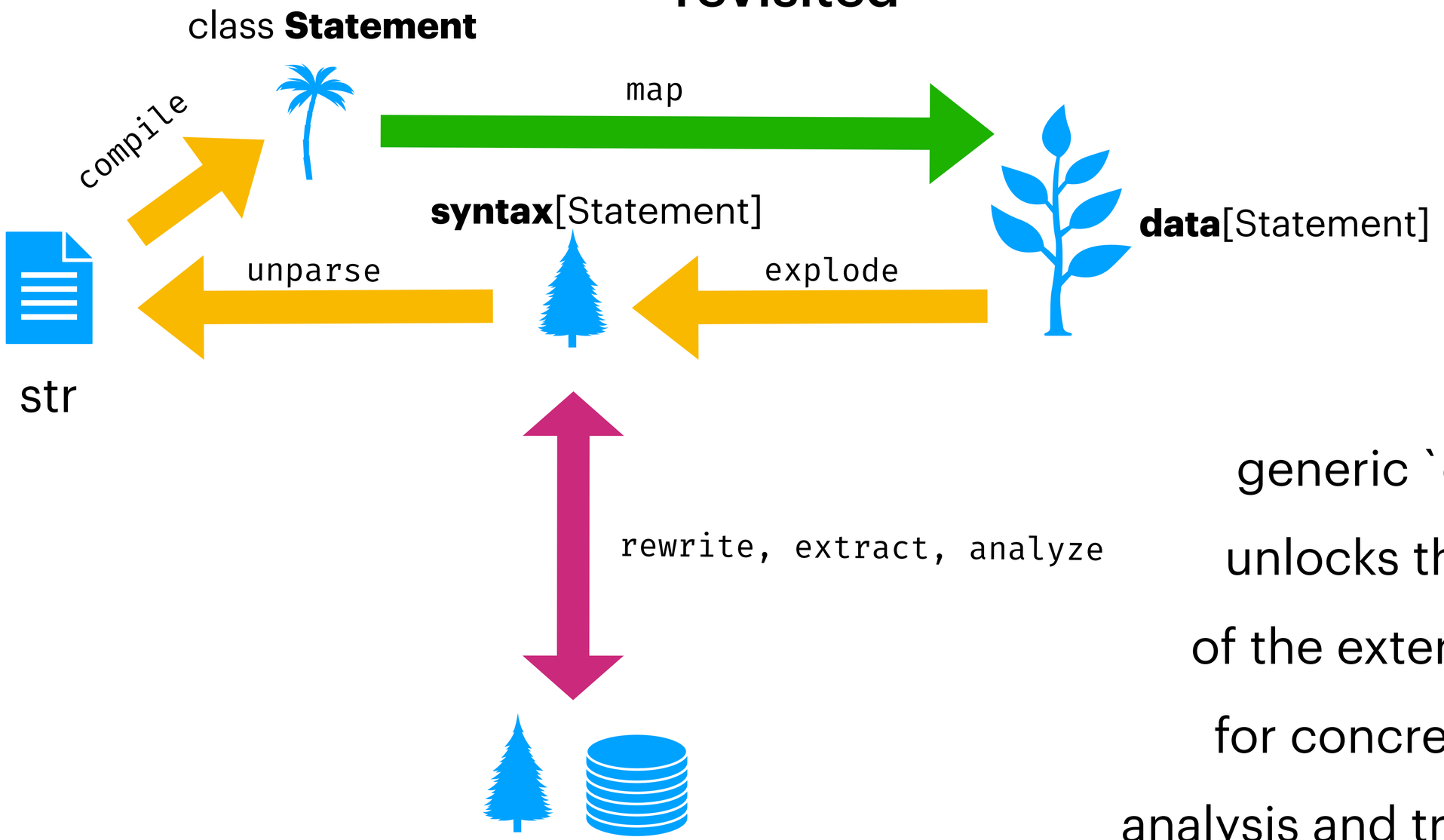
The AST contract

an AST is not just any ADT

```
bool astNodeSpecification(node n, str language = "java", bool checkNameResolution=false, bool checkSourceLocation=true) {
    // get a loc from any node if there is any.
    loc  pos(node y) = (loc f := (y.src?|unknown:///|(0,0))) ? f : |unknown:///|(0,0);
    int  begin(node y) = begin(pos(y));                    int  end(node y) = end(pos(y));
    int  begin(loc l)  = l.offset;                         int  end(loc l) = l.offset + l.length;
    bool leftToRight(loc l, loc r) = end(l) ≤ begin(r); bool leftToRight(node a, node b) = leftToRight(pos(a), pos(b));

    if (checkSourceLocation) {
        // all nodes have src annotations
        assert all(/node x := n, x.src?);

        // siblings are sorted in the input, even if some of them are lists
        assert all(/node x := n, [*_, node a, node b, *_] := getChildren(x), leftToRight(a,b));
        assert all(/node x := n, [*_, node a, [node b, *_], *_] := getChildren(x), leftToRight(a,b));
        assert all(/node x := n, [*_, [*_, node a], node b, *_] := getChildren(x), leftToRight(a,b));
        assert all(/node x := n, [*_, [*_, node a], [node b, *_], *_] := getChildren(x), leftToRight(a,b));
        assert all(/[*_, node a, node b, *_] := n, leftToRight(a,b));

        // children positions are included in the parent input scope
        assert all(/node parent := n, /node child := parent, begin(parent) ≤ begin(child), end(child) ≤ end(parent));
    }
}
```

Concrete syntax for external parsers

revisited



generic `explode`

unlocks the power

of the external parser

for concrete syntax

analysis and transformation

Summary

- Needed to write a generic explode
- But the type system of Rascal did not allow for this
- Added (generic) syntax roles
- This enables concrete syntax for external parsers