Evolution, extension, modularity

Tijs van der Storm

CWI
Centrum Wiskunde & Informatica

university of groningen

Recap

- Grammar -> Parser -> Parse Tree -> ~~AST~~
- Name resolution: recover referential structure
- Checking: find errors not captured by syntax
- Semantics: interpret or compile
- Transformations: at the core of SLE
- Today: after “release” 0.1 -> language **evolution**
  - and how to engineer to support it

Software evolution

Lehman’s laws (some of them)

Manny Lehman

- Continuing change: software must adapt to changes in the environment
- Increasing complexity: maintenance increases complexity unless measures to mitigate are taken.
- Continuing growth: software has to accumulate more features to keep users satisfied.
- Declining quality: without rigorous counter measures, the quality of a system will decline.

Software evolution

different kinds of maintenance

- Corrective: fixing bugs
- Preventive: anticipating things, like code quality problems, or changes in libraries etc.
- Adaptive: respond to changes in environment, security problems in dependencies, deprecation of frameworks, new versions etc.
- Perfective: improvements to, e.g., user-friendliness, performance, …

Language evolution: possible changes

- Changes to the language design:
  - new constructs (or deprecation), additional checks, change in semantics, …
- Changes to the implementation:
  - performance improvements, different back-end, new parser algorithm, …
- Changes w.r.t. code quality:
  - refactoring, rearchitecting, documentation
- Some changes affect users, others may not.

A brief and incomplete history of Java

Java	Year	Language Features	Library Features	Other
1.0	1996	[base language]	Applets	
1.1	1997	Inner classes		
1.2	1998		Swing API, Collections	
1.3	2000		JNDI API	**Hotspot JVM**
1.4	2002	Assertions	Regular expressions	
5	2004	**Generics**, for-each		
6	2006		Scripting API	
7	2007	Strings in switch, Try-with-resources		
8	2014	**Lambdas**, default methods	Stream API	
9	2017	Modules, private methods in interfaces		
10	2018	Var keyword		
11	2018		Improved HTTP API	
12-16	*Minor things and previews*			
17	2021	Sealed classes, Pattern matching instanceof		
18-20	*Minor things and previews*			
21	2023	Pattern matching in switch		Generational ZGC

https://en.wikipedia.org/wiki/Java_version_history

Evolution is inevitable

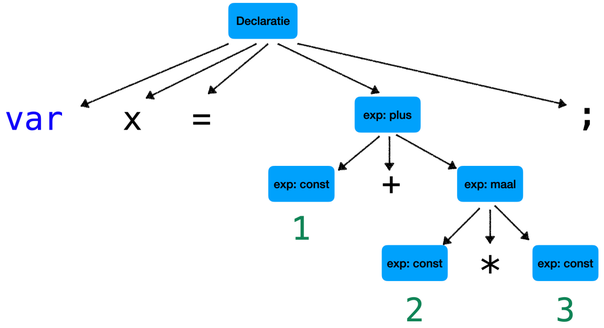
how to facilitate effective maintenance?

- Separation of concerns: “one concern, one module” (ideally)
  - e.g., separate parsing from type checking
- Modular decomposition: isolate and encapsulate (information hiding)
  - hide implementation details behind an interface
- Layering: built higher-level components on top of foundational components
  - e.g., intermediate languages
- Reuse: “code we don’t write, is code we don’t have to maintain”

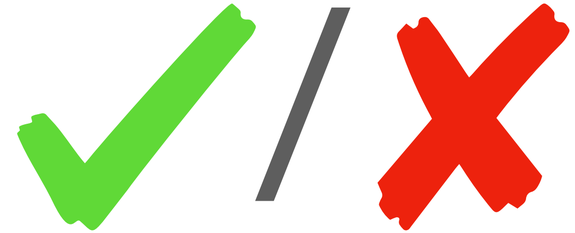
```
var x = 1 + 2 * 3;
```

Separation of concerns

Parse



Check



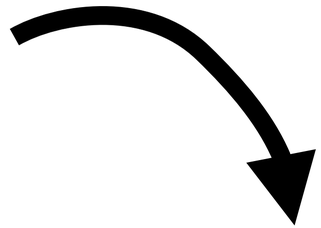
Execute

Result

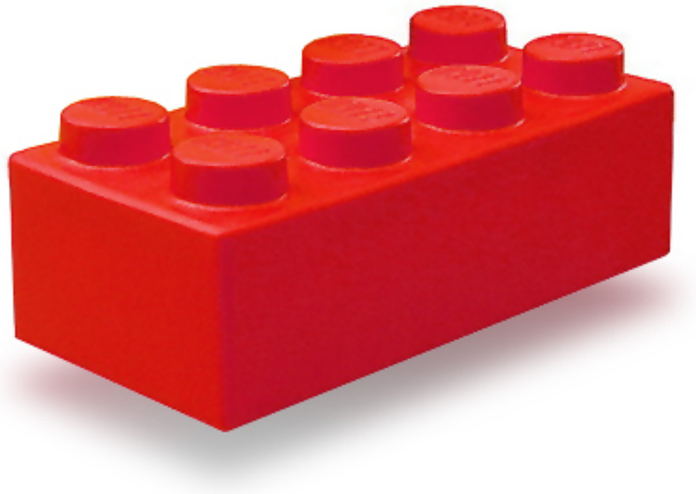
Detour: the expression problem

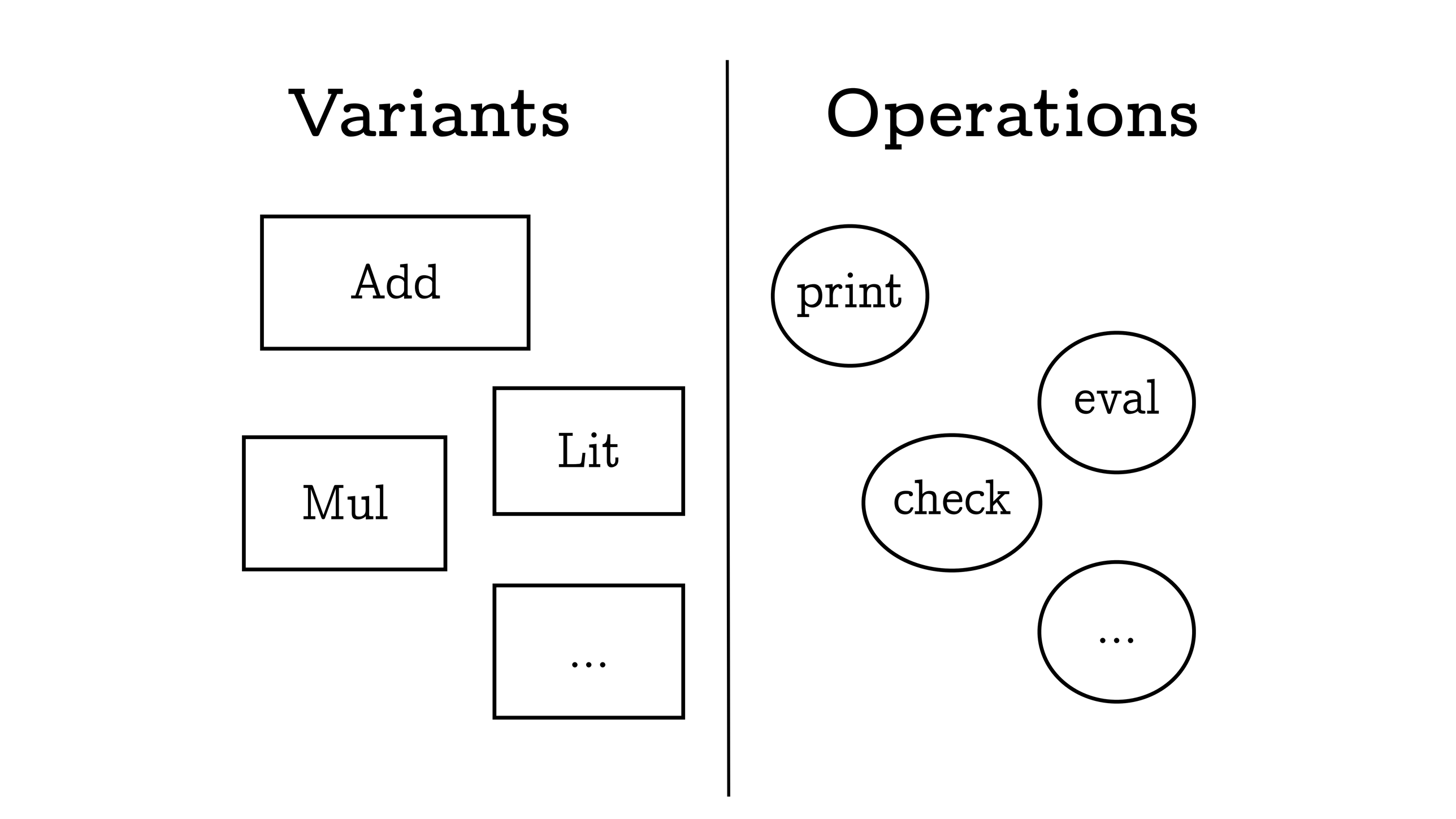
Extensibility

Multiplication feature

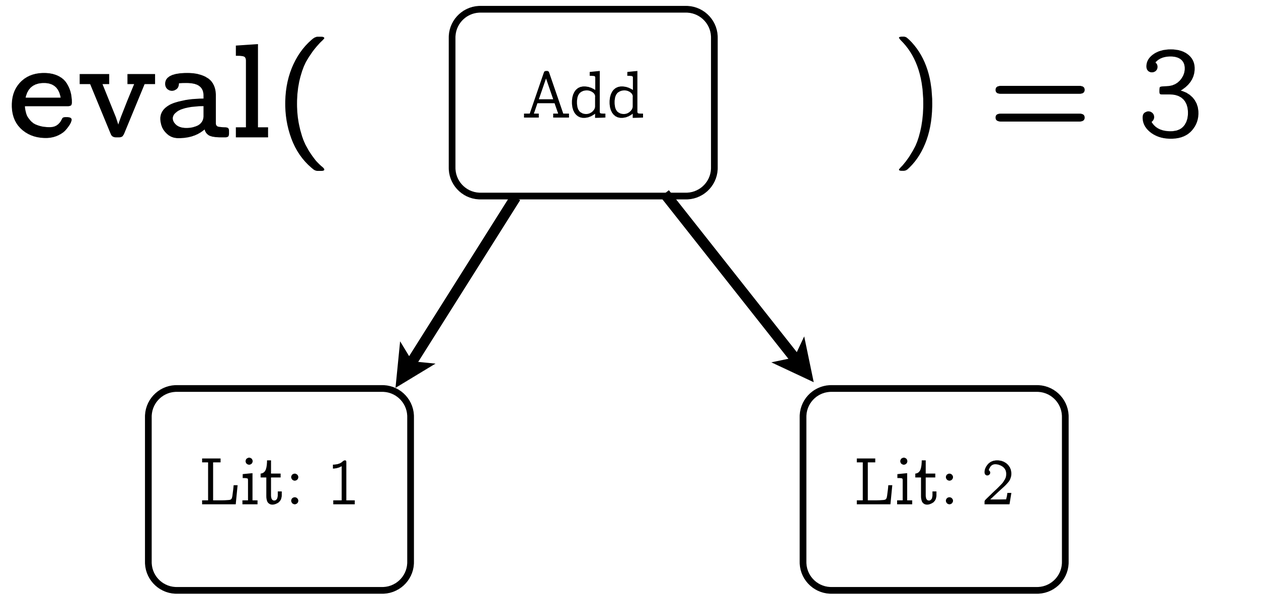


Language with addition

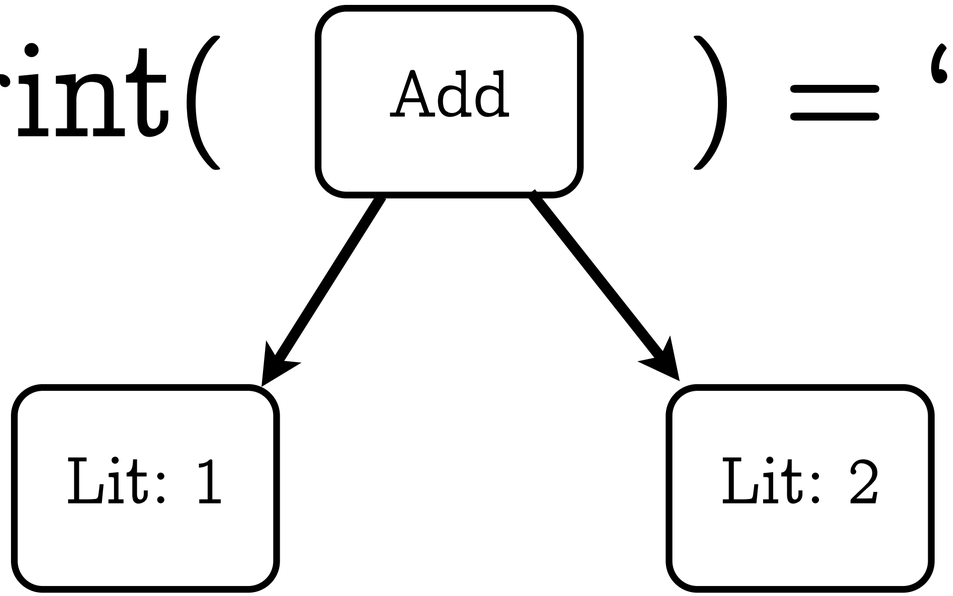



Variants
Operations
Add
Mul
Lit
...
print
eval
check
...

Evaluate operation



Print operation

print(

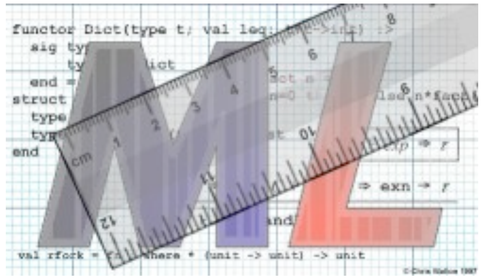
) = “1 + 2”

Variants

OO languages

Operations

FP languages



The Expression Problem

Philip Wadler, 12 November 1998

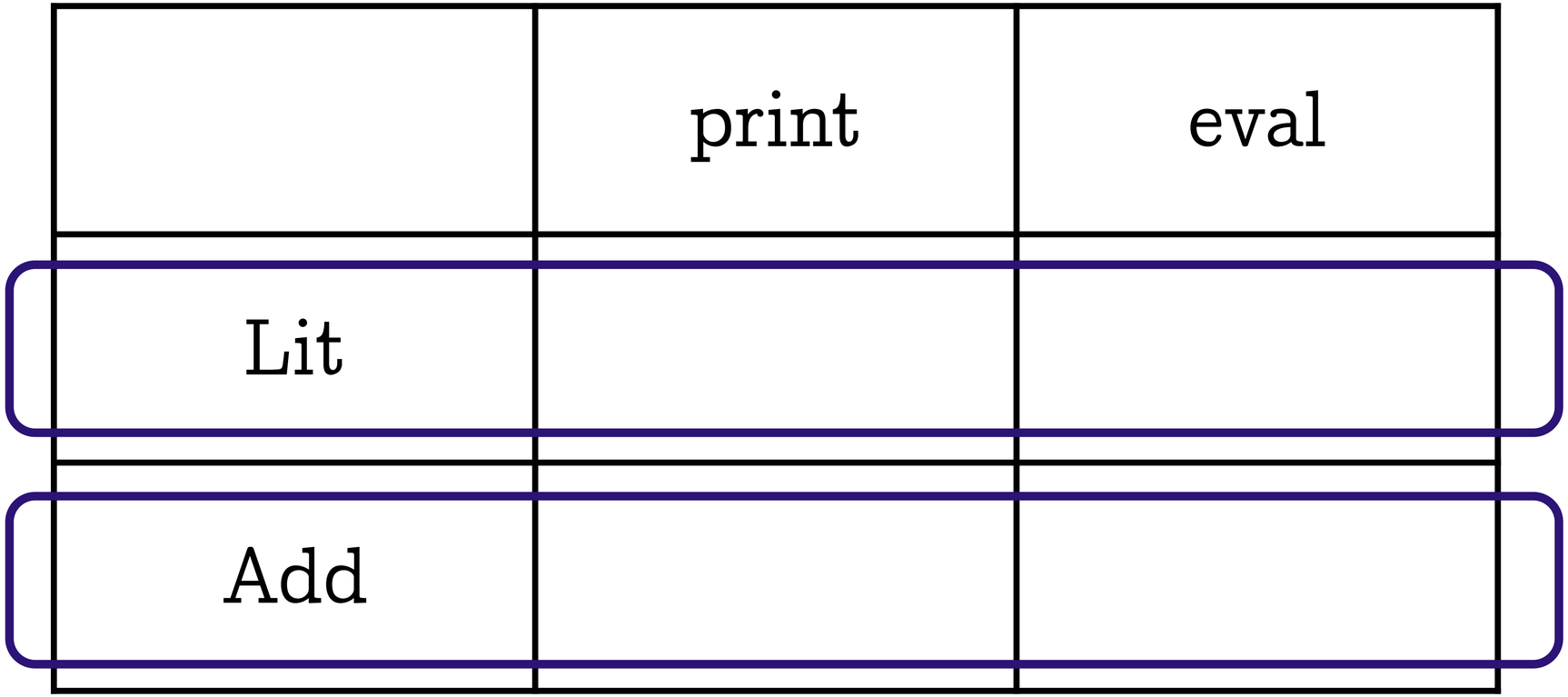
The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string).

http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

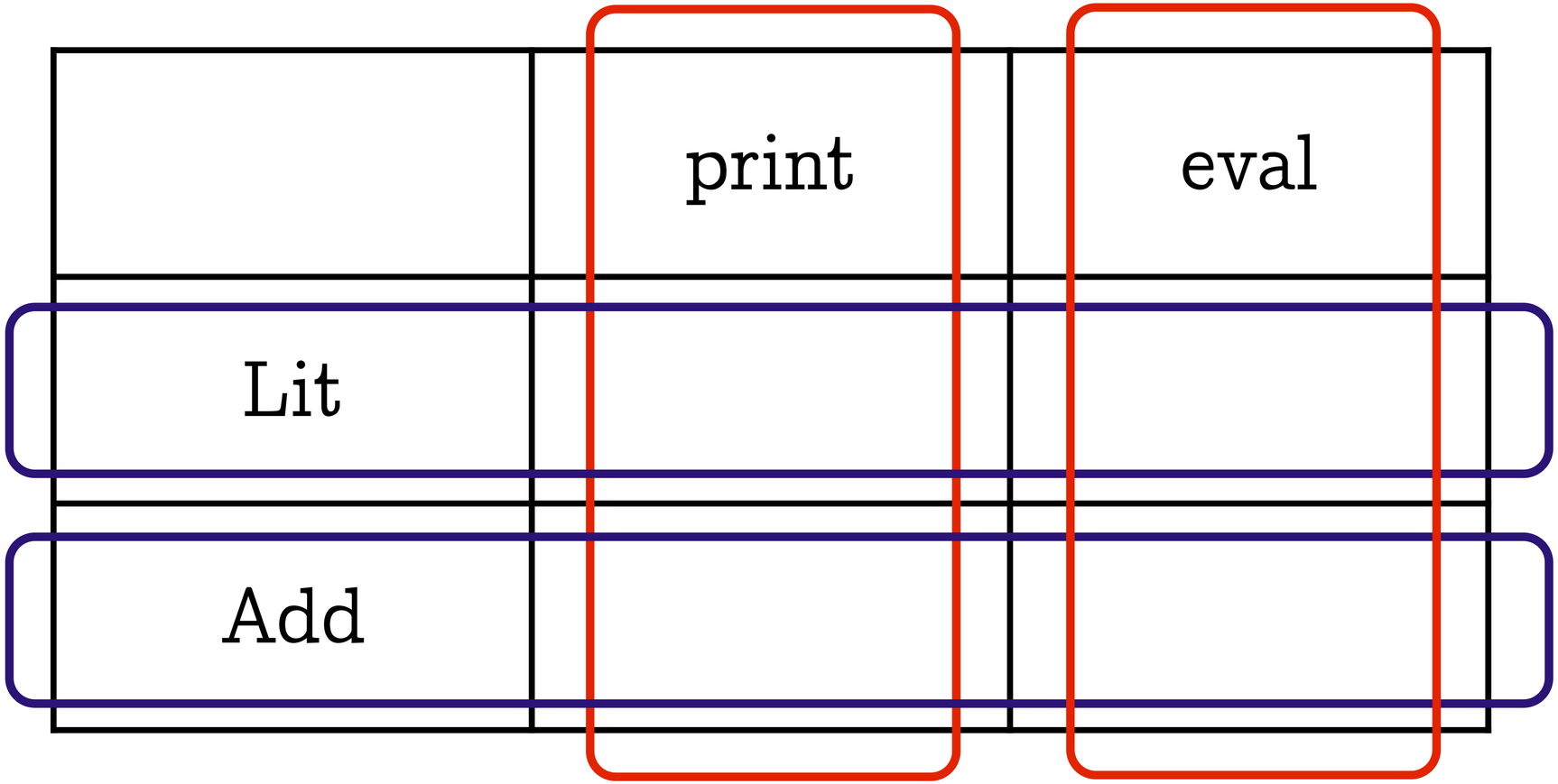
Decomposition

	print	eval
Lit		
Add		

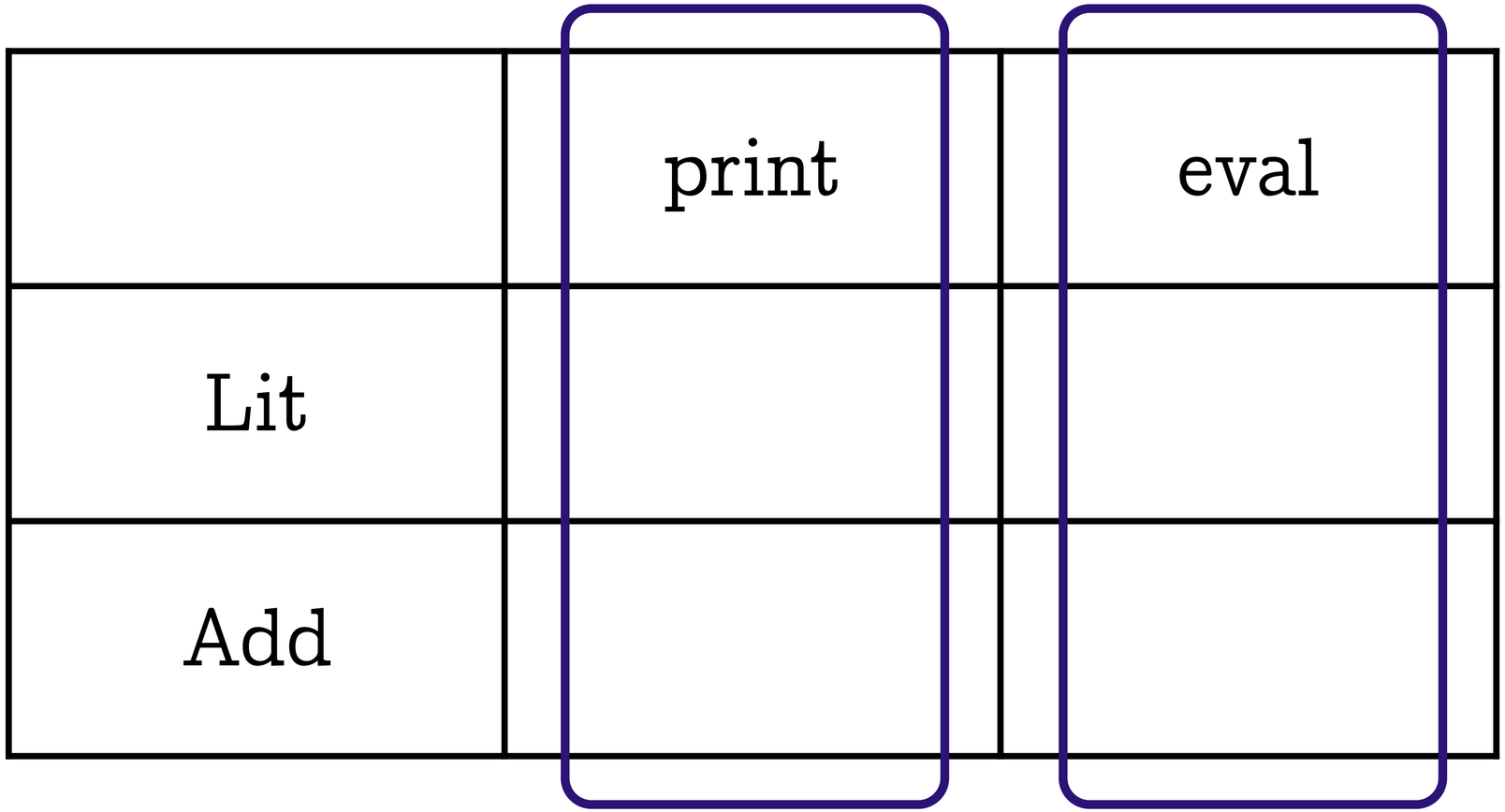
Object-oriented



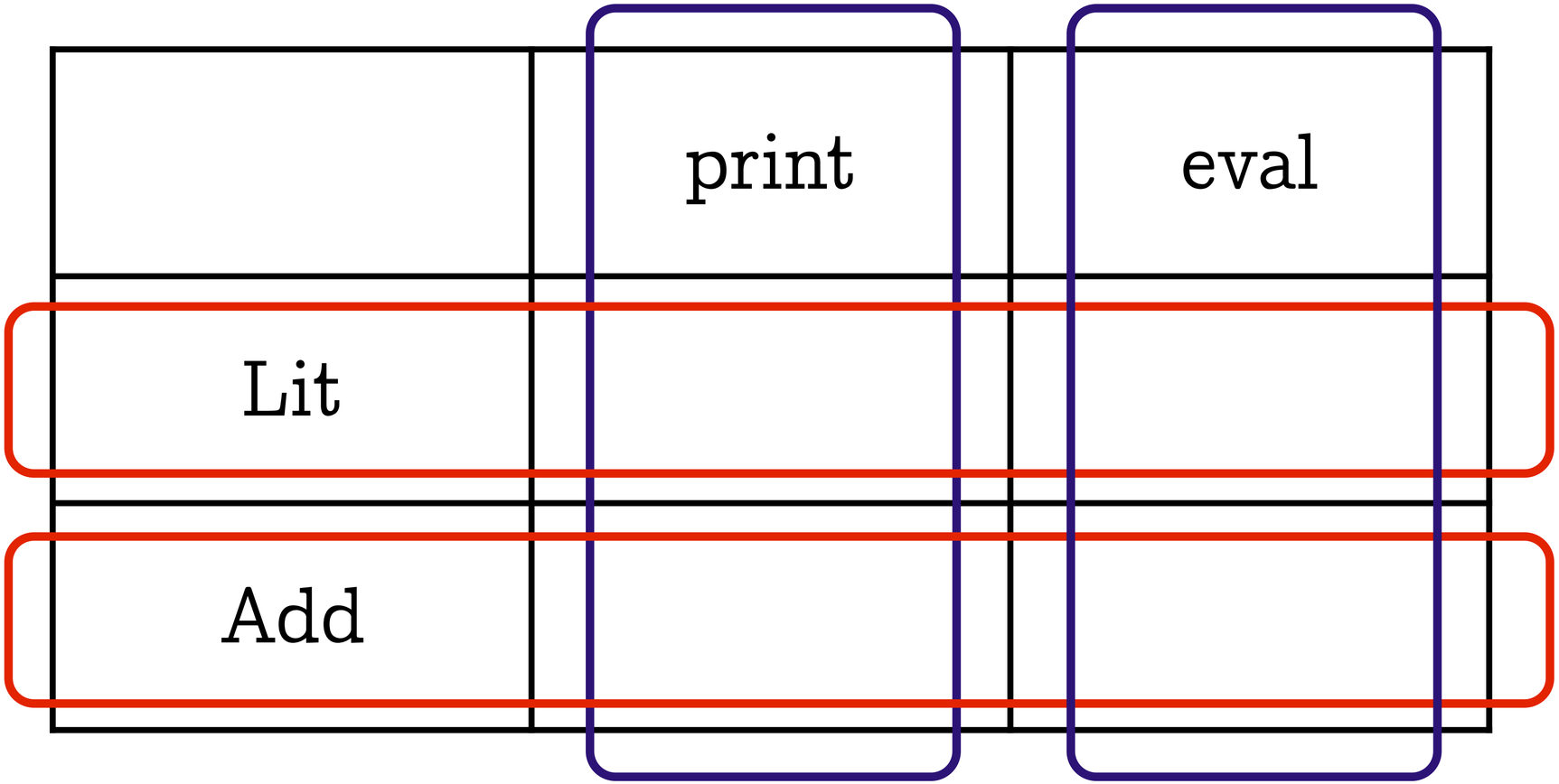
Object-oriented

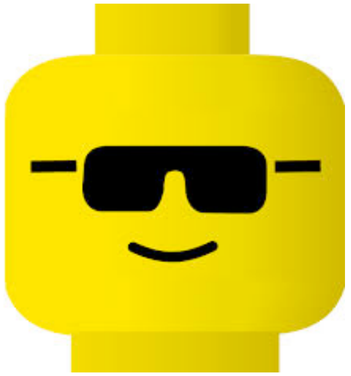
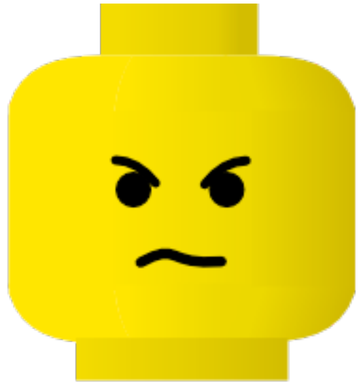
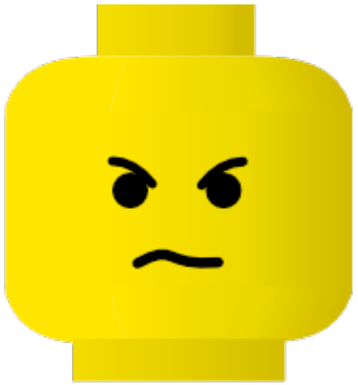
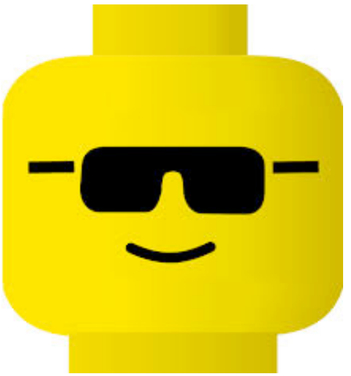


Functional



Functional



	Variants	Operations
Object-oriented style		
Functional style		

Solving the expression problem

	print	eval
Lit		
Add		

Extensibility for the Masses

Practical Extensibility with Object Algebras

Bruno C. d. S. Oliveira[1] and William R. Cook[2]

[1]National University of Singapore
bruno@ropas.snu.ac.kr
[2] University of Texas, Austin
wcook@cs.utexas.edu

Rascal modularity in action

Oberon-0 Language Levels

- $L1$: Basic control flow statements, constant, type and variable declarations, assignments and expressions.
- $L2$: Extension with FOR-statements and CASE-statements.
- $L3$: Definition of (nested) procedures and procedure call statements.
- $L4$: Support for arrays and records, including subscript and field selection expressions and assignment statements.

	T_1 SYNTAX	T_2 BIND	T_3 CHECK	T_4 DESUGAR	T_5 COMPILE	T_6 EVAL	$T'_{7,8}$ NORMALIZE	T_7 TOJAVA	T_8 TOJVM	T_9 CFLOW
L_1	244	162	146	–	60	211	–	–	–	121
L_2	80	44	26	39	–	–	–	–	–	71
L_3	73	117	39	–	106	167	–	–	–	36
L_4	90	67	53	–	48	117	743	78	116	–
Total	487	390	264	39	214	495	743	78	116	228

Modular language implementation in Rascal

– experience report

Bas Basten [a], Jeroen van den Bos [f], Mark Hills [d], Paul Klint [a,c], Arnold Lankamp [e], Bert Lisser [a], Atze van der Ploeg [b], Tijs van der Storm [a,c], Jurgen Vinju [a,c]

AddLit.rsc U

sle-master-course > src > expr > AddLit.rsc > { } expr

Import in new Rascal terminal

```
1  module expr::AddLit
2
3  extend lang::std::Layout;
4
5  syntax Expr
6    = Lit
7    | bracket "(" Expr ")"
8    | left Expr "+" Expr
9    ;
10
11 lexical Lit = [0-9]+;
12
13
```

Mul.rsc U

sle-master-course > src > expr > Mul.rsc > ...

Import in new Rascal terminal

```
1  module expr::Mul
2
3  extend expr::AddLit;
4
5  syntax Expr
6     = left Expr "*" Expr;
7
8
```

EvalAddLit.rsc U

sle-master-course > src > expr > EvalAddLit.rsc > { }

Import in new Rascal terminal

```
1  module expr::EvalAddLit
2
3  import expr::AddLit;
4  import String;
5
6  int eval((Expr)`<Lit n>`) = toInt("<n>");
7
8  int eval((Expr)`(<Expr e>)`) = eval(e);
9
10 int eval((Expr)`<Expr lhs> + <Expr rhs>`)
11     = eval(lhs) + eval(rhs);
12
13 test bool testAdd() = eval((Expr)`1 + 2`)
```

EvalMul.rsc U

sle-master-course > src > expr > EvalMul.rsc > ...

Import in new Rascal terminal

```
1  module expr::EvalMul
2
3  import expr::Mul;
4  extend expr::EvalAddLit;
5
6  int eval((Expr)`<Expr lhs> * <Expr rhs>`)
7      = eval(lhs) * eval(rhs);
8
9
10 test bool testMul() = eval((Expr)`2 * 3`) == 6;
11
12 test bool testAddMul()
13     = eval((Expr)`1 + (2 * 3)`) == 7;
```

In the lab

- Pick one of the provided language extensions and implement them
- Think about:
  - which operations (check, eval, etc.) are affected
  - how to modularize? (no copy paste, no reimplementation, reuse as much as possible)
  - can you organize the code so that existing (“base QL”) code remains the same (minimize modifications to original code)
- In other words: try to **extend** rather than **change** your code.