REPL-first languages

L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
ltvanbinsbergen@acm.org

March 10, 2021

Deriving REPLs and Notebooks for DSLs

From DSL Specification to Interactive Computer Programming Environment

Pierre Jeanjean
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
pierre.jeanjean@inria.fr

Benoit Combemale
University of Toulouse
Toulouse, France
benoit.combemale@irit.fr

Olivier Barais
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
olivier.barais@irisa.fr

Figure: SLE2019

Bacatá: Notebooks for DSLs, Almost for Free

Mauricio Verano Merino[a,d], Jurgen Vinju[a,b], and Tijs van der Storm[b,c]

a Eindhoven University of Technology, The Netherlands
b Centrum Wiskunde & Informatica, The Netherlands
c University of Groningen, The Netherlands
d Océ Technologies B.V., The Netherlands

Figure: Art, Science, and Engineering of Programming

Deriving REPL/Notebook – commonalities

- READ: Identify entry points, i.e. the alternatives in syntactic root

Deriving REPL/Notebook – commonalities

- READ: Identify entry points, i.e. the alternatives in syntactic root
- EVAL: Connect entry points with evaluation function in DSL interpreter

Deriving REPL/Notebook – commonalities

- READ: Identify entry points, i.e. the alternatives in syntactic root
- EVAL: Connect entry points with evaluation function in DSL interpreter
- PRINT: Specify function to convert evaluation result to string

Deriving REPL/Notebook – commonalities

- READ: Identify entry points, i.e. the alternatives in syntactic root
- EVAL: Connect entry points with evaluation function in DSL interpreter
- PRINT: Specify function to convert evaluation result to string
- LOOP:

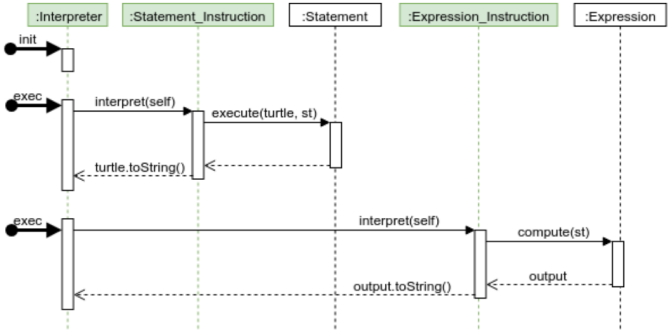


Figure 8. Overall Execution Flow for Logo

Deriving REPL/Notebook – commonalities

- READ: Identify entry points, i.e. the alternatives in syntactic root
- EVAL: Connect entry points with evaluation function in DSL interpreter
- PRINT: Specify function to convert evaluation result to string
- LOOP:

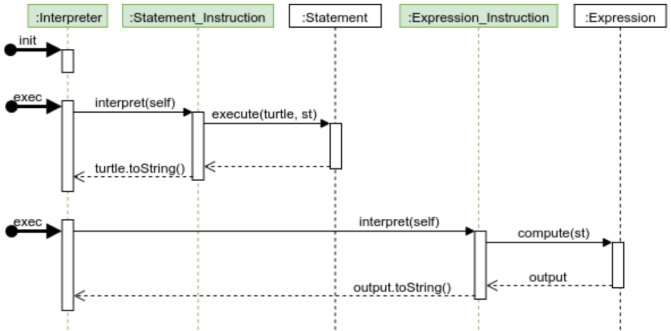


Figure 8. Overall Execution Flow for Logo

How does one execution affect the next?

Formal model based on structural operational semantics

A **language** L is a structure $\langle P, \Gamma, \gamma^0, I \rangle$ with:

- P a set of programs,
- Γ a set of configurations (containing semantic entities, attributes, algebraic effects, etc..),
- γ^0 an initial configuration with $\gamma^0 \in \Gamma$ and
- I a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \to \Gamma$.

$$\mathit{interpreter} : \mathit{program} \times \mathit{config} \to \mathit{config}$$

Sufficiently general to capture at least all (deterministic) languages that can have their semantics expressed as a transition function (e.g. using small-step or big-step semantics)

Formal model based on structural operational semantics

A **language** L is a structure $\langle P, \Gamma, \gamma^0, I \rangle$ with:

- P a set of programs,
- Γ a set of configurations (containing semantic entities, attributes, algebraic effects, etc..),
- γ^0 an initial configuration with $\gamma^0 \in \Gamma$ and
- I a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \to \Gamma$.

$$\mathit{interpreter} : \mathit{program} \times \mathit{config} \to \mathit{config}$$

Sufficiently general to capture at least all (deterministic) languages that can have their semantics expressed as a transition function (e.g. using small-step or big-step semantics)

Note that the interpreter can be applied repeatedly, i.e. that effects can be composed

Observation..!

REPLs with incremental execution implement a language with the following property:

Observation..!

REPLs with incremental execution implement a language with the following property:

A **sequential language** is a language in which $p_1; p_2$ is a (syntactically) valid program iff p_1 and p_2 are valid programs and iff $p_1; p_2$ is equivalent to 'doing' p_1 and then p_2

Observation..!

REPLs with incremental execution implement a language with the following property:

A **sequential language** is a language in which $p_1; p_2$ is a (syntactically) valid program iff p_1 and p_2 are valid programs and iff $p_1; p_2$ is equivalent to 'doing' p_1 and then p_2

$$[\![p_1; p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

Observation..!

REPLs with incremental execution implement a language with the following property:

A **sequential language** is a language in which $p_1; p_2$ is a (syntactically) valid program iff p_1 and p_2 are valid programs and iff $p_1; p_2$ is equivalent to 'doing' p_1 and then p_2

$$[\![p_1; p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

Formally

A language $\langle P, \Gamma, \gamma^0, I \rangle$ is *sequential* if there is an operator $\otimes$ such that for every $p_1, p_2 \in P$ and $\gamma \in \Gamma$ it holds that $p_1 \otimes p_2 \in P$ and that $I_{p_1 \otimes p_2}(\gamma) = (I_{p_2} \circ I_{p_1})(\gamma)$

Idea..! REPLization

Distinguish between REPL language and base language (e.g. JShell vs Java)

Idea..! REPLization

Distinguish between REPL language and base language (e.g. JShell vs Java)
Replization *is: extending a base language to a sequential variant*

Idea..! REPLization

Distinguish between REPL language and base language (e.g. JShell vs Java)
Replization *is: extending a base language to a sequential variant*

A Principled Approach to REPL Interpreters

L. Thomas van Binsbergen
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Mauricio Verano Merino
Eindhoven University of Technology
Eindhoven, The Netherlands
m.verano.merino@tue.nl

Pierre Jeanjean
Inria, University of Rennes, CRNS, IRISA
Rennes, France
pierre.jeanjean@inria.fr

Tijs van der Storm
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
University of Groningen
Groningen, The Netherlands
storm@cwi.nl

Benoit Combemale
University of Rennes, Inria, CNRS, IRISA
Rennes, France
benoit.combemale@irit.fr

Olivier Barais
University of Rennes, Inria, CNRS, IRISA
Rennes, France
olivier.barais@irisa.fr

Figure: Onward!2020

REPLization in Onward!2020

Distinguish between REPL language and base language (e.g. JShell vs Java)
Replization *is: extending a base language to a sequential variant*

REPLization in Onward!2020

Distinguish between REPL language and base language (e.g. JShell vs Java)
Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)

REPLization in Onward!2020

Distinguish between REPL language and base language (e.g. JShell vs Java)
Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)
2. Implement definitional interpreter by choosing Γ and in terms of base interpreter

REPLization in Onward!2020

Distinguish between REPL language and base language (e.g. JShell vs Java)
Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)
2. Implement definitional interpreter by choosing Γ and in terms of base interpreter
3. Add phrase composition operator to the language (it is now sequential by definition)

$$[\![p_1 \otimes p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

REPLization in Onward!2020

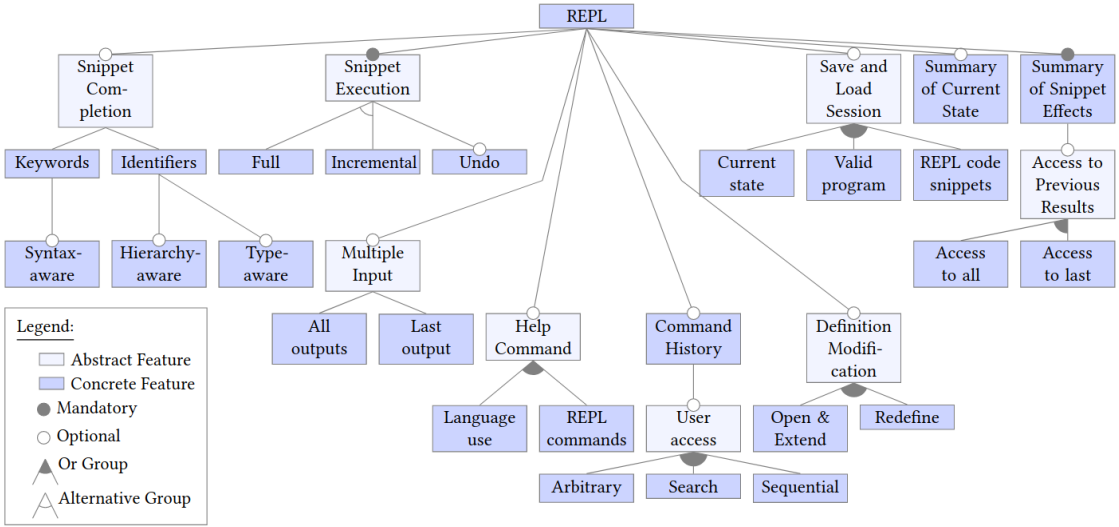
Distinguish between REPL language and base language (e.g. JShell vs Java)
Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)
2. Implement definitional interpreter by choosing Γ and in terms of base interpreter
3. Add phrase composition operator to the language (it is now sequential by definition)

$$[\![p_1 \otimes p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

- The effect of one phrase on the next is determined by its modifications to $\gamma \in \Gamma$

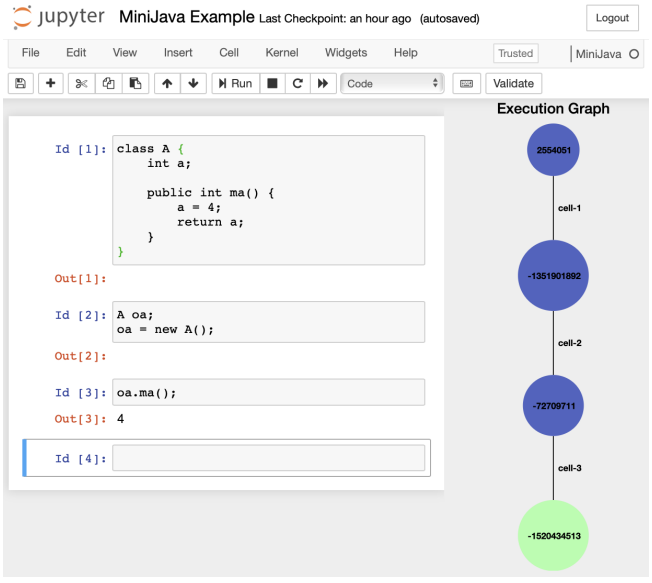
Onward!2020 (feature model)



Onward!2020 (feature table)

		CLing	JShell	Python	IPython	C# REPL	Node.js	PHP	PsySH	SQLite	R	Swift	Gore	Octave	Rappel	iRB
Snippet Execution	Incremental	●	●	●	●	●	●	●	●	●	●	●		●	●	●
	Full												●			
	Undo	●														
Summary of Current State		●	●		●	●			●	●				●	●	
Summary of Snippet Effects		●	●	●	●	●	●		●	●	●	●	●	●	●	●
Access to Previous Results	Access to last			●	●		●		●					●		●
	Access to all		●		●							●				
Multiple Input	Last output	●			●	●	●		●			●		●	●	●
	All outputs		●	●						●	●					
Snippet Completion	Keywords	●		●	●		●			●	●	●		●		●
	Syntax-aware	●			●							●				
	Identifiers	●	●	●	●	●	●	●	●	●	●	●	●	●		●
	Type-aware		●				–				–				–	
	Hierarchy-aware	●	●	●	●	●	●		●		●		●	●	–	
Definition Modification	Redefine		● [1]	●	● [1]	●					● [1]	●	●	● [1]	–	●
	Open & Extend														–	●
Help Command	REPL commands	●	●	●	●	●	●		●	●		●	●		●	
	Language use			●	●						●		●	●		
Command History (User Access)	Sequential	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	Search	●	●	●	●	●	●	●	●	●	●		●	●		●
	Arbitrary		●		●									●		
Save and Load Session	Current state									●	●			○		
	REPL code snippets		●		●		●			○	○					
	Valid programs				●		●			○	○		●			

Onward!2020 (MiniJava case study)




```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

Figure: Interpreter for extended MiniJava

Onward!2020 (QL case study)

```
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
  }
}
```

Figure: QL form

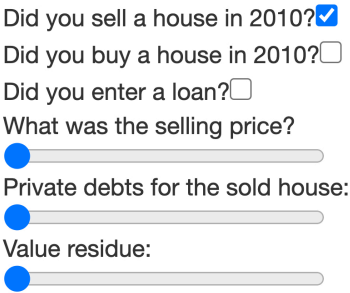


Figure: Rendering

Onward!2020 (eFLINT case study)

```
+seller("Alice")
+buyer("Bob")
+duty-to-deliver(seller("Alice"),buyer("Bob"))
+duty-to-pay(buyer("Bob"),seller("Alice"))
+amount(10)
+pay(buyer("Bob"),seller("Alice"),amount(10))
+asset-id("Meat")
+deliver(seller("Alice"),buyer("Bob"),asset-id("Meat"))
query successful
query successful
#9 > :o
actions & events:
1. deliver(seller("Alice"),buyer("Bob"),asset-id("Meat")) (ENABLED)
2. pay(buyer("Bob"),seller("Alice"),amount(10)) (ENABLED)
3. suspend-delivery(seller("Alice"),buyer("Bob")) (DISABLED)
4. tick() (ENABLED)
#9 > :4
-clock(0)
+clock(1)
#10 > :4
violations:
  violated duty!: duty-to-pay(buyer("Bob"),seller("Alice"))
-clock(1)
+clock(2)
+suspend-delivery(seller("Alice"),buyer("Bob"))
#11 > suspend-delivery(Alice,Bob)
violations:
  violated duty!: duty-to-pay(buyer("Bob"),seller("Alice"))
#12 > :revert 9
#9 > suspend-delivery(Alice,Bob)
not a compliant action
#9 >
```

Figure: eFLINT command-line REPL

Figure: eFLINT web-interface

Idea..!

Idea..! REPL-first languages

REPL-first is:

Designing and implementing your language as a sequential language from the get-go

Hypothesis

The iterative execution of the definitional interpreter of a sequential language is the essential **building block** of *all* language services related to interpretation

Idea..! REPL-first languages

REPL-first is:

> *Designing and implementing your language as a sequential language from the get-go*

Hypothesis

The iterative execution of the definitional interpreter of a sequential language is the essential **building block** of *all* language services related to interpretation

- Command-line REPLs, notebooks, and servers (Onward!2020)

Idea..! REPL-first languages

REPL-first is:

Designing and implementing your language as a sequential language from the get-go

Hypothesis

The iterative execution of the definitional interpreter of a sequential language is the essential **building block** of *all* language services related to interpretation

- Command-line REPLs, notebooks, and servers (Onward!2020)
- Exploring interpreter as a bookkeeping device on top of definitional interpreter
  - Enables generic back-end for **exploratory programming** (TFP2021)
  - Back-in-time (omnisicient) **debugging**

Idea..! REPL-first languages

REPL-first is:

> *Designing and implementing your language as a sequential language from the get-go*

Hypothesis

The iterative execution of the definitional interpreter of a sequential language is the essential **building block** of *all* language services related to interpretation

- Command-line REPLs, notebooks, and servers (Onward!2020)
- Exploring interpreter as a bookkeeping device on top of definitional interpreter
  - Enables generic back-end for **exploratory programming** (TFP2021)
  - Back-in-time (omnisicient) **debugging**
- Delta-operations as phrases to support **live programming**

Idea..! REPL-first languages

REPL-first is:

Designing and implementing your language as a sequential language from the get-go

Hypothesis

The iterative execution of the definitional interpreter of a sequential language is the essential **building block** of *all* language services related to interpretation

- Command-line REPLs, notebooks, and servers (Onward!2020)
- Exploring interpreter as a bookkeeping device on top of definitional interpreter
  - Enables generic back-end for **exploratory programming** (TFP2021)
  - Back-in-time (omnisicient) **debugging**
- Delta-operations as phrases to support **live programming**
- Fluid, bidirectional moves between GUI-actions and code for GUI-interfaces[1]

[1] *mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks.* Mary Beth Kery et al.

Exploring interpreter algorithm

The *reachability graph* for a configuration $\gamma \in \Gamma$ of a language $\langle P, \Gamma, \gamma^0, I \rangle$ contains all the configurations γ' that are reachable by executing programs $p \in P$ using I. Nodes are configurations, edges are labelled with programs

Exploring interpreter algorithm

The *reachability graph* for a configuration $\gamma \in \Gamma$ of a language $\langle P, \Gamma, \gamma^0, I \rangle$ contains all the configurations γ' that are reachable by executing programs $p \in P$ using I. Nodes are configurations, edges are labelled with programs

An *exploring interpreter* for a language $\langle P, \Gamma, \gamma^0, I \rangle$ is an algorithm constructing a subgraph of the reachability graph from γ^0 by performing one of the following actions:

Exploring interpreter algorithm

The *reachability graph* for a configuration $\gamma \in \Gamma$ of a language $\langle P, \Gamma, \gamma^0, I \rangle$ contains all the configurations γ' that are reachable by executing programs $p \in P$ using I. Nodes are configurations, edges are labelled with programs

An *exploring interpreter* for a language $\langle P, \Gamma, \gamma^0, I \rangle$ is an algorithm constructing a subgraph of the reachability graph from γ^0 by performing one of the following actions:

Algorithm

- **execute**(p): take $\gamma' = I_p(\gamma)$ and (p given as input, γ current configuration):
  - add γ' to the set of nodes (if new), and
  - add $\langle \gamma, p, \gamma' \rangle$ to the set of edges (if new).
- **revert**(γ): take γ as the current configuration (with γ given as input and in the graph).
- **display**: produce a structured representation of the current graph, distinguishing the current configuration in the graph from the other configurations.

REPL-first languages

L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
ltvanbinsbergen@acm.org

March 10, 2021