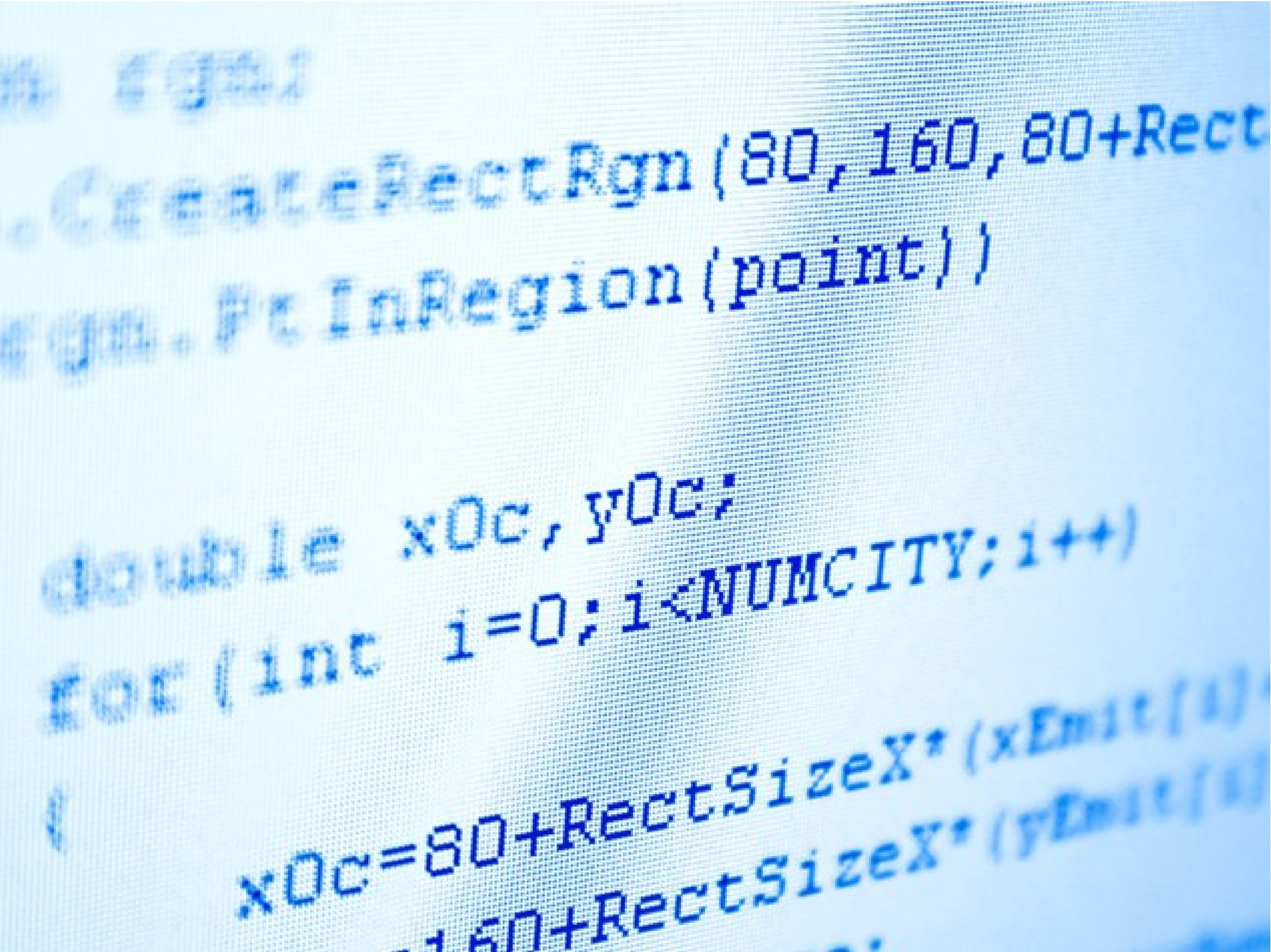
# Towards Modular Computer Language Components

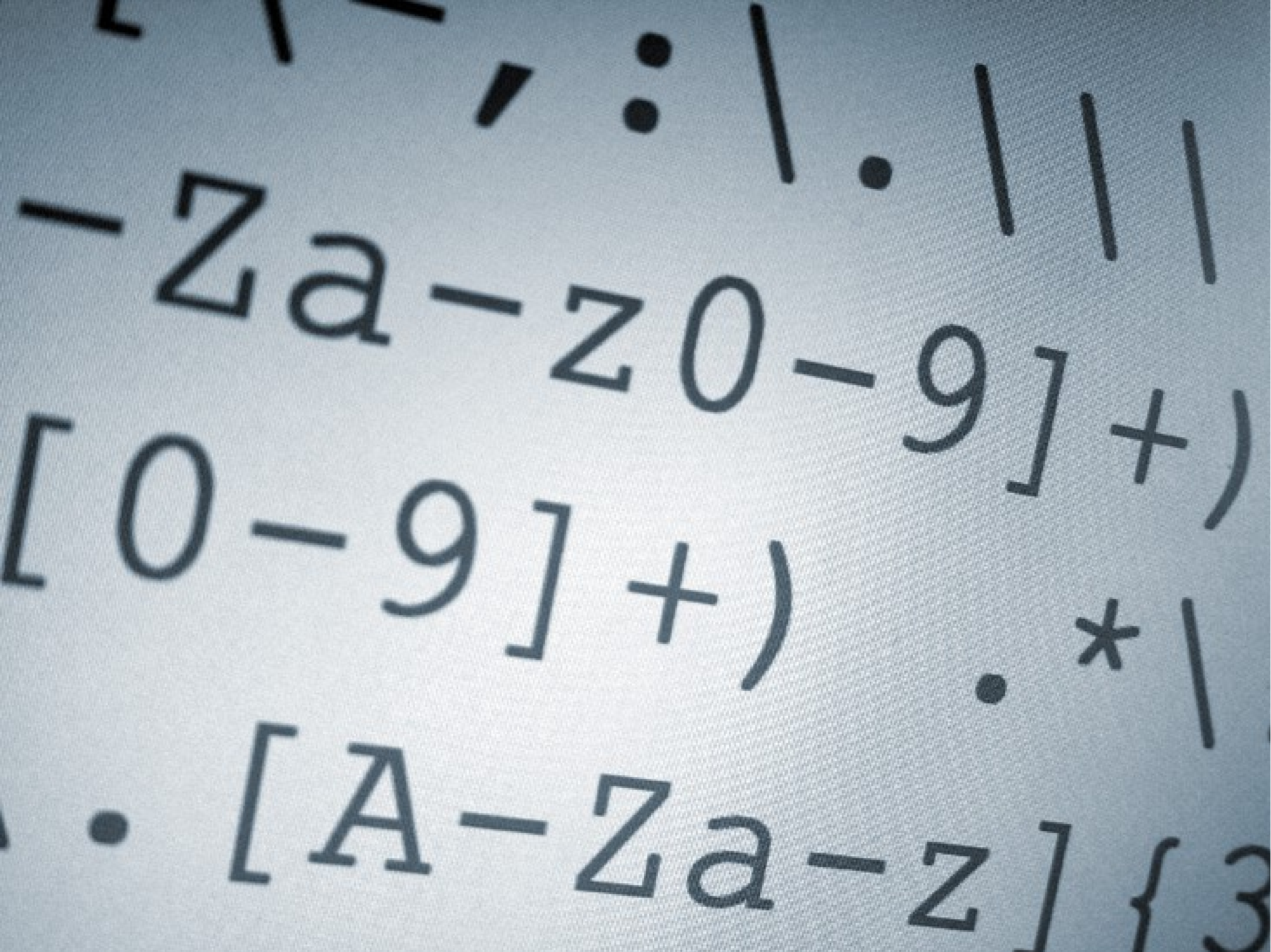
**Tillmann Rendel**
*University of Tübingen*

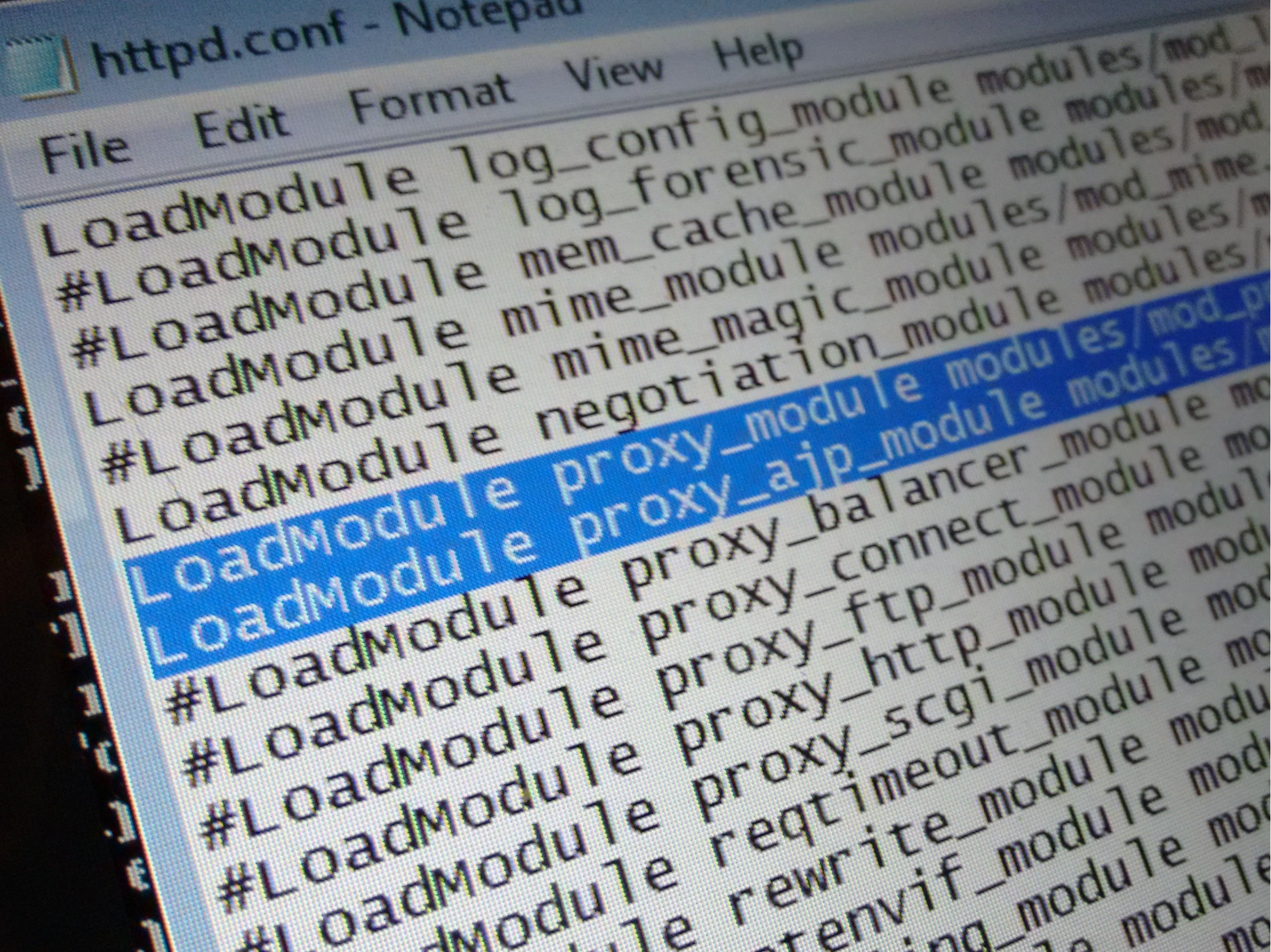
Presentation at the colloquium of the Oregon State University's
School of Electrical Engineering and Computer Science
Corvallis, October 27, 2014

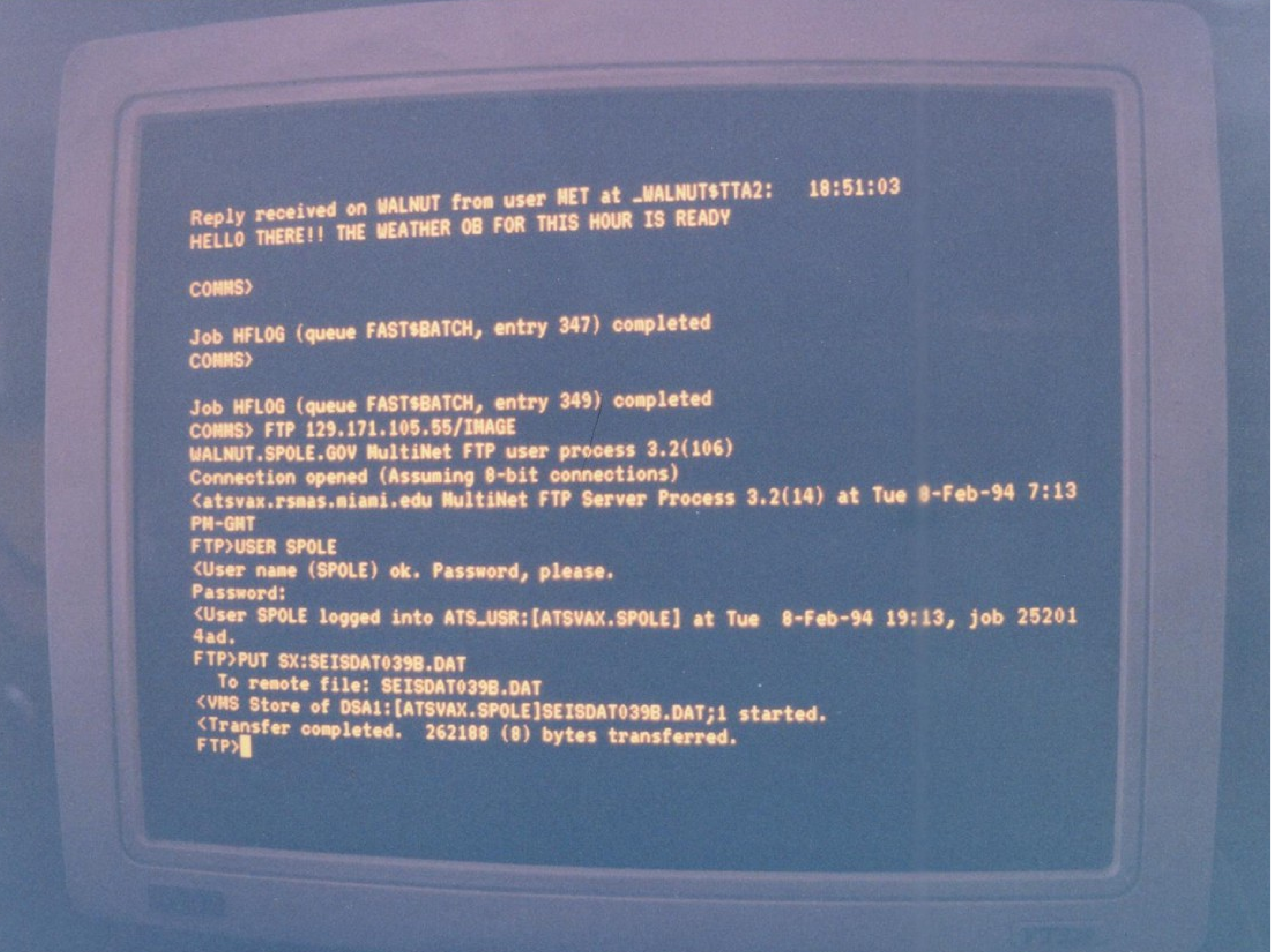

Lang·try (lăng
1929. British a
affair with Edw
lan·guage (lă
beings of voice
these sounds

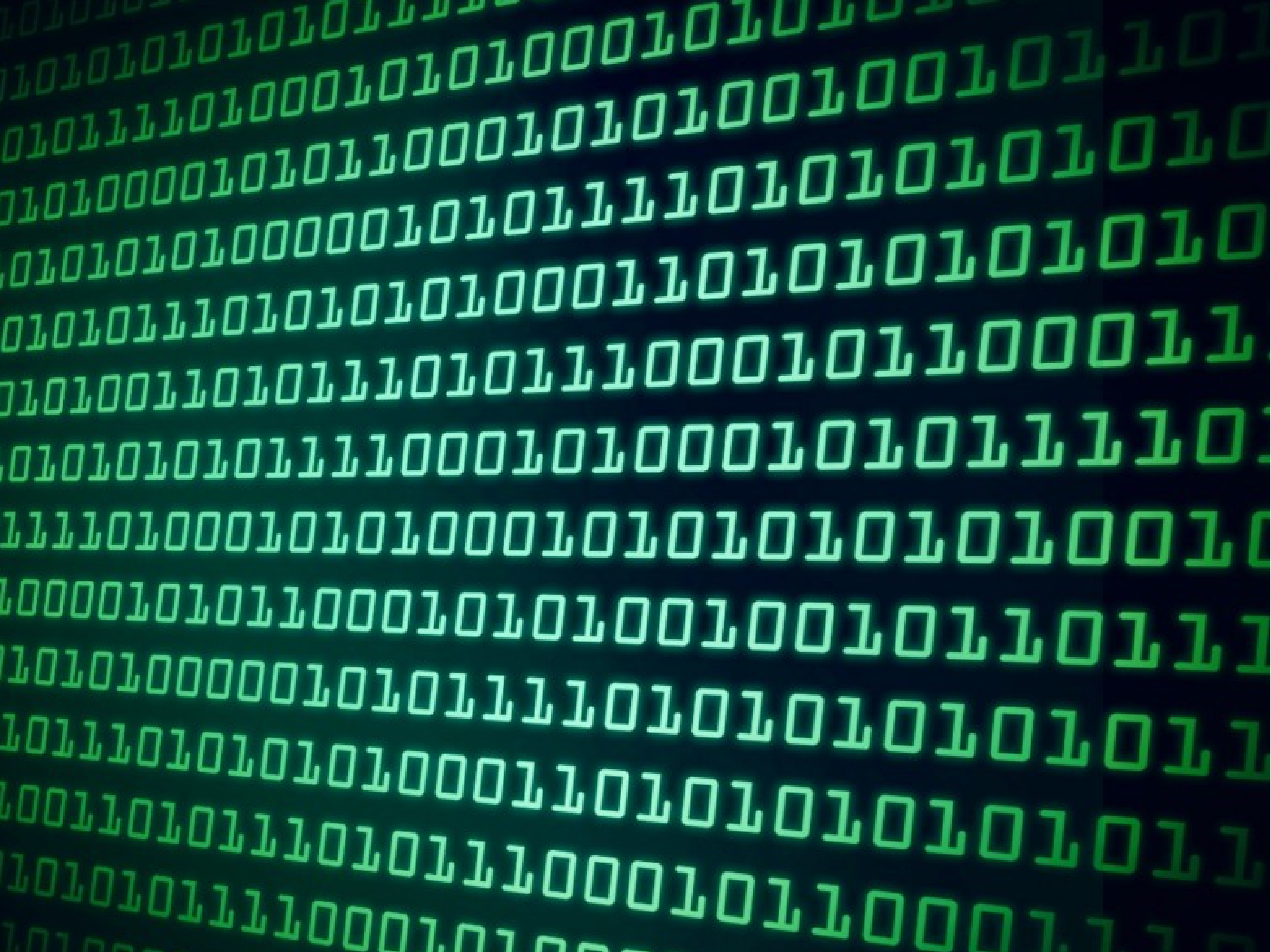

CreateRectRgn(80,160,80+Rect
InRegion(point))
xOc,yOc;
i=0;i<NUMCITY;i++)
xOc=80+RectSizeX*
160+RectSizeX*


<!DOCTYPE HTML
<html>
<head>
<meta name="TITLE"
<meta name="
<meta name="DESCRIPTION"
<link rel="stylesheet"
<script language="
</head>
<body bgcolor="
width="

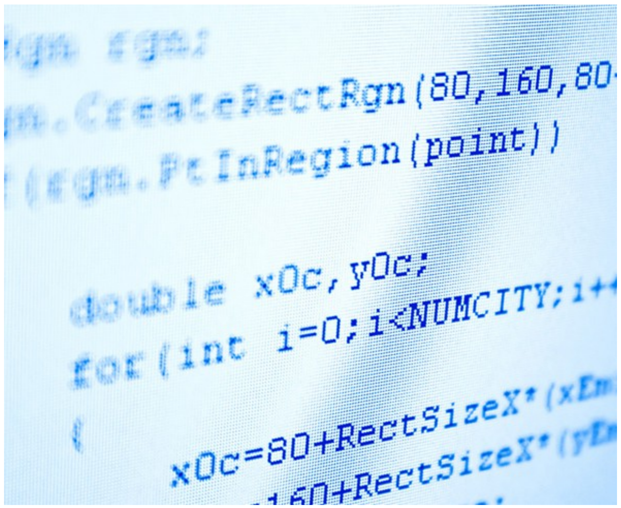

-Za-z0-9]+)
[0-9]+) .*\
.[A-Za-z]{3

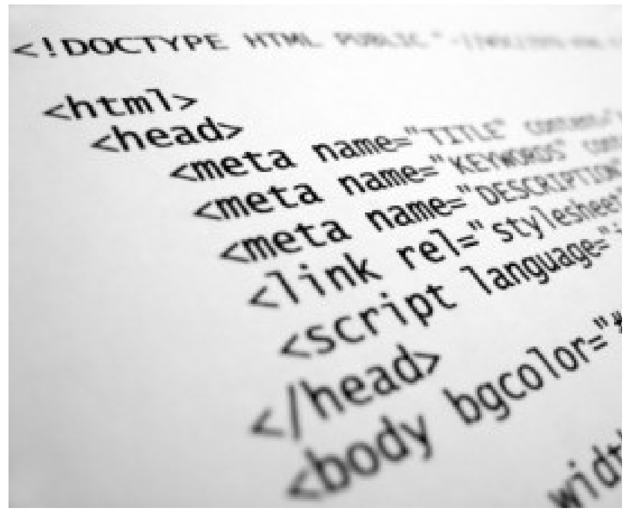

httpd.conf - Notepad
File Edit Format View Help
LoadModule log_config_module
#LoadModule log_forensic_module
#LoadModule mem_cache_module
LoadModule mime_module
#LoadModule mime_magic_module
LoadModule negotiation_module
LoadModule proxy_module
LoadModule proxy_ajp_module
#LoadModule proxy_balancer_module
#LoadModule proxy_connect_module
#LoadModule proxy_ftp_module
#LoadModule proxy_http_module
#LoadModule proxy_scgi_module
#LoadModule reqtimeout_module
#LoadModule rewrite_module

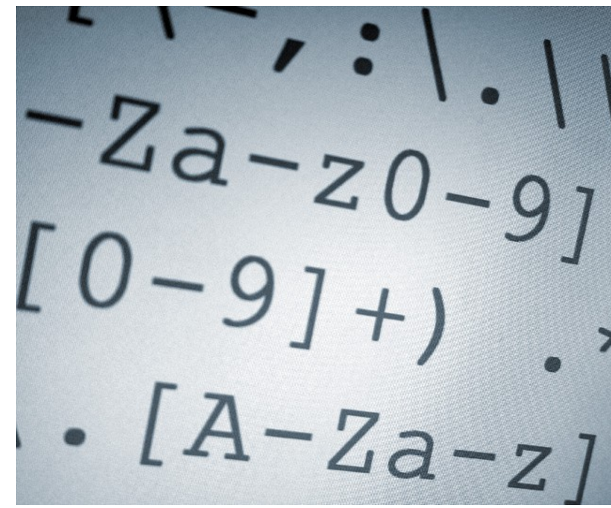

Reply received on WALNUT from user NET at _WALNUT$TTA2: 18:51:03
HELLO THERE!! THE WEATHER OB FOR THIS HOUR IS READY
COMMS>
Job HFLOG (queue FAST$BATCH, entry 347) completed
COMMS>
Job HFLOG (queue FAST$BATCH, entry 349) completed
COMMS> FTP 129.171.105.55/IMAGE
WALNUT.SPOLE.GOV MultiNet FTP user process 3.2(106)
Connection opened (Assuming 8-bit connections)
<atsvax.rsmas.miami.edu MultiNet FTP Server Process 3.2(14) at Tue 8-Feb-94 7:13 PM-GMT
FTP>USER SPOLE
<User name (SPOLE) ok. Password, please.
Password:
<User SPOLE logged into ATS_USR:[ATSVAX.SPOLE] at Tue 8-Feb-94 19:13, job 252014ad.
FTP>PUT SX:SEISDAT039B.DAT
To remote file: SEISDAT039B.DAT
<VMS Store of DSA1:[ATSVAX.SPOLE]SEISDAT039B.DAT;1 started.
<Transfer completed. 262188 (8) bytes transferred.
FTP>

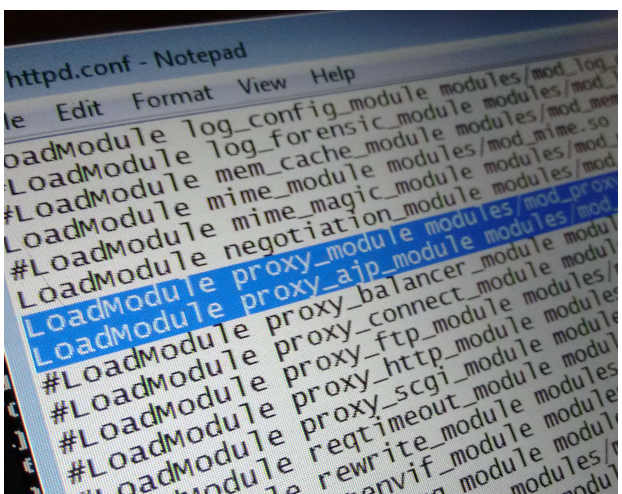


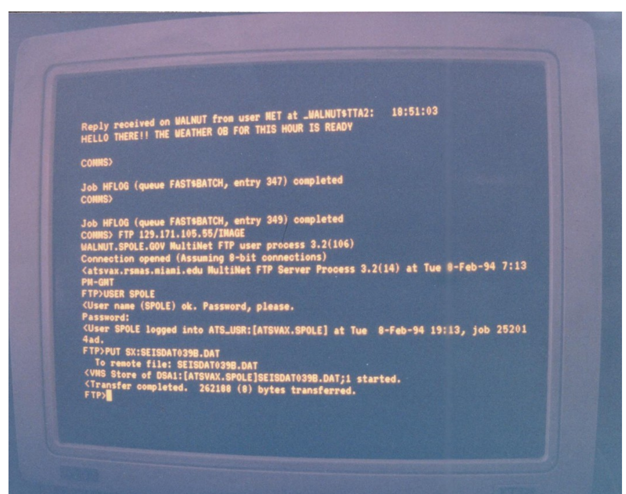
# Computer Languages

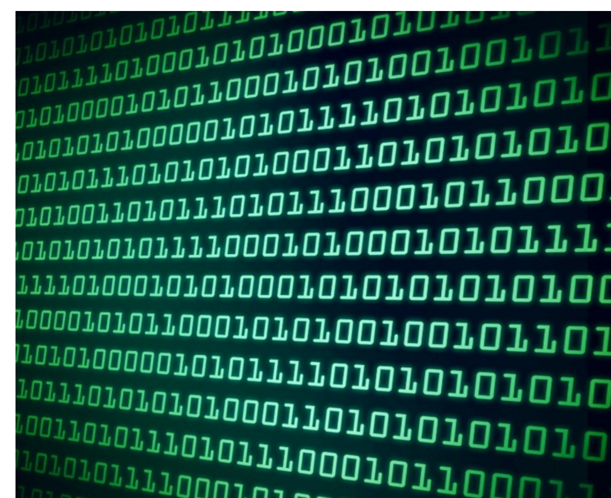




# Economy of Computer Languages

- Language maker invests effort into language design and implementation.
- Language user invests effort into language learning and use.
- Language user benefits from language use.
- Return on investment?

# Programming Language (PL)

- Languages for general-purpose computing
- Distill computing paradigm into a PL
- Invest effort into PL design and implementation
- Reuse PL for many software projects
- Use by programmers

- Language author:
  team of language engineers

# Domain-Specific Languages (DSL)

- Languages for just one application domain
- Distill domain knowledge into the DSL
- Invest effort into DSL design & implementation
- Reuse DSL for many programs in the domain
- Use by domain experts (non-programmers)

- Language author:
  domain expert + language engineer

# Language-Oriented Programming

- Languages for just one software project
- Express component interface as language
- Invest effort into component design & impl.
- Reuse DSL for many clients of the component
- Use by other team members (programmers)

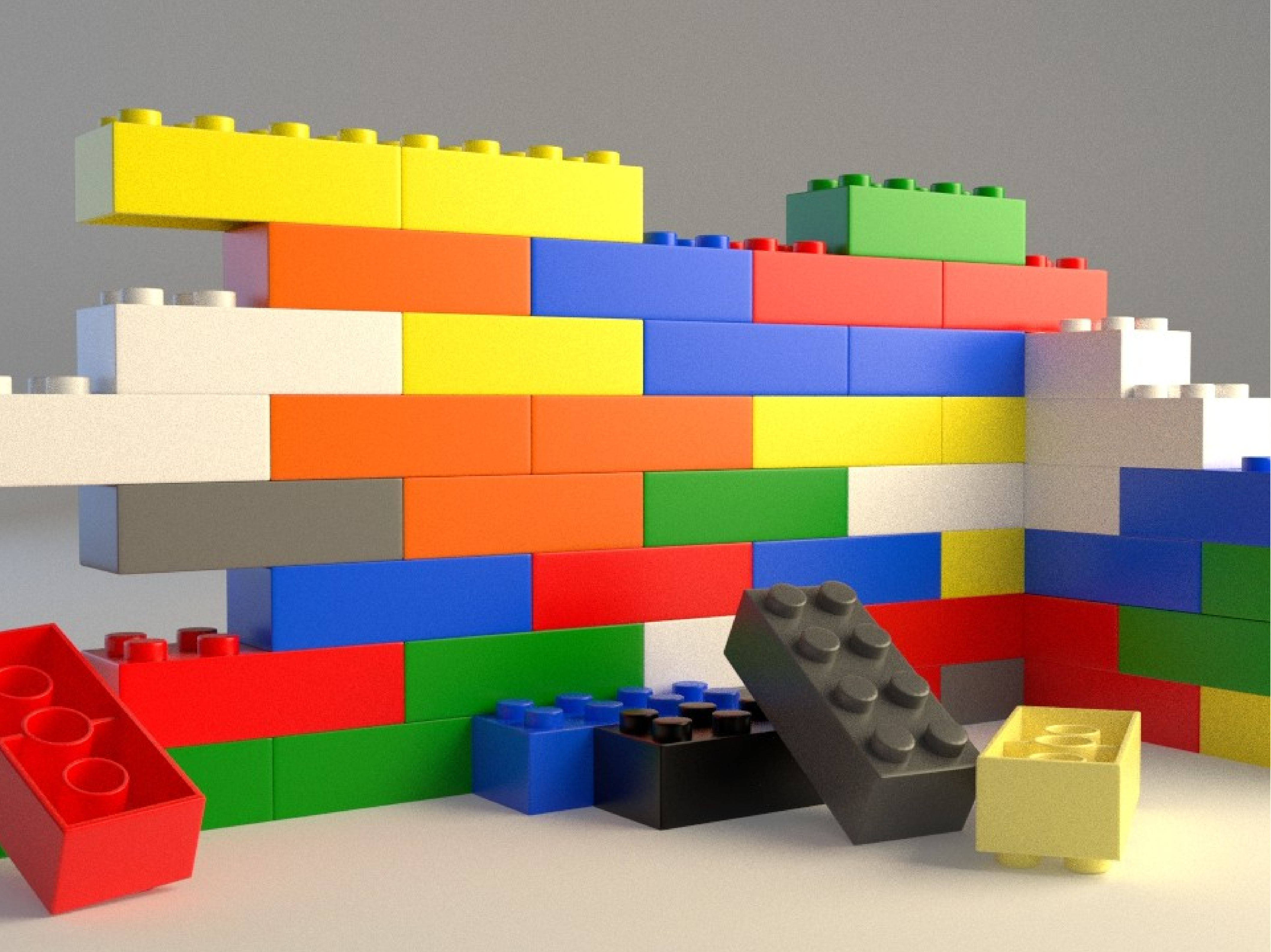
- Language author:
  software engineer

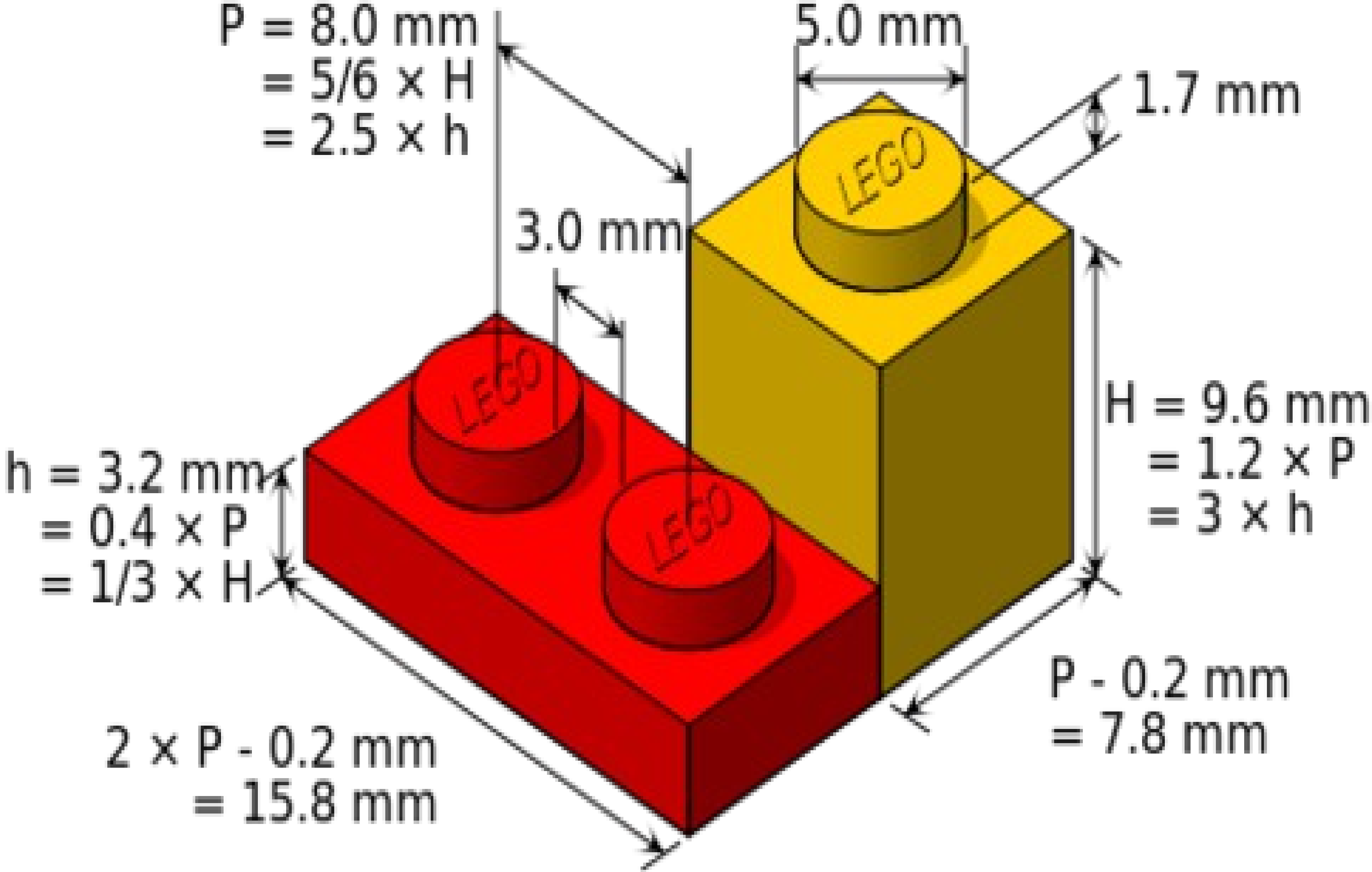
# How to invest less and gain more?

# Reusable Language Components

- Can languages be reusable components?

- Reuse whole languages inside another language?
- Reuse fragments of a language?
- Build a new language from bits and pieces?
- Reuse language design concepts?
- Reuse language implementation artifacts?
- Reuse language ecosystems?




P = 8.0 mm
= 5/6 × H
= 2.5 × h
5.0 mm
1.7 mm
3.0 mm
LEGO
LEGO
LEGO
H = 9.6 mm
= 1.2 × P
= 3 × h
h = 3.2 mm
= 0.4 × P
= 1/3 × H
P - 0.2 mm
= 7.8 mm
2 × P - 0.2 mm
= 15.8 mm

# What is in a Language?

- Syntax
  *(What are the programs?)*
- Static Semantics
  *(Which programs are legal?)*
- Dynamic Semantics
  *(What do the programs mean?)*
- Editors
  *(How to write the programs?)*
- Interpreters and Compilers
  *(How to run the programs?)*
- Development Tools
  *(How to interact with a program?)*
- Ecosystem
  *(Which other programs are there?)*

# What is in a Language?

- Syntax
  *(What are the programs?)*
- Static Semantics
  *(Which programs are legal?)*
- Dynamic Semantics
  *(What do the programs mean?)*
- Editors
  *(How to write the programs?)*
- Interpreters and Compilers
  *(How to run the programs?)*
- Development Tools
  *(How to interact with a program?)*
- Ecosystem
  *(Which other programs are there?)*

# What is in a Language?

- Syntax
  *(What are the programs?)*
- Static Semantics
  *(Which programs are legal?)*
- Dynamic Semantics
  *(What do the programs mean?)*
- Editors
  *(How to write the programs?)*
- Interpreters and Compilers
  *(How to run the programs?)*
- Development Tools
  *(How to interact with a program?)*
- Ecosystem
  *(Which other programs are there?)*

# What is in a Language?

- Syntax
  *(What are the programs?)*
- Static Semantics
  *(Which programs are legal?)*
- Dynamic Semantics
  *(What do the programs mean?)*
- Editors
  *(How to write the programs?)*
- Interpreters and Compilers
  *(How to run the programs?)*
- Development Tools
  *(How to interact with a program?)*
- Ecosystem
  *(Which other programs are there?)*

# What is in a Language?

- Syntax
  *(What are the programs?)*
- Static Semantics
  *(Which programs are legal?)*
- Dynamic Semantics
  *(What do the programs mean?)*
- Editors
  *(How to write the programs?)*
- Interpreters and Compilers
  *(How to run the programs?)*
- Development Tools
  *(How to interact with a program?)*
- Ecosystem
  *(Which other programs are there?)*

# What is in a Language?

- Syntax
  *(What are the programs?)*
- Static Semantics
  *(Which programs are legal?)*
- Dynamic Semantics
  *(What do the programs mean?)*
- Editors
  *(How to write the programs?)*
- Interpreters and Compilers
  *(How to run the programs?)*
- Development Tools
  *(How to interact with a program?)*
- Ecosystem
  *(Which other programs are there?)*

# What is in a Language?

- Syntax
  *(What are the programs?)*
- Static Semantics
  *(Which programs are legal?)*
- Dynamic Semantics
  *(What do the programs mean?)*
- Editors
  *(How to write the programs?)*
- Interpreters and Compilers
  *(How to run the programs?)*
- Development Tools
  *(How to interact with a program?)*
- Ecosystem
  *(Which other programs are there?)*

# What is in a Language?

- Syntax
  *(What are the programs?)*
- Static Semantics
  *(Which programs are legal?)*
- Dynamic Semantics
  *(What do the programs mean?)*
- Editors
  *(How to write the programs?)*
- Interpreters and Compilers
  *(How to run the programs?)*
- Development Tools
  *(How to interact with a program?)*
- Ecosystem
  *(Which other programs are there?)*

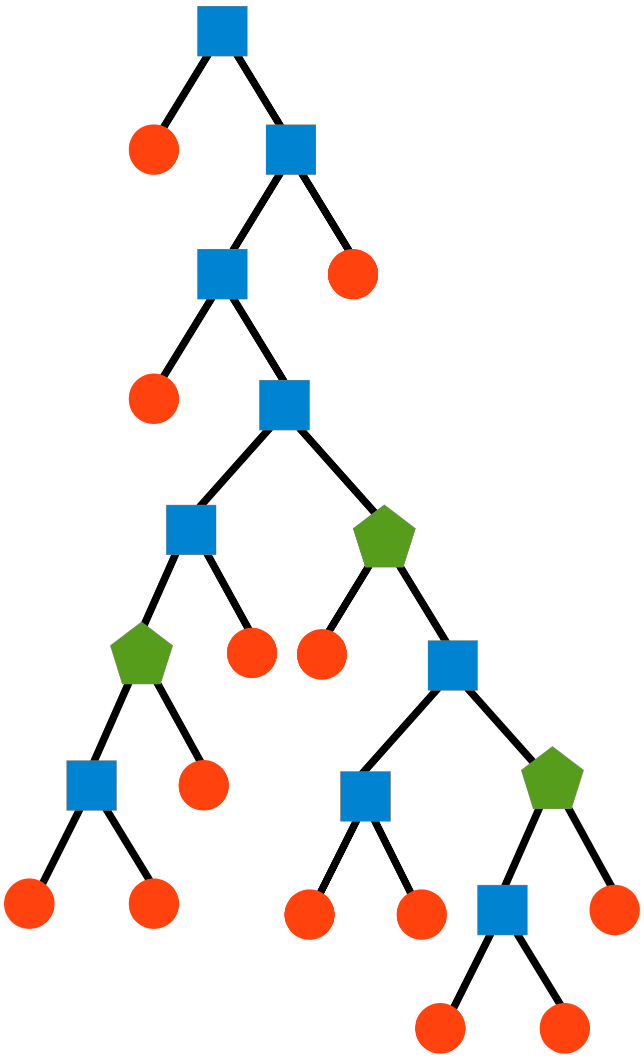
# How is a Language Structured?

# How is a Language Structured?

- strings
- pictures
- abstract syntax trees

# How is a Language Structured?

- strings
- pictures
- abstract syntax trees



# What is Language Composition?

Sebastian Erdweg, Paolo G. Giarrusso, Tillmann Rendel.
**Language Composition Untangled.**
In *Proceedings of Workshop on Language Descriptions, Tools, and Applications*, 2012.

# Language Composition Untangled

- Extension
  *(can extend a language unchanged)*
- Unification
  *(can merge two languages unchanged)*
- Self-Extension
  *(can implement language extension in the language itself)*
- Incremental Extension
  *(can extend extensions)*
- Extension Unification
  *(can unify extensions)*

Erdweg et al. (2012)

# How to Compose Syntax?

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, Klaus Ostermann.
**SugarJ: Library-based Syntactic Language Extensibility.**
In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages & Applications*, 2011.

# SugarJ

```
import regexp.RegExp;
import pairs.Pair;

public class Test {
    RegExp r = /(a|b)c/;
    String s = “hello“;
    (String, Int) pair = (“answer“, 42)
    (RegExp, String) pair = (/ab*/, “text“);
}
```

Erdweg et al. (2011)

# SugarJ

```
import regexp.RegExp;
import pairs.Pair;

public class Test {
    RegExp r = /(a|b)c/;
    String s = “hello“;
    (String, Int) pair = (“answer“, 42)
    (RegExp, String) pair = (/ab*/, “text“);
}
```

Erdweg et al. (2011)

# SugarJ

```
import regexp.RegExp;
import pairs.Pair;

public class Test {
    RegExp r = /(a|b)c/;
    String s = “hello“;
    (String, Int) pair = (“answer“, 42)
    (RegExp, String) pair = (/ab*/, “text“);
}
```

*imports extend the language*

Erdweg et al. (2011)

# SugarJ

```
import regexp.RegExp;
import pairs.Pair;

public class Test {
    RegExp r = /(a|b)c/;
    String s = "hello";
    (String, Int) pair = ("answer", 42)
    (RegExp, String) pair = (/ab*/, "text");
}
```

*another import*

Erdweg et al. (2011)

# SugarJ

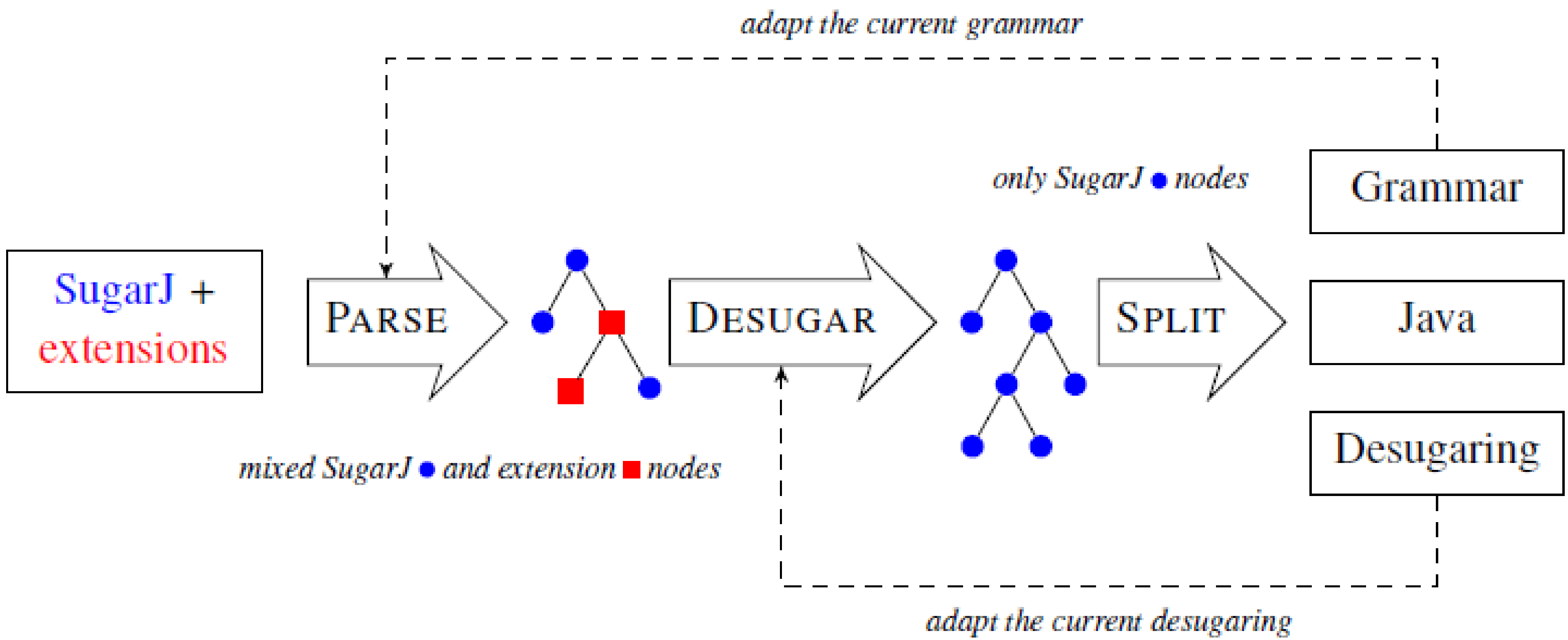
**import** regexp.RegExp;
**import** pairs.Pair;

**public class** Test {
    RegExp r = /(a|b)c/;
    String s = “hello“;
    (String, Int) pair = (“answer“, 42);
    (RegExp, String) pair = $\boxed{\text{(/ab*/, “text“);}}$
}

*language extensions compose*

Erdweg et al. (2011)

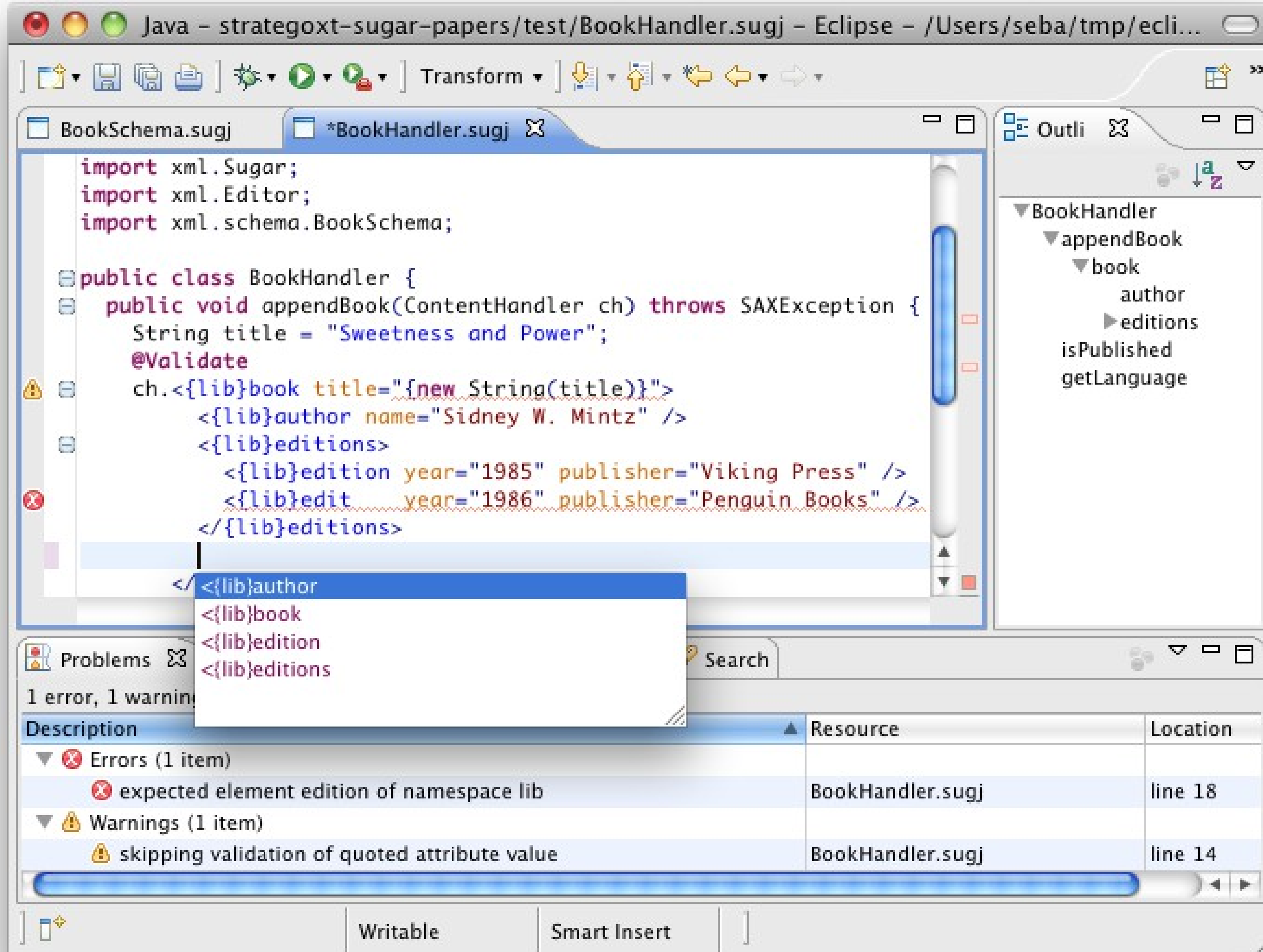
# SugarJ Implementation




Erdweg et al. (2011)

# How to Compose Editors?

Sebastian Erdweg, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, Eelco Visser. **Growing a Language Environment with Editor Libraries.** In *Proceedings of Conference on Generative Programming and Component Engineering*, 2011.




Erdweg et al. (2011)

# How to Compose Interpreters?

Christian Hofer, Klaus Ostermann, Tillmann Rendel, Adriaan Moors.
**Polymorphic Embedding of DSLs.**
In *Proceedings of Conference on Generative Programming and Component Engineering*, 2008.

# How to Compose Compilers?

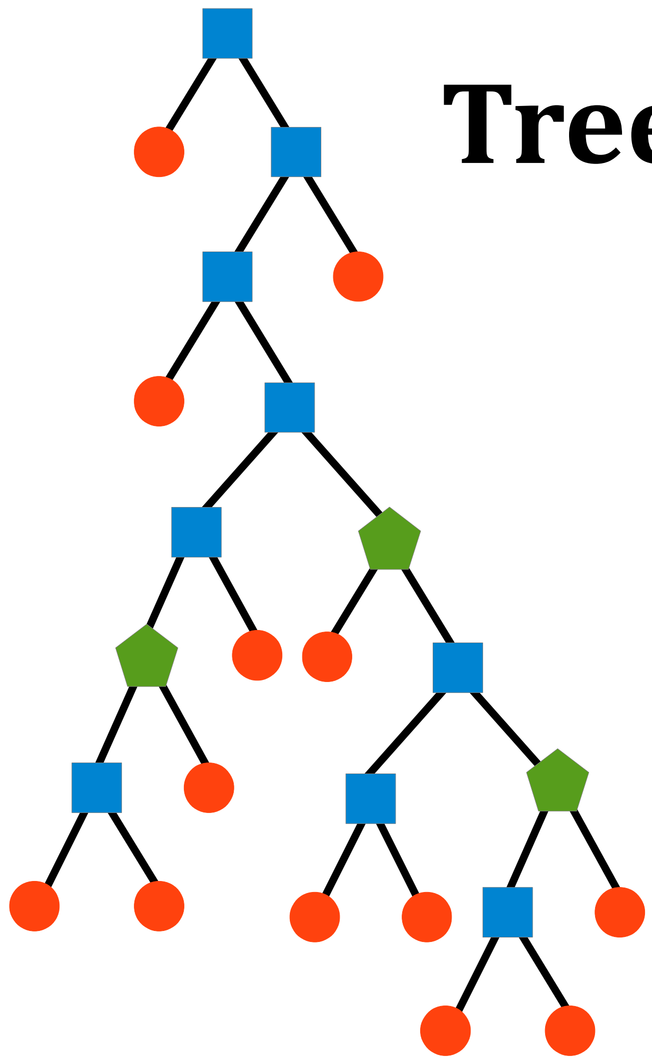
- Macro Systems
- Extensible Compilers
- Attribute Grammars

Tillmann Rendel, Jonathan Brachthäuser, Klaus Ostermann.
**From Object Algebras to Attribute Grammars.**
In *Proceedings of Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

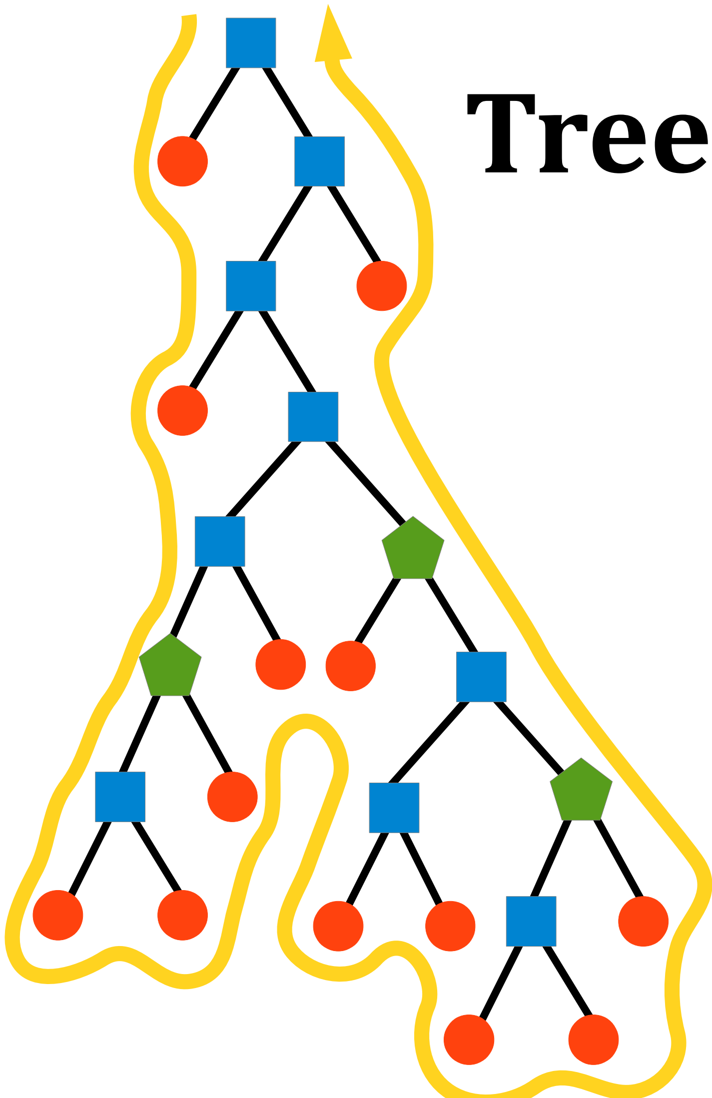
# Tree Traversals

Rendel et al. (2014)

# Tree Traversals

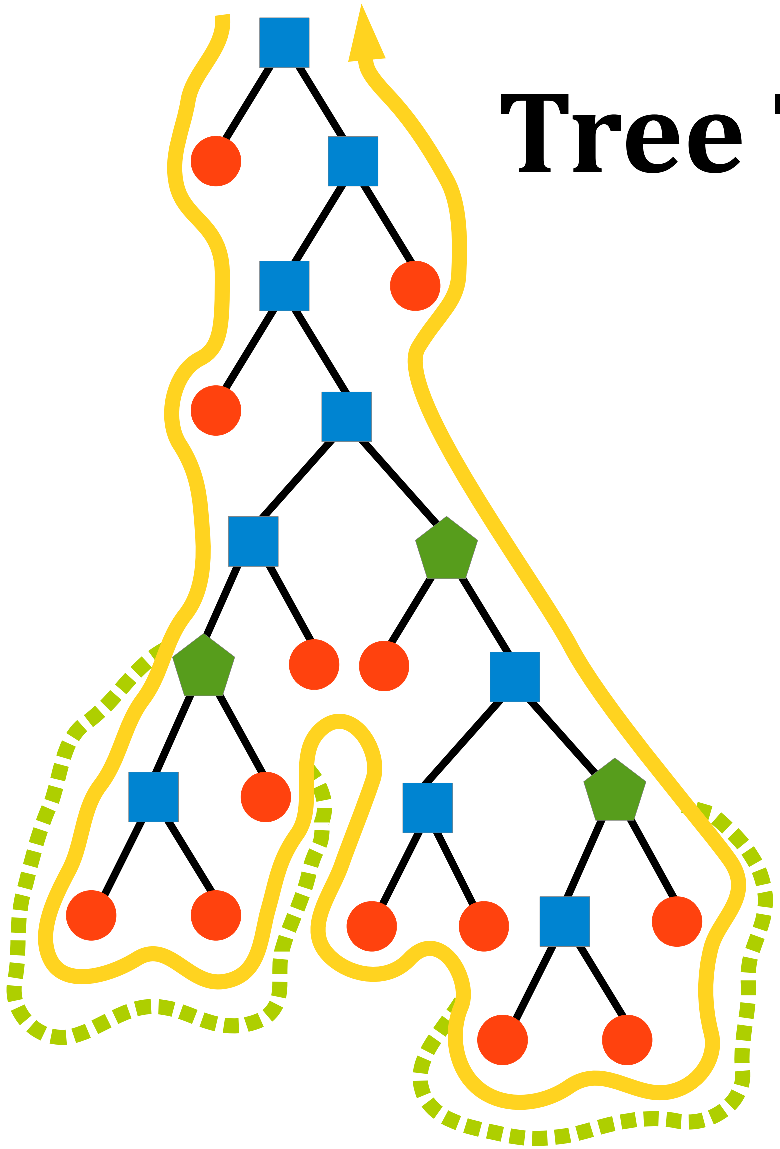


Rendel et al. (2014)



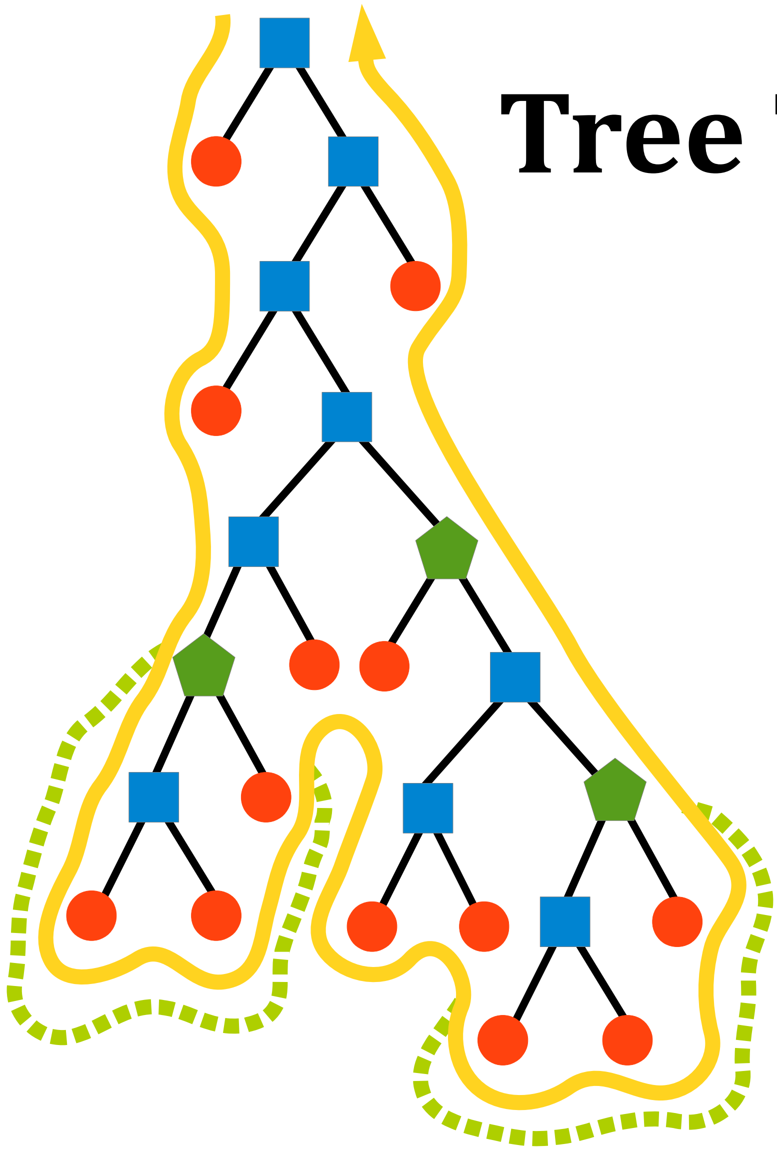
# Tree Traversals

Rendel et al. (2014)



# Tree Traversals

Rendel et al. (2014)



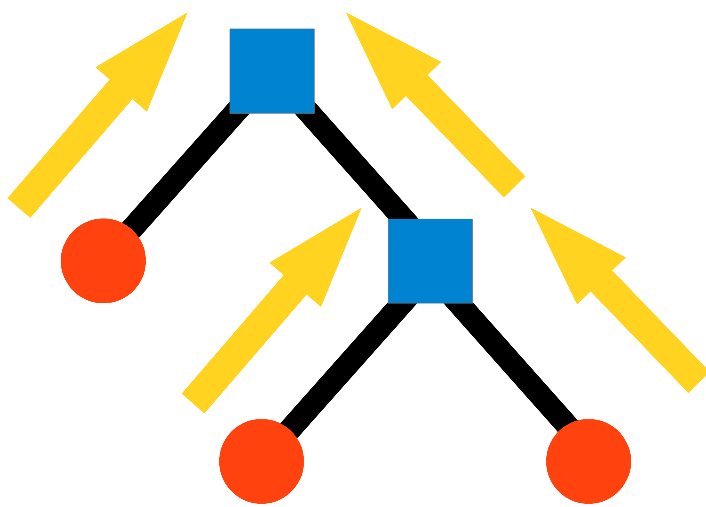
# Tree Traversals

*Represent tree traversals as first-class reusable components (in Scala)!*

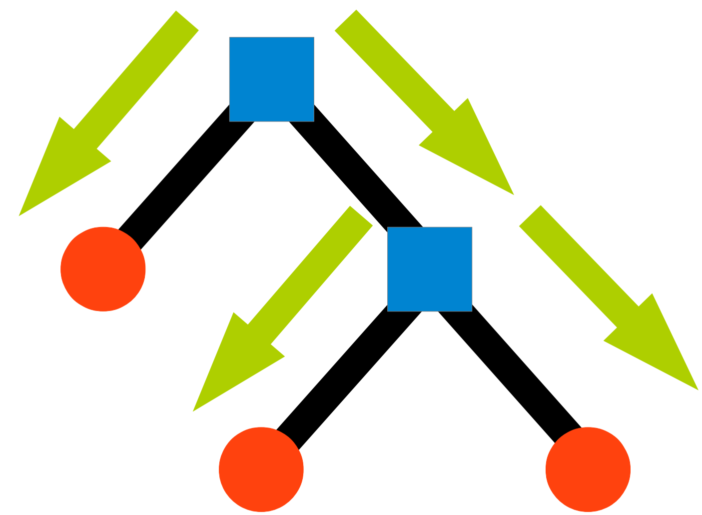
Rendel et al. (2014)

# Traversal Components

Bottom-Up Dataflow



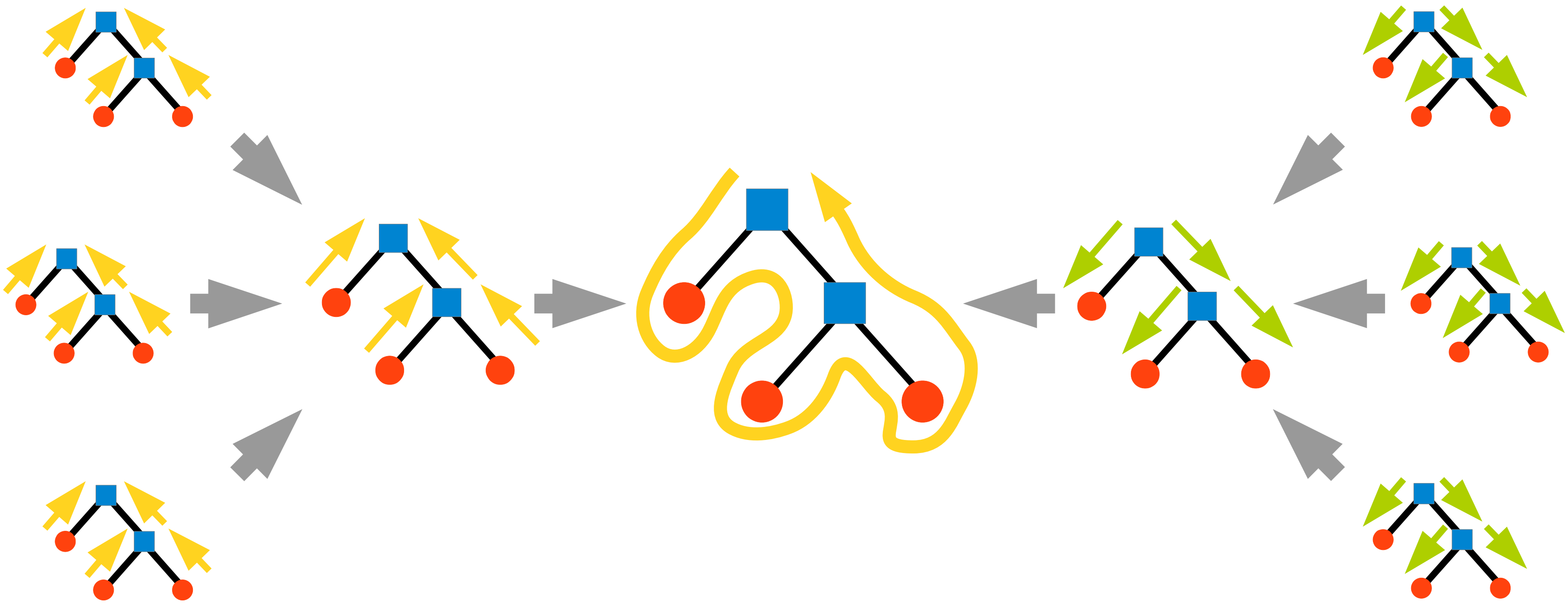
Top-Down Dataflow



- First-class Scala values
- Dependencies checked by Scala type system

Rendel et al. (2014)

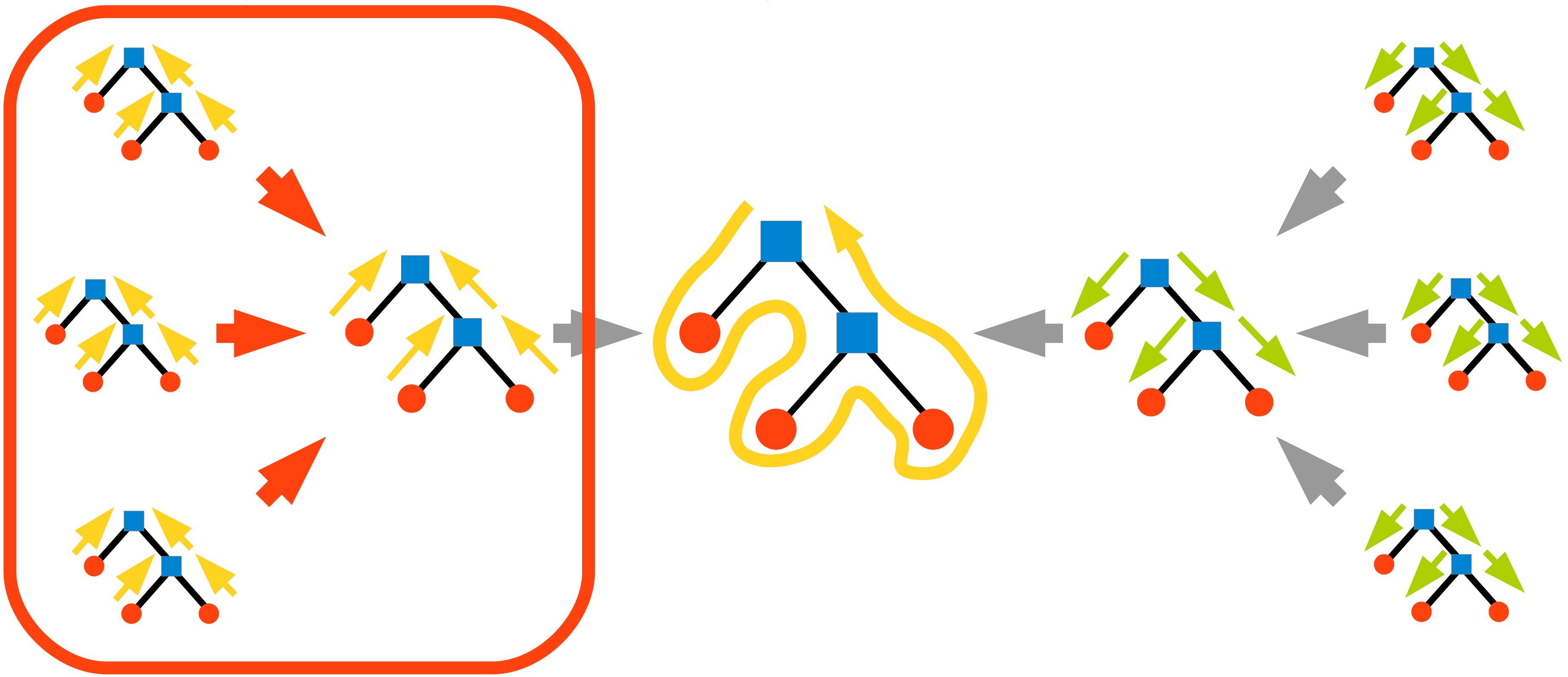
# Traversal Composition



Rendel et al. (2014)

# Traversal Composition

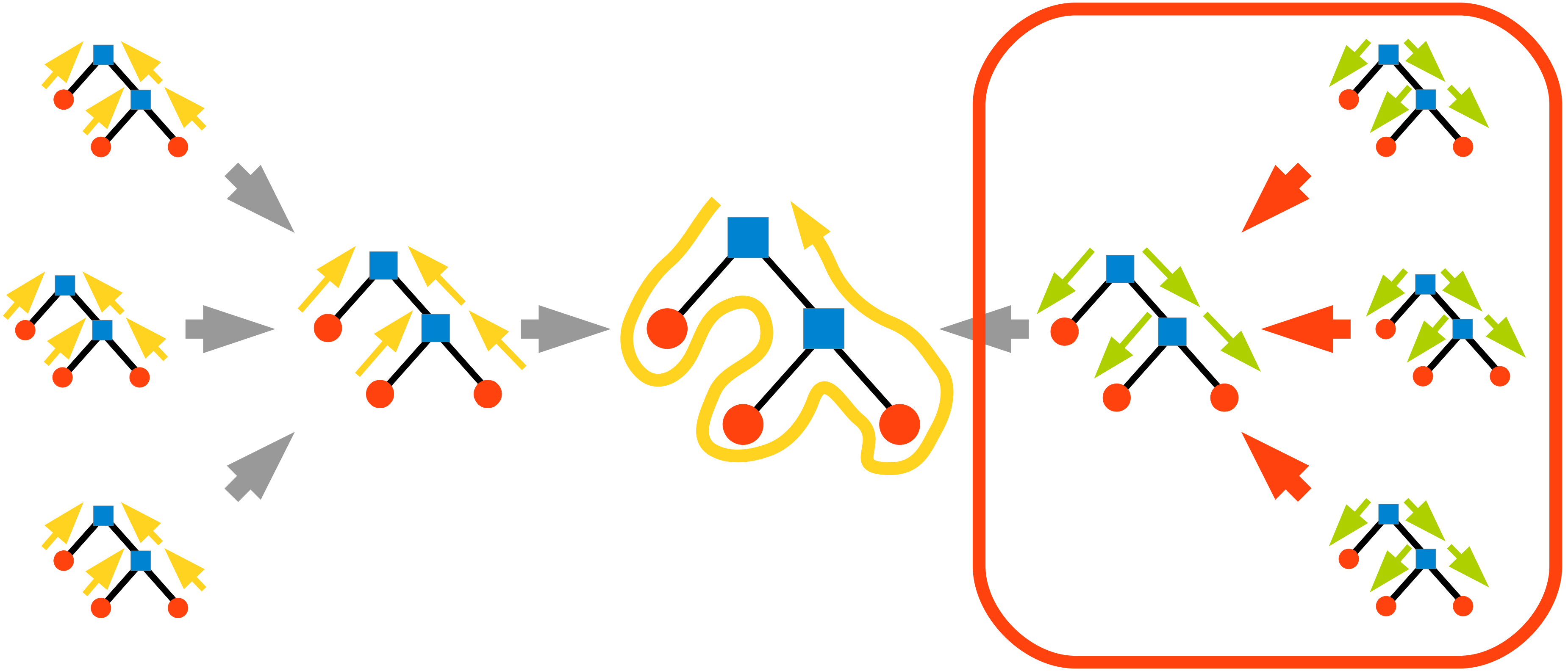
*compose all bottom-up traversals*



Rendel et al. (2014)

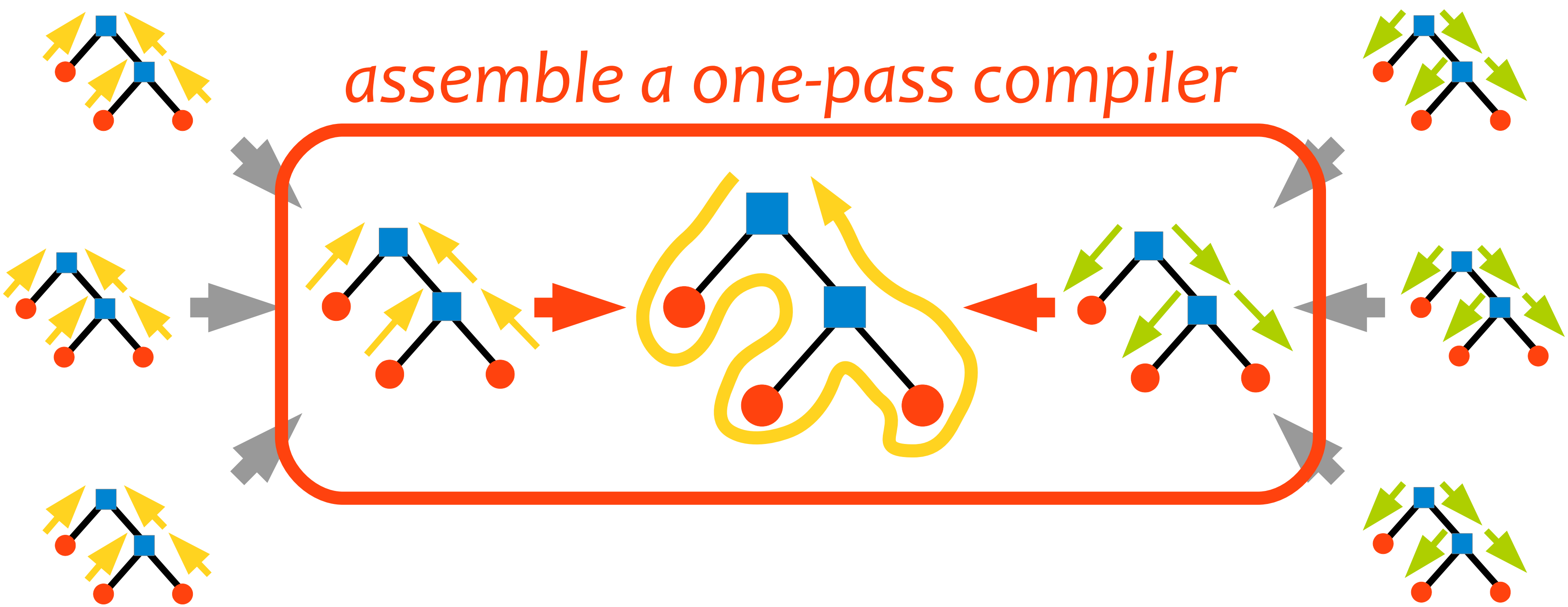
# Traversal Composition

*compose all top-down traversals*



Rendel et al. (2014)

# Traversal Composition




Rendel et al. (2014)

# Monolithic compiler

1 file
807 lines of Java code
entangled

```java
static void parseFunction(Map funcs, Map prototypes) {
  Map vars = new HashMap();
  String name;
  checkToken(tINT);
  skipToken(tID);
  name = tIdValue;
  skipToken(tLPAR);
  int args = parseFormals(vars);
  skipToken(tRPAR);
  skip();
  if (tKind==tSEMI) {
    nextToken();
    if (prototypes.containsKey(name))
      compileError("duplicate declaration of "+name);
    if (funcs.containsKey(name) && args!=funcs.get(name))
      compileError("conflicting declaration of "+name);
    prototypes.put(name,args);
  } else {
    if (funcs.containsKey(name))
      compileError("duplicate implementation of "+name);
    if (prototypes.containsKey(name) && args!=prototypes.get(name))
      compileError("conflicting implementation of "+name);
    funcs.put(name,args),
    code(".method "+name);
    code(".args "+(args+1));
    parseBody(args,vars,funcs,prototypes);
    code("bipush 0");
    code("ireturn");
  }
}
```

Rendel et al. (2014)

# Modularized compiler

ca. 25 files
1620 lines of Scala code
modular

# How to Compose Ecosystems?

- Common target language helps

# How to Compose Static Semantics

Tillmann Rendel, Klaus Ostermann, Christian Hofer.
**Typed Self-Representation.**
In *Proceedings of the International Conference on Programming Language Design and Implementation*, June 2009.

# Typed Self-Representation

*Can we embed*
*a statically typed language*
*into itself?*

Rendel et al. (2009)

# Type-Safe Self-Evaluation

<T> T **eval**(expr: Expr<T>)

**eval :** forall T . Expr T →T

Rendel et al. (2009)

# The Expr<T> Family of Types

- **Representation**
  quote(t) : Expr<T> *if and only if* t : T
- **Adequacy**
  expr : Expr<T> *implies* t : T *with* quote(expr) = t *exists*
- **First Class Interpretations**
  *there are operations on* Expr<T> *values*
- **Self Interpretation**
  t : T *implies* eval<T>(quote(t)) == t
- **Reflection**
  quote(t) *exhibits the intensional structure of* t

Rendel et al. (2009)

# The Language $F_\omega^*$

- Pure Lambda Calculus

- Terms, Types, and Kinds
- Terms are classified by Types
- Types are classified by Kinds
- Kinds are classified by Kinds, too

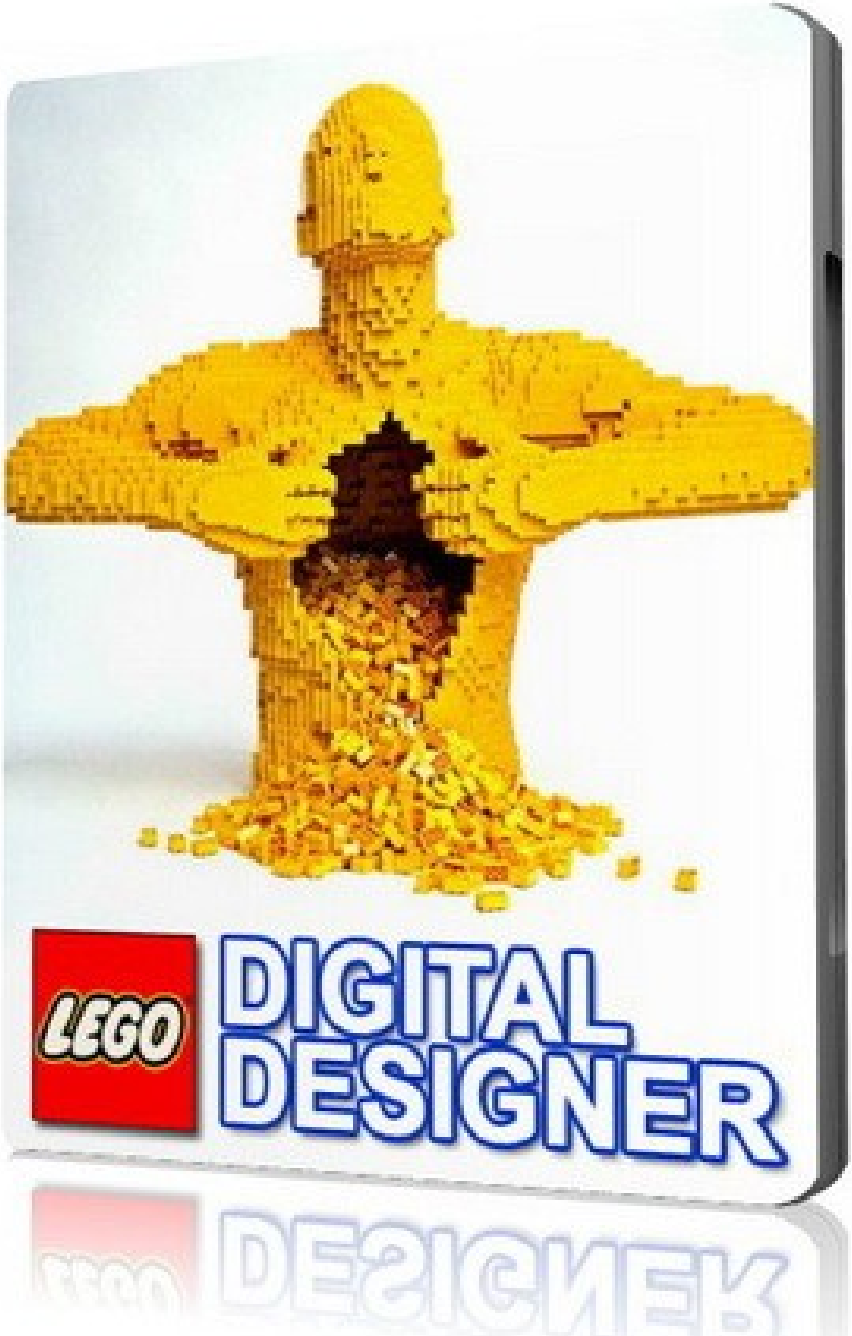
- Expr<T> is implemented with Church encoding

Rendel et al. (2009)

# Related Work

- **Metacircularity in the Polymorphic Lambda-Calculus**
  by Frank Pfenning and Peter Lee.
  In Theoretical Computer Science 89(1), 1991.
- **Typed Self-Representation**
  by Tillmann Rendel, Klaus Ostermann and Christian Hofer.
  In Proc. of PLDI, 2009.
- **Typed Self-Interpretation by Pattern Matching**
  by Barry Jay and Jens Palsberg.
  In Proc. of ICFP, 2011.
- **Self-Representation in Girard's System U.**
  by Matt Brown and Jens Palsberg.
  To appear in Proc. of POPL, 2015.

Rendel et al. (2009)


LEGO
DIGITAL
DESIGNER

# How to Design Languages?

Paolo G. Giarrusso, Tillmann Rendel, Klaus Ostermann,
Eric Walkingshaw.
**Formal Semantics as a Language Designer's Toolbox: A case for semantics-inspired language design.**
Presentation at *Workshop on Domain-Specific Language Design and Implementation*, October 2014

How can a

**programmer/**

**language designer**

learn to design languages that are

**elegant and usable?**

Giarrusso et al. (2014)

# Formal Semantics

- Semanticists know a lot about languages
  (it's their job)
- Semanticists know a lot about elegance
  (they are mathematicians)
- Mathematical elegance has pragmatic advantages

  Elegant = powerful and simple, less to learn

Giarrusso et al. (2014)

Can formal semantics guide a programmer/language designer towards an elegant and usable design?

Giarrusso et al. (2014)

# Problem 1

- *Problem*: Formal semantics is a lot of work.
- *Proposed Solution*: Don't actually formalize the semantics, just let the insights of formal semantics guide your design process.

Giarrusso et al. (2014)

# Problem 2

- *Problem*: The language of the semanticists is not understandable to the working programmer/language designers
- *Proposed Solution*: Package the insights from formal semantics as **language design patterns**.

Giarrusso et al. (2014)

# Language Design Patterns

- Patterns work for software design,
  we want to adapt them for language design
- Use terms that make sense to the working programmer/language designer

Giarrusso et al. (2014)

| | |
|---|---|
| *name* | **Bound & Binding Occurrences** |
| *problem* | How to structure names? |
| *solution* | Distinguish bound and binding occurrences of names. Each bound occurrences refers to a binding occurrence. |
| *effects* | You can reason about the naming structure of a program in terms of „this name here is bound there“ |

Giarrusso et al. (2014)

*name* **Bound & Binding Occurrences**

*pro*

*solu*

*ef*

*name* **Lexical Scoping**

*problem* Which bound occurrence refers to which binding occurrence?

*solution* All bound occurrences in a continuous region of the source file bind to the same binding occurrence.

*effects* You can reason about the binding structure statically.

Giarrusso et al. (2014)

*name* **Bound & Binding Occurrences**

*pro*

*solu*

*ef*

*name* **Lexical Scoping**

*pro*

*solu*

*ef*

*name* **Associated Scoping**

*problem* Which bound occurrence refers to which binding occurrence?

*solution* Attach the scoping information to a domain-specific entity in your language design.

*effects* Your binding structure supports your domain integration.

Giarrusso et al. (2014)

| | |
|---|---|
| *name* | **Meaning** |
| *problem* | How to specify the semantics? |
| *solution* | Map every program to its meaning. |
| *effects* | Allows to identify programs that mean the same but work differently internally. |

Giarrusso et al. (2014)

*name* **Meaning**

*pro*

*solu*

*ef*

*name* **Simple Meaning**

*problem* How to structure the meaning?

*solution* Choose the simplest thing that works.

*effects* Carefully choosing the meaning helps you focus your design on your domain.

Giarrusso et al. (2014)

*name* **Meaning**

*name* **Simple Meaning**

*name* **Recursive Meaning**

*problem* How to define the meaning mapping?

*solution* Map each *phrase* of the program to its meaning.

*effects* You can explain what a part of a program means.

Giarrusso et al. (2014)

*name* **Meaning**

*name* **Simple Meaning**

*name* **Recursive Meaning**

*name* **Compositional Meaning**

*problem* How to define the meaning mapping?

*solution* Define the meaning of a phrase in terms of the meaning of its subphrases.

*effects* The meaning of a phrase is the phrase's interface. Allow code moving without changing meaning.

Giarrusso

| | |
|---|---|
| *name* | **Type Structure** |
| *problem* | How to structure the primitives? |
| *solution* | Structure your language design around the available types of values. Think of the primitives as the interfaces of the types. |
| *effects* | Easier to not forget primitives. Structuring principle also for documentation. |

Giarrusso et al. (2014)

*name* **Type Structure**

*name* **Constructor**

*problem* Which operations for a type?

*solution* Provide constructors for making new values of a type.

*effects* User programs can create values of the type.

Giarrusso et al. (2014)

*name* **Type Structure**

*name* **Constructor**

*name* **Destructor**

*problem* Which operations for a type?

*solution* Provide destructors for getting information out of values of a type.

*effects* User programs can use values of the type.

Giarrusso et al. (2014)

*name* **Type Structure**

*name* **Constructor**

*name* **Destructor**

| | |
|---|---|
| *Name* | **Information Preservation** |
| *problem* | How to balance constructors and destructors? |
| *solution* | Provide enough destructors to get all information out of an constructed value.<br>Provide enough constructors to recreate a destructed value. |
| *effects* | No identity and no secrets. |

Giarrusso

# Language Design Patterns ...

- guide the design process
  („*think of all constructors*")
- structure the design
  („*separate constructors and destructors*")
- highlight design choices
  („*which kind of scoping is appropriate?*")
- explain effects („*user programs can ...*")
- interact („*if a compositional meaning is a phrase's interface, a simple meaning is a better interface*")

Giarrusso et al. (2014)

# Conclusion

- Computer languages matter
- There are many computer languages
- Package domain knowledge in languages
- Structure component interfaces as languages
- Reuse language design concepts
- Reuse language implementation artifacts

# Conclusion

- Computer languages matter
- There are many computer languages

- Package domain knowledge in languages
- Structure component interfaces as languages

- Reuse language design concepts
- Reuse language implementation artifacts

*Thanks*