The basic skill set of software language engineering

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
Germany

This is sort of an advertisement for the upcoming textbook on software languages:


http://www.softlang.org/book
What’s to come today?

• An index-based story
• A profile based on languages
• A walk through the TOC
• The underlying repository

Let’s run this a bit more like a discussion, not as a pure tutorial.
An index-based story
<table>
<thead>
<tr>
<th>Abstract Domain</th>
<th>Compositionality</th>
<th>EBNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Interpretation</td>
<td>Conclusion</td>
<td>Embedded System</td>
</tr>
<tr>
<td>Abstract Syntax</td>
<td>Concrete Syntax</td>
<td>Expression Evaluation</td>
</tr>
<tr>
<td>Abstraction</td>
<td>Context-Free Grammar</td>
<td>External DSL</td>
</tr>
<tr>
<td>Acceptance</td>
<td>Continuation</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>Acceptor</td>
<td>CPS</td>
<td>Fixed Point</td>
</tr>
<tr>
<td>Acme</td>
<td>Control Flow</td>
<td>Function</td>
</tr>
<tr>
<td>Algebraic Signature</td>
<td>Data Flow Analysis</td>
<td>Function Application</td>
</tr>
<tr>
<td>Alpha Conversion</td>
<td>Derivation Sequence</td>
<td>Function Symbol</td>
</tr>
<tr>
<td>Assignment</td>
<td>Derivation Tree</td>
<td>Functional Language</td>
</tr>
<tr>
<td>Axiomatic Semantics</td>
<td>Directional Type</td>
<td>Functional Program</td>
</tr>
<tr>
<td>Big-Step ...</td>
<td>Domain Model</td>
<td>Galois Connection</td>
</tr>
<tr>
<td>BNF</td>
<td>Domain Theory</td>
<td>Goto</td>
</tr>
<tr>
<td>Bottom</td>
<td>DSL</td>
<td>Grammar</td>
</tr>
<tr>
<td>Compiler</td>
<td>EBNF</td>
<td>Haskell</td>
</tr>
<tr>
<td>Compiler Optimization</td>
<td>Expression Evaluation</td>
<td>Homomorphism</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Imperative Programming</td>
</tr>
</tbody>
</table>
• inference rule
• inlining
• internal DSL
• interpreter
• Java
• JavaScript
• judgment
• language definition
• lattice
• logic
• loop unrolling
• many-sorted term
• MDE
• meta-variable
• metalevel
• metametalevel
• method chaining
• model
• modeling language
• nonterminal
• normal form
• object model
• operational semantics
• parse tree
• parser
• parsing
• partial evaluation
• partial order
• Peano
• polymorphic function
• polymorphism
• pragmatics
• pre-term
• premise
• preservation
• production
• profile
• program analysis
• program specialization
• progress
• Prolog
• recursive function
• residual program
• rule
• semantic algebra
• semantic action
• **semantics**
• signature
• slicing
• **splice**
• small-step ...
• sort
• start symbol
• statement execution
• substitution
• **syntax**
• syntax checker
• syntax tree
• System F
• term
• terminal

• text-to-model
• **text-to-object**
• **textual syntax**
• top
• transition system
• **type**
• type safety
• **type checking**
• type derivation
• type inference
• **type system**
• UML
• variable
• visual syntax
• **well-formedness**
A profile based on languages
Sample languages (syntax definition)

• BGL — Basic Grammar Language (‘BNF’)
• EGL — Extended Grammar Language (‘EBNF’)
• BSL — Basic Signature Language
• ESL — Extended Signature Language
• MML — Ecore-like Metamodelling Language
• DDL — SQL-like Data Definition Language
Sample languages (GPLs, DSLs)

• BNL — Binary Number Language
• BTL — Basic TAPL Language
• FSML — FSM Language
• TBL — Tree-based Buddy Language
• GBL — Graph-based Buddy Language
• BFPL — Basic Functional Programming Language
• EFPL — Extended Functional Programming Language
• BIPL — Basic Imperative Programming Language
• EIPL — Extended Imperative Programming Language
Implementation languages

• Haskell — for most (‘all’) illustrations
• ANTLR — whenever it works
• Python — for a few illustrations
• Java — for a few illustrations
• XML, XSD, JSON, JSON Schema — a bit
• Prolog — last chapter; do ‘everything’ again
A walk through the TOC
Software Languages
A slightly Haskell-biased introduction

- Part I — Introduction
- Part II — Syntax and metaprogramming
- Part III — Semantics and types
- Part IV — Consolidation
Software Languages
A slightly Haskell-biased introduction

- **Part I — Introduction**
  - Preface
  - The notion of software language
  - Story of a language

- Part II — Syntax and metaprogramming
- Part III — Semantics and types
- Part IV — Consolidation
Software Languages
A slightly Haskell-biased introduction

• Part I — Introduction
  • Preface
  • The notion of software language
    • Software language examples
    • Classification of software languages
    • The lifecycle of software languages
    • Software languages in software engineering
    • Software languages in academia
  • Story of a language

• Part II — Syntax and metaprogramming

• Part III — Semantics and types

• Part IV — Consolidation
The software language lifecycle

- Language design
- Language definition
- Language implementation
- Language evolution
- Language usage
- Domain analysis
- Language retirement
Compilation

Grammar

Rules for type system etc.

Rules for code generation

Parser

Semantic analysis

Enriched parse tree

Code generator

Source code

Parse tree

Enriched parse tree

Machine code

Compilation
Re-engineering

Grammar → Parser → Parse tree → Semantic analysis → Enriched parse tree → Transformation → Transformed parse tree

Rules for type system etc. → Enriched parse tree

Rules for transformation
Fact extraction in reverse engineering

Parser

Semantic analysis

Enriched parse tree

Fact extraction

Grammar

Rules for type system etc.

Rules for fact extraction

Source code

Extracted facts
Software Languages
A slightly Haskell-biased introduction

• Part I — Introduction
  • Preface
  • The notion of software language
  • Story of a language
    • Language concepts
    • Internal DSL style
    • Textual syntax
    • Parsing text to objects
    • Language-agnostic models
    • Language constraints
    • Interpretation
    • Code generation
    • Visual syntax

• Part II — Syntax and metaprogramming
• Part III — Semantics and types
• Part IV — Consolidation
The DSML FSML
Parsing (text-to-object) for FSML

```plaintext
fsm : statedecl* EOF ;
statedecl :
   { boolean initial = false; }.
   ( 'initial' { initial = true; } )?
   'state' stateid
   { fsm.getStates().add(new State($stateid.text, initial)); }
   '}' transition[$stateid.text]* '}'
   ;
transition[String source] :
event
   { String action = null; }
   ( '/' action { action = $action.text; } )?
   { String target = source; }
   ( '->' stateid { target = $stateid.text; } )?
   { fsm.getTranss().add(new Trans(source, $event.text, action, target)); }
   ';
   ;
```

At Acme, different approaches were used over the years to represent FSML models. Initially, they used a Java-based object model. Later, they favored an XML representation, thereby getting independent of Java. More recently, JSON was favored.
Software Languages
A slightly Haskell-biased introduction

• Part I — Introduction

• Part II — Syntax and metaprogramming
  • Abstract syntax—foundations
  • Abstract syntax—implementations
  • Basics of metaprogramming
  • Concrete syntax—foundations
  • Concrete syntax—implementations

• Part III — Semantics and types

• Part IV — Consolidation
Syntax mapping

Input text → Scanning → Parsing → Token streams

Token streams → Parsing → CST

CST → Abstraction → AST

AST → Resolution → ASG

ASG → Text-to-model

AST → Model-to-text → Output text

Output text → Formatting

Formatting → Parsing → CST

CST → Formatting

Parsing → Scanning

Scanning → Input text
Parse trees
(CSTs)
Mapping CSTs to ASTs

Concrete syntax tree

```
fork((number, number, [n(bits), n(rest)]),
    [fork((many, bits, [n(bit), n(bits)]),
        [fork((one, bit, [t('1')]),
            [leaf('1')]),
        fork((single, bits, [n(bit)]),
            [fork((zero, bit, [t('0')]),
                [leaf('0')])])]),
    fork((integer, rest, []), [
        []])).
```

Abstract syntax tree

```
number(
    many(
        one
        single(
            zero)),
    integer).
```
Parser generation

MyLang.g4 → java org.antlr.v4.Tool → MyLangParser.java → MyLangDriver.java → Parse tree

MySample → Code generation-time data flow

Code-level dependencies

Runtime data flow

Parser generator (ANTLR)
Programmer-provided artifacts
Generated code

© 2016, Software Languages Team, University of Koblenz-Landau, Ralf Lämmel
Mapping grammars to signatures

concrete [rational] rest : '.' bits ;

versus

abstract symbol rational : bits -> rest ;

syntax
Resolution (AST-to-ASG)

<table>
<thead>
<tr>
<th>TBL</th>
<th>GBL</th>
</tr>
</thead>
</table>
| newtype World = World {  
   getPersons :: [Person]  
} | newtype World = World {  
   getPersons :: [IORef Person]  
} |
| data Person = Person {  
   getName :: String,  
   getBuddy :: Maybe String  
} | data Person = Person {  
   getName :: String,  
   getBuddy :: Maybe (IORef Person)  
} |
Software Languages
A slightly Haskell-biased introduction

• Part I — Introduction

• **Part II — Syntax and metaprogramming**
  • Abstract syntax—foundations
  • Abstract syntax—implementations
  • **Basics of metaprogramming**
  • Concrete syntax—foundations
  • Concrete syntax—implementations

• Part III — Semantics and types

• Part IV — Consolidation
Pretty printing for FSML

initial state locked {
    ticket/collect -> unlocked;
    pass/alarm -> exception;
}

state unlocked {
    ticket/eject;
    pass -> locked;
}

...
Template processing for FSML

```
Fsm {
    State {
        initial=True,
        stateid="locked",
        transitions=[
            Transition { event="ticket", action=Just "collect", target=Just "unlocked" },
            Transition { event="pass", action=Just "alarm", target=Just "exception" }
        ]
    },
    State {
        initial=False,
        stateid="unlocked",
        transitions=[
            Transition { event="ticket", action=Just "eject", target=Nothing },
            Transition { event="pass", action=Nothing, target=Just "locked" }
        ]
    },
...
```

Data

```
("main", "$fsm.states.state()…$"),
("state", "... state ...")
("transition", "... -> ...")
```

Rendering

```
initial state locked {
    ticket/collect -> unlocked;
    pass/alarm -> exception;
}
state unlocked {
    ticket/eject;
    pass -> locked;
}
...
```

Templates
State id renaming for FSML

**Input**

*initial state* locked {
  ticket / collect -> unlocked;
  pass / alarm -> exception;
}

*state* unlocked {
  ticket / eject;
  pass -> locked;
}

...
Composition for FSML

```
initial state locked {
    ticket / collect -> unlocked;
}
state unlocked {
    ticket / eject;
    pass -> locked;
}
state locked {
    pass / alarm -> exception;
}
state exception {
    ticket / eject;
    pass;
    mute;
    release -> locked;
}
```

```
initial state locked {
    ticket / collect -> unlocked;
    pass / alarm -> exception;
}
state unlocked {
    ticket / eject;
    pass -> locked;
}
state locked {
    pass / alarm -> exception;
    pass -> locked;
}
state exception {
    ticket / eject;
    pass;
    mute;
    release -> locked;
}
```
Software Languages
A slightly Haskell-biased introduction

• Part I — Introduction

• Part II — Syntax and metaprogramming

• Part III — Semantics and types
  • Operational semantics
  • Type systems
  • Denotational semantics

• Part IV — Consolidation
Operational semantics (big step)

zero → zero

\[ e → n \]
\[ \text{succ}(e) → \text{succ}(n) \]

evaluate Zero = Just Zero
evaluate (Succ e) =
  \[
  \text{case } \text{evaluate } e \text{ of }
  \]
  (Just n) →
  \[
  \text{if } \text{isNat} \ n
  \]
  then Just (Succ n)
  else Nothing
Nothin → Nothing

© 2016, Software Languages Team, University of Koblenz-Landau, Ralf Lämmel
Type systems

zero : nat

e : nat

---

succe : nat

[t~succ]

[t~zero]

zero : nat

const: nat

false

true

ToInt: int

typeOf (If e0 e1 e2) = typeOf (IsZero e) = typeOf (Pred e) = typeOf Zero = Just Nat

wellTyped e =

def_typeOf (If e0 e1 e2) = typeOf (If e0 e1 e2) =
def_typeOf (IsZero e) = typeOf (IsZero e) =
def_typeOf (Pred e) = typeOf (Pred e) =
def_typeOf Zero = Just Nat

wellTyped :: Expr

false

true

Int

just True

just False

Nothing

typeOf e0
typeOf e1
typeOf e2

case typeof e of

(Just Nat) -> Just Nat

default Nothing

---

 BFPL and BIPL implemented in Haskell, Prolog, and other metalanguages. Further examples of type checkers can be found in the repository of the Software Language Book. For instance, we can implement the type system of BTL in Haskell, Prolog, or any other language. In the following illustration, we strive again for a very systematic mapping of BTL's typing rules to Haskell function equations. Further examples of type checkers can be found in the repository of the Software Language Book. For instance, we can implement the type system of BTL in Haskell, Prolog, or any other language.
Denotational semantics

\[
\begin{align*}
\mathcal{S}[\text{skip}] &= \text{skip} \\
\mathcal{S}[\text{assign}(x, e)] &= \text{assign } x (\mathcal{E}[e]) \\
\mathcal{S}[\text{seq}(s_1, s_2)] &= \text{seq } (\mathcal{S}[s_1]) (\mathcal{S}[s_2]) \\
\mathcal{S}[\text{if}(e, s_1, s_2)] &= \text{if } (\mathcal{E}[e]) (\mathcal{S}[s_1]) (\mathcal{S}[s_2]) \\
\mathcal{S}[\text{while}(e, s)] &= \text{while } (\mathcal{E}[e]) (\mathcal{S}[s])
\end{align*}
\]

execute \( \text{Skip} = \text{skip}' \)
execute \( \text{(Assign } x \text{ e)} = \text{assign}' x (\text{evaluate } e) \)
execute \( \text{(Seq } s_1 \text{ s2)} = \text{seq}' (\text{execute } s_1) (\text{execute } s_2) \)
execute \( \text{(If } e \text{ s1 s2)} = \text{if}' (\text{evaluate } e) (\text{execute } s_1) (\text{execute } s_2) \)
execute \( \text{(While } e \text{ s)} = \text{while}' (\text{evaluate } e) (\text{execute } s) \)
Software Languages
A slightly Haskell-biased introduction

• Part I — Introduction

• Part II — Syntax and metaprogramming

• Part III — Semantics and types

• Part IV — Consolidation
  • Metaprogramming techniques
  • Logic programming to the rescue
  • Postface
Software Languages
A slightly Haskell-biased introduction

• Part I — Introduction

• Part II — Syntax and metaprogramming

• Part III — Semantics and types

• Part IV — Consolidation
  • Metaprogramming techniques
    • Term rewriting
    • Attribute grammars
    • Multi-stage programming
    • Partial evaluation

• Logic programming to the rescue
• Postface
-- Laws on expressions
\[ x + y = y + x \]
\[ x * y = y * x \]

-- Implementation based on abstract object syntax
\[
\text{commute} :: \text{Expr} \to \text{Maybe} \text{Expr} \\
\text{commute} (\text{Binary Add} \; x \; y) = \text{Just} \; \text{Binary Add} \; y \; x \\
\text{commute} (\text{Binary Mul} \; x \; y) = \text{Just} \; \text{Binary Mul} \; y \; x \\
\text{commute} _ = \text{Nothing}
\]

-- Implementation based on concrete object syntax
\[
\text{commute} :: \text{Expr} \to \text{Maybe} \text{Expr} \\
\text{commute} [\text{el} | \; x + \; y \; |] = \text{Just} \; [\text{el} | \; y + \; x \; |] \\
\text{commute} [\text{el} | \; x * \; y \; |] = \text{Just} \; [\text{el} | \; y * \; x \; |] \\
\text{commute} _ = \text{Nothing}
\]
Term rewriting incl. strategies

![Diagram of term rewriting strategies](image)

- **all**
- **one**
- **fulltd**
- **oncebu**
Attributed trees

number 5.25

many

many 5

many

rational

many .25

many

rational

many .25

single

single

one 4

one 1

zero 0

zero 0

one .25

one .25

one .25

one .25

‘1’

‘0’

‘1’

‘.’

‘0’

‘1’

© 2016, Software Languages Team, University of Koblenz-Landau, Ralf Lämmel
Attribute dependencies
Yet Another SLR
(Software Language Repository)

The underlying repository

https://github.com/softlang/yas/
./languages/BIPL/Haskell/Language/BIPL/Sign/Domains.hs
./languages/BIPL/Haskell/Language/BIPL/Sign/Fix.hs
./languages/BIPL/Haskell/Language/BIPL/Sign/Sample.hs
./languages/BIPL/Haskell/Language/BIPL/Sign/V1/Analysis.hs
./languages/BIPL/Haskell/Language/BIPL/Sign/V2/Analysis.hs
./languages/BIPL/Haskell/Language/BIPL/Syntax.hs
./languages/BIPL/Haskell/Language/ML/Machine.hs
./languages/BIPL/Haskell/Language/ML/Sample.hs
./languages/BIPL/Haskell/Language/ML/Syntax.hs
./languages/BIPL/Haskell/Main.hs
./languages/BNL/ESL/haskell/as.hs
./languages/BNL/Haskell/Language/BNL/Acceptor.hs
./languages/BNL/Haskell/Language/BNL/Conversion.hs
./languages/BNL/Haskell/Language/BNL/Parser.hs
./languages/BNL/Haskell/Language/BNL/Syntax.hs
./languages/BNL/Haskell/Main.hs
./languages/BSL/Haskell/Language/BSL/Conformance.hs
./languages/BSL/Haskell/Language/BSL/Sample.hs
./languages/BSL/Haskell/Language/BSL/Syntax.hs
./languages/BSL/Haskell/Main.hs
./languages/BTL/Haskell/Language/BTL/BigStep.hs
./languages/BTL/Haskell/Language/BTL/BigStepWithGuards.hs
./languages/BTL/Haskell/Language/BTL/Closure.hs
./languages/BTL/Haskell/Language/BTL/Domains.hs
./languages/BTL/Haskell/Language/BTL/Dynamics.hs
./languages/BTL/Haskell/Language/BTL/Interpreter.hs
./languages/BTL/Haskell/Language/BTL/Nf.hs
./languages/BTL/Haskell/Language/BTL/Sample.hs
./languages/BTL/Haskell/Language/BTL/Interpreter.hs
./languages/BTL/Haskell/Language/BTL/Nf.hs
./languages/BTL/Haskell/Language/BTL/Sample.hs
./languages/BTL/Haskell/Language/BTL/SmallStep.hs
./languages/BTL/Haskell/Language/BTL/Syntax.hs
./languages/BTL/Haskell/Language/BTL/Typing.hs
./languages/BTL/Haskell/Language/BTL/Universal/Sample.hs
./languages/BTL/Haskell/Language/BTL/Universal/Term.hs
./languages/BTL/Haskell/Main.hs
./languages/EGL/Haskell/Language/EGL/Interpreter.hs
./languages/EGL/Haskell/Language/EGL/Lexer.hs
./languages/EGL/Haskell/Language/EGL/Parser.hs
./languages/EGL/Haskell/Language/EGL/Sample.hs
./languages/EGL/Haskell/Language/EGL/Syntax.hs
./languages/EGL/Haskell/Main.hs
./languages/EL/Haskell/Language/EL/Lexer.hs
./languages/EL/Haskell/Language/EL/MoreRules.hs
./languages/EL/Haskell/Language/EL/Normalizer.hs
./languages/EL/Haskell/Language/EL/Parser.hs
./languages/EL/Haskell/Language/EL/QQ/Lexer.hs
./languages/EL/Haskell/Language/EL/QQ/MoreRules.hs
./languages/EL/Haskell/Language/EL/QQ/Parser.hs
./languages/EL/Haskell/Language/EL/QQ/Rules/Associate.hs
./languages/EL/Haskell/Language/EL/QQ/Rules/Commute.hs
./languages/EL/Haskell/Language/EL/QQ/Rules/Simplify.hs
./languages/EL/Haskell/Language/EL/QQ/Rules.hs
./languages/EL/Haskell/Language/EL/QQ/Syntax.hs
./languages/EL/Haskell/Language/EL/QuasiQuoter.hs
./lib/Haskell/Data/Generics/SimpleStrategySchemes.hs
./lib/Haskell/Data/Generics/StrategyBasics.hs
./lib/Haskell/Data/Generics/StrategyPrimitives.hs
./lib/Haskell/Data/Generics/StrategySchemes.hs
./lib/Haskell/Data/Graph.hs
./lib/Haskell/Data/Term.hs
./lib/Haskell/Data/TermRep.hs
./lib/Haskell/Text/Parsec/Utilities.hs
./samples/Haskell/Powerehs
./samples/Haskell/TestTypedPower.hs
./samples/Haskell/TestUntypedPower.hs
./samples/Haskell/TypedPower.hs
./samples/Haskell/UntypedPower.hs
Grammars

\[
\begin{align*}
S & \rightarrow aS \mid bX \\
X & \rightarrow aX \mid bY \\
Y & \rightarrow aY \mid bZ \mid \Lambda \\
Z & \rightarrow aZ \mid \Lambda 
\end{align*}
\]

\[ S \rightarrow aS \mid bX \\
X \rightarrow aX \mid bY \\
Y \rightarrow aY \mid bZ \mid \Lambda \\
Z \rightarrow aZ \mid \Lambda \]
\[ S \rightarrow aS | bX \\
X \rightarrow aX | bY \\
Y \rightarrow aY | bZ | \Lambda \\
Z \rightarrow aZ | \Lambda \]
\[
S \rightarrow aS \mid bX \\
X \rightarrow aX \mid bY \\
Y \rightarrow aY \mid bZ \mid \Lambda \\
Z \rightarrow aZ \mid \Lambda
\]
symbol number: bits # rest -> number;
symbol single: bit -> bits;
symbol many: bit # bits -> bits;
symbol zero: -> bit;
symbol one: -> bit;
symbol integer: -> rest;
symbol rational: bits -> rest;

Signatures
symbol number: bits # rest -> number;
symbol single: bit -> bits;
symbol many: bit # bits -> bits;
symbol zero: -> bit;
symbol one: -> bit;
symbol integer: -> rest;
symbol rational: bits -> rest;
symbol number: bits # rest -> number;
symbol single: bit -> bits;
symbol many: bit # bits -> bits;
symbol zero: -> bit;
symbol one: -> bit;
symbol integer: -> rest;
symbol rational: bits -> rest;
./languages/Term/graph/iterate.pro
./languages/Term/graph/deref.pro
./languages/Term/graph/normal.pro
./languages/Term/graph/search.pro
./languages/Text/textEq.pro
./languages/Text/textLoc.pro
./languages/ueber/macros/bgl-and-bsl.pro
./languages/ueber/macros/bmpl.pro
./languages/ueber/macros/egl.pro
./languages/ueber/macros/esl.pro
./languages/ueber/macros/forall.pro
./languages/ueber/macros/fxy.pro
./languages/ueber/macros/graph.pro
./languages/ueber/macros/include.pro
./languages/ueber/macros/lal.pro
./languages/ueber/macros/mml.pro
./languages/ueber/macros/parse.pro
./languages/ueber/macros/pickyParse.pro
./languages/ueber/macros/ppl.pro
./languages/ueber/macros/rules.pro
./languages/ueber/macros/test.pro
./languages/ueber/ueberApply.pro
./languages/ueber/ueberMain.pro
./languages/ueber/ueberNorm.pro
./languages/ueber/ueberOk.pro
./languages/ueber/ueberReport.pro
./languages/ueber/ueberSub.pro
./languages/ueber/ueberVerify.pro
./lib/Prolog/assoc.pro
./lib/Prolog/data.pro
./lib/Prolog/dynamic.pro
./lib/Prolog/generics.pro
./lib/Prolog/higher-order.pro
./lib/Prolog/http.pro
./lib/Prolog/io.pro
./lib/Prolog/logvars.pro
./lib/Prolog/multifile.pro
./lib/Prolog/scanning.pro
./lib/Prolog(strategyBasics.pro
./lib/Prolog(strategySchemes.pro
./lib/Prolog/syntax.pro
./lib/Prolog/test.pro
./lib/Prolog/ueber.pro
./main.pro
Discussion

• Many roads to Rome — this is one SLE treatment
• Focus on late BSc, early MSc + further reading
• Bits of PLT, CC, MDE — where needed
• Too many technology options — we opt for Haskell
• Prolog would be easier — if readers were willing
Thanks!

Please get in touch, if you like to be an early “adopter”.


http://www.softlang.org/book