Domain-specific Languages

Course "Software Language Engineering"

University of Koblenz-Landau
Department of Computer Science
Ralf Lämmel
Software Languages Team
What’s a DSL?
An example of a DSL

... 102:45:44 Aldrin: Okay. Engine Stop.
102:45:45 Aldrin: ACA out of Detent.
102:45:57 Duke: We copy you down, Eagle.
102:45:58 Armstrong (on-board): Engine arm is off. (Pause) Houston, Tranquility Base here. The Eagle has landed.
...

Simplified, domain-specific grammar and terminology focused on “Moon landing”.

© 2012-14, Ralf Lämmel, Software Languages Team, University of Koblenz-Landau, and collaborators (see 1st slide)
Domain-specific notation

✗ matrixA.multiply(matrixB);

✔ matrixA * matrixB
The word *domain* in DSL refers to "an area or sphere of knowledge, influence, or activity." Focusing on a domain gives you a *context* -- a logical framework within which you can evolve models for an application.
A:hover { color: #FF0000; text-decoration: none; }
GCC=cxx -I.
all : clean compile
compile : myprog
myprog: MyProg.cc Util.o
   $(GCC) -o myprog MyProg.cc Util.o
Util.o : Util.h Util.cc
   $(GCC) -c Util.cc
clean :
   /bin/rm -f myprog Util.o
General-purpose language

- Turing complete
- Well understood and widely used
- Applicable to a wide range of problems
GPL Pros

- Excellent support through IDEs
- Easy to find experienced developers
- Growth through libraries
GPL Cons

- Growth of language is (often) not possible
- Can be very verbose
- Lack of abstractions
Domain-specific language

- Small language targeted at a particular problem domain
- Expressive in its own domain
- Often declarative
DSL Pros

- Tailored to a particular application domain
  [Mernik 2005]
- Expressive and easy to use
  [Hudak 1998]
- Little languages, little maintenance, increased productivity
  [Deursen 1997]
- Communication with domain experts
  [Fowler, Domain-specific languages, Addison-Wesley 2010]
DSL Cons

- Extra investment
- Language engineering
- Learning a new language
- Weak IDE support
- Editor, Debugger, Refactoring
- Migration might be difficult
- Evolving into generality
Aspects of DSL implementation
Implementation styles

- **Internal**: APIs, combinator libraries, fluent APIs
- **External**: a different language independent of host language
- **Embedded**: extend host language with syntax & semantics

---

Workbenches support embedded language development.
External vs. internal DSLs

An external or *free-standing* DSL is designed to be independent of any particular language.

An internal or *embedded* DSL, on the other hand, is designed and implemented using a host language.
External

- DSLs that *use a different* syntax to the main language that uses them (if any).

Examples

- make, flex, yacc, bison
- XPath, SQL
- sed, awk
External Pros

- Language Designer
  - Tools for compiler construction can be used

- Language User
  - Simpler to use than a GPL
External Cons

- **Language Designer**
  - Expensive to implement

- **Language User**
  - Weak tool support
  - Yet another language to learn
  - Often targeted towards a particular GPL
  - Often difficult to closely integrate with GPL
CSS (external DSL)

A:hover { color: #FF0000; text-decoration: none; }

This slide has been extracted from “Domain Specific Languages” by Lukas Renggli (course slides). See license at the end of the deck.
Makefile (external DSL)

GCC=cxx -I.
all : clean compile
compile : myprog
myprog: MyProg.cc Util.o
   $(GCC) -o myprog MyProg.cc Util.o
Util.o : Util.h Util.cc
   $(GCC) -c Util.cc
clean :
   /bin/rm -f myprog Util.o
<project name="AnExampleProject" default="jarit" basedir=".">
<property name="src" location="src"/>
<property name="build" location="build"/>
<property name="distrib" location="distrib"/>
<target name="compile" description="compile your Java code from src into build">
<javac srcdir="${src}" destdir="${build}"/>
</target>
<target name="jarit" depends="compile" description="jar it up">
<jar jarfile="${distrib}/AnExampleProject.jar" basedir="${build}"/>
</target>
</project>
DSLs that *share the same* syntax to the main language that uses them.

**Examples**

- Parsec (Haskell)
- PetitParser (Smalltalk)
- rake, rspec (Ruby)
- jQuery (JavaScript)
- pypy (Python)
Internal

- A subset of the host language is used.
- Popular in Lisp, Scheme, Ruby, Smalltalk and JavaScript.
Internal Pros

- **Language Designer**
  - No special tools required
  - No new grammar required

- **Language User**
  - Intermixable with GPL
  - Tools continue to work
  - No new language to learn

This slide has been extracted from “Domain Specific Languages” by Lukas Renggli (course slides). See license at the end of the deck.
Internal Cons

- Limited expressivity
- Unnecessary syntactic noise
- Constrained by host language
Rake (internal in Ruby)

```
ORIGINAL = 'input.dat'
BACKUP = 'input.dat.bak'
task :default => BACKUP
file BACKUP => ORIGINAL do |task|
cp task.prerequisites[0], task.name
end
```
#slightly modified version of example from http://gant.codehaus.org/
includeTargets << gant.targets.Clean

cleanPattern << [ '**/*', '**/*.bak']
cleanDirectory << 'build'
target (stuff : 'A target to do some stuff') {
    println 'Stuff'
depends clean
echo message : 'A default message from Ant'
otherStuff()
}
target (otherStuff : 'A target to do some other stuff') {
    println 'OtherStuff'
echo message : 'Another message from Ant'
clean()
}
setDefaultTarget stuff

Generates "ant" (XML).
Rather than using the highly parameterized `ParsecT` type, we specialize the types of the parser combinators readily to \( P \) as used in the present contribution.

--- Sequential composition

\[
(\gg\gg) :: P \ a \rightarrow (a \rightarrow P \ b) \rightarrow P \ b
\]

--- Sequential composition; forget first result

\[
(\gg) :: P \ a \rightarrow P \ b \rightarrow P \ b
\]

--- Predictive choice

\[
(<|>) :: P \ a \rightarrow P \ a \rightarrow P \ a
\]

--- EBNF "*"

\[
\text{many} :: P \ a \rightarrow P \ [a]
\]
Internal DSL approaches

- Fluent APIs
- Operator Overloading
  - C++, C#, Smalltalk, Python, Ruby, ...
- Annotations
  - Java, C#, Smalltalk, Python, ...
- Parse Tree Manipulation
  - C#, Smalltalk
- Macros / Program generation
  - LISP, Scheme, Template Haskell, Converge
# Taxonomy for internal, external and embedded languages

<table>
<thead>
<tr>
<th></th>
<th>Syntax</th>
<th>Semantics</th>
<th>Host Integration</th>
<th>Tool Integration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>Internal languages make a creative use of the host language. They integrate seamlessly into the host language and tools, but their syntax and semantics is strictly constrained.</td>
</tr>
<tr>
<td>External</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>External languages are independent of the host language. This makes them difficult to integrate into the host language and development tools.</td>
</tr>
<tr>
<td>Embedded</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>Embedded languages combine the advantages of internal and external languages. Ideally an embedded language uses the same executable representation as the host and integrates well with tools.</td>
</tr>
</tbody>
</table>
Language workbenches

- IDEs designed for building DSLs.
- Common representation of host and domain specific languages.
- Examples
  - Spoofax, Sugar/J
  - JetBrains Meta Programming System (MPS)
  - openArchitectureWare
Hello

the simplest concept instance Hello

text = hello text

class Hello

public class Hello extends Object implements Interface {

  public Hello() {

    public static void main(String[] args) {

      System.out.println("$Hello");

    }

  }

}
This slide has been extracted from "Domain Specific Languages" by Lukas Renggli (course slides). See license at the end of the deck.
Simple patterns for internal DSL implementation
Example

```
Processor p = new Processor(2, Processor.Type.i386);
Disk d1 = new Disk(150, Disk.UNKNOWN_SIZE, null);
Disk d2 = new Disk(75, 7200, Disk.Interface.SATA);
return new Computer(p, d1, d2);
```
Method chains

```java
computer()
  .processor()
    .cores(2)
    .i386()
    .end()
  .disk()
    .size(150)
    .end()
  .disk()
    .size(75)
    .speed(7200)
    .sata()
    .end()
  .end()
.end();
```

Indentation only for inspiration.
The “end” methods return the “parent”. The other methods return “this”.
Imperative sequences

computer();
  processor();
    cores(2);
    i386();
  disk();
    size(150);
  disk();
    size(75);
    speed(7200);
  sata();
end();

Indentation only for inspiration. The methods must switch focus.
 Nested method calls

```java
computer(
    processor(
        cores(2),
        Processor.Type.i386),
    disk(
        size(150)),
    disk(
        size(75),
        speed(7200),
        Disk.Interface.SATA));
```

These are static methods which replace “functional constructors”.

© 2012-14, Ralf Lämmel, Software Languages Team, University of Koblenz-Landau, and collaborators (see 1st slide)
Literal Collections

```
[:computer,
  [:processor,
    [:cores, 2],
    [:type, :i386]],
[:disk,
  [:size, 150]],
[:disk,
  [:size, 75],
  [:speed, 7200],
  [:interface, :sata]]]
```
Closures

```
computer() do | c |
  c.processor() do | p |
    p.cores(2)
    p.i386()
  end
  c.disk().size(150)
  c.disk() do | d |
    d.size(75)
    d.speed(7200)
    d.sata()
  end
end

b.computer() do
  b.processor() do
    b.cores(2)
    b.i386()
  end
  b.disk().size(150)
  b.disk() do
    b.size(75)
    b.speed(7200)
    b.sata()
  end
end
```
For comparison: translation

```
computer:
  processor:
    cores 2
    i386
  disk:
    size 150
  disk:
    size 75
    speed 7200
    sata

Processor p = new Processor(2,
  Processor.Type.i386);
Disk d1 = new Disk(150,
  Disk.UNKNOWN_SIZE,
  null);
Disk d2 = new Disk(75,
  7200,
  Disk.Interface.SATA);
return new Computer(p,
  d1, d2);
```
Fluency?
Do something with each name in a collection!

```java
for(int i = 0; i < names.size(); i++)
{
    String name = (String) name.get(i);
    //...
}
```

Not quite fluent.
Do something with each name in a collection!

```
Iterator iter = name.iterator();
while(iter.hasMore())
{
    String name = (String) iter.next();
    // ...
}
```

Still not too fluent.
Do something with each name in a collection!

```java
for(String name : names)
{
    // ...
}
```

That’s fluent.
names.each { name ->
  //...
}

That’s very fluent.
JSONObject json = new JSONObject()
    .accumulate("key1", "value1")
    .accumulate("key2", "value2");
Repeated context in Java

```java
java.util.ArrayList<String> cart = new java.util.ArrayList<String>();
cart.add("Milk");
cart.add("Juice");
cart.add("Apple");

System.out.printf("My cart has %d items.", cart.size());
```
Conceivable approach with method chaining

ChainingArrayList<String> cart = new ChainingArrayList<String>();

cart.
   .add("Milk").
   .add("Juice").
   .add("Apple");

System.out.printf("My cart has %d items.", cart.size());
"With" in Groovy

cart = []

cart.with {
  add "Milk"
  add "Juice"
  add "Apple"

  println "My cart has $size items."
}

"With" in JS

cart = new java.util.ArrayList()

with(cart)
{
    add("Milk")
    add("Juice")
    add("Apple")

    println("My cart has " + size() + " items.")
}
<table>
<thead>
<tr>
<th>Function Type</th>
<th>C</th>
<th>C++</th>
<th>C#</th>
<th>Java</th>
<th>Javascript</th>
<th>Lisp</th>
<th>Haskell</th>
<th>Ruby</th>
<th>Smalltalk</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.1 Function Sequence</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2.1.2 Function Nesting</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2.1.3 Function Chaining</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2.1.4 Higher-Order Functions</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2.1.5 Language Literals</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2.1.6 Operator Overloading</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2.1.7 Meta-Annotations</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2.1.8 Program Generation</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>2.1.9 Macro Programming</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
</tbody>
</table>

The applicability of various internal language patterns in common programming languages. ○ not supported, ● partly possible using workarounds, libraries or language extensions, ● full support.
Embedding approaches
## Taxonomy for pidgin, creole and argot embedded languages

<table>
<thead>
<tr>
<th>Pidgin</th>
<th>○</th>
<th>●</th>
<th>●</th>
<th>A pidgin is a simplified form of the host language. It introduces a new vocabulary and new semantics to the code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creole</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>A creole changes the syntax of the host language (and therefore also the vocabulary) and defines new semantics.</td>
</tr>
<tr>
<td>Argot</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>An argot changes the semantics of the existing language without affecting its syntax.</td>
</tr>
</tbody>
</table>
The remaining combinations

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Vocabulary</th>
<th>Semantics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal Language</td>
<td>○</td>
<td>•</td>
<td>○</td>
</tr>
<tr>
<td>Host Language</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

(New syntax w/o semantics does not make sense.)
SLE meets NLP

In the domain of natural language, a “pidgin” is “a grammatically simplified form of a language used for communication between people not sharing a common language”; a “creole” is “a mother tongue formed from the contact of two languages through an earlier pidgin stage”; an “argot” is a “jargon or slang of a particular group or class” [Jewell and Abate, 2005].
Heterogenous language integration

- At least one of the languages (metalanguage or DSL) is largely, or completely *ignorant of the existence* of the other languages.

- Examples
  - Preprocessors
  - Stratego/XT, TXL
Homogenous language integration

- All languages are specifically designed to work with each other.

- Examples
  - LISP/Scheme Macros, Template Haskell
  - Nemerle, Metalua (Lua), xTc (C)
  - Converge, Diesel
Multiple context-dependent languages

Switching between different languages should be possible at arbitrary points and not enforce the use of special syntactic markers. It should be practicable to **mix and match different language extensions and the host language**. Language changes should be scopable at a fine-grained level, to make it possible to use several otherwise conflicting language extensions in the same compilation unit.
Homogeneous tool integration

A uniform tool set is important to software engineers. To ease the development and use of embedded languages all development activities should happen in the same familiar programming environment of the host language. No special code browser, editors, debuggers or source control should be necessary; all existing tools should continue to work transparently with different languages. To facilitate debugging a precise bi-directional connection between the original source, the various transformation stages and the final executable code should be maintained.

Wanted!

- Homogenous language integration
- Multiple context-dependent languages
- Homogenous tool integration (editors, debuggers, etc.)
Table 2. The transformation of new languages to Jolt is used to transform new languages to Jolt. Even if all languages are built on an object-oriented pattern matcher based on Parsing Expression Grammars that abstract syntax and executable representation for other languages. OMeta [14], a language called Pepsi. Jolt is a Lisp-like language that serves as a common programming paradigm and is bootstrapped in itself using a Smalltalk-like programming facilities targeted at code generation.

2.2 Meta-Programming Systems

Meta-Programming Systems are programming languages that come with meta-programming facilities targeted at code generation. MetaBorg, MontiCore, and Khepera are examples of Meta-Programming Systems. Cola is bootstrapped in itself using a Smalltalk-like programming facilities targeted at code generation.

<table>
<thead>
<tr>
<th>Type</th>
<th>System</th>
<th>Host Language</th>
<th>Pidgin Creole Argot</th>
<th>Multiple Languages Homogeneous Tools Homogeneous Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helvetia</td>
<td>Smalltalk</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>System</th>
<th>Host Language</th>
<th>Pidgin Creole Argot</th>
<th>Multiple Languages Homogeneous Tools Homogeneous Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extensible Compilers</td>
<td>Java Annotation Processing, ableJ</td>
<td>Java</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td></td>
<td>Dryad</td>
<td>Java</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>JastAddJ</td>
<td>Java</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>Polyglot</td>
<td>Java</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td></td>
<td>Xoc</td>
<td>C</td>
<td>✓ ✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>Meta Programming Systems</td>
<td>Cola</td>
<td>(various)</td>
<td>✓ ✓ •</td>
<td>✓ ✓ •</td>
</tr>
<tr>
<td></td>
<td>Converge</td>
<td>(Python)</td>
<td>✓ ✓ •</td>
<td>✓ ✓ •</td>
</tr>
<tr>
<td></td>
<td>MetaOCaml</td>
<td>OCaml</td>
<td>✓ ✓ •</td>
<td>✓ ✓ •</td>
</tr>
<tr>
<td></td>
<td>Scheme</td>
<td>Scheme</td>
<td>✓ ✓ •</td>
<td>✓ ✓ •</td>
</tr>
</tbody>
</table>

*Extensible Compilers* are best described as open toolboxes that provide entry points into the tool chain to extend and change the host language. *Meta-Programming Systems* are programming languages that come with meta-programming facilities targeted at code generation.
<table>
<thead>
<tr>
<th>Type</th>
<th>System</th>
<th>Host Language</th>
<th>Pidgin</th>
<th>Creole</th>
<th>Argot</th>
<th>Multiple Languages</th>
<th>Homogeneous Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>JetBrains MPS</td>
<td>(Java)</td>
<td>.</td>
<td>✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
</tr>
<tr>
<td></td>
<td>Intentional Software</td>
<td>(C#)</td>
<td>?</td>
<td>✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
</tr>
<tr>
<td>Workbenches</td>
<td>openArchitectureWare</td>
<td>Java</td>
<td>.</td>
<td>✓</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>Java Development Tools</td>
<td>Java</td>
<td>.</td>
<td>.</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
</tr>
<tr>
<td></td>
<td>IDE Metatooling Platform</td>
<td>Java</td>
<td>.</td>
<td>.</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
</tr>
<tr>
<td></td>
<td>WholePlatform</td>
<td>Java</td>
<td>.</td>
<td>✓</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>Katahdin</td>
<td>(C#)</td>
<td>.</td>
<td>✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
</tr>
<tr>
<td></td>
<td>Ceteva XMF</td>
<td>(Java)</td>
<td>.</td>
<td>✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
</tr>
<tr>
<td>Language</td>
<td>Khepera</td>
<td>C</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
</tr>
<tr>
<td>Transformation</td>
<td>MontiCore</td>
<td>Java</td>
<td>.</td>
<td>✓</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Systems</td>
<td>MetaBorg</td>
<td></td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
<td>✓✓✓</td>
</tr>
</tbody>
</table>

*Language Workbenches* are characterized by a specialized IDE with a well-defined workflow to specify and use different languages. Language designers are required to follow clearly defined steps to describe syntax, semantics and editor behavior of a new language. *Language transformation* systems define languages through the transformation and composition of language models.
Homogenous embedding with HELVETIA

HELVETIA is an extensible system that intercepts the compilation pipeline of the Smalltalk host language to seamlessly integrate language extensions.
The pidgin shows an improvement over the original Smalltalk codex but our goal

The following code addresses these issues:

rectangle {

label {

shape {

transformation is specified at the AST level using two transformation rules that

is transformed transparently into the code from

implement a

Smalltalk parserx it is not semantically valid. For example numbers do not

1 , 2) - }2 , 1) = rectangle

1 , 1) = label

column = fill.
column = grow.
row = fill.
row = grow.
3.
2.
2.
2.
2.
2.
2.
2.
2.
2.
2.
2.

colspan: 2;
position: 1 , 2;
text: [ :each | each name ];
borderWidth: 1;
borderColor: ublack;
text: [ :each | each name ];

3.
2.
2.
2.
2.
2.
2.
2.

height: 100.
width: 200;

borderWidth: 1.
borderColor: ublack;
text: [ :each | each name ];

5.1 Homogeneous Language Integration

Listing 1

Pidgin: Eliminating syntactic noise.

erent shapes is cumbersome as in this case the host

column is an unknown variable.
Section 4.1

ux the above pidgin example

Methodx and

Listing 2.

Homogenous language integration

The code compilation pipeline showing multiple interception paths.
Helvetia Pros

- Tools continue to work
- Homogenous Embedding
- Relatively simple implementation
- Incremental development of DSLs
Helvetia **Cons**

- Pidgin DSL is constrained by syntax
- Creole DSL requires you to write a parser and specify a transformation
- Language definitions are in “scripts”, there is no real model behind it
In the domain of natural language, a *pidgin* is a grammatically simplified form of a language used for communication between people not sharing a common language.
Floating Point Numbers

digit = "0" | "1" | ... | "9" ;
number = [ "-" ] digit { digit }
       [ "." digit { digit } ] ;
JParsec Parser
(plain internal DSL)

```java
final Pattern digit = Patterns.range('0', '9');

final Pattern number =
    Patterns.seq(Patterns.isChar('-').optional(),
                digit.many(1),
               Patterns.seq(Patterns.isChar('.'),
                            digit.many(1)).optional());
```
PetitParser
(Advanced internal DSL)

digit
    ^ $0 asParser / $1 asParser / ... / $9 asParser

number
    ^ $- asParser optional , self digit plus , ($.
        asParser , self digit plus) optional
Pidgin PetitParser
(Proper embedding)

digit
$0 / $1 / ... / $9

number
$- optional, digit plus, ($.$, digit plus) optional
Pidgin transformations

**DSLTreePattern** new
expression: `'\#literal`
do: [ :ast | `\`(`,\`(ast) asParser) ]

**DSLTreePattern** new
eexpression: `'variable`
do: [ :ast | `\`(\`(self `\`(ast name))) ]

**DSLTreePattern** new
eexpression: `\statement`
do: [ :ast | `\`(`,\`(ast statement)) ]
In the domain of natural language, a creole is a mother tongue formed from the contract of two languages through an earlier pidgin stage.
digit = "0" | "1" | ... | "9" ;
number = [ "-" ] digit { digit } [ "." digit { digit } ] ;
production = identifier "=" choice ";" ;
choice = sequence { "/" sequence } ;
sequence = element { element } ;
element = option / repetition / literal / identifier ;
option = "[" choice "]" ;
repetition = "{" choice "}" ;
literal = "" string "" / "" string "" ;
option = "[" choice "]" ;

```
(` `second) optional)
```
Homogenous tool integration

Traditional Smalltalk debugger with language specific syntax highlighting stepping through an EBNF for parsing.
Quasiquoting
(needed for metaprogramming)

http://101companies.org/wiki/Quasi-quotatation
<table>
<thead>
<tr>
<th></th>
<th>Lisp</th>
<th>Scheme</th>
<th>Template</th>
<th>Haskell</th>
<th>Quasiquote</th>
<th>Smalltalk</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quote</strong></td>
<td>`</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quasiquote</strong></td>
<td>``</td>
<td></td>
<td>`[</td>
<td>...</td>
<td>]`</td>
<td></td>
</tr>
<tr>
<td><strong>Unquote</strong></td>
<td>`,</td>
<td></td>
<td><code>$\{ ... \}$</code></td>
<td></td>
<td>``,</td>
<td></td>
</tr>
<tr>
<td><strong>Splice</strong></td>
<td>`,@</td>
<td></td>
<td><code>$&lt; ... &gt;$</code></td>
<td></td>
<td>``@</td>
<td></td>
</tr>
</tbody>
</table>
Quasiquote example
(Compute code for $x^3$)

\textbf{raise: aNode to: anInteger}
\begin{verbatim}
anInteger = 0
ifTrue: [ ^``1 ].
anInteger = 1
ifTrue: [ ^ aNode ].
^``(`,\(self\) raise: aNode to: anInteger - 1) * `\,aNode)
\end{verbatim}

\textbf{power3: aNumber}
\begin{verbatim}
^`@\(self\) raise:\`\`aNumber to: 3)
\end{verbatim}

aNode would be an AST node for an expression.
Quasiquote decompiled

**raise: aNode to: anInteger**

```smalltalk
anInteger = 0
    ifTrue: [ ^ RBLiteralNode value: 1 ].
anInteger = 1
    ifTrue: [ ^ aNode ].
^ RBMessageNode
    receiver: (self raise: aNode to: anInteger - 1)
    selector: #* arguments: (Array with: aNode)
```

**power3: aNumber**

```smalltalk
^ aNumber * aNumber * aNumber
```
More HELVETIA examples

HELVETIA is an extensible system that intercepts the compilation pipeline of the Smalltalk host language to seamlessly integrate language extensions.

Traditional Mondrian Forms Builder API
(Example of internal DSL)

```smalltalk
aBuilder row grow.  " defines row sizing "
aBuilder row fill.

aBuilder column grow.  " define column sizing "
aBuilder column fill.

aBuilder x: 1 y: 1 add: (LabelShape new
    text: [ :each | each name ];
    borderColor: #black;
    borderWidth: 1;
    yourself).

aBuilder x: 1 y: 2 w: 2 h: 1 add: (RectangleShape new
    borderColor: #black;
    borderWidth: 1;
    width: 200;
    height: 100;
    yourself)
```

A UML package shape in Mondrian.
Issues with the code

1. The variable `aBuilder` is referenced in every rule as an entry point to construct and configure the different parts of the forms.
2. The specification of the cells and their content is repetitive and rather hard to read.
3. The instantiation of different shapes is cumbersome as in this case the host language syntax is rather verbose.
2. The specification of the cells and their content is repetitive and rather hard to read.

3. The instantiation of different shapes is cumbersome as in this case the host language syntax is rather verbose.

The following code addresses these issues:

row = grow.
row = fill.
column = grow.
column = fill.

(1, 1) = label
text: [ :each | each name ];
borderColor: #black;
borderWidth: 1.

(1, 2) - (2, 1) = rectangle
borderColor: #black;
borderWidth: 1;
width: 200;
height: 100.

While the above code is syntactically valid and is parsed by the standard Smalltalk parser, it is not semantically valid. For example numbers do not implement a method, and column is an unknown variable.

A UML package shape in Mondrian.
The transformation for the pidgin

```smalltalk
MondrianPidgin class >> rowColumnTransformation

<transform>
  ^ TreeRule new
  expression: 'row = `@expr';
  expression: 'column = `@expr';
  action: [ :ast |
    ast swapWith: ``(aBuilder
      `, (ast receiver)
      `, (ast at: `@expr')) ]
```

```smalltalk
MondrianPidgin class >> cellTransformation

<transform>
  ^ TreeRule new
  expression: `(`@x , `@y) = `@expr';
  expression: `(`@x , `@y) - (`@w , `@h) = `@expr';
  action: [ :ast |
    ast swapWith: ``(aBuilder
      x: `, (ast at: `@x')
      y: `, (ast at: `@y')
      w: `, (ast at: `@w' ifAbsent: [ 1 ])
      h: `, (ast at: `@h' ifAbsent: [ 1 ])
      add: ``, (Shapes at: (context at: `var') name)
      new `, (ast at: `@expr')) ]
```

The transformation rules are split into two methods. Each of these methods is tagged with the method annotation <transform> (lines 2 and 12), so that the compiler knows that it has to apply these transformations before performing semantic analysis. Each rule consists of two match expressions (lines 4–5 and 14–15) to find particular parse-tree nodes. This functionality is part of the Refactoring Engine [...] and is provided by the host environment. [...] these patterns match the specific constructs we introduced [...].
The transformation for the pidgin

```Smalltalk
MondrianPidgin class>>rowColumnTransformation
  <transform>
    ^ TreeRule new
    expression: 'row = `@expr';
    expression: 'column = `@expr';
    action: [ :ast |
      ast swapWith: `^(aBuilder
        `,,(ast receiver)
        `,,(ast at: `@expr'))) ]

MondrianPidgin class>>cellTransformation
  <transform>
    ^ TreeRule new
    expression: `(\x , \y) = `@expr';
    expression: `(\x , \y) - `(\w , \h) = `@expr';
    action: [ :ast |
      ast swapWith: `^(aBuilder
        x: `,(ast at: `\x')
        y: `,(ast at: `\y')
        w: `,(ast at: `\w' ifAbsent: [ 1 ])
        h: `,(ast at: `\h' ifAbsent: [ 1 ])
        add: `^(\x,(Shapes at: (context at: `\var') name)
          new `,(ast at: `\expr'))) ]
```

The action blocks (lines 6–9 and 16–23) perform a transformation on the matched AST node. For example the first action block transforms expressions of the form `row = grow` into `aBuilder row grow. [...]. Everything that follows the quasiquote meta-character `\` is delayed in execution and represents the AST of the enclosed expression at runtime. Similarly everything that follows the unquote meta-character `, is again executed when performing the code and is used to combine smaller delayed values (e.g., from matched AST nodes) to larger ones. A third operator to compile, evaluate and splice in the result at compile-time is available too, but not used in the examples of this paper.
Creole: A CSS-like syntax

The code above does not follow Smalltalk syntax. At this point, the assumption of a pidgin relying on the host syntax starts to get in our way.

The solution is to allow the definition of a new parser that handles the creole syntax. We typically also want to integrate the new language constructs with the host language or with other language constructs. In our example, the code text: [ :each | each name ] provides such a case in which we parameterize the shape specification with a Smalltalk expression.


© 2012-14, Ralf Lämmel, Software Languages Team, University of Koblenz-Landau, and collaborators (see 1st slide)
The pidgin shows an improvement over the original Smalltalk codex but our goal is to obtain an even more concise CSSylike language as in the listing below:

```smalltalk
class CSSParser

rules = { rule }
rule = selector "{" declarations ""}
selector = #identifier
declarations = declaration { ";" declaration }
declaration = #keywordMessage

CSSParser new parse: 'body
text: [ :each | each name ]
row = fill.
column = grow.
colspan: 2;
position: 1, 2;
width: 200;
height: 100.'
```

This grammar looks very similar to Extended Backus-Naur Form (EBNF) [...]. In fact, it is a DSL for parser generators implemented in Helvetia. As an extension to EBNF we allow productions to reference grammar rules of other languages. The name of external grammar rules are prefixed with a hash character #. For example, the CSS selector is simply a Smalltalk identifier, and the declaration of a property is a keyword message (a Smalltalk method name with arguments, but without receiver) of the host language.
Translator of the Mondrian Creole

We create a new subclass of CSSParser called CSSTranslator, to reuse the abstract grammar definition and to augment it with productions to transform the parse tree nodes to the host language AST [...]. Again we use quasiquoting to build the AST of the host language. Two of CSSTranslator’s parse tree transformations look like in the following listing. The other grammar productions are similarly defined.

```plaintext
CSSTranslator>>rules
  ^ super rules ==> [ :ast | ``(buildOn: aBuilder `,ast) ]

CSSTranslator>>rule
  ^ super rule ==> [ :ast | self transform: (ast at: 'selector')
                    declarations: (ast at: 'declarations') ]
```
Parser registration for the Mondrian Creole

```ruby
MondrianCreole class>>cssParser
  <parse>
  ^ CSSTranslator
```
Highlighting for the Mondrian Creole

```smalltalk
CSSHighlighter>>selector
  ^ super selector == [ :ast | ast -> TextEmphasis underlined ]

MondrianCreole class>>cssHighlighter
  <highlight>
    ^ CSSHighlighter
```
Homogenous tool integration

Traditional Smalltalk debugger with language specific syntax highlighting stepping through a mixture of Smalltalk and the Mondrian creole.
An Argot: Transactional Memory

incrementBy: anInteger
value := value + anInteger

When running the above method from within a transaction, the change is deferred to the end of the transaction, instead of incrementing the variable immediately. This allows the system to check for conflicts and revert the changes if necessary. Thus, even if the source code looks exactly the same, its behavior changes.
Specifying the Transactional Memory Argot

```olang
Object class>>transformAtomic
  <attribute>
    ^ ConditionRule new
      if: [ :context | context isTransactional ]
      then: (TreeRule new
        expression: '\receiver \msg: \args' do: [ :ast |
          ast swapWith: '\((',(ast at: '\receiver')
          ,('__atomic__', (ast at: '\msg:'))
          ,(ast at: '\args') ) ];
        expression: '\var := \expr' do: [ :ast |
          ast swapWith: '\(',(self
            atomicInstVarAt: ',',(ast binding index)
            put: ',',(ast at: '\expr')) ) ];
        expression: '\var' do: [ :ast |
          ast swapWith: '\(',(self
            atomicInstVarAt: ',',(ast binding index)) ) ])
```

The code uses the <attribute> method annotation (line 2), to tell the compiler that the rules are expected to run after the symbols have been resolved (attributed). Line 4 makes sure that the transformation is only performed when compiling code for the transactional context. Lines 5–16 implement the actual transformations.
Registration of the argot

Object class>>compileTransactional: aContext
    <parser>
    aContext isTransactional ifFalse: [
        aContext copy
        beTransactional;
        perform ].
    ^ nil

To trigger the compilation of a transactional and a non-transactional version of every method we hook into the parser using the <parser> annotation. We copy the compilation context and spawn a new compilation path for the transaction context.
Wrap-up
Summary

• DSL implementation concerns syntax, semantics, debugging, highlighting, traceability, and possibly others.
• DSLs may be implemented internally, externally, or via embedding.
• Programming environments get increasingly capable of non-trivial embedding approaches.
You are free:

- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

Under the following conditions:

- **Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

- **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.