Domain Specific languages (DSLs)

[...] a good programmer in these times does not just write programs. [...] a good programmer does language design, though not from scratch, but building on the frame of a base language. -Guy Steele Jr.

Definition

*A **Domain Specific Language** is a computer programming language focused on a particular domain -Martin Fowler*

DSLs - a first example

An example of a DSL is an Antlr grammar for integers, comments and lower case letters.

```
integer : (HEX_PREFIX | OCTAL_PREFIX)? DIGITS;
DIGITS : '1'..'9' '0'..'9'*;
ML_COMMENT : '/*' ( options {greedy=false;} : . )* '*/' ; .
NO_LOWERCASE_LETTERS : (~('a'..'z'))+;
```

DSLs - a second example

Here, we create an SQL table, insert and select records from it.

```
CREATE TABLE Persons
(
PersonID int,
LastName varchar(255),
FirstName varchar(255),
Address varchar(255),
City varchar(255)
);

insert into Persons values(1,'Eichberg','Michael','Hochschulstr. 10','Darmsta
select Address from Persons where LastName='Eichberg';
```

Other examples of DSLs are : HTML, SQL, Yacc, Antlr ...

What distinguishes DSLs is their high expressiveness inside the boundaries of a particular domain and limited expressiveness outside the boundaries of that domain.

So, a single project may use many DSLs in different parts where every language is targeted towards a specific domain.

Reasons for using DSLs

- A DSL is designed with a *domain expert* in mind. Hence, it is easier for domain experts, who are usually not software developers, to understand DSL code than code in a general purpose programming language(GPL).
- The code is (up to a certain level) self-documented.
- It may be easier to apply formal methods on DSLs as reasoning can be done within the domain semantics.

Desired properties of DSLs

- Precise representation of domain concepts
- Composability with other DSLs/GPLs
- Reasonable performance
- Infrastructure reusability (parser, compiler, linker, ...)
- High modularity (i.e., modular checking and compilation)
- Static (type) safety
- Tool support for debugging/syntax highlighting/code completion

Tradeoffs when building a standalone DSL

When building a classical (standalone) DSL you need to implement your own parser/compiler (basically the whole tool suite). However, the DSL can be tailored to the domain's needs.

DSLs Takeaway

	DSLs
Precise representation of domain concepts	++
Tool support	+/-
Infrastructure reuse	--
Modularity	-
Composability	--
Static safety	+/-

7

A standalone DSL is excellent for representing domain concepts. An example is ANTLR.

Developing tools is usually a lot of work, but there are basically no limits.

There is no infrastructure reuse as the whole "package" (grammar/parser/compiler/...) has to be built from scratch.

Composability is typically not supported.

Embedding of a DSL

- Example: SQLj (An embedding of SQL in Java)

```
sql private static iterator EmployeeIterator(String, String, BigDecimal);
...
EmployeeIterator iter;
#sql [ctx] iter = {
  SELECT LASTNAME
       , FIRSTNME
       , SALARY
    FROM DSN8710.EMP
   WHERE SALARY BETWEEN :min AND :max
};
do {
  #sql {
    FETCH :iter
     INTO :lastname, :firstname, :salary
  };
  // Print row...
} while (!iter.endFetch());
iter.close();
```


- The DSL has its own syntax which is different from the syntax of the host language.
- A special compiler is needed to compile/to preprocess the code. Afterwards it is passed to the host language compiler.
- Tools need to be adapted.

Embedding of DSLs Takeaway

Heterogenous Embedding

	DSLs	Embedding of DSLs
Precise representation of domain concepts	++	o
Tool support	+/-	-
Infrastructure reuse	--	o
Modularity	-	o
Composability	--	-
Static safety	+/-	+

Embedding of a DSL in a host language usually provides some level of static type safety.

Embedding of a DSL in a host language will make it not composable as it has its own compiler/interpreter that differs from that of the host language.

The tool support is just for the host language not for the embedded language.

Embedded Domain Specific Languages (EDSLs)

Embedding a DSL into an already existing general purpose programming language such that it only uses the GPLs mechanisms will greatly reduce the implementation effort.

The embedded language will make use of the features of the host language.

But on the other side embedding the DSL as a library makes it **tightly coupled** to the host language.

There are many approaches for embedding a DSL into a general purpose language.

Embedded Interpreter

Make use of the host language parser and compiler by embedding the DSL using host language expressions.

The written syntax is interpreted by a host language interpreter

This is known as the **Interpreter pattern**

Pure Embedding

Similar to a traditional library approach, however, domain specific semantics are taken into consideration while constructing the language. In particular, domain types are implemented as host language types.

However the host language and DSL are tightly coupled.

Embedded DSLs Takeaway

	DSLs	Embedding of DSLs	Embedded DSLs
Precise representation of domain concepts	++	o	-
Tool support	+/-	-	+/-
Infrastructure reuse	--	o	++
High Modularity	-	o	o
Composability	--	-	+
Static safety	+/-	+	+/-

Embedded DSLs are bound by the syntax of the host language which limits the possibility to precisely capture the domain concepts.

Concerning tool support, there is tool support (syntax highlighting, code completion,...) for the host language but there **is no** tool support for the embedded DSL (e.g. slick).

The static safety depends on the type safety of the host language. So if the host language is dynamically typed then there is no type safety for the EDSL.

Host Languages used for Building DSLs

- Many of today's general purpose languages facilitate the development of EDSLs. For example: (Java,) Haskell, Scala,...
- Scala is particularly well suited due to its rich set of advanced features: path-dependent types, mixin composition, syntactic flexibility, advanced type system, ...

Case Study- The *Regions* language

- Derived from the image processing domain.
- It is basically for one domain specific type **Regions**.
- Along with operations required for describing regions and building more complex regions.
- For demonstration purposes, the language is kept simple. However, it can be easily extended without changing the basic language structure.

Not relevant for the exam.

The *Regions* language - cont

```
trait Regions{

   type Vector = (double,double)
   type Region

   def univ : Region
   def empty : Region
   def circle : Region
   def scale(v : Vector, x : Region) : Region
   def union(x : Region, y : Region) : Region
}

def program(semantics : Regions) : semantics.Region ={
   import semantics._

   val ellipse24=scale ((2,4), circle)
   union (univ,ellipse24) //The returned expression
}
```

Not relevant for the exam.

Here two type members are declared:

- **Vector** as a pair of doubles
- **Region** as an abstract domain type that will be defined in subclasses.

Beside type members, operations are defined to represent

- empty regions
- universal regions: that include every point
- circle to represent a unit circle aroung the origin of the coordinate space
- scaling a region by a by a vector v resulting in a new scaled region
- and the union between regions

A client of the Regions interface is defined having a dependent return type that depends on the **semantics** implementation of type

Various semantics for the *Regions* type

After defining the trait of the Regions language, we can now define various semantics that extend this trait and implement abstract types and operations.

An example of the operations are :

- checking if a point is inside or outside a region
- pretty printing of region operations

Not relevant for the exam.

The *Evaluation* trait

```
trait Evaluation extends Regions{

   type Region = Vector => boolean

   def univ : Region = p => true
   def empty : Region  = p => false
   def circle : Region = p => p._1*p._1 + p._2*p._2 < 1
   def scale(v : Vector, x : Region) : Region = p => x(p._1/v._1,p._2/v._2)
   def union(x : Region, y : Region) : Region = p => x(p) || y(p)
}

object Eval extends Evaluation
```

Not relevant for the exam.

A region is seen as a set of points

Points inside the region are mapped to *true* and those outside are mapped to *false*

For example, the universal region contains all points so it will always output true. On the other hand, the empty region does not contain any point so it will always output false.

Also in circle, a point is seen is a part of the unit circle if the square sum of its components is less than or equal one.

In union, we can use characteristic functions of regions we want to combine to map all points to true which are mapped to true by any of the composed regions.

The pretty printing trait

```
trait Printing extends Regions {

type Region = String

   def univ : Region = "univ"
   def empty : Region = "empty"
   def circle : Region = "circle"
   def scale(v : Vector, x : Region) : Region =
      "scale(" + v + ", " + x + ")"

   def union(x : Region, y : Region) : Region =
      "union(" + x + ", " + y + ")"
}

object Print extends Printing
```


Mapping the domain type *Regions* to a string and ouputing the corresponding string representation for every operation.

Optimizing the *Regions* DSL

```
trait Optimization extends Regions {

   val semantics : Regions

   type Region = (semantics.Region, boolean)

   def univ : Region = (semantics.univ, true)
   def empty : Region = (semantics.empty, false)
   def circle : Region = (semantics.circle, false)

   def scale(v : Vector, x : Region) : Region =
      if (x._2) (semantics.univ, true)
      else (semantics.scale(v, x._1), false)

   def union(x : Region, y : Region) : Region =
     if (x._2 || y._2) (semantics.univ, true)
     else (semantics.union(x._1, y._1), false)
}

// prints "union(univ, scale((2.0,4.0), circle)"
println(program(Print))

object OptimizePrint extends Optimization { val semantics = Print }

// prints "(univ, true)"
println(program(OptimizePrint))
```


The *Region* here is represented as a pair of unoptimized region and a boolean value that indicates whether this region is statically known to be universal or not

For example, in the *union* operation, if any of the regions is the universal region then the resulting region is universal.

In the Optimization trait, the actual optimization is done in the *union* and *scale* operations. Where these operations can make use of the additional information given in the *univ, empty and circle* regions to shorten evaluation paths.

Since the Optimization trait can work on any region semantics, therefore, it can work with the Pretty printing and optimization semantics defined before.