Domain-Specific Programming Languages

Torben Mogensen

Programming Language Design

GPL Versus DSL

General-purpose languages (GPLs) are programming languages intended for a broad range of problem domains, and typically for use by trained programmers.

Domain-specific languages (DSLs) are programming languages intended for either a narrow range of problem domains or for use by a certain type of non-programmers (and sometimes both).

Mostly a matter of degree: Few languages are completely general purpose, nor are they normally useless outside a single, narrow domain.

Why Should You Design a DSL?

- A DSL can potentially increase productivity within the domain.
- A DSL can use notation and concepts known to domain experts, which can help domain experts to understand programs and allow them to write simple programs.
- A DSL can be written to either guarantee certain properties or make it possible to analyse programs to determine these.
- An implementation of a DSL can employ domain-specific optimisations not usually found in implementations of GPLs.
- A program written in a DSL can have multiple different well-defined semantics.
- Training non-programmers to use a DSL may be easier than training them to use a GPL.
- If some element of the problem domain changes, it may be sufficient to handle this in the implementation of the DSL rather than having to rewrite all programs. Example: Dates.

Why Should You *Not* Design a DSL?

- The problem domain may not be specific enough or may not have features that lend themselves to domain-specific notation or optimisation.
- Designing, implementing and maintaining a DSL is a significant investment.
- Interfacing with other software may be difficult.
- If the intended users already know a GPL, training them to use a DSL is an extra investment.
- A program with command-line options or menu choices may suffice.

How do You Design a DSL?

1. Identify the problem domain.
2. Collect domain knowledge.
3. Domain analysis and theory building.
4. Concrete language design.
5. Implementation.

May need iteration. Should always consider the needs and knowledge of the user over that of the designer / implementer. Nevertheless, good knowledge of language design and implementation needed to avoid pitfalls.

Before You Continue

Exercise 11.1:

- Discuss to what extent JSON and XML can be considered DSLs.
- Discuss to what extent a notation for regular expressions is a DSL.

How do You Implement a DSL?

Embedded language: The language is represented within an existing language, either as data structure or program structure. Type system of host language can help ensure syntactic correctness of embedded programs.

Preprocessor: Domain-specific features are expanded by a preprocessor into existing features in an existing language, leaving all other parts unchanged.

Compiler/interpreter modification: A compiler or interpreter for an existing language is modified to add, modify or remove features from original language as needed.

Stand-alone language: Implemented using a traditional interpreter or compiler with its own parser, type system etc.

Example DSL: SQL SELECT Statement

Grammar for subset of SQL SELECT statement:

SelectStatement	→	`SELECT` *Columns* `FROM` **TableName** *WhereClause*
Columns	→	**ColumnName**
Columns	→	*Columns* , *Columns*
WhereClause	→	`WHERE` *Condition*
WhereClause	→	
Condition	→	*Value* **RelOp** *Value*
Condition	→	*Condition* `AND` *Condition*
Value	→	**ColumnName**
Value	→	**Constant**

An example of a `SELECT` statement is

```
SELECT name FROM students WHERE enrollmentYear < 2015
```

Implementation as Embedded Language

Implement inside existing language.

+ It is usually easier to make a suitable data structure or program structure within an existing language than it is to make or modify an interpreter or compiler.
+ A programmer using the DSL can use abstraction and structuring features of the GPL around the DSL parts.
+ It is easy to interface programs written in the DSL with other programs.
– You are limited to use the syntax of the implementation language.
– Error messages are often expressed in terms of the underlying implementation language and not in terms of concepts from the DSL.
– It may be impossible to statically verify domain-specific properties, so these may have to be checked at runtime.

Representing Syntax in Embedded Language

- As a string, e.g.,
  `"SELECT name FROM students WHERE enrollmentYear < 2015"`
  - Readable syntax, but requires parser
  - Syntax not checked at compile time
  - Possible to build syntax at run time
  - SQL injection!
- As a datatype (as in ML, F#, or Haskell), e.g.,

```
SELECTFROM (["name"], "students",
            WHERE (Column "enrollmentYear" << Number 2015))
```

  - Fairly readable syntax, no parser required
  - Syntax checked at compile time (type system of host language)
  - Possible to build syntax at run time
- Using method chaining, e.g.,

```
Database("students").
    WHERElt(Column("enrollmentYear"),Number(2015)).SELECT("name")
```

  - Less readable syntax
  - Syntax checked at compile time (type system of host language)
  - No parser or match/case-statement required
  - Not possible to build syntax at run time

Implementation by Preprocessor

+ Domain-specific notation can (to some extent) be used.
+ Simple domain-specific properties can be verified statically.
+ Compile-time errors can be expressed in terms of domain-specific concepts – as long as the user only uses domain-specific features.
+ A programmer using the DSL can use the abstraction and structuring features of the GPL around the DSL parts.
+ It is easy to interface programs written in the DSL with other programs.
– Run-time errors are in terms of expanded code, not original code.
– Unless full parser is written, syntax errors reported by preprocessor may be primitive.

Implementation by Compiler/Interpreter Modification

+ Domain-specific notation can fairly freely be used.
+ More complex domain-specific properties can be verified statically.
+ Both compile-time and run-time errors can be expressed in terms of domain-specific concepts.
+ A programmer using the DSL can use the abstraction and structuring features of the GPL, but restrictions can can be imposed.
+ It is easy to interface programs written in the DSL with other programs.
– Requires access to source code of compiler/interpreter.
– Tie-in to specific (eventually obsolete?) compiler/interpreter.

Implementation as Stand-Alone Language

Complete, traditional implementation.

+ Full domain-specific notation can be used, including diagrams and complex formula notation.
+ Domain-specific properties can be guaranteed or verified statically.
+ Errors can be expressed in terms of domain-specific concepts.
+ More advanced domain-specific optimisations can be used.
+ It is possible to implement multiple different semantics for the same language.
– Requires implementation and maintenance of a full compiler or interpreter.
– Interfacing with other programs may be difficult. It is usually not difficult to make other programs call programs written in the DSL, but the converse may compromise domain-specific properties that are otherwise guaranteed or verifiable.

Before You Continue

Exercise 11.2:

The DSL that has arguably been most successful in being used by non-programmers is Excel, the spreadsheet program in Microsoft's Office package.

Discuss which properties of Excel makes it suitable for non-programmers, and which properties that might hinder this.

Examples of DSLs

Scratch: A language for teaching children to program.

TEX and LATEX: Languages for type-setting.

Graphviz: A language for writing graph-like diagrams.

Troll: A language for rolling dice and calculating probabilities thereof.

CRL: A language for cashflow reengineering.

Futhark: A language for programming graphics processors.

Scratch

Intended for teaching children how to program.

- Visual LEGO-inspired syntax.
- Problem domain is making interactive 2D animations and games.
- Nevertheless, has advanced features such as concurrency and recursion.

TEX and LATEX

TEX was designed by Donald Knuth to typeset "The Art of Computer Programming". Includes Turing-complete macro language – a BASIC interpreter has been programmed in TEX.

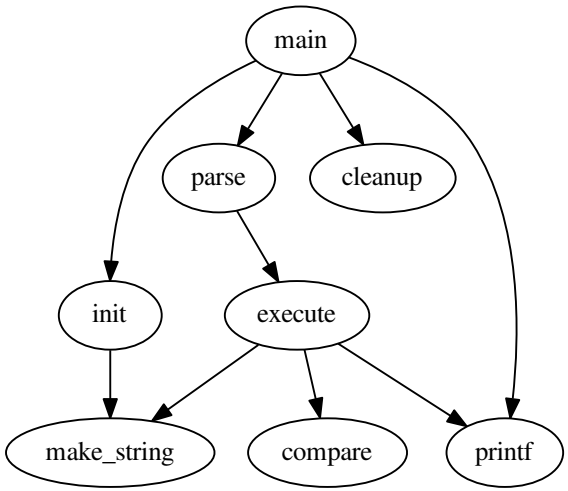
LATEX built on top of TEX by Leslie Lamport. Adds features for structuring a document. It is more widely used than raw TEX.

LATEX is the de-facto standard for writing articles, reports, and textbooks in the natural sciences, but has also been used to typeset music notation, circuit diagrams, role-playing games, and much more.

Graphviz

Graphviz (`https://graphviz.org/`) is a domain-specific language for visualising trees and graphs: You describe a graph using textual notations, and Graphviz will convert this to an image.

```
digraph G {
  main -> parse -> execute
  main -> {init, cleanup}
  execute -> {make_string, printf, compare}
  init -> make_string
  main -> printf
}
```



Troll

A language for rolling dice and calculating probabilities thereof (two different semantics for one language).

Dice can be combined in a myriad of ways. Examples:

- Count the number of different results obtained when rolling five ten-sided dice:

```
count different 5d10
```

- Generate random attributes for D&D:

```
"Str |>Dex|>Con|>Int|>Wis|>Chr"
|| 6'sum largest 3 4d6
```

See more at `https://topps.diku.dk/~torbenm/troll.msp`.

CRL

A language for creating financial papers (loans, etc.) from others. Example:

```
declarations
  fraction = 0.7
input
  cashflow a, b
structure
  c = a + b
  (d,e) = Divide(c, fraction)
output
  cashflow d,e
```

Uses *linear types* to prevent multiple use of the same money: Every variable is used *exactly* once.

Futhark

Functional language for doing parallel computations on graphics processors. Not intended for complete programs, but for compute-intensive cores that are called from other programs.

Uses `map`, `filter`, `reduce`, and `scan` or equivalent functions expressed as loops. Uses *uniqueness types* to allow arrays to be destructively overwritten.

Matrix multiplication:

```
fun main(x: [n][m]i32, y: [m][p]i32): [n][p]i32 =
  map (\xr -> map (\yc ->
                     reduce (+) 0 (zipWith (*) xr yc))
                  (transpose y))
  x
```

Before You Continue

Which DSLs not mentioned in the slides do you know, and what makes them domain-specific?