A tutorial on Lisp extensibility
Impact on DSL design and implementation

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/~didier

ILC 2012 – Sunday, October 21st
Introduction
The Challenges in DSL design and implementation

- Orthogonal expertise
  - Application domain
  - Language design and implementation

- DSLs vs GPLs
  - ≠ syntax
  - ≠ semantics

- DSLs and GPLs need to be completely different.

Really?
Table of contents

1 DSL Overview
   - Taxonomy of DSLs
   - Extensibility at a glance

2 A concrete example
   - Building the DSL
   - Externalizing it

3 Conclusion
   - Discussion
Taxonomy of DSLs

[Fowler, 2005, Tratt, 2008]

- **Standalone / External**
  - Full language
  - Needs parser, compiler / interpreter
  - Lex / Yacc, ANTLR etc.

- **Embedded / Internal**
  - **Heterogeneous**
    - Program transformation outside the host language
    - Stratego, Silver etc.
  - **Homogeneous**
    - Extended host language
Extensibility at a glance I
[van Deursen et al., 2000, Vasudevan and Tratt, 2011]

- User-defined data types and (infix) operators
  Algol 68 [Denert et al., 1975, Pagan, 1979]

- Operator overloading
  C++ [McNamara and Smaragdakis, 2000]

- Compile-Time Meta-Programming
  C++ templates [Prud’homme, 2006]
  Template Haskell, Meta OCaml [Czarnecki et al., 2004]
  Meta Lua, Converge, Nermerle [Tratt, 2005, Fleutot and Tratt, 2007, Skalski et al., 2004]...

- Functional languages (Haskell, ML)
Extensibility at a glance II
[van Deursen et al., 2000, Vasudevan and Tratt, 2011]

- Forth: Operator extensibility + CTMP
  [Ahson and Lamba, 1985]
- Scala [Rompf et al., 2011]
- Marginally: TeX, m4
Example
Command-line options highlighting

Usage: advanced [-hd] [+d] [OPTIONS] cmd [OPTIONS]

Available commands: push pull.
Use 'cmd --help' to get command-specific help.
- **h**, --help
  Print this help and exit.
- **(+d)**, --debug[=on/off]
  Turn debugging on or off.
  **Fallback**: on
  **Environment**: DEBUG

- Properties (bold, underline, foreground color...)
- Faces (localized property set)
- Themes (face trees)
Step 1: the basic thing

```lisp
(setq default-theme
  (make-instance 'face :name 'toplevel
    :background 'black
    :subfaces (list (make-instance 'face :name 'option
          :foreground 'white
          :subfaces (list (make-instance 'face :name 'syntax
                    :bold t
                    :foreground 'cyan)
                (make-instance 'face :name 'usage
                    :foreground 'yellow))))))
```

Problems with `make-instance`:

1. Class name exposed
2. Name argument optional
Step 2: an instantiation wrapper

```lisp
(setq default-theme
  (make-face 'toplevel
    :background 'black
    :subfaces (list (make-face 'option
                     :foreground 'white
                     :subfaces (list (make-face 'syntax
                                      :bold t
                                      :foreground 'cyan)
                        (make-face 'usage
                                      :foreground 'yellow))))))
```

Problems:

1. Explicit `toplevel` face name
2. Explicit creation of sub-faces `list`
3. Lots of calls to `make-face`
4. Lots of quoting
Step 3: a theme creation wrapper
Solution to problem #1

```lisp
(setq default-theme
  (make-theme
   :background 'black
   :subfaces (list (make-face 'option
                      :foreground 'white
                      :subfaces (list (make-face 'syntax
                                         :bold t
                                         :foreground 'cyan)
                                      (make-face 'usage
                                                      :foreground 'yellow))))))
```

Problems:

1. **Explicit toplevel face name**
2. **Explicit creation of sub-faces list**
3. **Lots of calls to make-face**
4. **Lots of quoting**
Extensibility
Didier Verna

DSL Overview
Taxonomy
Extensibility
Example
Internal
External
Conclusion
Discussion

Step 4: CLOS and the MOP
Solution to problem #2

Generic functions, methods

(defmethod func ((arg1 class1) arg2 ...) body)

- Methods are *outside* the classes (ordinary function calls)
- Multiple dispatch (multi-methods)

The CLOS Meta-Object Protocol (MOP)

- **CLOS itself** is object-oriented
  - The CLOS MOP: a *de facto* implementation standard
  - The CLOS components (classes *etc.*) are (meta-)objects of some (meta-)classes

- *initialize-instance* is a generic function
Step 4: CLOS and the MOP
Solution to problem #2

Problems:
1. Explicit `toplevel` face name
2. Explicit creation of sub-faces list
3. Lots of calls to `make-face`
4. Lots of quoting
Step 5: syntax extension
Solution to problem #3

- **readtable**: currently active syntax extensions table
- **macro character**: special syntactic meaning
- **reader macro**: implements macro character behavior
Step 5: syntax extension
Solution to problem #3

Problems:

1. Explicit toplevel face name
2. Explicit creation of sub-faces list
3. Lots of calls to make-face
4. Lots of quoting
Step 6: macros
Solution to problem #4

- Ordinary Lisp functions
- Work on chunks of code (as data)
- Transform expressions into a new expression
- Compile-time effect
- Control over evaluation
Step 6: macros
Solution to problem #4

Problems:

1. Explicit top-level face name
2. Explicit creation of sub-faces list
3. Lots of calls to make-face
4. Lots of quoting

```lisp
(setq default-theme
  (define-theme
   :background black
   :face { option :foreground white
    :face { syntax :bold t :foreground cyan }
    :face { usage :foreground yellow }
   }
  ))
```
Final result

Looks pretty much like a DSL to me...

```plaintext
;; My personal theme with so cool colors

:background black
:face {option :foreground white
    :face {syntax :bold t :foreground cyan}
    :face {usage :foreground yellow}}
```

- read, eval and possibly compile
Conclusion

- Impact of GPL on DSL design and implementation
- Key GPL aspect: extensibility
- Embedded homogeneous approach
  - A single language
  - DSL infrastructure smaller
  - DSL both internal and external
- Common Lisp
  - Functional, Imperative, Object-Oriented
  - MOP
  - CTMP
  - Syntax extension
  - read, eval, compile
Suboptimal syntax **ok but...**

**Not ok:**
- [Fowler, 2010]: “external DSLs have their own custom syntax and you write a full parser to process them”
- [Kamin, 1998, Czarnecki et al., 2004]: “a prerequisite for embedding is that the syntax for the new language be a subset of the syntax for the host language”
- BTW, same disagreement at the semantic level (MOP)

**Poor error reporting**
- Research: [Tratt, 2008]
- Lisp: ? (but Cf. condition system & restarts)
Controversial aspects of extensibility

- **Dynamic typing**
  - **pros:** end-user friendly
  - **cons:** run-time type errors / checking
  - Research: [Taha and Sheard, 1997]
  - Hybrid languages (Cf. Racket)

- **Lazy Evaluation**
  - **pros:** infinite data structures, new control primitives *etc.*
  - **cons:** pure functional languages only
  - Lisp: lazyness through macros (not as straightforward), but side-effects for free, and still functional.
The root of (Lisp) extensibility

- Reflexion
  - Introspection
  - Intercession
- Implementation
  - By API
  - Structural reflexion (program)
  - Behavioral reflexion (language)

Didier Verna

Bibliography II


Bibliography IV


