COSC345 Week 2

Little Languages

3 and 6 March 2015

Richard A. O’Keefe
What are they?


Why are they?

Reduce the semantic gap.

Reduce the labour of coding.

Reduce the cost of debugging.
Reduce the semantic gap

Requirements are stated in businessese. There are those who say that businessese is not a human language.

Specifications are stated in some formal or semi-formal notation. With luck, it has a semantics, so checkable.

Code is written in programming languages, the semantics of which is not related to the application domain.

The gaps are large; translation is costly and error-prone.
Reduce the labour of coding.

The less we write, the less we wrong.

Typing takes time.

If we can shrink the amount to be written, we may be able to reduce direct costs and errors.

Error diagnosis may be easier to do or understand.
Bettini’s Example (XML)

<!-- 2 + 5N lines to describe N people -->

<people>
    <person>
        <name>James</name>
        <surname>Smith</surname>
        <age>50</age>
    </person>
    <person employed="employed">
        <name>John</name>
        <surname>Anderson</surname>
        <age>40</age>
    </person>
</people>
Processing the XML version

Using an uncommonly simple XML interface:

```smalltalk
(OxusParser parseFile: 'bettini.xml')
descendants: #person do: [:person |
    family_name := (person firstElementChild: #surname) text.
given_name  := (person firstElementChild: #name)   text.
age         := (person firstElementChild: #age)    text asInteger.
employed    := person hasElementChild: #employed.
...].
```
Bettini’s example (AWK)

Smith, James;50
Anderson, John;40;employed

Read in AWK using

BEGIN { FS = ";" }
{
    full_name = $1; age = $2; employed = $3 == "employed"
    split(full_name, parts, /, /)
    family_name = parts[1]; given_name = parts[2]
    ... 
}
What did we just see?

A general purpose data language (2+5N) lines and a special purpose one (1N).

The general purpose language lets us say a lot of things we did not mean.

The special purpose one only just lets us say what we want; oodles of mistakes are impossible.

The general purpose data language requires a complex parser and complex processing.

The special purpose one is trivial to process.

Lower data entry costs!
Reduce the cost of debugging.

Error rate approximately language independent

About 1 error per 25 lines before checking

Fewer lines means fewer mistakes.

Debugging at high semantic level.
Another example.

I have a file DARWIN of UNIX functions with short glosses and an indication of which systems I know have them.

43--S---  pthread_barrierattr_setpshared  Barrier private or shared?
43DLSUBC  pthread_cancel         Kill a thread.

Task: “Find which functions are in the 4th edition of the Single Unix Specification but not in Mac OS X (Darwin)”.

Execute from the shell:

awk '/^4.-/ {print $2}' DARWIN

My best Java code is 22 lines and had 3 mistakes.
Example : make(1)

Make basically has definitions

Variable=Text

and rules

Targets: Sources
    Commands

These are IF-THEN rules: if any target is missing or older than some corresponding source, execute the commands to bring it up to date.
“A warning take by me”

Little languages are easy to copy

— DOS/Windows nmake is basically make

— but different

Little languages may be easy to extend

— include file is popular

Adding warts to a diamond...

— GNU make.
Example: lex(1)

Write lexical analysers for little languages or even general programming languages, using regular expressions.

A lex program to strip out XML. Definitions, then pattern-action rules.

N [._a-zA-Z][-_.:a-zA-Z0-9]*
S [ \t\n]+
V (["\]^"\)*["\] | [’][^’]*[’])
%
"<"{N}({S}{N}][-{S}?={S}?{V})*{S}?[/]?">" {} 
"</"{N}{S}?">" {} 
"<!--"(-?[^-])*"-->" {} 
"&#"[0-9]+;" {putchar(atoi(yytext+2));}
"\n"| . {putchar(yytext[0]);}
Example: yacc(1)

Write parsers for little languages or even general programming languages, using unambiguous context-free grammars. Token definitions, then pattern-action rules.

http://www.cs.otago.ac.nz/cosc345/resources/json.htm is a grammar for a little programming language that has JSON as a sublanguage. The pure JSON rules are in blank; the extensions are blue.
Example: Forth

Invented by Chuck Moore

Powerful, tiny interpreter

Unconventional

Preceded by SNIBBOL (MINT)

Forth is at the heart of OpenFirmware (non-x86 machines, said to beat EFI).

Postscript is related.
Example: the shell, sh(1)

Only one data type: string.

Specialised syntax for running programs.

Structured control statements.

Originally no functions.

But can run scripts as programs.

Special ${...}$ embedded sublanguage.
Example: awk(1)

The classic UNIX scripting language

BEGIN { actions } initialisation

END { actions } summary actions

function f(args) { actions } functions

pattern { actions } per-record.

gvpr(1) AWK like but streams of graphs
Example: dot(1)

Describes directed or undirected graphs

Graphs have names and attributes

Nodes have names and attributes

Edges have labels and attributes

Layout is done by dot(1), neato(1), circo(1), ...

http://www.cs.otago.ac.nz/cosc345/resources/inc.htm
Example : pic(1)

A declarative notation for box-and-line diagrams.

Embedded in Troff or TeX or convert to HTML

gpic(1) is part of the groff distribution

http://www.cs.otago.ac.nz/cosc345/resources/er.htm


Much easier for me than hand drawing.

Easy to generate from a script.

See also grap(1), chem, gnuplot
Category : DATA languages

Instances define a particular data value

JSON www.json.org

XML www.w3.org/TR/REC-xml/

RDF www.w3.org/RDF/

dot www.graphviz.org/content/dot-language

Category : Data description languages

Items define data (sub)languages

ASN.1

Google protocol buffers

Avro, Thrift (rivals of protobufs)

SGML/XML Document Type Descriptors

XML Schemas

JSON Schemas json-schema.org/

...
Category: Query Languages

- select
- project
- aggregate
- transform
- typically not join
- on streams of records, trees, graphs
- XPath, XSLT, XQuery, SQL
Category: Pattern/Action languages

Much overlap with query languages

lex(1), yacc(1), awk(1), gvpr(1)

CSS, arguably

Arguably make(1)

ant and maven, maybe?

Package dependency rules
Category: programming languages

Forth

Scheme (50-page manual)

TeX

Lua (extension language)

TCL (Tool Control Language)

Various game extension languages
Category: ad hoc scaffolding

http://www.cs.otago.ac.nz/cosc345/json.d/dis.c

209 lines of repetitive C

http://www.cs.otago.ac.nz/cosc345/json.d/dis.awk

44 lines of simple AWK to generate it

Writing a generator was 21% of the work.

http://www.cs.otago.ac.nz/cosc345/json.d/run.c

864 lines, 663 so far *should* be generated!
How it can save work

There were several bugs in dis.c

— Fix *one* place in dis.awk, fix all!

The VM instruction set is not finalised.

— Change instruction list, regenerate!

Debugging easier; maintenance easier.
Free-standing vs embedded

Free-standing needs own data and control structures.

Free-standing gets own syntax.

Free-standing needs own tracing, debugging, ...

Embedded shares host data and control structures.

Embedded has to live within host syntax.

Embedded shares host tracing, debugging, but

in host terms, not DSL terms.
Example: free-standing XML.

<html>
  <head><title>Greeting</title></head>
  <body>
    <h1>Hello world!</h1>
    <p>Here is one paragraph.</p>  
    <p>And another.</p>
  </body>
</html>
Example: hosted XML (Scheme)

`'(html
  (head (title "Greeting"))
  (body
    (h1 "Hello world!"
    (p "Here is one paragraph."
    (p "And another.")))

30
What hosting buys you:

(define (book-by-category c i n)
  (define (book b)
    '(li "AUTHOR: " ,(child-text 'AUTHOR b) (br)
      "TITLE: " (i ,(child-text 'TITLE b))))

    '( (h3 "CATEGORY: " ,(first c))
      (ol
       ,(map book (rest c))))))
We can even embed in C

if (want_e_mail_and_or_id >= 0) {
  . <a name=value(p, "id", 0) id=value(p, "id", 0)>
  . ^first_child_named(p, "name")
  . </a>
} else {
  . ^first_child_named(p, "name")
}
Costs

Work to develop DSL.

Work to maintain DSL.

Documentation and training for users.

There is no silver bullet.

Silver bullets don’t work that well anyway.

You don’t have to do it all by hand...
Modern demands for DSLs:

- syntax colouring
- syntax checking/hints/completion in editor
- navigate to definition/documentation
- automatic build
- debugging, including tracing
- and profiling
How to satisfy those demands?

Embedding

DSL toolkit like Xtext

Metaprogramming toolkit like Rascal

See http://www.rascal-mpl.org
Where next?

Team organisation

HTML is a little (data) language

CSS is a little (data annotation) language

JavaScript is a little (programming) language

Powerful because of the restrictions it omits

and the browser it lives in.