Domain specific languages: why? how? and where next?

Laurence Tratt

http://tratt.net/laurie/

King's College London

2011/11/22
A question

What’s this?
What’s this?

```c
#include <stdlib.h>

#define IS_DAY(t, d) ...

void
validTickets(time_t t, int **tick_offs, int *num_ticks)
{
    int num_ticks_alloc = START_SIZE;
    *num_ticks = 0;
    *ticks = malloc(num_ticks_alloc * sizeof(int));
    #define CHECK_SPACE if (*num_ticks > num_ticks_alloc) { \n        num_ticks_alloc *= 2; \n        ticks = realloc(ticks, num_ticks_alloc * sizeof(int)); \n        if (ticks == NULL) \n            errx(1, "malloc failed"); \n    } \n    for (int i = 0; i < all_num_ticks; i++) { \n        ticket tick = all_ticks[i]; \n        if (IS_DAY(t, MON) && !IS_BANK_HOLIDAY(t)) { \n            if (tick.timed && IS_AFTER(t.after)) \n                *tick_offs[(*num_ticks)++] = CHEAP_DAY_RETURN; \n            else \n                *tick_offs[(*num_ticks)++] = DAY_RETURN;
        } \n    } ...
}
```
A question

What’s this?

Is it a language for computers or a language for railway timetables?
To express a solution we need a language.
To express a solution we need a language.

On computers we turn to General Purpose Languages (GPLs)—e.g. Java, C#, C++, Python, Ruby...
The situation

- To express a solution we need a language.
- On computers we turn to General Purpose Languages (GPLs)—e.g. Java, C#, C++, Python, Ruby...
- For new or unusual problems, GPLs are nearly always great.
- But not always for repetitive tasks. Why?
Why do we have GPLs?

- Let’s take Java.
- Main features: packages, classes, functions, static types, garbage collection, variables, if, while, for, and so on.
Let’s take Java.

Main features: packages, classes, functions, static types, garbage collection, variables, if, while, for, and so on.

Really: building blocks.
Building blocks

- Virtually anything can be built with them...

Photo: David Iliff (licence)
Building blocks

- ...but it can be repetitive.

Photo: Mark Murphy (licence)
**GPLs summary**

- *Low level* building blocks.
- Virtually any task will need some (often all) of the building blocks.
GPLs summary

- *Low level* building blocks.
- Virtually any task will need some (often all) of the building blocks.
- But few naturally map onto them.
GPLs summary

- *Low level* building blocks.
- Virtually any task will need some (often all) of the building blocks.
- But few naturally map onto them.
- Very general; jacks of all trades, masters of none.
- The railway timetable uses only a tiny fraction of a GPLs power...
My GPL is better than yours

- But wait—my favourite language is better than Java!
My GPL is better than yours

- But wait—my favourite language is better than Java!
My GPL is better than yours

- But wait—my favourite language is better than Java!

(l-r) Java, C++, Python, C#, Haskell

Source: Library & Archives Canada (licence)
My GPL is better than yours

- But wait—my favourite language is better than Java!
- GPLs are nearly all extremely similar.
- We magnify small differences for cultural reasons.
- They’re all jack of all trades, master of none.
DSLs—the basic idea

- DSL: a small language targeted at a specific class of problems.
- Allows you to specify repetitive tasks with small amounts of variation.
- ‘Do one thing and do it well.’
SQL (databases)

```python
if way1_id == highway_id and \
    dist1 + _DISTANCE_FROM_OTHER_HIGHWAY / Geo.metres_at_latitude(lat) < dist2:
    found = True

if found:
    related.append(['w', way_id, dst])

# Node case

for node_id, dst, node_geom in c.fetchall():
    c2.execute('"
    SELECT
        ways.id, ST_Distance(ST_GeomFromEWKT(%(node_geom)s), ways.linearring) AS dst
    FROM ways, way_tags
    WHERE
        ST_DWithin(ST_GeomFromEWKT(%(node_geom)s), ways.bbox, %(max_dst)s)
    AND node.id = ways.id AND way_tags.k='highway'
    ORDER BY dst
    LIMIT 2
    "', dict(node_geom=node_geom, \
        max_dst=MAX_DISTANCE_OF_BUILDING / Geo.metres_at_latitude(lat)))
    assert c2.rowcount > 0 and c2.rowcount <= 2
    found = False
    if c2.rowcount == 1:
        found = True
    else:
        way1_id, dst1 = c2.fetchone()
        way2_id, dst2 = c2.fetchone()
```
DSL examples

- make (software builds)

```make
Makefile.Stdlib - /home/tratt/src/converge/current/lib/

%.cvb: %.cv
   %-CONVERGE_VM% $< $CONVERGE_CC -I Stdlib -o $@

%.cvb: 
   $< $CONVERGE_CC -o $@

all: Stdlib.cv1

minimal: $[MINIMAL_OBJC]

install: all
   $INSTALL -d $DESTDIR$[conlibdir]
   $INSTALL -c -m 444 Stdlib.cv1 $DESTDIR$[conlibdir]

ifdef TARGET
   CROSS_OBJC = $[ALL_OBJC:.cvb=.TARGET.cv]

   .TARGET.cv: .cv
      % $CONVERGE_CC -T $TARGET -I $CONVERGE_COMPILER_DIR -o $@

ifdef TARGET
   CROSS_OBJC = 
   % $CONVERGE_CC $CONVERGE_CC -T $TARGET -o Stdlib.$TARGET.cv1 Stdlib.$TARGET.cv

cross-clean:
   rm -f $[CROSS_OBJC] Stdlib.$TARGET.cv1
endif

Stdlib.cv1: $[ALL_OBJC]
   $< $CONVERGE_CC -1 -o Stdlib.cv1 Stdlib.cv $[ALL_OBJC]

clean:
   rm -f $[ALL_OBJC] Stdlib.cv1
```
Hardware DSLs

Question: are DSLs only for low-level software activities?
Hardware DSLs

- **Question:** are DSLs only for low-level software activities?
- **Verilog:** hardware description language.

```verilog
module counter (clk, rst, enable, count);
    input clk, rst, enable;
    output [3:0] count;
    reg [3:0] count;

    always @ (posedge clk or posedge rst)
        if (rst) begin
            count <= 0;
        end else begin : COUNT
            while (enable) begin
                count <= count + 1;
                disable COUNT;
            end
        end
endmodule
```

Source: Deepak Kumar Tala
Why would we want DSLs?

- DSLs are good when we do the same type of task repeatedly.
- But is that it?
Consideration 1: accessibility

- Programming is how we tell computers what to do.

Pros / cons:
+ Can allow non-programmers to do programming-like things.
- Sometimes complexity is fundamental.

Income tax:
- 2010-2011 allowance:
  - age < 65: £6,475
  - age >= 65 and age <= 74: £9,490
  - age > 74: £9,640
- reduction: if income > £100,000 then max(0, allowance - ((income - £100,000) / 2))

Tax rules source: HMRC

L. Tratt
http://tratt.net/laurie/
Consideration 1: accessibility

- Programming is how we tell computers what to do.
- Many (most?) people struggle with programming...

Income tax:

- 2010-2011:
  - Allowance:
    - Age < 65: £6,475
    - Age >= 65 and age <= 74: £9,490
    - Age > 74: £9,640
  - Reduction: if income > £100,000 then
    \[
    \max(0, \text{allowance} - \frac{(\text{income} - £100,000)}{2})
    \]

Tax rules source: HMRC

Pros / cons:

+ Can allow non-programmers to do programming-like things.
- Sometimes complexity is fundamental.

L. Tratt  
http://tratt.net/laurie/
Consideration 1: accessibility

- DSLs can remove complex confusing features.
Consideration 1: accessibility

- DSLs can remove complex confusing features.

```plaintext
income tax {
    2010-2011 {
        allowance {
            age < 65: £6,475
            age >= 65 and age <= 74: £9,490
            age > 74: £9,640

            reduction: if income > £100,000 then
                max(0, allowance - ((income - £100,000) / 2))
        }
    }
}
```

Tax rules source: HMRC
Consideration 1: accessibility

- DSLs can remove complex confusing features.

```
income tax {
  2010-2011 {
    allowance {
      age < 65: £6,475
      age >= 65 and age <= 74: £9,490
      age > 74: £9,640

      reduction: if income > £100,000 then
        max(0, allowance - ((income - £100,000) / 2))
    }
  }
}
```

Tax rules source: HMRC

Pros / cons:

+ Can allow non-programmers to do programming-like things.
Consideration 1: accessibility

- DSLs can remove complex confusing features.

```javascript
income tax {
  2010-2011 {
    allowance {
      age < 65: £6,475
      age >= 65 and age <= 74: £9,490
      age > 74: £9,640

      reduction: if income > £100,000 then
        max(0, allowance - ((income - £100,000) / 2))
    }
  }
}
```

Tax rules source: HMRC

Pros / cons:
- Can allow non-programmers to do programming-like things.
- Sometimes complexity is fundamental.
Consideration 2: implementation flexibility

- Virtually all programming is done in imperative languages.
Consideration 2: implementation flexibility

- Virtually all programming is done in imperative languages.
- Advantage: explicitness.
Consideration 2: implementation flexibility

- Virtually all programming is done in imperative languages.
Consideration 2: implementation flexibility

- Virtually all programming is done in imperative languages.
- DSLs are an abstraction over a domain.
Consideration 2: implementation flexibility

- **SQL:**
  
  ```sql
  SELECT * FROM nodes WHERE node.parent=NULL;
  ```

- **C:**
  
  ```c
  table *nodes = get_table(db, "nodes");
  cursor *c = mk_cursor(nodes);
  row *r;
  results res = mk_results();
  while ((r = get_next(c)) != null) {
    if (get_column(r, "parent") == null) {
      add_result(res, r);
    }
  }
  ```

How do you make parallelized versions of each?

- **C:** rewrite your program (pthreads etc.).

- **SQL:** a cleverer SQL implementation.

Pros / cons:

+ Moves the burden from programmer to language implementer.

- Over-abstraction can preclude some reasonable programs.
Consideration 2: implementation flexibility

- SQL:
  
  ```sql
  SELECT * FROM nodes WHERE node.parent=NULL;
  ```

- C:
  
  ```c
  table *nodes = get_table(db, "nodes");
  cursor *c = mk_cursor(nodes);
  row *r;
  results res = mk_results();
  while ((r = get_next(c)) != null) {
    if (get_column(r, "parent") == null)
      add_result(res, r);
  }
  ```

- How do you make parallelized versions of each?
Consideration 2: implementation flexibility

- **SQL:**
  
  \[
  \text{SELECT * FROM nodes WHERE node.parent=NULL;}
  \]

- **C:**
  
  ```c
  table *nodes = get_table(db, "nodes");
  cursor *c = mk_cursor(nodes);
  row *r;
  results res = mk_results();
  while ((r = get_next(c)) != null) {
    if (get_column(r, "parent") == null)
      add_result(res, r);
  }
  ```

How do you make parallelized versions of each?

- **C:** rewrite your program (pthreads etc.).
Consideration 2: implementation flexibility

- **SQL:**
  
  ```sql
  SELECT * FROM nodes WHERE node.parent=NULL;
  ```

- **C:**
  
  ```c
  table *nodes = get_table(db, "nodes");
  cursor *c = mk_cursor(nodes);
  row *r;
  results res = mk_results();
  while ((r = get_next(c)) != null) {
    if (get_column(r, "parent") == null)
      add_result(res, r);
  }
  ```

  How do you make parallelized versions of each?

  - **C:** rewrite your program (pthreads etc.).
  - **SQL:** a cleverer SQL implementation.

---

L. Tratt | http://tratt.net/laurie/ | DSLs | 2011/11/22 | 14 / 47
Consideration 2: implementation flexibility

- **SQL:**
  
  ```
  SELECT * FROM nodes WHERE node.parent=NULL;
  ```

- **C:**
  
  ```
  table *nodes = get_table(db, "nodes");
  cursor *c = mk_cursor(nodes);
  row *r;
  results res = mk_results();
  while ((r = get_next(c)) != null) {
    if (get_column(r, "parent") == null)
      add_result(res, r);
  }
  ```

How do you make parallelized versions of each?

- C: rewrite your program (pthreads etc.).
- SQL: a cleverer SQL implementation.

**Pros / cons:**
- Moves the burden from programmer to language implementer.
Consideration 2: implementation flexibility

- **SQL:**
  
  ```sql
  SELECT * FROM nodes WHERE node.parent=NULL;
  ```

- **C:**
  
  ```c
  table *nodes = get_table(db, "nodes");
  cursor *c = mk_cursor(nodes);
  row *r;
  results res = mk_results();
  while ((r = get_next(c)) != null) {
    if (get_column(r, "parent") == null)
      add_result(res, r);
  }
  ```

How do you make parallelized versions of each?

- C: rewrite your program (pthreads etc.).
- SQL: a cleverer SQL implementation.

**Pros / cons:**

+ Moves the burden from programmer to language implementer.
- Over-abstraction can preclude some reasonable programs.
Consideration 3: Economics

- The bottom line: does it save money?
Consideration 3: Economics

- The bottom line: does it save money?
- If you’re using someone else’s DSL: almost certainly yes.
- But if you need to build a DSL: it depends.
Consideration 3: Economics

Source: P. Hudak 'Modular domain specific languages and tools'

+ It can save serious amounts of money.
- Short-term hit for long-term gain.

---

L. Tratt  
http://tratt.net/laurie/
Consideration 3: Economics

- It can save serious amounts of money.
- Short-term hit for long-term gain.

Source: P. Hudak ‘Modular domain specific languages and tools’

Diagram:
- Total SW cost
- Software life-cycle
- Conventional methodology
- DSL-based methodology
- Start-up costs: c1, c2

Source: L. Tratt http://tratt.net/laurie/
Consideration 3: Economics

It can save serious amounts of money.

Source: P. Hudak ‘Modular domain specific languages and tools’
Consideration 3: Economics

+ It can save *serious* amounts of money.
- Short-term hit for long-term gain.

Source: P. Hudak ‘Modular domain specific languages and tools’
What defines a DSL?

- [Inherently subjective and ill-defined. But... ]
What defines a DSL?

- [Inherently subjective and ill-defined. But... ]
- Has a well-defined problem domain.
What defines a DSL?

- Inherently subjective and ill-defined. But...
- Has a well-defined problem domain.
- Has its own syntax.
- Practically speaking: its own implementation
What DSLs aren’t

- Haskell and Ruby people talk about ‘internal DSLs’.
- Just a [clever?] way of using libraries.
- IMHO: not DSLs. Better called fluent interfaces.
**make:** standalone

```
Makefile.Stdlib - /home/tratt/src/converge/current/lib/

%.cvb: %.cv
  $(CONVERGE_VM) $(CONVERGE_GC) -I Stdlib -o $@ $<

%.cvb:
  $(CONVERGE_VM) $(CONVERGE_GC) -o $@ $<

all: Stdlib.cvl

minimal: $(MINIMAL_OBJS)

install: all
  $(INSTALL) -d $(DESTDIR)$(stdlibdir)
  $(INSTALL) -c -m 444 Stdlib.cvl $(DESTDIR)$(stdlibdir)

ifdef TARGET
CROSS_OBJS = $(ALL_OBJS).cvb-$(TARGET).cvb

%.$(TARGET).cvb: %.cv
  $(CONVERGE_VM) $(CONVERGE_GC) -T $(TARGET) -I $(CONVERGE_COMPILER_DIR) -o $@ $<

%.$(TARGET).cvb: %.cv
  $(CONVERGE_VM) $(CONVERGE_GC) -T $(TARGET) -o $@ $<

$(TARGET)
  $(CONVERGE_VM) $(CONVERGE_GC) -l -T $(TARGET) -o Stdlib.$(TARGET).cvl Stdlib.$(TARGET).cvb $

cross-clean:
  rm -f $(CROSS_OBJS) Stdlib.$(TARGET).cvl
endif

Stdlib.cvl: $(ALL_OBJS)
  $(CONVERGE_VM) $(CONVERGE_GC) -l -o Stdlib.cvl Stdlib.cvl $(ALL_OBJS)

clean:
  rm -f $(ALL_OBJS) Stdlib.cvl
```
- SQL: embedded, syntactically distinct, run-time
SQL: embedded, syntactically distinct, compile-time

```python
if d.has_key("addr:city"):
    addr.append(d["addr:city"])

return ", " .join(addr)

return None

def _pp_way(way_id, highway_name, lat, lon):
    do = OSM_DB.get_db()

    found = False
    for k, v in do.iter_tags(way_id=way_id):
        if v == highway_name:
            found = True
            break

    addr = []
    d = _lookup Hud(v)
    if d.has_key("addr:interpolation"):
```
- UML: diagrammatic
DSL flavours

- Metro systems: diagrammatic
DSL implementation techniques

A representative sample:

- Stand alone.
- **Converge** (embedded, homogeneous).
- **Stratego** (embedded / standalone, heterogeneous).
- **Intentional** (embedded, heterogeneous).
- **MPS** (embedded, homogeneous).
- **Xtext** (standalone, heterogeneous).
Part II: The Converge Language
Converge has a number of influences. Relevant ones include:

- is dynamically, but strongly typed (think Python).
- is compiled to bytecode and run by a VM (think Java).
- can perform compile-time meta-programming (as Template Haskell, but probably easiest to think of macros in LISP/Scheme).
- can have its syntax extended (think MetaBorg).
Hello world
Compile-time meta-programming

This is the tricky, interesting bit. Code (as trees, not text) is programmatically generated.
Compile-time meta-programming

This is the tricky, interesting bit. Code (as trees, not text) is programmatically generated.

Expression \( 2 + 3 \) evaluates to 5 as one expects.

Splice \( \$<x> \) evaluates \( x \) at compile-time; the AST returned overwrites the splice.

Quasi-quote \[ | 2 + 3 | \] evaluates to a hygienic AST representing \( 2 + 3 \).

Insertion \[ | 2 + $\{x\} | \] ‘inserts’ the AST \( x \) into the AST being created by the quasi-quotes.
func expand_power(n, x):
    if n == 0:
        return [1]
    else:
        return [x * expand_power(n - 1, x)]

func mk_power(n):
    return [
        func (x):
            return x * x * x * 1
    ]

power3 := mk_power(3)

means that power3 looks like:

power3 := func (x):
    return x * x * x * 1

by the time it is compiled to bytecode.
printf
What use is compile-time meta-programming?

- Now we have a modern programming language with macros...
- ...we can ‘compile’ arbitrary strings at compile time and...
- ...a DSL input is really just a string...
What use is compile-time meta-programming?

- Now we have a modern programming language with macros...
- ...we can ‘compile’ arbitrary strings at compile time and...
- ...a DSL input is really just a string...
- But that is far as previous approaches have got...
Part III: DSLs in Converge
DSL creation in Converge

- DSLs use a simple layer on top of compile-time meta-programming.
- The sole language feature for DSLs is the *DSL block*.
- Allows embedding arbitrary strings using the indentation based syntax.
But first... parsing!

- Parsing is about finding the structure of text.
- Many ways to do this, but we’ll look at one.
- Languages (natural or computer) have an underlying grammar.
Parsing is about finding the structure of text.
Many ways to do this, but we’ll look at one.
Languages (natural or computer) have an underlying grammar.
Simple English grammar:
   sentence ::= subject verb object
   e.g. Bill hits Ben
Parsing phases

- Simplest way: tokenize then parse.
- Tokenize: split input up into individual tokens. [e.g. in English split words by the presence of spaces or punctuation]. Creates list of tokens.
- Parse: work out the structure of the tokens relative to the grammar. Creates a parse tree.
Parsing phases

- Simplest way: tokenize then parse.
- Tokenize: split input up into individual tokens. [e.g. in English split words by the presence of spaces or punctuation]. Creates list of tokens.
- Parse: work out the structure of the tokens relative to the grammar. Creates a parse tree.
- Tokenization is generally easy.
- Parsing isn’t: use a grammar formalism to help.
Context Free Grammars (CFGs) can express most programming languages.

Earley parsing can parse any CFG, so use that.

Backus-Naur Form (BNF): the standard(ish) way of specifying CFGs.

A very simple calculator grammar:

\[
E ::= \text{INT} \"+\" \text{INT} \\
\quad | \text{INT} \"\*\" \text{INT}
\]

Now we can do a ‘yes/no’ parse of 2 + 3 and 6 * 2.
Context Free Grammars (CFGs) can express most programming languages.

Earley parsing can parse any CFG, so use that.

Backus-Naur Form (BNF): the standard(ish) way of specifying CFGs.

A very simple calculator grammar:

\[
E ::= \text{INT} \ "+" \ \text{INT} \\
    | \ \text{INT} \ "\ast" \ \text{INT}
\]

Now we can do a ‘yes/no’ parse of \(2 + 3\) and \(6 \ast 2\).

But ‘yes/no’ isn’t very useful: build parse trees.
Self-referencing rules

- A better calculator:
  \[
  E ::= E \ "+" \ E \\
  \mid E \ "\ast" \ E \\
  \mid \text{INT}
  \]

- What parse tree will we get for 2 + 3 * 4?
A better calculator:

\[
E ::= E \text{ "+" } E \\
| E \text{ "*" } E \\
| \text{ INT}
\]

What parse tree will we get for \(2 + 3 \times 4\)?

Resolve ambiguity with precedences:

\[
E ::= E \text{ "+" } E \%\text{precedence 0} \\
| E \text{ "*" } E \%\text{precedence 10}
\]

Higher precedences are preferred.
A better calculator:

\[
E ::= E \ "+" \ E \\
| E \ "\*" \ E \\
| INT
\]

What parse tree will we get for \(2 + 3 \times 4\)？

Resolve ambiguity with precedences:

\[
E ::= E \ "+" \ E \ %\text{precedence} \ 0 \\
| E \ "\*" \ E \ %\text{precedence} \ 10
\]

Higher precedences are preferred.

An aside: in general, it’s not known how to statically detect ambiguities in arbitrary CFGs. Ambiguities are sort-of run-time errors.
A simplified EBNF grammar... for EBNF!

Grammar ::= Rule*

Rule ::= ID "::=" Prod ( "|" Prod )* 

Prod ::= Expr*

Expr ::= ID
       | STRING
       | "(" Expr* ")"
       | Expr "*"

[Don’t worry if this makes your head hurt for the moment.]
Simplifying parsing

- Hudak: syntax extension is bad. (Because parsing is horrid).
- Converge aims to make parsing easy.
- Converge’s tokenizer (a.k.a. lexer) is designed for use by non-Converge languages.
- It can be told to parse new keywords and ‘unknown’ symbols.
- Converge has a built in Earley parser; can parse *any* CFG.
- Writing a grammar for an Earley parser is easy.
Another problem with new syntax: error reporting goes out of the window.

Languages with macro systems provide little or no error reporting.

DSL development is intolerable without accurate error reporting.
Another problem with new syntax: error reporting goes out of the window.

Languages with macro systems provide little or no error reporting.

DSL development is intolerable without accurate error reporting.

Converge has evolved a unique approach to error reporting.

Errors identify file name, line number, and column numbers.
‘Src info’ a (src path, src offset, src len) triple.
‘Src info’ concept pervasive: tokenizer, parser, ASTs, bytecode generator, and VM.
Every token, AST element, and bytecode instruction associated with one or more src infos. Trivial to pinpoint errors as having occurred within a DSL block.
Users can add extra src info to AST elements in various ways.
e.g. To associate the AST built by a quasi-quote with both the quasi-quote and a position in a DSL, use this syntax:

```<node[1].src_infos>| ${foo}[0] |]```
Hudak noted: as DSLs evolve they increasingly resemble a GPL.

Many stand alone DSLs have hackish, buggy, expression languages.
Hudak noted: as DSLs evolve they increasingly resemble a GPL.
Many stand alone DSLs have hackish, buggy, expression languages.
If the standard Converge tokenizer is used for a DSL, Converge’s expression language can be embedded within the DSL.
Code reuse at its best!

```c
$<<table>>:
  8:25 "Exeter St. Davids"
    if day == "Saturday" | Date::is_public_holiday():
      ["Cheap", "Premium"]
    else:
      ["Premium"]
  10:20 "Salisbury"
    ["Cheap", "Premium"]
  11:15 "Woking"
    ["Cheap", "Premium"]
  11:49 "London Waterloo"
```
The Converge DSL process

Converge does not mandate a process, but the following naturally presents itself:

1. Use the Converge tokenizer.
Converge does not mandate a process, but the following naturally presents itself:

1. Use the Converge tokenizer.
2. Write a CFG.
The Converge DSL process

Converge does not mandate a process, but the following naturally presents itself:

1. Use the Converge tokenizer.
2. Write a CFG.
3. Write a translation class (from parse tree to AST).
The Converge DSL process

Converge does not mandate a process, but the following naturally presents itself:

1. Use the Converge tokenizer.
2. Write a CFG.
3. Write a translation class (from parse tree to AST).
4. Test, debug, modify etc.

Converge gives you huge assistance for everything but step 5!
Converge does not mandate a process, but the following naturally presents itself:

1. Use the Converge tokenizer.
2. Write a CFG.
3. Write a translation class (from parse tree to AST).
4. Test, debug, modify etc.
5. Deploy finished DSL.
Converge does not mandate a process, but the following naturally presents itself:

1. Use the Converge tokenizer.
2. Write a CFG.
3. Write a translation class (from parse tree to AST).
4. Test, debug, modify etc.
5. Deploy finished DSL.

Converge gives you huge assistance for everything but step 5!
Converge started circa 2004.

Converge 1.2 released July 2011.

Pre-built binaries for Linux / OpenBSD / OS X / Windows.


Currently working on a new RPython-based VM: about 2/3 complete and about 4x faster than the old VM (aiming to get ˜6-8x faster).

https://github.com/ltratt/converge/tree/pypyvm/pypyvm
Current state of affairs

- Converge started circa 2004.
- Converge 1.2 released July 2011.
- Pre-built binaries for Linux / OpenBSD / OS X / Windows.
- Currently working on a new RPython-based VM: about 2/3 complete and about 4x faster than the old VM (aiming to get Õ-8x faster).

Part IV: The future
What we want: arbitrary composition of languages.
What we want: arbitrary composition of languages.
But we fail at step 1: parsing. Why?

The union of 2 LR-compatible grammars may not be LR-compatible (similarly LL etc.).
But e.g. Earley parsing can parse any CFG. Problem solved?
Composing known unambiguous grammars may lead to an ambiguous grammar...
...but we can't statically uncover ambiguity for CFGs in general.
Always worried that the next input will cause unrecoverable ambiguity.
PEGs are inexpressive (no arbitrary left-recursion).
As far as I can tell, no good solution known.
What we want: arbitrary composition of languages.
But we fail at step 1: parsing. Why?
The union of 2 LR-compatible grammars may not be LR-compatible (similarly LL etc.).
What we want: arbitrary composition of languages.
But we fail at step 1: parsing. Why?
The union of 2 LR-compatible grammars may not be LR-compatible (similarly LL etc.).
But e.g. Earley parsing can parse any CFG. Problem solved?
What we want: arbitrary composition of languages.

But we fail at step 1: parsing. Why?

The union of 2 LR-compatible grammars may not be LR-compatible (similarly LL etc.).

But e.g. Earley parsing can parse any CFG. Problem solved?

Composing known unambiguous grammars may lead to an ambiguous grammar...

...but we can’t statically uncover ambiguity for CFGs in general.

Always worried that the next input will cause unrecoverable ambiguity.
What we want: arbitrary composition of languages.

But we fail at step 1: parsing. Why?

The union of 2 LR-compatible grammars may not be LR-compatible (similarly LL etc.).

But e.g. Earley parsing can parse any CFG. Problem solved?

Composing known unambiguous grammars may lead to an ambiguous grammar...

...but we can’t statically uncover ambiguity for CFGs in general.

Always worried that the next input will cause unrecoverable ambiguity.

PEGs are inexpressive (no arbitrary left-recursion).
What we want: arbitrary composition of languages.

But we fail at step 1: parsing. Why?

The union of 2 LR-compatible grammars may not be LR-compatible (similarly LL etc.).

But e.g. Earley parsing can parse any CFG. Problem solved?

Composing known unambiguous grammars may lead to an ambiguous grammar...

...but we can’t statically uncover ambiguity for CFGs in general.

Always worried that the next input will cause unrecoverable ambiguity.

PEGs are inexpressive (no arbitrary left-recursion).

As far as I can tell, no good solution known.
Beyond parsing

- Syntax directed editing has no composition problems...
- ...but tried and rejected in the 80s.
Beyond parsing

- Syntax directed editing has no composition problems...
- ...but tried and rejected in the 80s.
- MPS shows it can be (at least) semi-palatable.
- [Maybe the Intentional tool, if we ever get to play with it.]
Next major challenge: composing language implementations.

Not Java + C++ (yet).

What are the correct units to break languages down into? How to integrate compilers? What types of languages are mutually exclusive? What about efficiency? Nice editors? etc. etc.

Initially a language design issue (language semantics to follow?).
Next major challenge: composing language implementations.
Not Java + C++ (yet).
What are the correct units to break languages down into? How to integrate compilers? What types of languages are mutually exclusive? What about efficiency? Nice editors? etc. etc.
Initially a language design issue (language semantics to follow?).
My attempt: Foundries.
Unifying compilers and editors; languages, programs, and editors interact with meta-programming.
Attempt to tackle the problem bit by bit, bottom up.
Current status:
Next major challenge: composing language implementations.
Not Java + C++ (yet).
What are the correct units to break languages down into? How to integrate compilers? What types of languages are mutually exclusive? What about efficiency? Nice editors? etc. etc.
Initially a language design issue (language semantics to follow?).
My attempt: Foundries.
Unifying compilers and editors; languages, programs, and editors interact with meta-programming.
Attempt to tackle the problem bit by bit, bottom up.
Current status: barely started.
Further reading

- Fowler: *Language workbenches*
- Stahl, Völter: *Model-Driven Software Development*
- Vasudevan, Tratt: *Comparative study of DSL tools*
There are more DSLs in existence than we first think...
...and there will be a lot more.
There are more DSLs in existence than we first think...
...and there will be a lot more.
When DSLs are the right tool, they can lead to real savings.
But we’re still searching for more general tooling.
There are more DSLs in existence than we first think...
...and there will be a lot more.
When DSLs are the right tool, they can lead to real savings.
But we’re still searching for more general tooling.

Thanks for listening