Language Factories

Tony Clark[1] Laurence Tratt[2]

[1]School of Computing
Thames Valley University
London, UK

[2]Bournemouth University, UK

Onward! 2009

Outline

Language Horticulture

...a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language.
...from now on, a main goal in designing a language should be to plan for growth.

Guy L. Steele, Jr. Growing a language, 1998

Growing a Language

From little acorns...

Large structured languages grow...

Growing a Language

From little acorns...

Large structured languages grow...

But It Takes Too Long and is Difficult to Control

Defining Languages is Hard

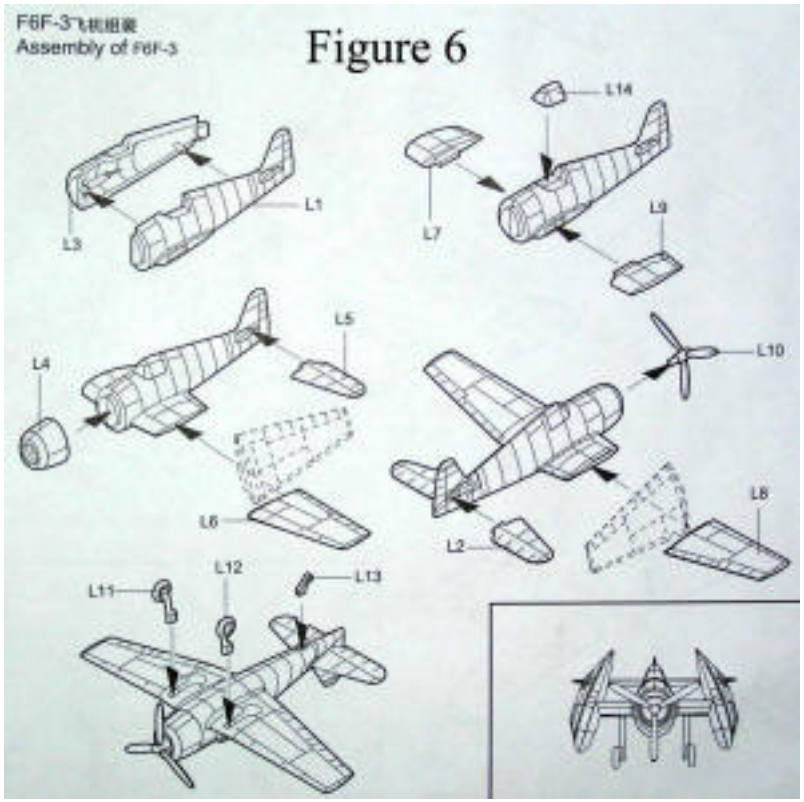
- Lots of interest in DSLs: malleable syntax and semantics.
- Using GPLs to implement DSLs is not the best way.
- DSL definition is a specialist activity.
- DSLs are a *desire lacking a philosophy.*

Current Situation

- Languages are difficult to evolve.
- Languages evolve slowly: how long between major language versions?
- Technologies are evolving to help with this:
  - Stratego
  - XMF
  - Converge
  - MPS
  - Oslo
- More needs to be done...

How Are Software Systems Designed?

- Components
- Reuse
- Interfaces
- Extension
- Change Analysis



Language Factories - Research Agenda

- A component-based approach to the definition and construction of languages, tools.
- Language Factories aim to support:
  - **Reuse** of common language components.
  - **Agile** language engineering.
  - Language **refactoring**.
  - Language **analysis** including **impact analysis**.
- **Product Lines** for Languages
- Users:
  - Language Factory Developers
  - Language Factory Users
  - Application Developers
  - Application Users

Realising Language Factories

- Different formalisms, tools, degrees of automation.
- Varying levels of precision:
  - **Virtual LFs**: designed and partly implemented.
  - **Idealised LFs**: language workbenches (Martin Fowler).
  - Many shared of grey in between (e.g. componentized abstract syntax).
- Can be realised with current languages (with compromises).

Language Components

Abstract Syntax	Single definition of a data type.
Concrete Syntax(es)	e.g. a context free grammar.
Syntactic Mapping	Concrete syntax(es) to abstract syntax.
Semantic Aspect(s)	Language meaning:

- Operational.
- Denotational.
- Types.
- Code Generation.
- Serialization.

Tooling	Composable editors, checkers, etc.
Constraints	Conditions on language use.
Interfaces	What is offered and what is required?

An Example Language

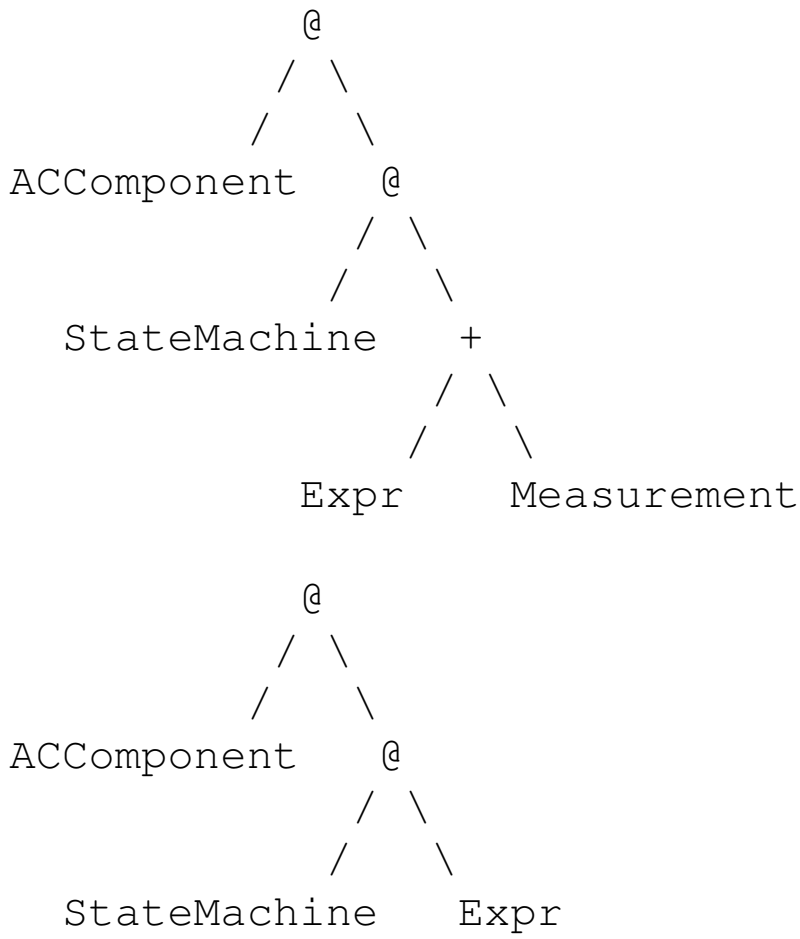
```
component Landing_gear(height:int, speed:float) {
  stm {
    state Moving_Up
    state Moving_Down
    state Deployed
    state Stowed
    transition up from Deployed to Moving_Up
      height_change[height>500ft and speed>100kn/s]
    transition down from Stowed to Moving_Down
      deploy
  }
}
```

A Language Is Composed of Parts

Given components:

- **Expr**
- **Measurements**
- **StateMachine(Guards)**
- **ACComponent(Body)**

Can combine and recombine:



A Reusable Expression Language

```
lang Expr:
  ast: Var(Str) | Add(Expr, Expr) | Num(Int)
  grammar:
    expr -> name:Id                    <Var(name)>
          | lhs:Expr '+' rhs:Expr      <Add(lhs, rhs)>
          | num:Int                    <Num(num)>

  semantics java:
    Var(x)       -> [j| ${x} |]
    Add(x, y)    -> [j| ${java(x)}.plus(${java(y)}) |]
    Num(x)       -> [j| new ExprInt(${x}) |]
  constraints:
    exists_class(ExprInt)
    exists_class(Expr)
    exists_static_method(Expr,plus)
end
```

A Reusable Expression Language

```
lang Expr:
  ast: Var(Str) | Add(Expr, Expr) | Num(Int)
  grammar:
    expr -> name:Id                      <Var(name)>
          | lhs:Expr '+' rhs:Expr        <Add(lhs, rhs)>
          | num:Int                      <Num(num)>

  semantics java:
    Var(x)       -> [j| ${x} |]
    Add(x, y)    -> [j| ${java(x)}.plus(${java(y)}) |]
    Num(x)       -> [j| new ExprInt(${x}) |]
  constraints:
    exists_class(ExprInt)
    exists_class(Expr)
    exists_static_method(Expr,plus)
end
```

A Reusable Expression Language

```
lang Expr:
  ast: Var(Str) | Add(Expr, Expr) | Num(Int)
  grammar:
    expr -> name:Id                    <Var(name)>
          | lhs:Expr '+' rhs:Expr      <Add(lhs, rhs)>
          | num:Int                    <Num(num)>
  semantics eval(env):
      Var(x)    -> lookup(env, x)
      Add(x, y) -> eval(x, env) + eval(y, env)
      Num(x)    -> x
  semantics java:
    Var(x)      -> [j| ${x} |]
    Add(x, y)   -> [j| ${java(x)}.plus(${java(y)}) |]
    Num(x)      -> [j| new ExprInt(${x}) |]
  constraints:
    exists_class(ExprInt)
    exists_class(Expr)
```

A Reusable Expression Language

```
lang Expr:
  ast: Var(Str) | Add(Expr, Expr) | Num(Int)
  grammar:
    expr -> name:Id                    <Var(name)>
          | lhs:Expr '+' rhs:Expr      <Add(lhs, rhs)>
          | num:Int                    <Num(num)>

  semantics java:
    Var(x)       -> [j| ${x} |]
    Add(x, y)    -> [j| ${java(x)}.plus(${java(y)}) |]
    Num(x)       -> [j| new ExprInt(${x}) |]
  constraints:
    exists_class(ExprInt)
    exists_class(Expr)
    exists_static_method(Expr,plus)
end
```

A Reusable Expression Language

```
lang Expr:
  ast: Var(Str) | Add(Expr, Expr) | Num(Int)
  grammar:
    expr -> name:Id                    <Var(name)>
          | lhs:Expr '+' rhs:Expr      <Add(lhs, rhs)>
          | num:Int                    <Num(num)>

  semantics java:
    Var(x)       -> [j| ${x} |]
    Add(x, y)    -> [j| ${java(x)}.plus(${java(y)}) |]
    Num(x)       -> [j| new ExprInt(${x}) |]
  constraints:
    exists_class(ExprInt)
    exists_class(Expr)
    exists_static_method(Expr,plus)
end
```

Representing Measurements

```
lang Measurement:
  ast: Ft(Float) | KnPerH(Float) | MiPerH(Float)
  grammar:
    measure -> dst:Float 'ft'    <Ft(dst)
             | dst:Float 'kn/h' <KnPerH(dst)>
             | dst:Float 'mph'  <MiPerH(dst)>
  semantics java:
    Ft(x)      -> [j| new ExprFeet(${x}) |]
    KnPerS(x) -> [j| new ExprKnPerH(${x}) |]
    MiPerS(x) ->
      [j| new ExprKnPerH(${x*0.869}) |]
  constraints:
      exists_class(ExprFeet)
      exists_class(ExprKnPerH)
end
```

Language Factories: The Research Agenda

- Need syntax reuse and composition.
- Need semantics (operational, translational, ...) reuse and composition.
- Need composable type systems.
- Need composable tools.
- Need language templates (components with parameters).
- Need LF meta-languages and type systems.
- Need LF workbenches.