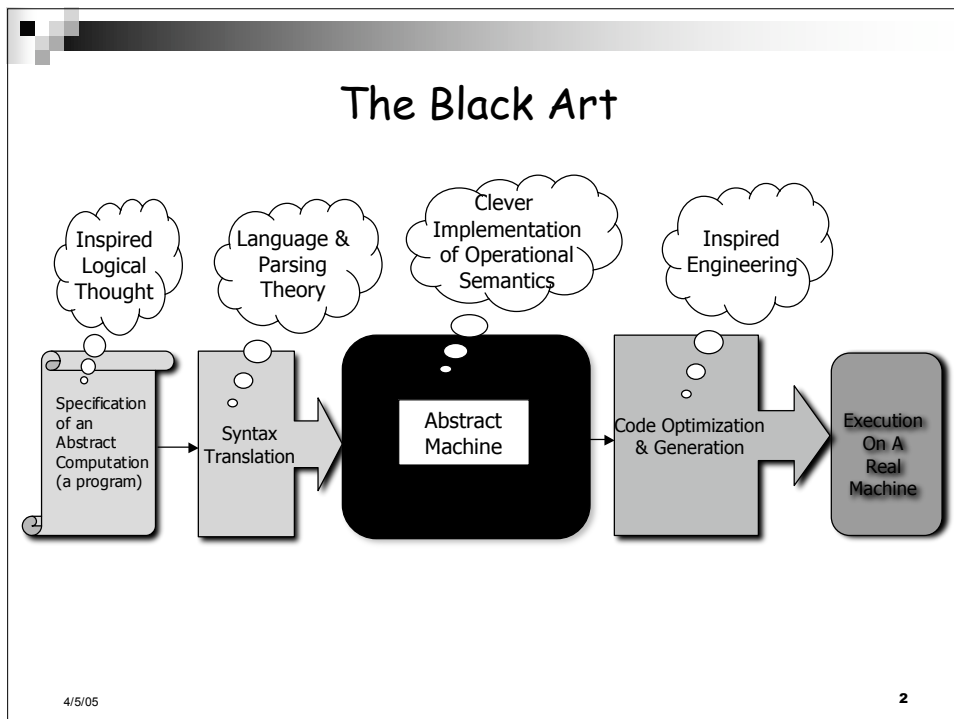


# Some Ideas for a DSL Toolkit

CS395T - Domain Specific Languages  
Greg Lavender  
Department of Computer Sciences  
The University of Texas at Austin



## What are the pieces of a DSL toolkit?

- A lot of "stuff"
  - Syntactic definition & language recognition
  - Semantics preserving transforms
  - Type systems and type checking
  - Abstract machine/interpreter construction
  - Efficient resource management
  - Feature-oriented libraries
  - Others?
- Can we unify this diverse collection of "stuff" into a coherent language construction toolkit?
  - Are there useful ways to formalize any of this?
  - Minimize the amount of custom code to be written?
  - Ensure some kind of semantic correctness & coherence
  - Provide a realistic set of tools that are useful to practitioners?
    - i.e., understandable by mere mortals, practical, efficient

4/5/05

3

## Piecemeal Solutions

- People just focus on one part of the problem
  - e.g., parsing or garbage collection or code generation
  - develop independent approaches in a specific context that are not easily integrated with other pieces done by other people
- Need to take a more holistic view of the system
  - Probably no one unifying theory
  - Lots of micro-theories that need to be stitched together
  - But perhaps there are some universal algebraic ideas that apply across all the pieces?
    - e.g., functional calculi, structure preserving morphisms, scale invariant compositional properties, higher-order logical operators, functors, etc.

4/5/05

4

## What formal tools do we have?

- Set theory, propositional & predicate logic
- Well-developed theories of syntax
  - Kleene regular languages/grammars
  - Chomsky hierarchy of languages/grammars
  - Practical experience implementing language systems
- Well-developed models of computation
  - Recursive function theory & Turing machines
    - Recursively enumerable sets
  - Finite state automata, pushdown automata, etc.
    - Algebraic theories, Universal algebra
- Higher-order functional calculi & combinatory logic
  - $\lambda$ -calculus and all its variants (e.g., poly  $\lambda$ -calculus, LCF, PCF)
- Well-developed theories of semantics
  - Denotational, Axiomatic, Operational, Algebraic, Categorical
  - Practical experience implementing various languages
    - logical, functional, imperative, object-oriented, concurrent, etc

4/5/05

## The Problem?

- How to get traction and make real progress?
  - takes years to learn all this formal theory stuff
  - takes years to learn how to think with theoretical stuff
  - takes years to learn how to implement something using it
  - takes years to learn how to re-implement what we got wrong
  - takes years to realize that we still didn't get it right
  - takes years to invent some new, hopefully better, ideas
  - take years to get them accepted by people whose careers are based on the other ideas that are not working out
  - then you die
  - then someone re-discovers the same problem in a different way and the process repeats itself
  - so in order to not waste a life, we use brute force to appear to make progress in one life-time (hopefully being well paid!)
    - not necessarily a bad strategy given a finite Newtonian Universe
      - but see E. B. Davies, Building Infinite Machines, Brit. J. Phil. Sci, 52(2001), 671-682 for an alternative idea

4/5/05

6

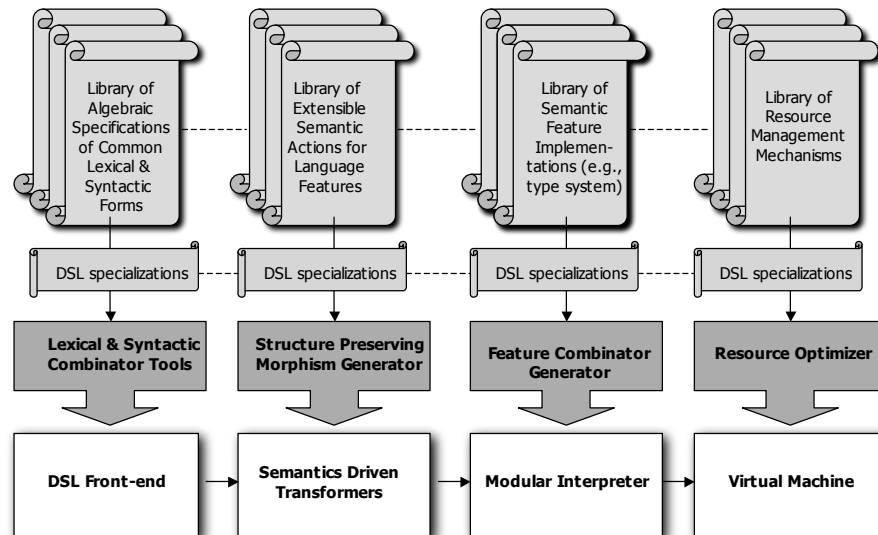
## The Solution?

- I don't have one (yet)
- I just have some ideas motivated by:
  - frustration using conventional languages and tools
  - disillusionment with the way real world software is usually developed
  - real concern about the quality of large software systems
  - a belief that the root of the problem is in the way we express partial solutions to computational problems and how they combine to form larger software systems
  - a gradually developed understanding of many different formal theoretical approaches, without being religiously attached to any one
  - some practical intuitions that lead me to believe that it is possible to apply some of these esoteric theories to the construction of practical software systems
    - but will have to bury the ideas inside the tools so that "inspired amateurs" will use them and be more effective while remaining (mostly) ignorant

4/5/05

7

## An Idea for a DSL Toolkit



4/5/05

8

## Language Definition/Recognition

- Lexical analysis tools
  - Well known theory of regular grammars and sets
  - We have many tools to choose from
  - What about a library of pre-defined subsets for common lexemes?
    - e.g., keywords, operators, commenting styles
    - define a "regular set constructor language" for the selecting the ones you want
    - allow override of default actions associated with token recognition
    - perhaps an "algebraic combinator tool" could be defined that assembles the selected subsets into the final regular set, checks consistency, and generates a token recognizer
    - Btw, it has already been shown by other researchers (e.g., J. Goguen, E. Manes) how to represent automata in terms of universal algebraic structures (e.g., categories), so that may be a good starting point for thinking about how to go about building a more generalized tool

4/5/05

9

## Language Definition/Recognition

- Syntax analysis tools
  - take your pick
    - however most are focused on a single implementation language (e.g., C or Java)
    - perhaps what is needed is a tool independent notation and the ability to generate the EBNF for a particular tool from this more general notation
  - AST construction library
    - Construction of an AST is a key step in the translation from raw syntax to initial semantics
      - semantic annotations, semantic error checking, etc.
    - an extensible tree walker for transforming ASTs
      - source-to-source translation
      - elaboration of initial semantics

4/5/05

10

## Language Definition/Recognition

- An idea: use an XML-based AST representation
  - use XML schema to define the tags and structure
    - allow tags to have annotations for future processing
  - define multiple schemas for various sublanguages
    - e.g., namespaces, expressions, type decls, blocks, stmts, procs, etc.
  - XML-based tools to modify/extend/combine schemas to create an intermediate language representation
    - an advantage is that AST is human readable
    - Can use standard XML editors to review
  - XML-based tools (XSLT) to do structural pattern matching and tree transformation
    - pretty printing for free (e.g., gxx -pp foo.xx)
    - clearer source-to-{source/bytecode} translation mapping
    - can "query" the AST for info about the program
      - facilitate code re-factoring analysis
    - structure preserving "morphisms" can be given categorical semantics and proven correct

4/5/05

11

## Extensibility

- At each step, need to allow for customization
  - Extensibility has to be a key feature
  - Can populate a library with common elements, but always need a way to incrementally extend and/or override defaults
  - Must ensure semantics are not violated however
    - that's the hard part!
  - No generally agreed upon way to do this
    - proliferation of similar but incompatible definitions
  - Need a model for correct composition
    - Need to be able to combine features in new ways
    - But also provide limits so that "bad" things don't result

4/5/05

12

## Specifying Composable Semantic Actions

- Lots of competing semantic models
  - Peter Mosses defined Action Semantics to be extensible and practical
  - Can sometimes give a direct semantics for each syntactic phrase
    - can usually handle recursion, exceptions, etc., as special cases
  - Composition of features is when things get complicated
    - Orthogonality checking?
    - How do we know when the composition of features results in inconsistency?
  - Need a formal higher-order model to express the relations between syntactic elements and semantic objects, but not too abstractly as we need to be able to generate an interpreter
    - Can Type Theory help?
    - Can Categorical Logic help?
      - A logic for relating syntax and semantics
      - See M. Barr & C. Wells: *Toposes, Triples and Theories*
      - See J. Lambek & P. Scott: *Higher-order Categorical Logic*

4/5/05

13

## Feature Implementation Libraries

- Well-understood implementation techniques for common language features
  - Basic Data Types (bool, byte, char, int, float, string, etc.)
  - Arrays/Vectors
  - Constructed Data Types (enums, structs, unions, etc.)
  - Conditionals
  - Sequencing and Iteration
  - Various kinds of (generic) Expressions
  - Order of evaluation
  - Function/Procedure calling & parameter passing
  - Various effects
  - Static and dynamic typing
  - Various scoping rules and systems
  - Class/Interface/Object abstractions
    - Inheritance & method dispatch
  - Constraint checking and type conversions mechanisms
- Etc.

4/5/05

14

## Type Systems

- Several well-developed logical theories
  - Church's simple types
  - Hindley-Milner (polymorphic)
  - Girard-Reynolds (polymorphic)
  - Curry-Howard (Propositions-as-types)
- Any one of which requires a substantial intellectual investment to understand fully
  - How do we organize the multitude of ideas about types as they occur in real languages?
    - Basic types, enumerated types, constructed types, recursive types, function types, polymorphic types, subtypes, type classes, dependent types, ad hoc types, type aliasing, type schemas, type inferencing, etc.
    - See
      - B. Pierce: *Types and Programming Languages*
      - P. Taylor: *Practical Foundation of Mathematics*
      - R. Crole: *Categories for Types*
      - A. Asperti & G. Longo: *Categories, Types and Structures*

4/5/05

15

## Modular Interpreters

- Generated from semantic transforms and compositional features
  - see G. Steele: "Building Interpreters by Composing Monads," for one idea using monads in Haskell
- Implements the executable semantics of the language
  - need a way to specify legal combinations of modular features
- Good software engineering required to modularize the features of an interpreter into incrementally extensible & usable components
  - such as done in the course textbook by Friedman & Wand using Scheme
    - GNU guile is a positive step in this direction as it exposes a number of APIs to its internal interpretive mechanisms
  - but may want to take a more formal approach to having the components generated or specialized from a library of components that can co-exist and satisfy various orthogonal feature properties

4/5/05

16

## Abstract Machines

- We've already talked about various kinds of abstract machines
  - still very much a black art
  - some abstract machines constructed for functional languages using strict adherence to theoretical models:
    - Structured operational semantics
    - combinator reduction
    - supercombinators
    - categorical abstract machine
  - Good to learn from these approaches, but they have been tied to a specific language or problems associated with a particular paradigm

4/5/05

17

## Resource Management

- Memory management
  - well-studied and good gc techniques available
- Concurrency
  - explicit vs implicit use of concurrency
  - synchronization
- Efficient run-time mechanisms
  - stack evaluator
  - exception handling
  - data structure usage
  - networking resources
  - I/O and file system interactions
  - concurrency and synchronization

4/5/05

18