Domain-specific language features

Ted Kaminksi and Eric Van Wyk

University of Minnesota

Global DSL 2013, July 2, 2013, Montpellier

DSLs are bad, or at least have some challenges...

- External
  - "No recourse to the law."
  - domain-specific syntax, analysis, and optimization
- Internal/Embedded
  - "just libraries", caveat: staging, Delite
  - composable
  - general purpose host language available

Maybe there are alternatives ...

Extensible Languages (Frameworks)

- pluggable domain-specific language extensions
- domain-specific syntax, analysis, and optimization
- composable
- general purpose host language available
- extension features translate down to host language

- Regexs, different whitespace

```
class Demo {
  int demoMethod ( ) {
    List<List<Integer>> dlist ;
    Regex ws = regex/\n\t / ;
    int SELECT ;
    connection c "jdbc:derby:/home/derby/db/testdb"
      with table person [ person_id INTEGER,
                          first_name VARCHAR,
                          last_name VARCHAR ] ,
           table details [ person_id INTEGER,
                           age INTEGER ] ;
    Integer limit = 18 ;
    ResultSet rs = using c query {
          SELECT age, gender, last_name
            FROM person , details
           WHERE person.person_id = details.person_id
             AND details.age > limit } ;
    Integer = rs.getInteger("age");
    String gender = rs.getString("gender");
    boolean b ;
    b = table ( age > 40       :  T * ,
                gender == "M" :  T F ) ;
  }
}
```

- natural syntax
- semantic analysis
- composable extensions
- SQL queries
- non-null pointer analysis
- tabular boolean expressions

```
#include <sdtio.h>

int main() {

    ... bits of SAC ...

    ... stencil specifications ...

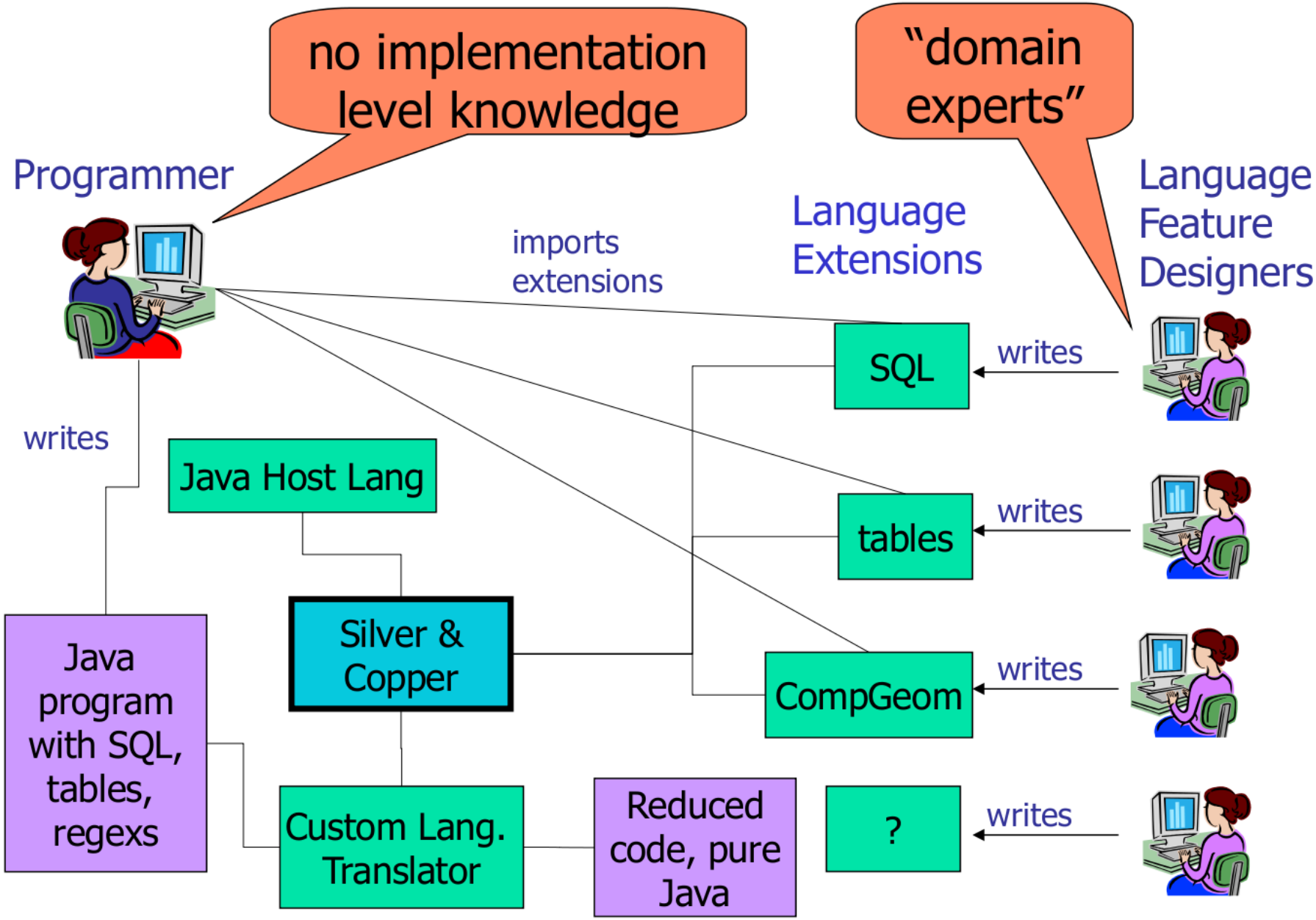
    ... computational geometry optimizations
        robustness transformations ...
}
```

Roles people play ...

1. host language designer

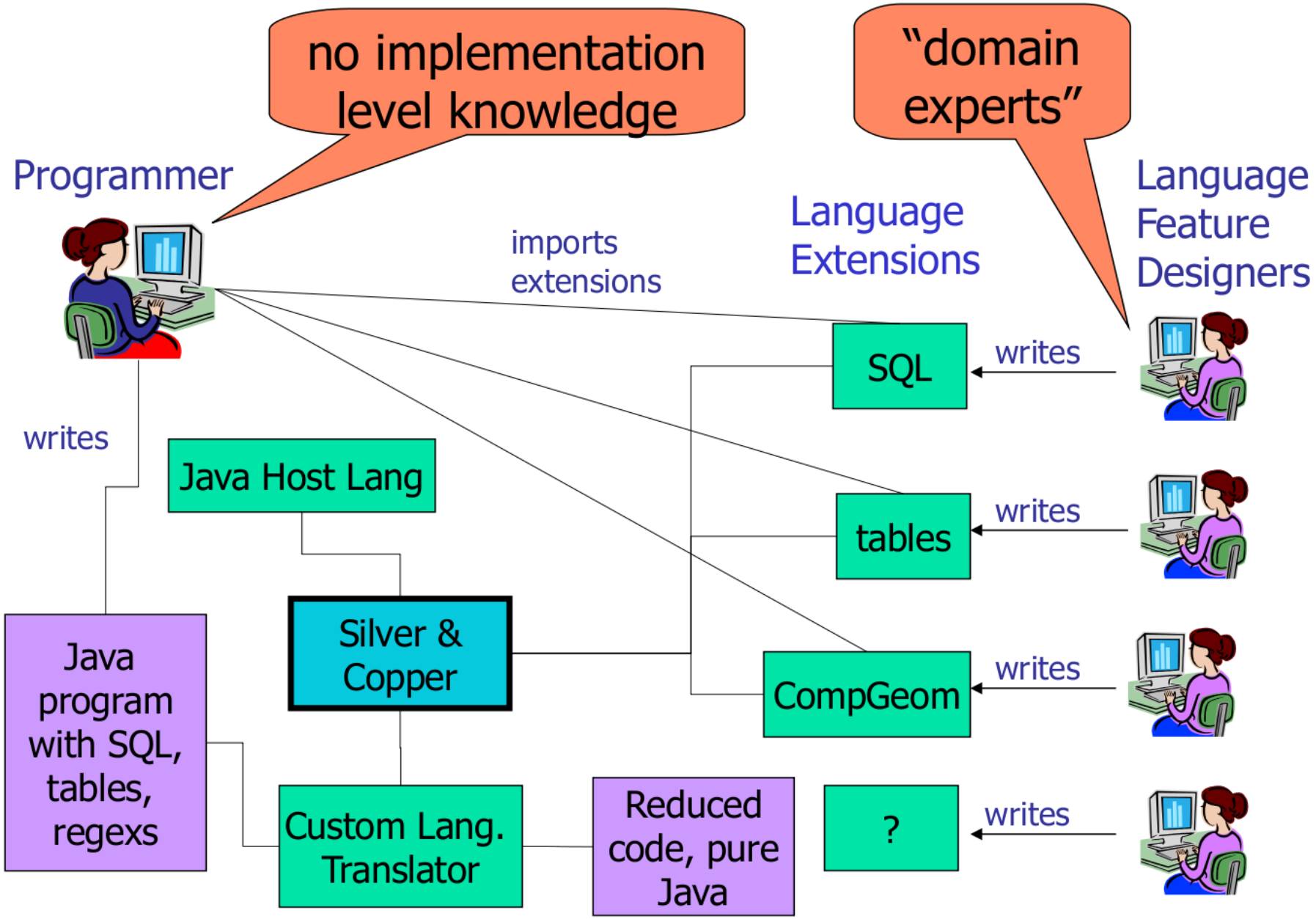
2. language extension designer

3. programmer
   - language extension user
   - has no language design and implementation knowledge


no implementation level knowledge
"domain experts"
Programmer
Language Extensions
Language Feature Designers
imports extensions
writes
SQL
writes
tables
writes
CompGeom
writes
?
writes
Java Host Lang
Silver & Copper
Java program with SQL, tables, regexs
Custom Lang. Translator
Reduced code, pure Java

An analog

- We can write type safe programs in assembly, Python, Ruby, Scheme, etc.
- We are guaranteed to write type safe program in Haskell, ML, etc.
- Some of use (happily) trade some expressibility for this safety.
- Can we build extensible languages and composable language extensions?
- Are there any guarantees of composability of language extensions?


no implementation level knowledge
"domain experts"
Programmer
Language Extensions
Language Feature Designers
imports extensions
writes
SQL
writes
tables
writes
CompGeom
writes
?
writes
Java Host Lang
Silver & Copper
Java program with SQL, tables, regexs
Custom Lang. Translator
Reduced code, pure Java

Developing language extensions

Two primary challenges:

1. composable syntax — enables building a parser
   - context-aware scanning [GPCE'07]
   - modular determinism analysis [PLDI'09]
   - Copper

2. composable semantics — analysis and translations
   - attribute grammars with forwarding, collections and higher-order attributes
   - set union of specification components
     - sets of productions, non-terminals, attributes
     - sets of attribute defining equations, on a production
     - sets of equations contributing values to a single attribute
   - modular well-definedness analysis [SLE'12a]
   - modular termination analysis [SLE'12b, KrishnanPhD]
   - Silver

Challenges in scanning

- Keywords in embedded languages may be identifiers in host language:

```
int SELECT ;
...
rs = using c query { SELECT last_name
                     FROM person WHERE ...
```

- Different extensions use same keyword

```
connection c "jdbc:derby:./derby/db/testdb"
    with table person [ person_id INTEGER,
                        first_name VARCHAR ];
...
b = table ( c1 :  T F ,
            c2 :  F * ) ;
```

Challenges in scanning

- Operators with different precedence specifications:

```
x = 3 + y * z ;
...
str =  /[a-z][a-z0-9]*\.java/
```

- Terminals that are prefixes of others

```
List<List<Integer>> dlist ;
...
x = y >> 4 ;
```

or

```
aspect ...  before ...  call( o.get*() )
...
x = get*3 ;
```

[from Bravenboer, Tanter & Visser's OOPSLA 06 paper on parsing AspectJ]

Need for context

- Problem: the scanner cannot match strings (lexemes) to proper terminal symbols in the grammar.

  “`SELECT`” – Select_kwd or Identifier
  “`table`” – SQL_table_kwd or DNF_table_kwd
  “>>” – RightBitShift_op or GT_op, GT_op

- Neither terminal has global precedence over the other.
  - maximal munch gives “>>” precedence over “>”
  - thus grammars are written to accommodate this

Need for context

- Traditionally, parser and scanner are disjoint.

  $$\boxed{\text{Scanner}} \rightarrow \boxed{\text{Parser}} \rightarrow \boxed{\text{Semantic Analysis}}$$

- In context aware scanning, they communicate

  $$\boxed{\text{Scanner}} \rightleftarrows \boxed{\text{Parser}} \rightarrow \boxed{\text{Semantic Analysis}}$$

Context aware scanning

- Scanner recognizes only tokens valid for current "context"
- keeps embedded sub-languages, in a sense, separate
- Consider:
  - ```
    chan in, out;
    for i in a { a[i] = i*i ; }
    ```
- Two terminal symbols that match "in".
  - `terminal IN 'in' ;`
  - ```
    terminal ID /[a-zA-Z_][a-zA-Z_0-9]*/
                    submits to {promela_kwd };
    ```
  - ```
    terminal FOR 'for' lexer classes {promela_kwd
    };
    ```
- example is part of AbleP [SPIN'11]
    ```

Allows parsing of embedded C code in host Promela

```
c_decl {
  typedef struct Coord {
    int x, y; } Coord;      }

c_state "Coord pt" "Global"  /* goes in state vector */
int z = 3;                   /* standard global decl */

active proctype example()
{ c_code { now.pt.x = now.pt.y = 0; };

  do :: c_expr { now.pt.x == now.pt.y }
          -> c_code { now.pt.y++; }
     :: else -> break
  od;
  };
}
```

Context aware scanning

- We use a slightly modified LR parser and a context-aware scanner.
- Context is based on the LR-parser state.
- Parser passes a set of valid look-ahead terminals to scanner.
- This is the set of terminals with *shift*, *reduce*, or *accept* entries in parse table for current parse state.
- Scanner only returns tokens from the valid look-ahead set.
- Scanner uses this set to disambiguate terminals.
  - *e.g.* SQL_table_kwd and DNF_table_kwd
  - *e.g.* Select_kwd and Identifier

Context aware scanning

- This scanning algorithm subordinates the disambiguation principle of maximal munch to the principle of disambiguation by context.
- It will return a shorter valid match before a longer invalid match.
- In `List<List<Integer>>` before "`>`", "`>`" in valid lookahead but "`>>`" is not.
- A context aware scanner is essentially an implicitly-moded scanner.
- There is no explicit specification of valid look ahead.
  - It is generated from standard grammars and terminal regexs.

- With a smarter scanner, LALR(1) is not so brittle.
- We can build *syntactically* composable language extensions.
- Context aware scanning makes composable syntax "more likely"
- But it does not give a guarantee of composability.

Building a parser from composed specifications.

$$\ldots \; CFG^{H} \cup^{*} \{CFG^{E_1}, \ldots, CFG^{E_n}\}$$

$$\begin{aligned} &\forall i \in [1, n].\mathit{isComposable}(CFG^{H}, CFG^{E_i}) \wedge \\ &\qquad\qquad \mathit{conflictFree}(CFG^{H} \cup CFG^{E_i}) \\ \Rightarrow \quad &\mathit{conflictFree}(CFG^{H} \cup \{CFG^{E_1}, \ldots, CFG^{E_n}\}) \end{aligned}$$

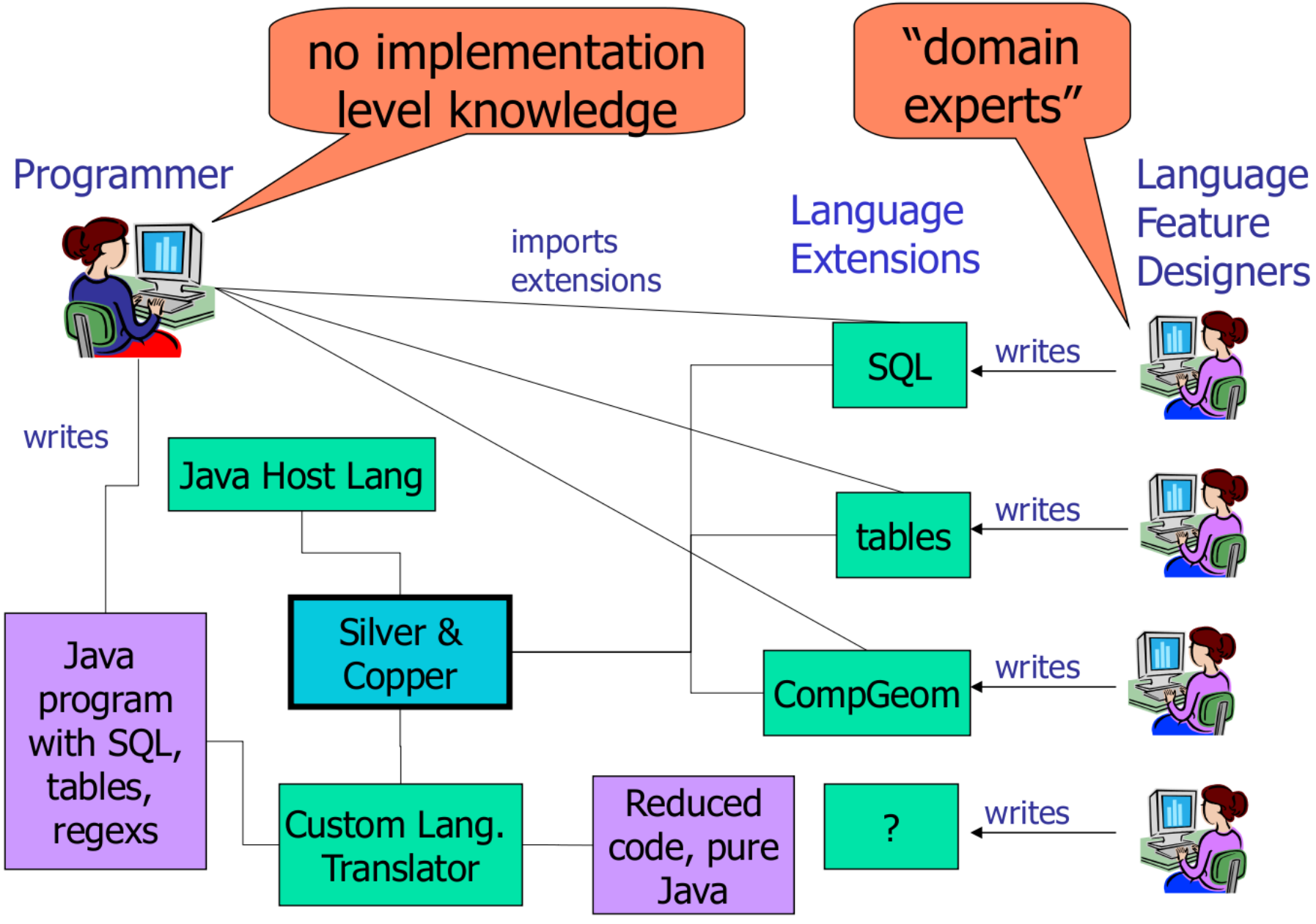
- Monolithic analysis - not too hard, but not too useful.
- Modular analysis - harder, but required [PLDI'09].
- Non-commutative composition of restricted LALR(1) grammars.

Building an attribute grammar evaluator from composed specifications.

$$\ldots\ AG^H \cup^* \{AG^{E_1}, \ldots, AG^{E_n}\}$$

$$\forall i \in [1, n].modComplete(AG^H, AG^{E_i})$$
$$\Rightarrow \quad complete(AG^H \cup \{AG_1^E, \ldots, AG_n^E\})$$

- Monolithic analysis - not too hard, but not too useful.
- Modular analysis - harder, but required [SLE'12a].


no implementation level knowledge
"domain experts"
Programmer
Language Extensions
Language Feature Designers
imports extensions
writes
SQL
writes
tables
writes
CompGeom
writes
?
writes
Java Host Lang
Silver & Copper
Java program with SQL, tables, regexs
Custom Lang. Translator
Reduced code, pure Java

- Modular
  - determinism analysis — scanning, parsing
  - well-definedness — attribute grammars
  - termination — attribute grammars

  are language independent analyses.

- Next: language specific analyses.
  Sebastian Erweg at ICFP'13 - modular type soundness

Expressiveness versus safe composition

Compare to

- other parser generators
- libraries

The modular compositionality analysis does not require context aware scanning.
But, context aware scanning makes it practical.

Tool Support

Copper – context-aware parser and scanner generator

- implements context-aware scanning for a LALR(1) parser
- lexical precedence
- parser attributes and disambiguation functions when disambiguation by context and lexical precedence is not enough
- currently integrated into Silver [SCP'10], an extensible attribute grammar system.

Thanks for your attention.

Questions?

http://melt.cs.umn.edu
evw@cs.umn.edu


Ted Kaminski and Eric Van Wyk.
Modular well-definedness analysis for attribute grammars.
In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 352–371. Springer-Verlag, September 2012.


Lijesh Krishnan and Eric Van Wyk.
Termination analysis for higher-order attribute grammars.
In *Proceedings of the 5th International Conference on Software Language Engineering (SLE 2012)*, volume 7745 of *LNCS*, pages 44–63. Springer-Verlag, September 2012.


Lijesh Krishnan.
Composable Semantics Using Higher-Order Attribute Grammars.
PhD thesis, University of Minnesota, Minnesota, USA, 2012.
`http://melt.cs.umn.edu/pubs/krishnan2012PhD/`.


Yogesh Mali and Eric Van Wyk.
Building extensible specifications and implementations of promela with AbleP.
In *Proc. of Intl. SPIN Workshop on Model Checking of Software*, volume 6823 of *LNCS*, pages 108–125. Springer-Verlag, July 2011.


August Schwerdfeger and Eric Van Wyk.
Verifiable composition of deterministic grammars.
In *Proc. of Conf. on Programming Language Design and Implementation (PLDI)*, pages 199–210. ACM, June 2009.


A. Schwerdfeger and E. Van Wyk.
Verifiable parse table composition for deterministic parsing.
In *2nd International Conference on Software Language Engineering*, volume 5969 of *LNCS*, pages 184–203. Springer-Verlag, 2010.


E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski.
Forwarding in attribute grammars for modular language design.
In *11th Conf. on Compiler Construction (CC)*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.


Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin.
Attribute grammar-based language extensions for Java.
In *Proc. of European Conf. on Object Oriented Prog. (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer-Verlag, 2007.


E. Van Wyk and A. Schwerdfeger.
Context-aware scanning for parsing extensible languages.
In *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*. ACM Press, October 2007.


Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan.
Silver: an extensible attribute grammar system.
Science of Computer Programming, 75(1–2):39–54, January 2010.