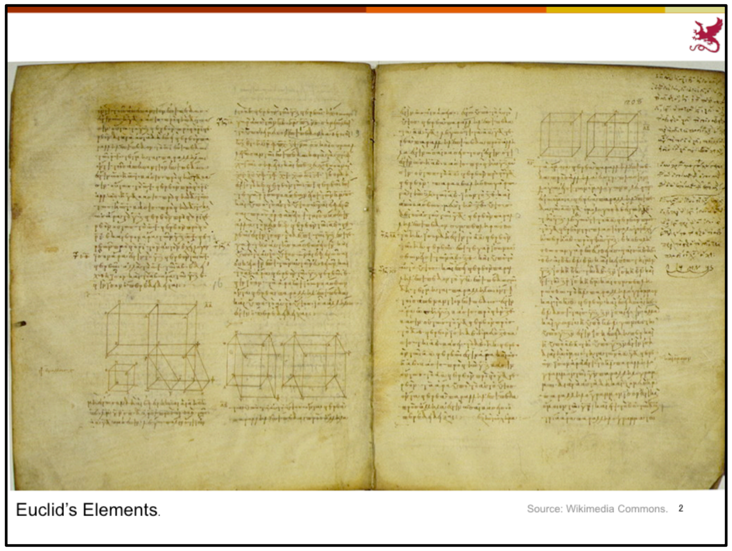
[ECOOP14]

Safely Composable Type-Specific Languages (TSLs)

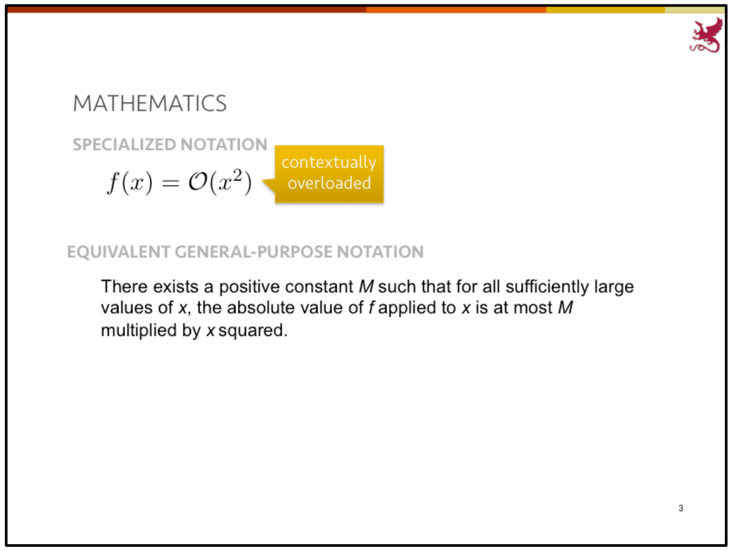
Cyrus Omar
Darya Kurilova
Ligia Nistor
Benjamin Chung
Alex Potanin *(Victoria University of Wellington)*
Jonathan Aldrich

School of Computer Science
Carnegie Mellon University


Euclid's Elements.

Source: Wikimedia Commons. 2

In Euclid's days, mathematics was mostly communicated in "long form", that is in prose. As you can see – a lot of words, some diagrams, but no symbolic constructions. In fact, modern mathematical notation didn't arise until about the 1600s.



MATHEMATICS

SPECIALIZED NOTATION

$f(x) = \mathcal{O}(x^2)$

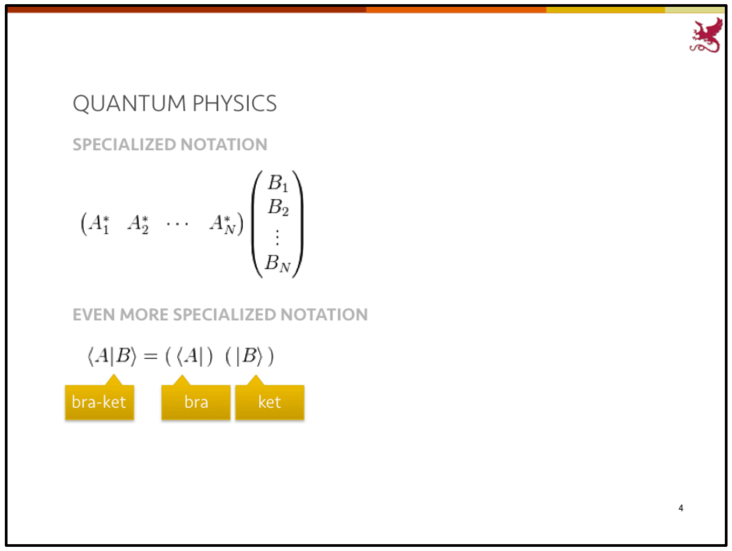
contextually overloaded

EQUIVALENT GENERAL-PURPOSE NOTATION

There exists a positive constant *M* such that for all sufficiently large values of *x*, the absolute value of *f* applied to *x* is at most *M* multiplied by *x* squared.

3

We've come a long way since then. Today, when we want to make a proposition about, say, the asymptoti c behavior of a function, we have a specialized notation available. This communicates the same idea muc h more efficiently, both for reader and writer, decreasing mental register pressure, if you will. In mathemati cs and mathematical science, coming up with new notation and, as in this example, repurposing existing notation has become almost a sport.



QUANTUM PHYSICS

SPECIALIZED NOTATION

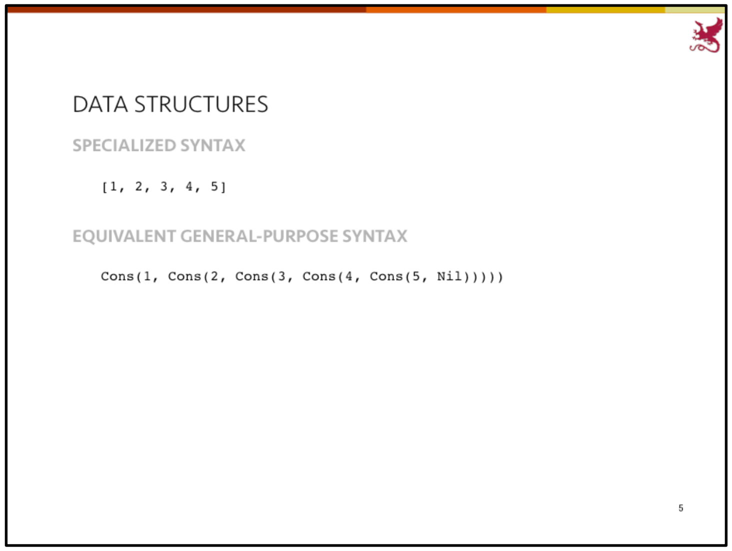
$$\begin{pmatrix} A_1^* & A_2^* & \cdots & A_N^* \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{pmatrix}$$

EVEN MORE SPECIALIZED NOTATION

$$\underbrace{\langle A|B\rangle}_{\text{bra-ket}} = (\underbrace{\langle A|}_{\text{bra}}) \; (\underbrace{|B\rangle}_{\text{ket}})$$

4

This process is almost fractal in nature. Mathematicians constantly develop new notations, often layering and composing prior notations, to capture the common idioms in their domain. For example, Dirac develo ped bra-ket notation in recognition of how common this particular combination of operations is in quantum physics. Bra-kets are composed of bra’s and ket’s.



DATA STRUCTURES

SPECIALIZED SYNTAX

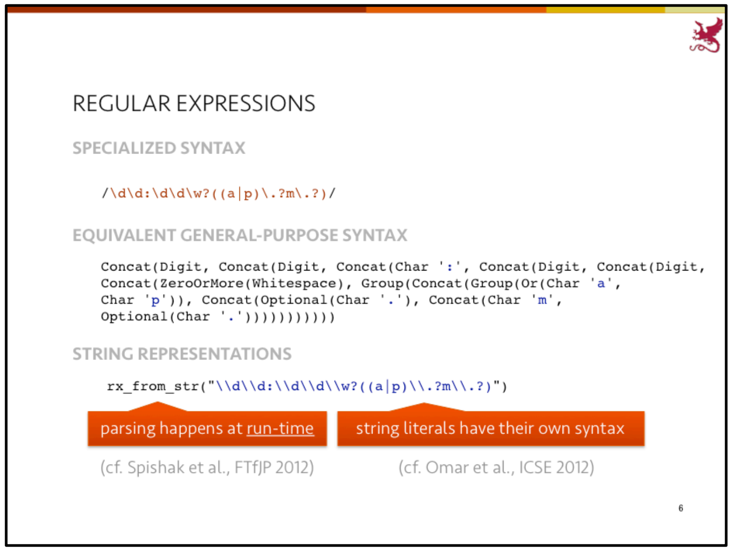
```
[1, 2, 3, 4, 5]
```

EQUIVALENT GENERAL-PURPOSE SYNTAX

```
Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
```

5

The concrete syntax of programming languages also includes specialized notations. For example, commo
n data structures like lists are often privileged with literal syntax like this.



REGULAR EXPRESSIONS

SPECIALIZED SYNTAX

```
/\d\d:\d\d\w?((a|p)\.?m\.?)/
```

EQUIVALENT GENERAL-PURPOSE SYNTAX

```
Concat(Digit, Concat(Digit, Concat(Char ':', Concat(Digit, Concat(Digit,
Concat(ZeroOrMore(Whitespace), Group(Concat(Group(Or(Char 'a',
Char 'p')), Concat(Optional(Char '.'), Concat(Char 'm',
Optional(Char '.'))))))))))
```

STRING REPRESENTATIONS

```
rx_from_str("\\d\\d:\\d\\d\\w?((a|p)\\.?m\\.?)")
```

parsing happens at run-time

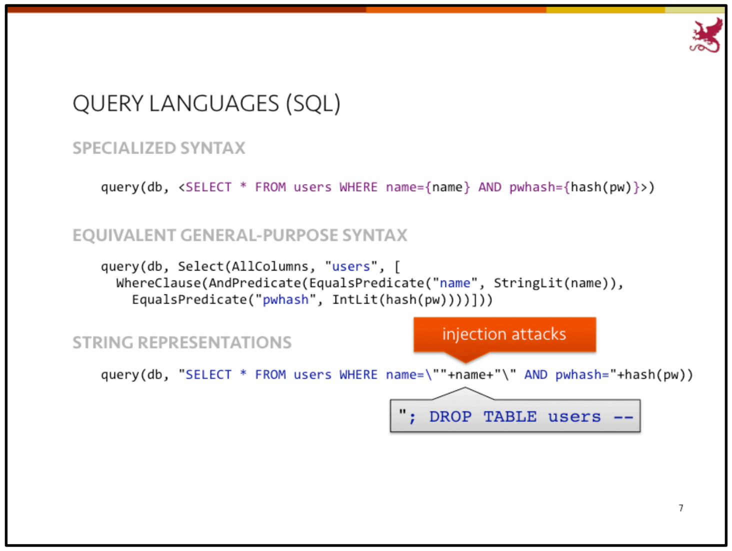
string literals have their own syntax

(cf. Spishak et al., FTfJP 2012)

(cf. Omar et al., ICSE 2012)

6

A few languages stray farther than this. For example some languages also include concrete syntax for reg ular expressions. You can achieve the same semantics in languages without this syntax, but to do it, you have to essentially encode the abstract syntax of regular expressions using data types. That is, you have to give up the benefits of concrete syntax. All too often, this is too hard a bargain, so programmers start s acrificing some semantic properties to partially recover the syntactic convenience of built in syntax. The m ost common technique is to use string representations. This is, we argue, a bad idea, for several reasons.



QUERY LANGUAGES (SQL)

SPECIALIZED SYNTAX

```
query(db, <SELECT * FROM users WHERE name={name} AND pwhash={hash(pw)}>)
```

EQUIVALENT GENERAL-PURPOSE SYNTAX

```
query(db, Select(AllColumns, "users", [
  WhereClause(AndPredicate(EqualsPredicate("name", StringLit(name)),
    EqualsPredicate("pwhash", IntLit(hash(pw))))]))
```

STRING REPRESENTATIONS

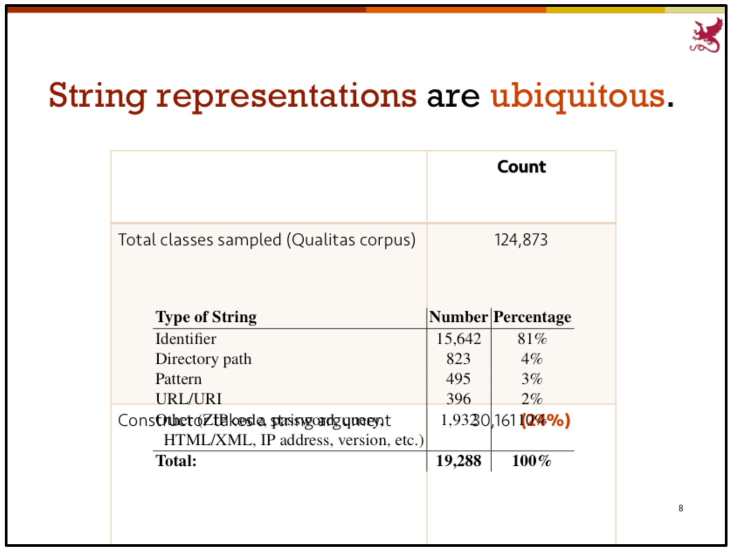
injection attacks

```
query(db, "SELECT * FROM users WHERE name=\""+name+"\" AND pwhash="+hash(pw))
```

```
"; DROP TABLE users --
```

7

Moreover, this can cause major security vulnerabilities. For example, consider SQL queries. Some langua ges include syntactic support for queries (LINQ), but to achieve the same semantics in other languages re quires again essentially encoding the abstract syntax of SQL using data types. So again, users attempt to recover the concrete syntax of SQL by using string representations, to devastating effect – injection attack s are one of the most serious security threats on the web today. Trivializing this compulsion is, I think, a m istake for the research community.



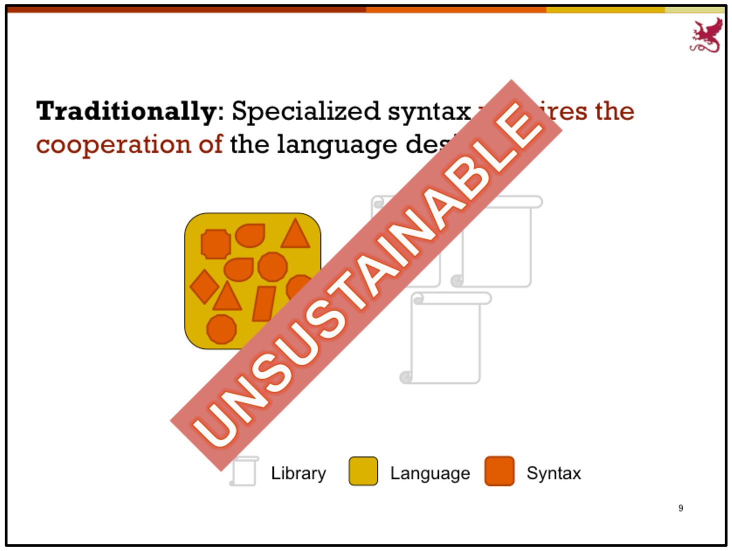
String representations are ubiquitous.

	Count
Total classes sampled (Qualitas corpus)	124,873
Constructor takes a string argument	30,161 (24%)

Type of String	Number	Percentage
Identifier	15,642	81%
Directory path	823	4%
Pattern	495	3%
URL/URI	396	2%
Other (ZIP code, password, query, HTML/XML, IP address, version, etc.)	1,932	10%
Total:	**19,288**	**100%**

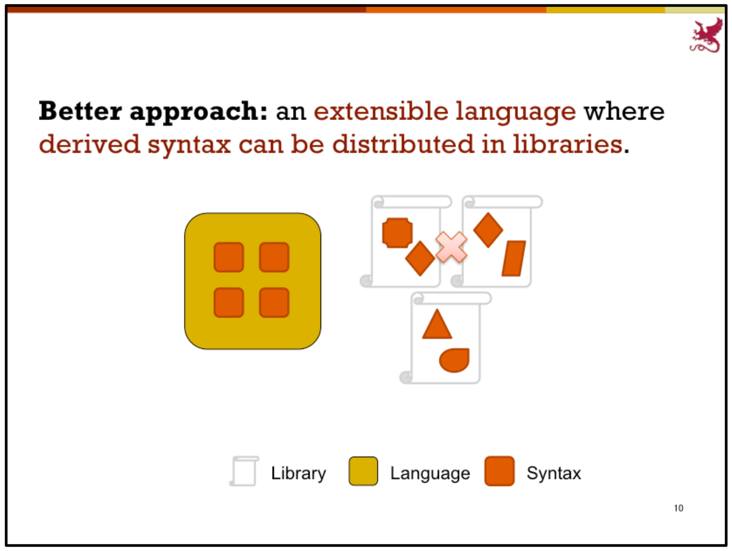
8

Indeed, these aren't isolated examples. In our paper, we conducted a small corpus analysis of Java progr ams, looking at constructor arguments of type String. We found that a rather alarming fraction of these stri ng arguments actually appear to capture some sort of parseable concrete syntax for something that might be encoded as a data type. You can see the breakdown in the table below – a lot of them are identifiers, but there are hundreds of other examples.



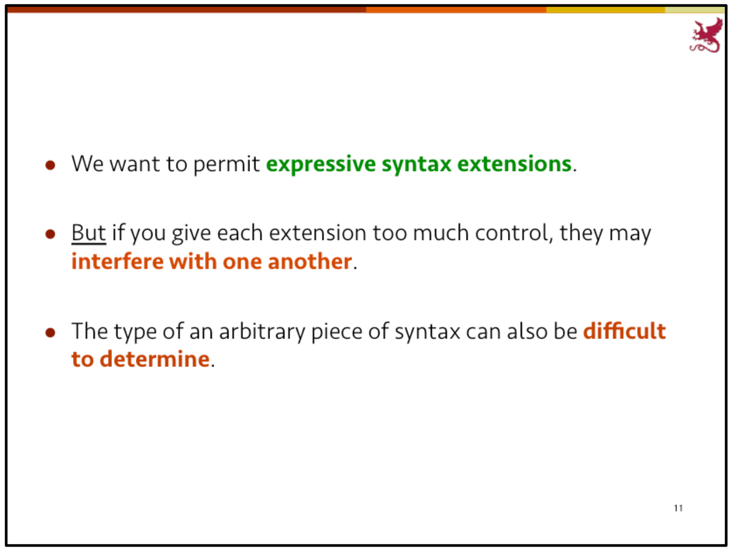
Traditionally: Specialized syntax ires the cooperation of the language des

If we want to decrease the cost of using richer representations, this data presents a major challenge to th e traditional PL design methodology, where new concrete syntax is slowly added to a language by its desi gners. Indeed, this methodology is unsustainable.

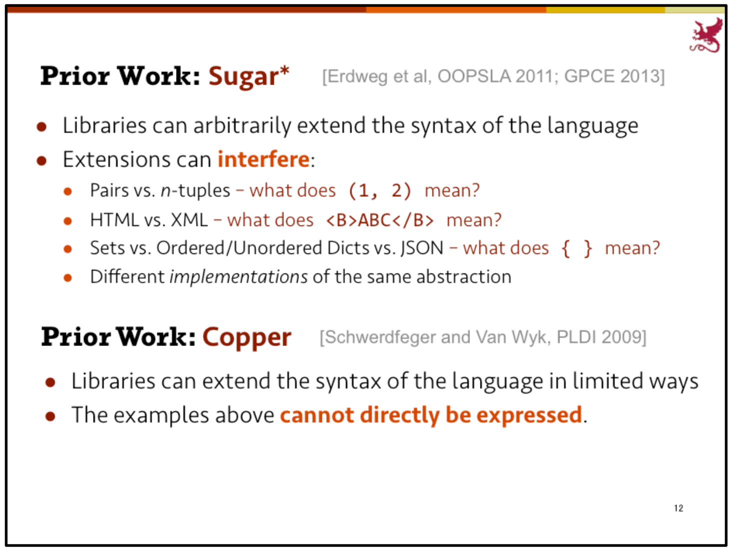


Better approach: an extensible language where derived syntax can be distributed in libraries.

A better approach, and one that more closely tracks mathematical practice, is to create an extensible lang uage building in clean, uniform syntax for the foundational constructs, but leaving derived forms to library authors.



- We want to permit **expressive syntax extensions**.
- But if you give each extension too much control, they may **interfere with one another**.
- The type of an arbitrary piece of syntax can also be **difficult to determine**.



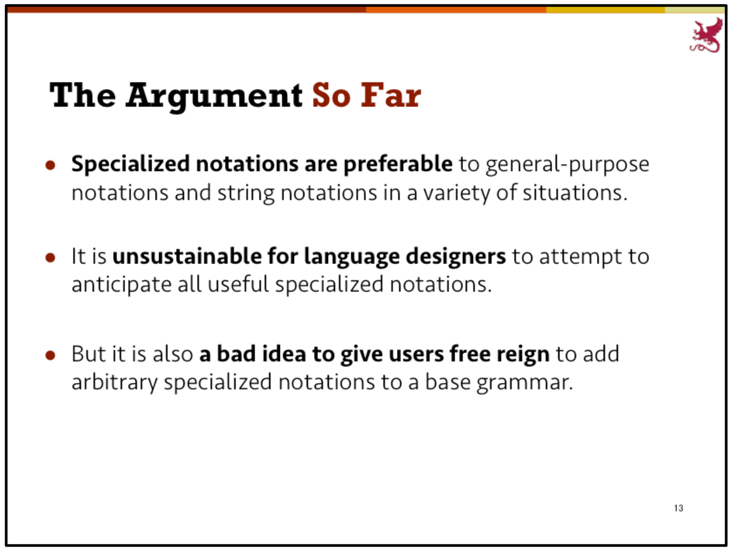
Prior Work: Sugar* [Erdweg et al, OOPSLA 2011; GPCE 2013]

- Libraries can arbitrarily extend the syntax of the language
- Extensions can **interfere**:
  - Pairs vs. *n*-tuples – what does `(1, 2)` mean?
  - HTML vs. XML – what does `<B>ABC</B>` mean?
  - Sets vs. Ordered/Unordered Dicts vs. JSON – what does `{ }` mean?
  - Different *implementations* of the same abstraction

Prior Work: Copper [Schwerdfeger and Van Wyk, PLDI 2009]

- Libraries can extend the syntax of the language in limited ways
- The examples above **cannot directly be expressed**.

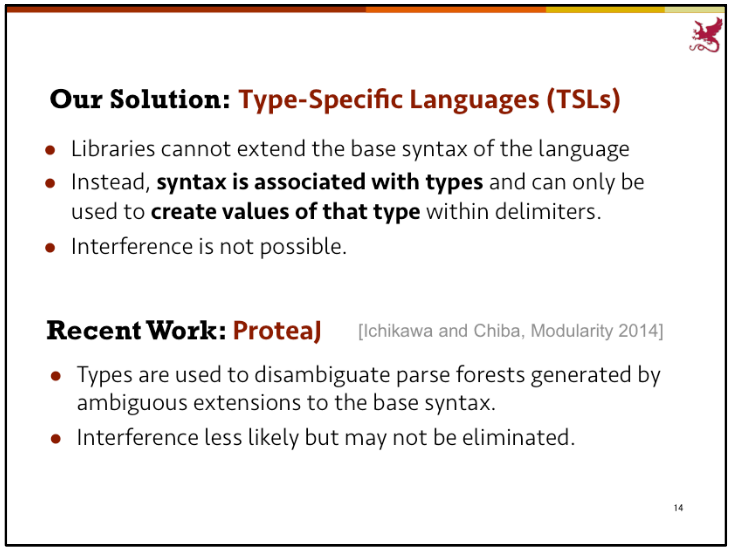
12



The Argument So Far

- **Specialized notations are preferable** to general-purpose notations and string notations in a variety of situations.
- It is **unsustainable for language designers** to attempt to anticipate all useful specialized notations.
- But it is also **a bad idea to give users free reign** to add arbitrary specialized notations to a base grammar.

13



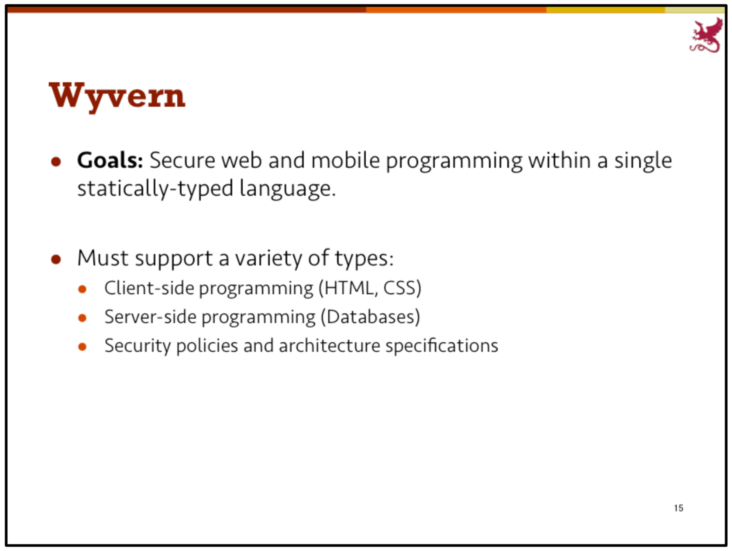
Our Solution: Type-Specific Languages (TSLs)

- Libraries cannot extend the base syntax of the language
- Instead, **syntax is associated with types** and can only be used to **create values of that type** within delimiters.
- Interference is not possible.

Recent Work: ProteaJ [Ichikawa and Chiba, Modularity 2014]

- Types are used to disambiguate parse forests generated by ambiguous extensions to the base syntax.
- Interference less likely but may not be eliminated.

14

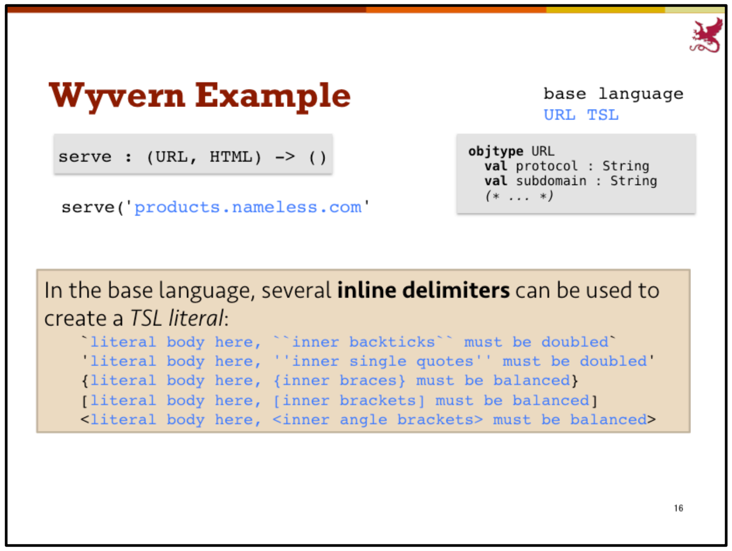


Wyvern

- **Goals:** Secure web and mobile programming within a single statically-typed language.
- Must support a variety of types:
  - Client-side programming (HTML, CSS)
  - Server-side programming (Databases)
  - Security policies and architecture specifications

15

We present our solution in the paper as a language design for Wyvern, a language being developed by o
ur group to support secure web and mobile programming within a single statically-typed language.



Wyvern Example

base language
URL TSL

```
serve : (URL, HTML) -> ()
```

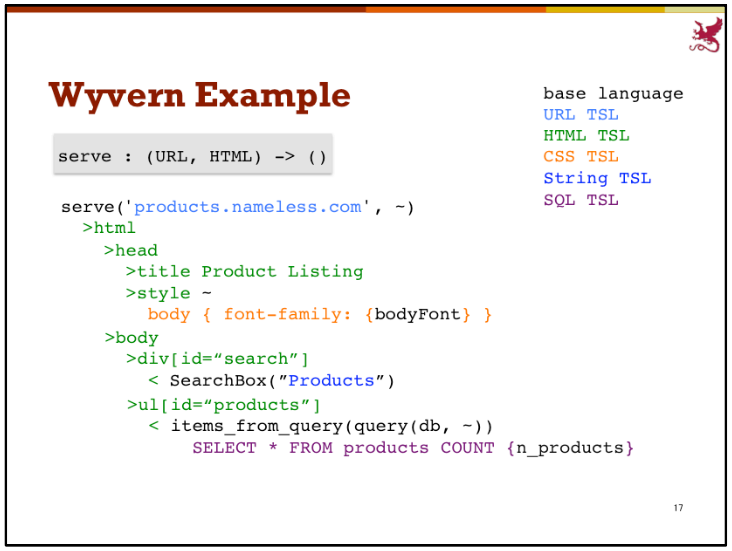
```
serve('products.nameless.com'
```

```
objtype URL
  val protocol : String
  val subdomain : String
  (* ... *)
```

In the base language, several **inline delimiters** can be used to create a *TSL literal*:

```
    `literal body here, ``inner backticks`` must be doubled`
    'literal body here, ''inner single quotes'' must be doubled'
    {literal body here, {inner braces} must be balanced}
    [literal body here, [inner brackets] must be balanced]
    <literal body here, <inner angle brackets> must be balanced>
```

16



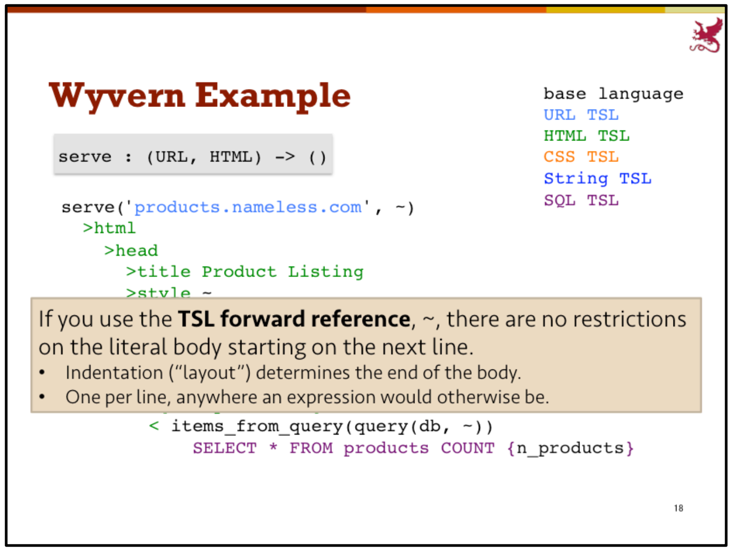
Wyvern Example

```
base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL
```

```
serve : (URL, HTML) -> ()
```

```
serve('products.nameless.com', ~)
  >html
    >head
      >title Product Listing
      >style ~
        body { font-family: {bodyFont} }
    >body
      >div[id="search"]
        < SearchBox("Products")
      >ul[id="products"]
        < items_from_query(query(db, ~))
            SELECT * FROM products COUNT {n_products}
```

17



Wyvern Example

```
serve : (URL, HTML) -> ()
```

```
serve('products.nameless.com', ~)
  >html
    >head
      >title Product Listing
      >style ~
```

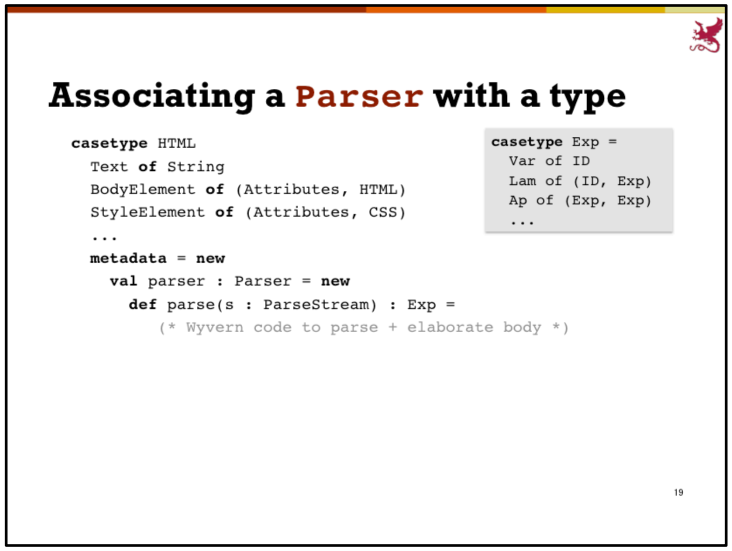
If you use the **TSL forward reference**, ~, there are no restrictions on the literal body starting on the next line.

- Indentation (“layout”) determines the end of the body.
- One per line, anywhere an expression would otherwise be.

```
          < items_from_query(query(db, ~))
              SELECT * FROM products COUNT {n_products}
```

```
base language
URL TSL
HTML TSL
CSS TSL
String TSL
SQL TSL
```

18

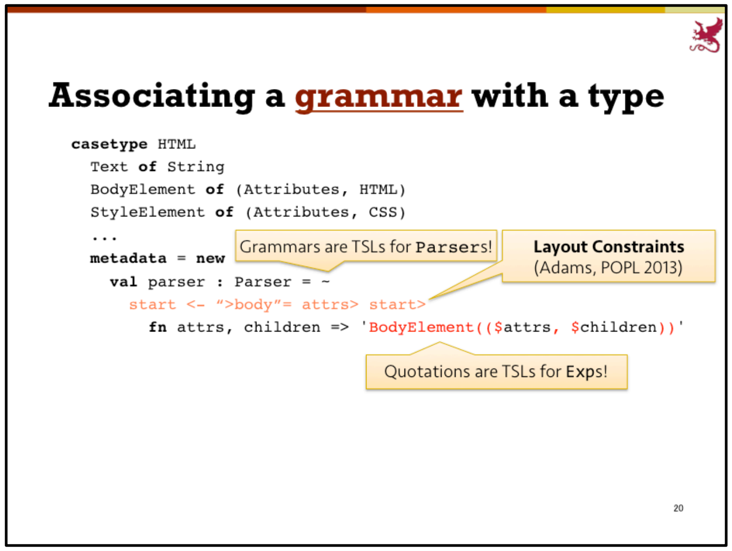


Associating a Parser with a type

```
casetype HTML
  Text of String
  BodyElement of (Attributes, HTML)
  StyleElement of (Attributes, CSS)
  ...
  metadata = new
    val parser : Parser = new
      def parse(s : ParseStream) : Exp =
         (* Wyvern code to parse + elaborate body *)
```

```
casetype Exp =
  Var of ID
  Lam of (ID, Exp)
  Ap of (Exp, Exp)
  ...
```

19



Associating a grammar with a type

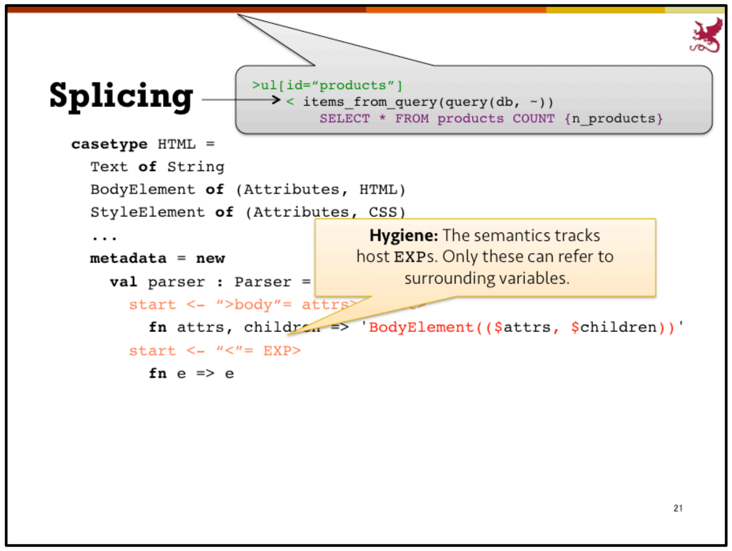
```
casetype HTML
  Text of String
  BodyElement of (Attributes, HTML)
  StyleElement of (Attributes, CSS)
  ...
  metadata = new
    val parser : Parser = ~
      start <- “>body”= attrs> start>
        fn attrs, children => 'BodyElement(($attrs, $children))'
```

Grammars are TSLs for `Parsers`!

Layout Constraints
(Adams, POPL 2013)

Quotations are TSLs for **Exp**s!

20



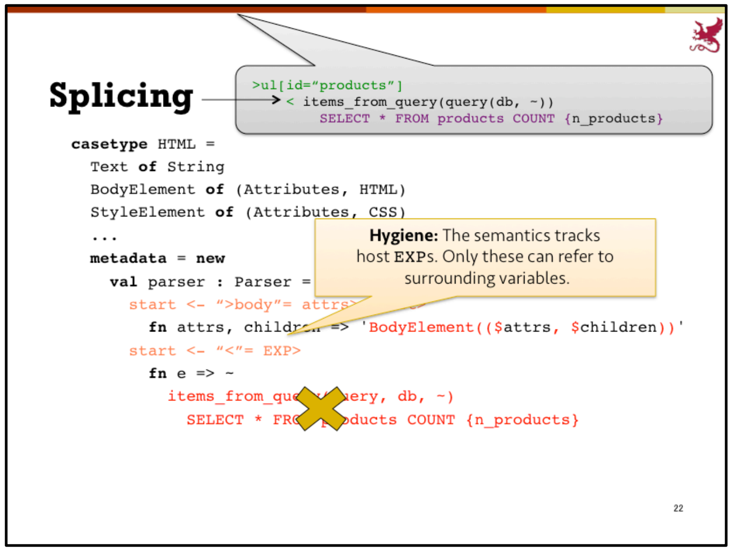
Splicing

```
>ul[id=“products”]
  < items_from_query(query(db, ~))
      SELECT * FROM products COUNT {n_products}
```

```
casetype HTML =
  Text of String
  BodyElement of (Attributes, HTML)
  StyleElement of (Attributes, CSS)
  ...
  metadata = new
    val parser : Parser =
      start <- “>body”= attrs>
        fn attrs, children => 'BodyElement(($attrs, $children))'
      start <- “<”= EXP>
        fn e => e
```

Hygiene: The semantics tracks host EXPs. Only these can refer to surrounding variables.

21



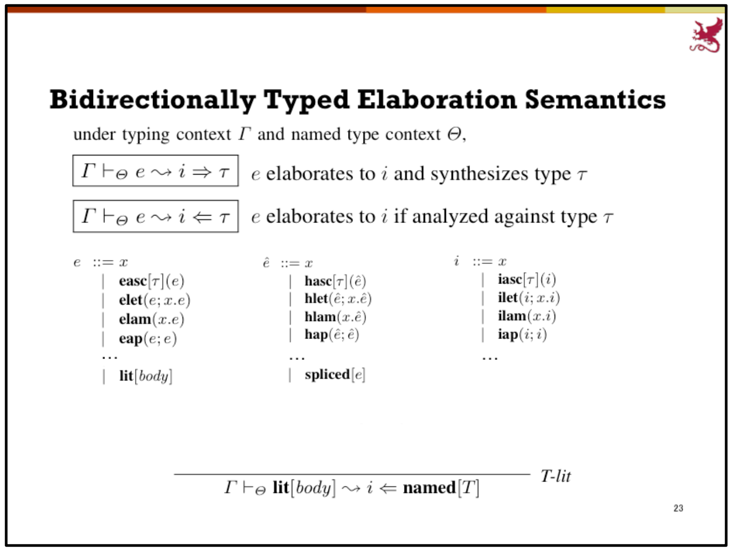
Splicing

```
>ul[id="products"]
  < items_from_query(query(db, ~))
      SELECT * FROM products COUNT {n_products}
```

```
casetype HTML =
  Text of String
  BodyElement of (Attributes, HTML)
  StyleElement of (Attributes, CSS)
  ...
  metadata = new
    val parser : Parser =
      start <- ">body"= attrs>
        fn attrs, childr    => 'BodyElement(($attrs, $children))'
      start <- "<"= EXP>
        fn e => ~
          items_from_que   uery, db, ~)
            SELECT * FRO   oducts COUNT {n_products}
```

Hygiene: The semantics tracks host EXPs. Only these can refer to surrounding variables.

22



Bidirectionally Typed Elaboration Semantics

under typing context Γ and named type context Θ,

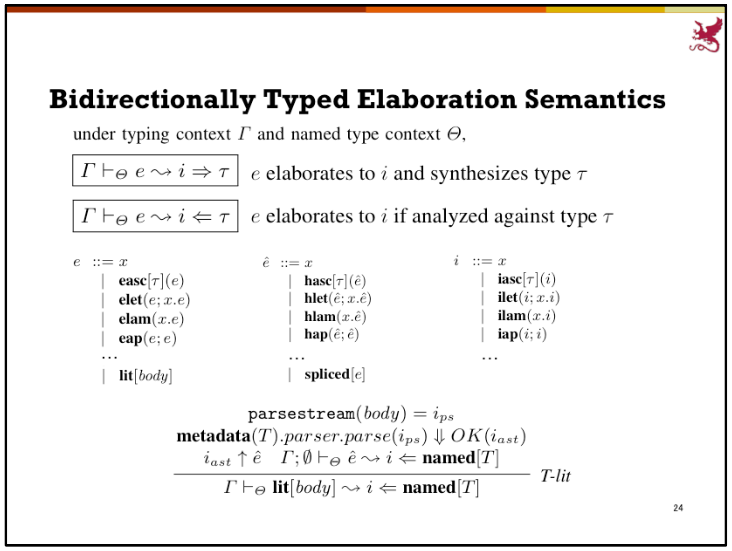
$\boxed{\Gamma \vdash_\Theta e \leadsto i \Rightarrow \tau}$ e elaborates to i and synthesizes type τ

$\boxed{\Gamma \vdash_\Theta e \leadsto i \Leftarrow \tau}$ e elaborates to i if analyzed against type τ

$$\begin{array}{lll}
e ::= x & \hat{e} ::= x & i ::= x \\
\quad \mid \ \mathbf{easc}[\tau](e) & \quad \mid \ \mathbf{hasc}[\tau](\hat{e}) & \quad \mid \ \mathbf{iasc}[\tau](i) \\
\quad \mid \ \mathbf{elet}(e; x.e) & \quad \mid \ \mathbf{hlet}(\hat{e}; x.\hat{e}) & \quad \mid \ \mathbf{ilet}(i; x.i) \\
\quad \mid \ \mathbf{elam}(x.e) & \quad \mid \ \mathbf{hlam}(x.\hat{e}) & \quad \mid \ \mathbf{ilam}(x.i) \\
\quad \mid \ \mathbf{eap}(e; e) & \quad \mid \ \mathbf{hap}(\hat{e}; \hat{e}) & \quad \mid \ \mathbf{iap}(i; i) \\
\quad \cdots & \quad \cdots & \quad \cdots \\
\quad \mid \ \mathbf{lit}[body] & \quad \mid \ \mathbf{spliced}[e] &
\end{array}$$

$$\frac{}{\Gamma \vdash_\Theta \mathbf{lit}[body] \leadsto i \Leftarrow \mathbf{named}[T]}\ \textit{T-lit}$$

23



Bidirectionally Typed Elaboration Semantics

under typing context $\mathit{\Gamma}$ and named type context $\mathit{\Theta}$,

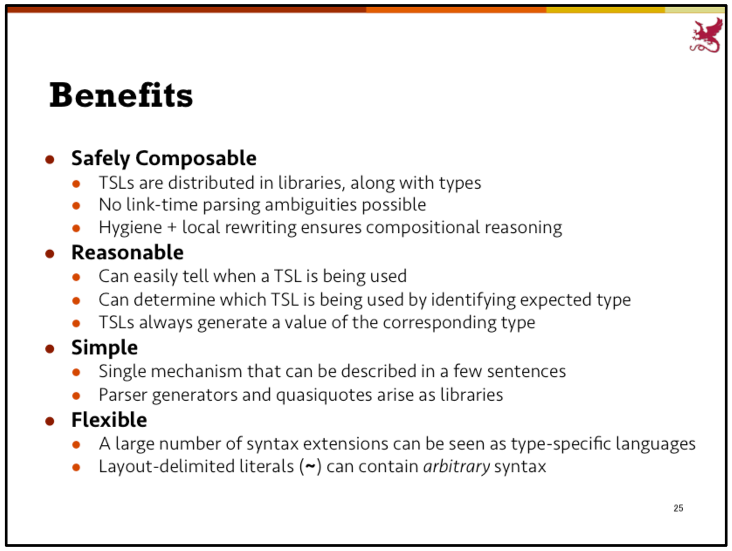
$\boxed{\Gamma \vdash_{\Theta} e \rightsquigarrow i \Rightarrow \tau}$ e elaborates to i and synthesizes type τ

$\boxed{\Gamma \vdash_{\Theta} e \rightsquigarrow i \Leftarrow \tau}$ e elaborates to i if analyzed against type τ

$$\begin{array}{lll}
e ::= x & \hat{e} ::= x & i ::= x \\
\quad \mid \mathbf{easc}[\tau](e) & \quad \mid \mathbf{hasc}[\tau](\hat{e}) & \quad \mid \mathbf{iasc}[\tau](i) \\
\quad \mid \mathbf{elet}(e; x.e) & \quad \mid \mathbf{hlet}(\hat{e}; x.\hat{e}) & \quad \mid \mathbf{ilet}(i; x.i) \\
\quad \mid \mathbf{elam}(x.e) & \quad \mid \mathbf{hlam}(x.\hat{e}) & \quad \mid \mathbf{ilam}(x.i) \\
\quad \mid \mathbf{eap}(e; e) & \quad \mid \mathbf{hap}(\hat{e}; \hat{e}) & \quad \mid \mathbf{iap}(i; i) \\
\quad \cdots & \quad \cdots & \quad \cdots \\
\quad \mid \mathbf{lit}[body] & \quad \mid \mathbf{spliced}[e] &
\end{array}$$

$$\frac{\begin{array}{c}\texttt{parsestream}(body) = i_{ps} \\ \mathbf{metadata}(T).parser.parse(i_{ps}) \Downarrow OK(i_{ast}) \\ i_{ast} \uparrow \hat{e} \quad \Gamma; \emptyset \vdash_{\Theta} \hat{e} \rightsquigarrow i \Leftarrow \mathbf{named}[T]\end{array}}{\Gamma \vdash_{\Theta} \mathbf{lit}[body] \rightsquigarrow i \Leftarrow \mathbf{named}[T]}\ \textit{T-lit}$$

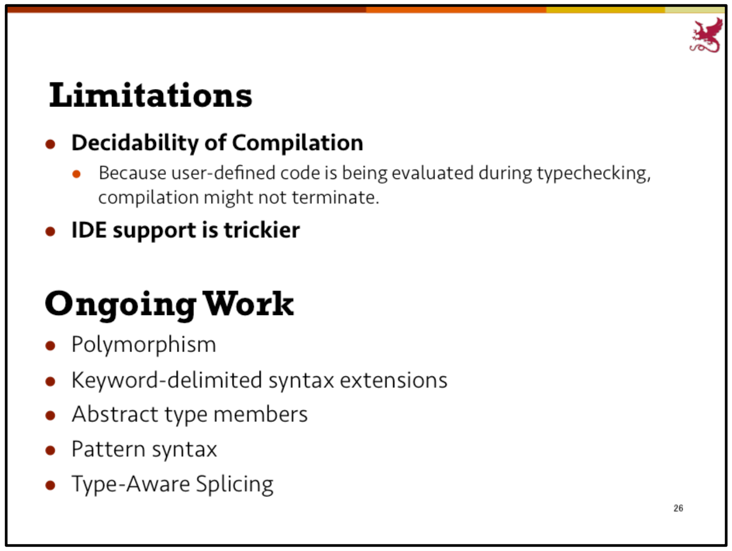
24



Benefits

- **Safely Composable**
  - TSLs are distributed in libraries, along with types
  - No link-time parsing ambiguities possible
  - Hygiene + local rewriting ensures compositional reasoning
- **Reasonable**
  - Can easily tell when a TSL is being used
  - Can determine which TSL is being used by identifying expected type
  - TSLs always generate a value of the corresponding type
- **Simple**
  - Single mechanism that can be described in a few sentences
  - Parser generators and quasiquotes arise as libraries
- **Flexible**
  - A large number of syntax extensions can be seen as type-specific languages
  - Layout-delimited literals (**~**) can contain *arbitrary* syntax

25



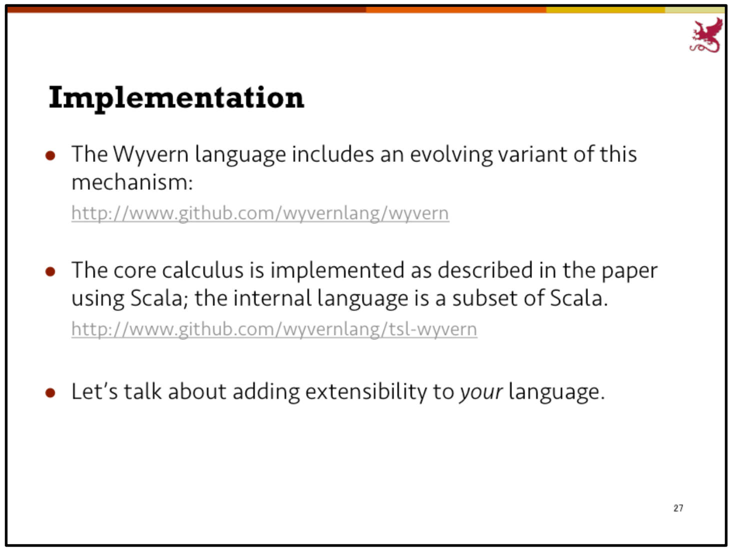
Limitations

- **Decidability of Compilation**
  - Because user-defined code is being evaluated during typechecking, compilation might not terminate.
- **IDE support is trickier**

Ongoing Work

- Polymorphism
- Keyword-delimited syntax extensions
- Abstract type members
- Pattern syntax
- Type-Aware Splicing

26



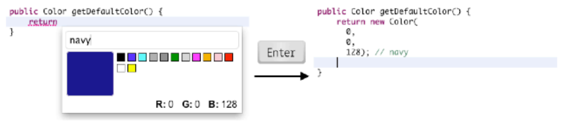
Implementation

- The Wyvern language includes an evolving variant of this mechanism:
  http://www.github.com/wyvernlang/wyvern
- The core calculus is implemented as described in the paper using Scala; the internal language is a subset of Scala.
  http://www.github.com/wyvernlang/tsl-wyvern
- Let's talk about adding extensibility to *your* language.

27

Types Organize Languages

- Types represent an organizational unit for programming languages and systems.
- They can be used for more than just ensuring that programs cannot go wrong:
  - **Syntax extensions** (this work)
  - **IDE extensions** (Omar et al., “Active Code Completion”, ICSE 2012)

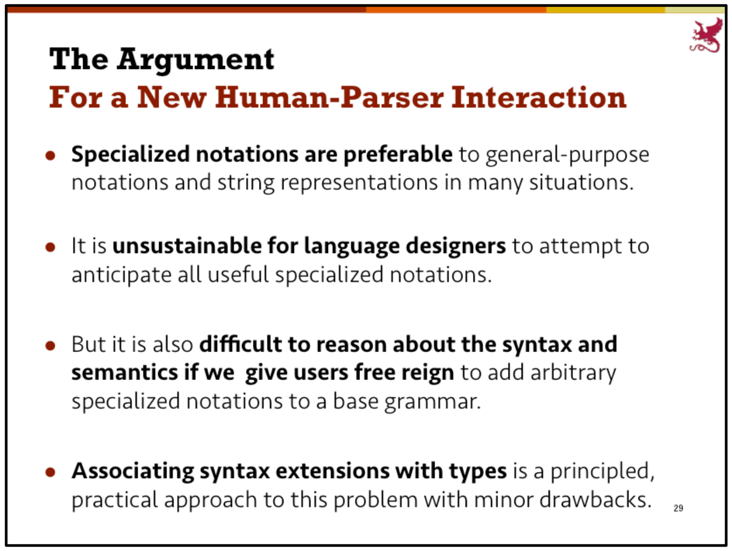


```
public Color getDefaultColor() {
    return
}
```

```
public Color getDefaultColor() {
    return new Color(
        0,
        0,
        128); // navy
    
}
```

(a) (b)

  - **Type system extensions** (submitted)



The Argument
For a New Human-Parser Interaction

- **Specialized notations are preferable** to general-purpose notations and string representations in many situations.
- It is **unsustainable for language designers** to attempt to anticipate all useful specialized notations.
- But it is also **difficult to reason about the syntax and semantics if we give users free reign** to add arbitrary specialized notations to a base grammar.
- **Associating syntax extensions with types** is a principled, practical approach to this problem with minor drawbacks.