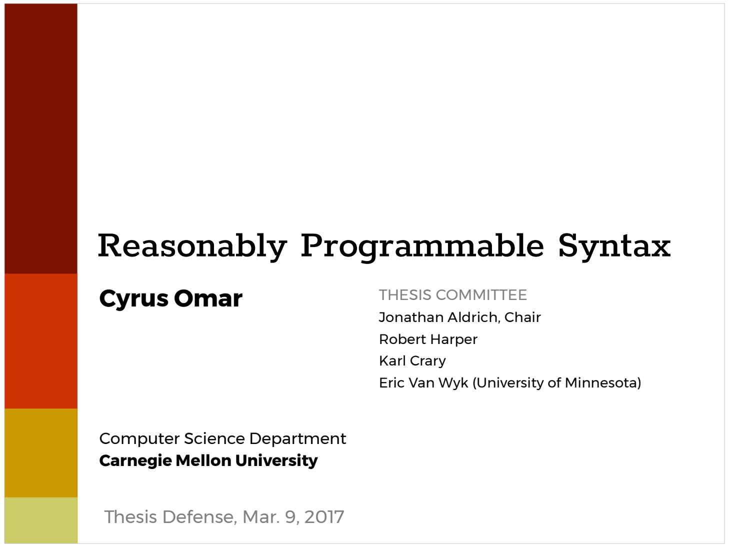


Reasonably Programmable Syntax

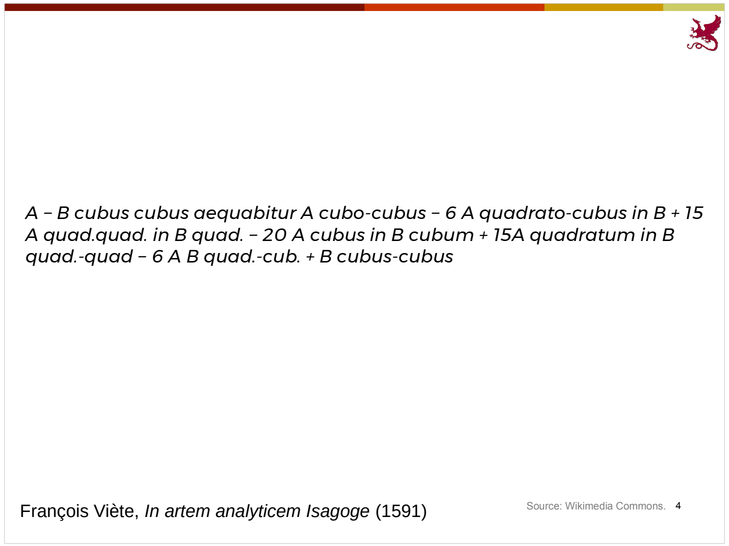
Cyrus Omar

THESIS COMMITTEE
Jonathan Aldrich, Chair
Robert Harper
Karl Crary
Eric Van Wyk (University of Minnesota)

Computer Science Department
Carnegie Mellon University

Thesis Defense, Mar. 9, 2017

Hey everyone, thanks for being here at my defense.



A – B cubus cubus aequabitur A cubo-cubus – 6 A quadrato-cubus in B + 15 A quad.quad. in B quad. – 20 A cubus in B cubum + 15A quadratum in B quad.-quad – 6 A B quad.-cub. + B cubus-cubus

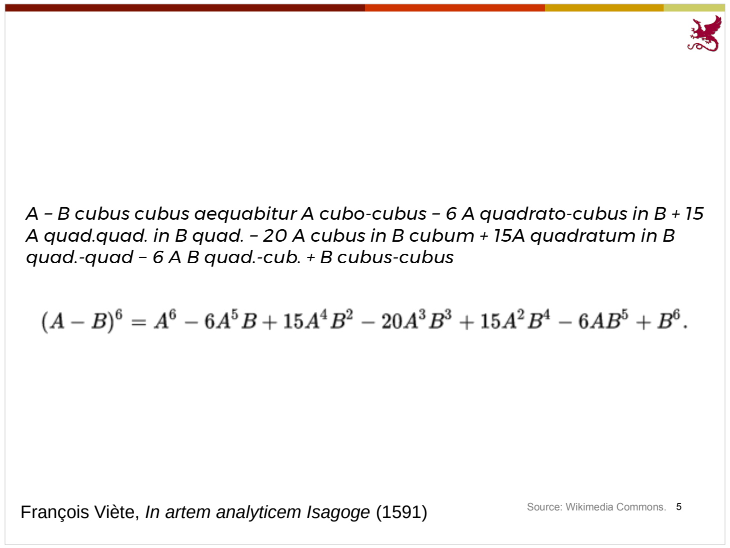
François Viète, *In artem analyticem Isagoge* (1591)

Source: Wikimedia Commons. 4

So I want to start with just a little bit of historical context. If you were a mathematician in 1591, your writing would look like this – you'd have variables like A and B – those had been around for a while – and you had notation for addition and subtraction, that had been invented only about 50 years earlier, but everything else, all the other operations and connectives in your mathematics would be written out in full Latin sentences.

What you're looking at is actually an equation – it's the expansion of A – B to the sixth power…

Now as you can imagine, as mathematicians started considering more sophisticated structures, this style became unwieldy, and indeed today ...



A – B cubus cubus aequabitur A cubo-cubus – 6 A quadrato-cubus in B + 15 A quad.quad. in B quad. – 20 A cubus in B cubum + 15A quadratum in B quad.-quad – 6 A B quad.-cub. + B cubus-cubus

$$(A - B)^6 = A^6 - 6A^5B + 15A^4B^2 - 20A^3B^3 + 15A^2B^4 - 6AB^5 + B^6.$$

François Viète, *In artem analyticem Isagoge* (1591)

Source: Wikimedia Commons. 5

... we would notate the same equation in this way. Using additional notational conventions that you're surely familiar with.

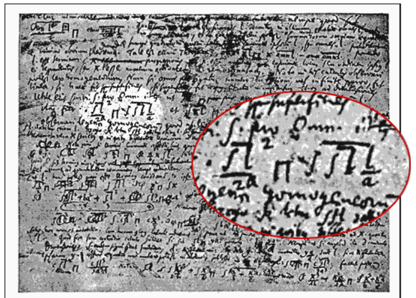
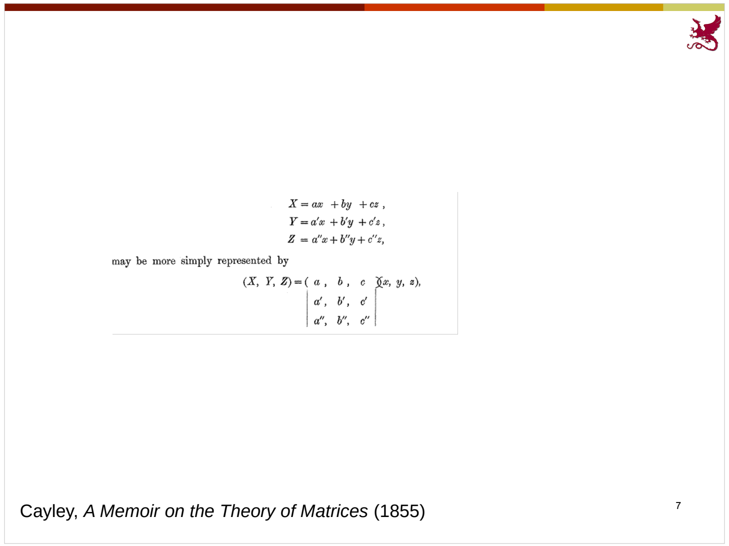


FIG. 124.—Facsimile of manuscript of Leibniz, dated Oct. 29, 1675, in which his sign of integration first appears. (Taken from C. I. Gerhardt's *Briefwechsel von G. W. Leibniz mit Mathematikern* [1899].)

Source: Wikimedia Commons. 6

This has become the usual thing – as mathematics advances, new notation follows.

Here a century later is Leibniz, who was a big fan of new notation, introducing the now-familiar symbol for integration.



$$X = ax + by + cz\,,$$
$$Y = a'x + b'y + c'z\,,$$
$$Z = a''x + b''y + c''z,$$

may be more simply represented by

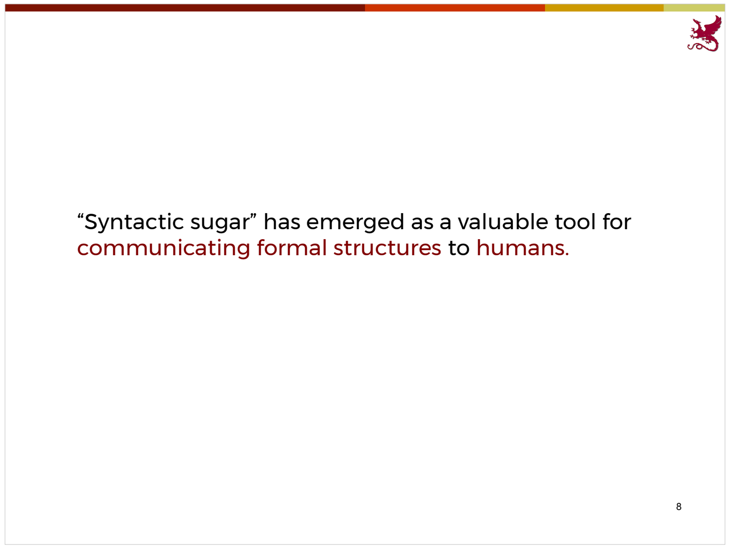
$$(X,\ Y,\ Z) = \left(\begin{array}{ccc} a\,, & b\,, & c \\ a'\,, & b'\,, & c' \\ a''\,, & b''\,, & c'' \end{array}\right)(x,\ y,\ z),$$

Cayley, *A Memoir on the Theory of Matrices* (1855)

7

This is Cayley with an early take on modern matrix notation. Still in it’s awkward teenage years.

There are many many more examples throughout modern mathematics and science.



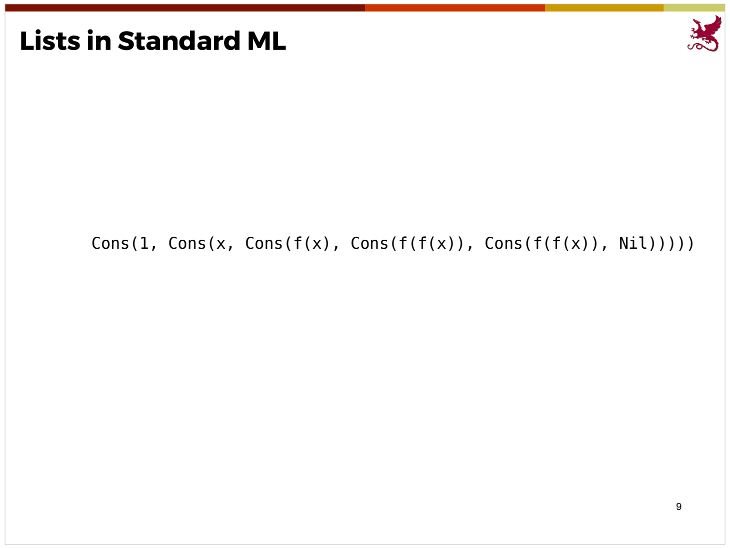
"Syntactic sugar" has emerged as a valuable tool for communicating formal structures to humans.

8

And this practice of inventing specialized syntactic forms strictly to abbreviate certain common idioms more concisely or somehow more suggestively has continued to this day – it has emerged…

And I do want to emphasize that this is about humans. There's the formal structure itself, which you might think of Platonically, and then there is the drawing and this too is a thing that you can also study and tweak. So keep that mind.

Now syntactic sugar is valuable not only in mathematical writing, but also in programming.



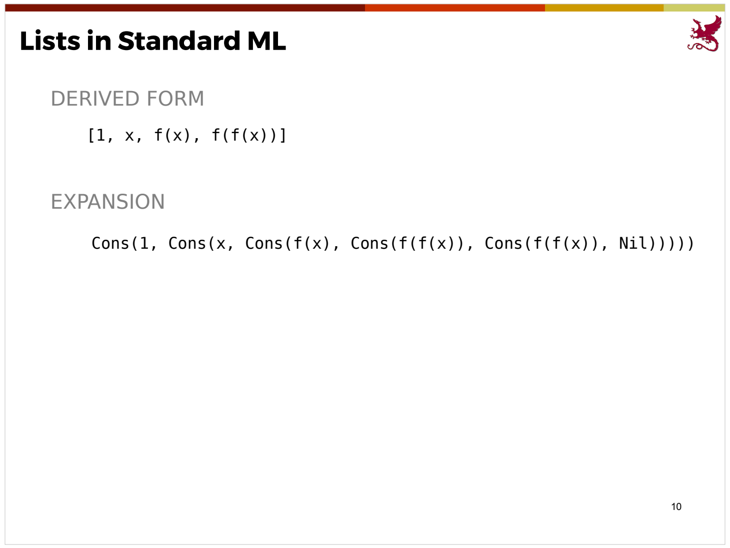
Lists in Standard ML

```
Cons(1, Cons(x, Cons(f(x), Cons(f(f(x)), Cons(f(f(x)), Nil)))))
```

9

So consider a general-purpose language like Standard ML where you can define a variety of datatypes, for example the list datatype which defines two constructors, Nil and Cons. You can string those together to form list expressions like this.

Semantically, this is great. But if you look at this as a drawing of a list, it’s a bit unsatisfying. In fact, if you’ll allow me to be a little bit facetious, it should remind you a bit of mathematics circa the 16th century where you have variables and numerals but everything else is written out laboriously in words. So that’s unsatisfying. Fortunately, the designers of Standard ML thought to include


Lists in Standard ML

DERIVED FORM

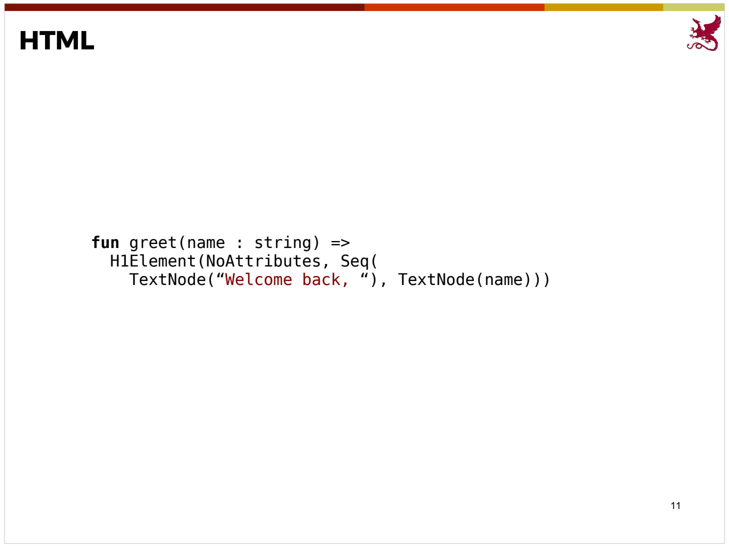
```
[1, x, f(x), f(f(x))]
```

EXPANSION

```
Cons(1, Cons(x, Cons(f(x), Cons(f(f(x)), Cons(f(f(x)), Nil)))))
```

10

…derived forms for list expressions (and also list patterns, I'll get back to list patterns in a moment.)in the textual syntax of the language. They look like that. A derived form is given meaning not directly but by its expansion to the basic forms, where you explicitly apply the constructors.



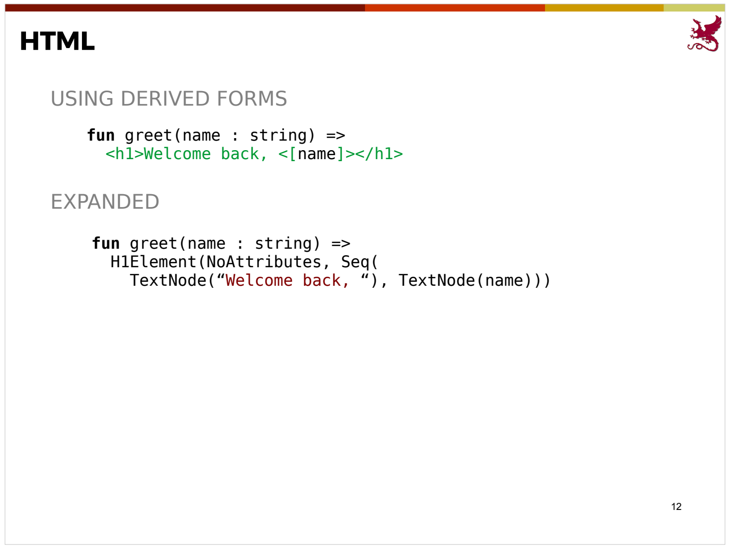
HTML

```
fun greet(name : string) =>
  H1Element(NoAttributes, Seq(
    TextNode(“Welcome back, “), TextNode(name)))
```

11

Now of course the list datatype is semantically ordinary. The designers of the language could also have given the same treatment to other datatypes and in fact some other languages do. For example, consider a datatype encoding HTML elements.

You might have constructors like H1Element and TextNode and so on, and this gets pretty laborious if you’re writing programs for the web. Moreover, there is a standardized syntax for HTML that people have for various reasons achieved consensus around



HTML

USING DERIVED FORMS

```
fun greet(name : string) =>
  <h1>Welcome back, <[name]></h1>
```

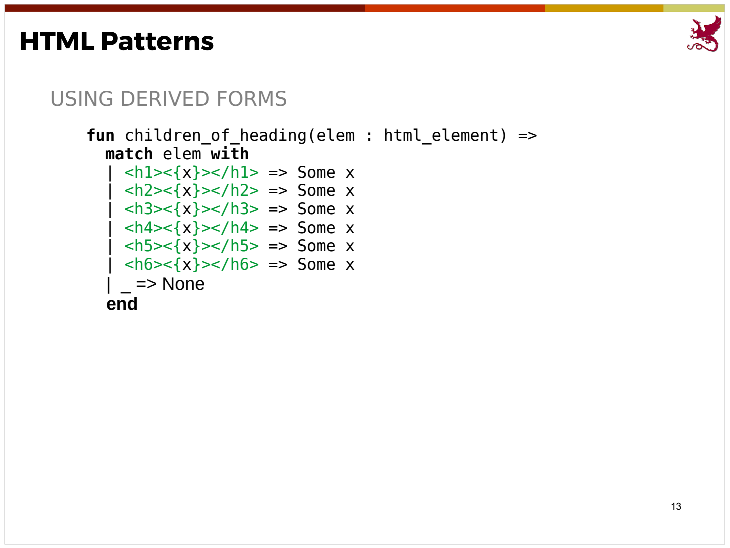
EXPANDED

```
fun greet(name : string) =>
  H1Element(NoAttributes, Seq(
    TextNode(“Welcome back, “), TextNode(name)))
```

12

So maybe you’d like derived syntax for expressions of this HTML element type, based on the standardized syntax for HTML extended, because we're programmatically generating HTML, with some forms for splicing in expressions of various types.

For example, here we have a spliced string form, and that means that, in the corresponding part of the expansion, that spliced expression of string type appears wrapped in a text node constructor.



HTML Patterns

USING DERIVED FORMS

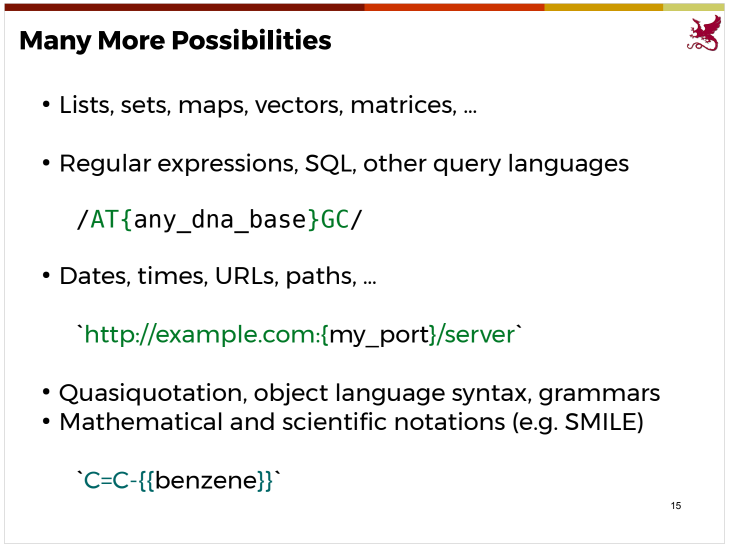
```
fun children_of_heading(elem : html_element) =>
  match elem with
  | <h1><{x}></h1> => Some x
  | <h2><{x}></h2> => Some x
  | <h3><{x}></h3> => Some x
  | <h4><{x}></h4> => Some x
  | <h5><{x}></h5> => Some x
  | <h6><{x}></h6> => Some x
  | _ => None
  end
```

13

Similarly with patterns matching values of type html_element.

So, there are languages that support such things. For example, Adam Chlipala's Ur/Web.

But now we’ve gotta return to this observation that these two datatypes are quite ordinary


Many More Possibilities

- Lists, sets, maps, vectors, matrices, …
- Regular expressions, SQL, other query languages

```
/AT{any_dna_base}GC/
```

- Dates, times, URLs, paths, …

```
`http://example.com:{my_port}/server`
```

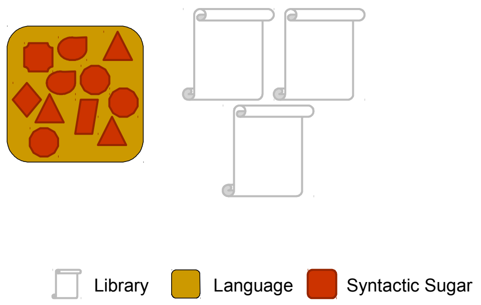
- Quasiquotation, object language syntax, grammars
- Mathematical and scientific notations (e.g. SMILE)

```
`C=C-{{benzene}}`
```

15

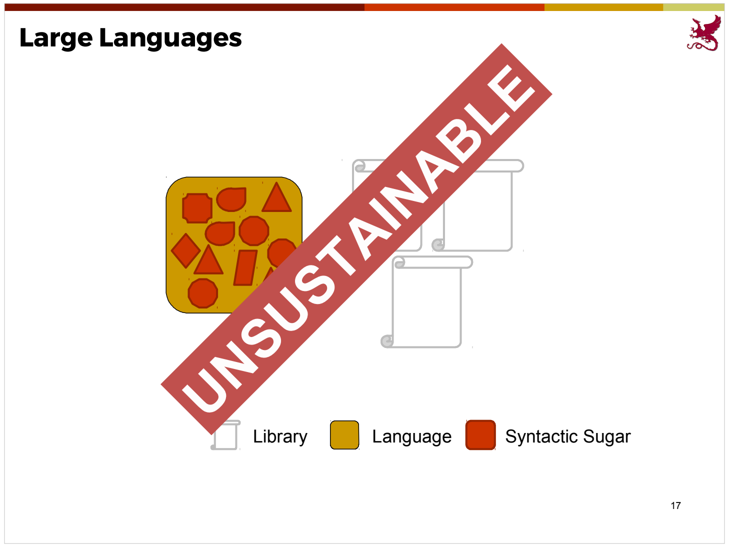
There are actually lots of examples like this…

Large Languages



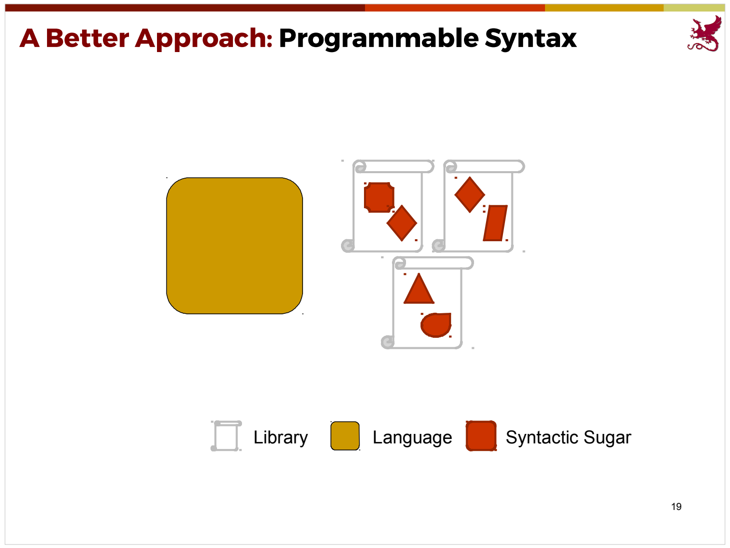
16

So if we take this approach where the language designer is going to decide a priori which constructs to privilege with derived forms, then the language and the standard library it is codefined with is going to start getting pretty large. And that means that alternative library designs will be at a distinct disadvantage as well.

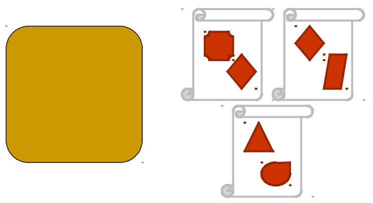


Large Languages

This is clearly unsustainable.



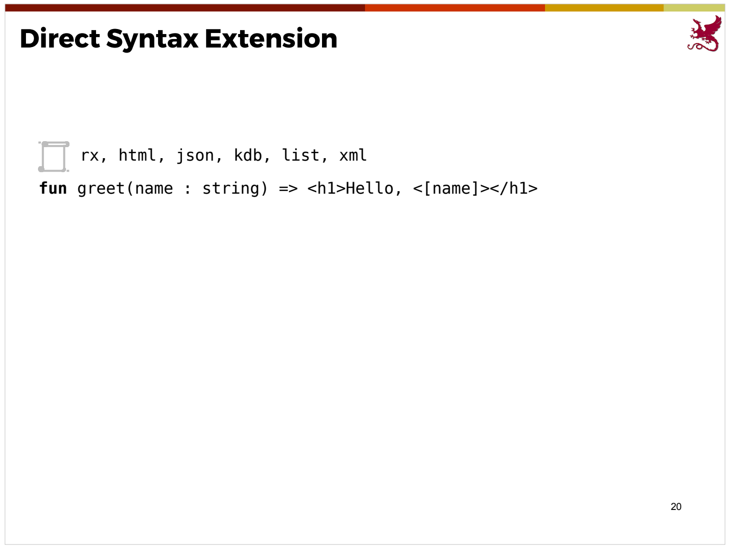
A Better Approach: Programmable Syntax



Library Language Syntactic Sugar

19

A better approach is to design a language with programmable syntax, meaning that it gives library provider the ability to define syntactic sugar themselves, by some mechanism.


Direct Syntax Extension

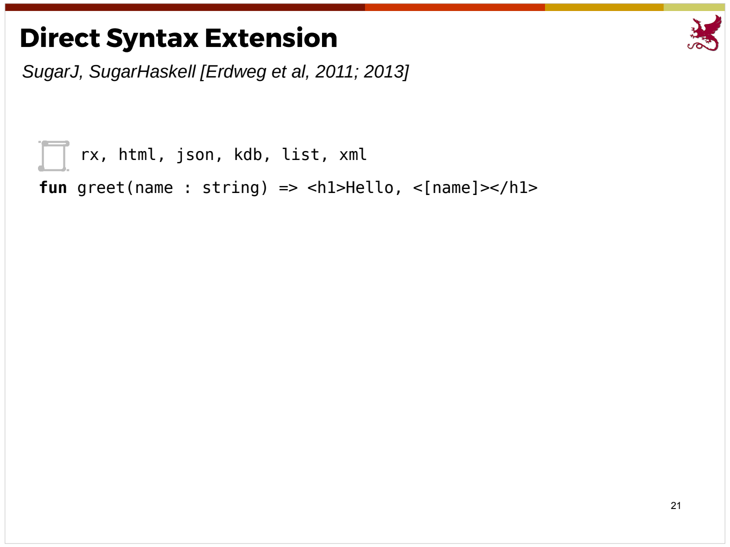
```
rx, html, json, kdb, list, xml
fun greet(name : string) => <h1>Hello, <[name]></h1>
```

20

So there have been some number of proposed mechanisms.

The most direct of these simply give library providers the ability to extend the context free grammar of the language with new derived forms.

And I want to actually talk about these first, because the problems with this approach are really what motivate my work.



Direct Syntax Extension

SugarJ, SugarHaskell [Erdweg et al, 2011; 2013]

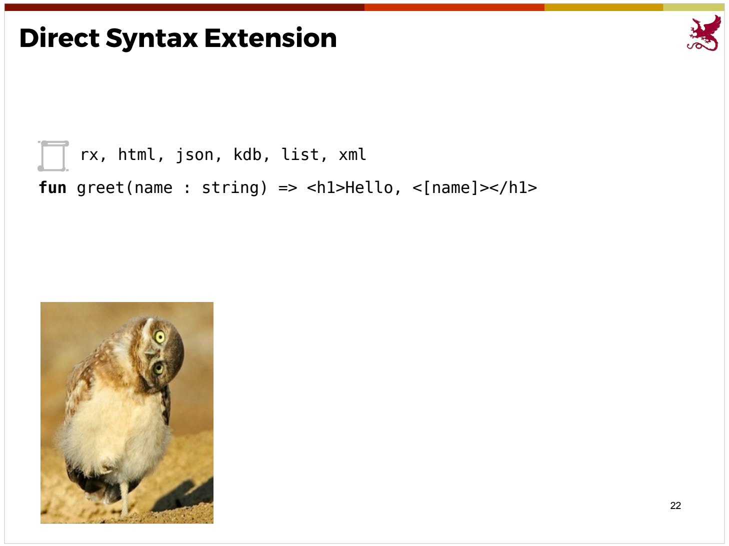
```
   rx, html, json, kdb, list, xml
fun greet(name : string) => <h1>Hello, <[name]></h1>
```

21

So first let’s talk about a system that takes this direct syntax extension approach to its extreme, and that’s the work by Sebastian Erdweg and colleagues on SugarJ and subsequent variations like SugarHaskell and others.

When you’re using system, you can come across a function greet that takes a string, name,

And then it uses what is apparently HTML syntax, like we’ve talked about, installed by one of these libraries.

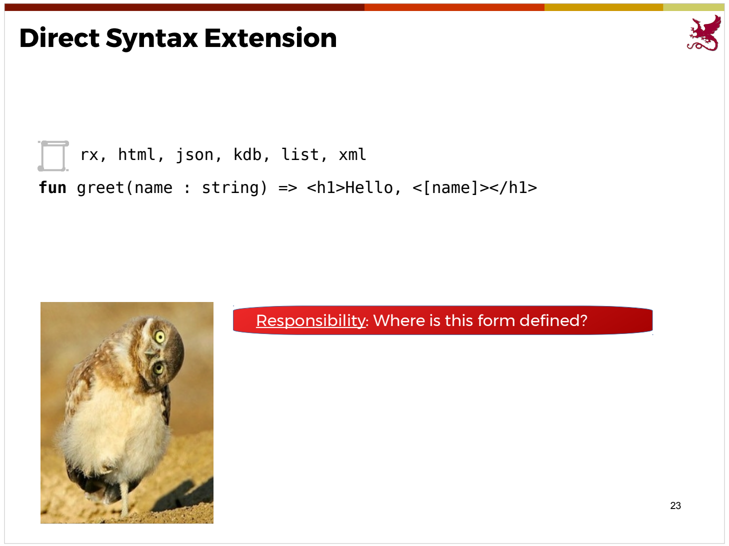


Direct Syntax Extension

```
rx, html, json, kdb, list, xml
fun greet(name : string) => <h1>Hello, <[name]></h1>
```

22

And here’s the first problem –



Direct Syntax Extension

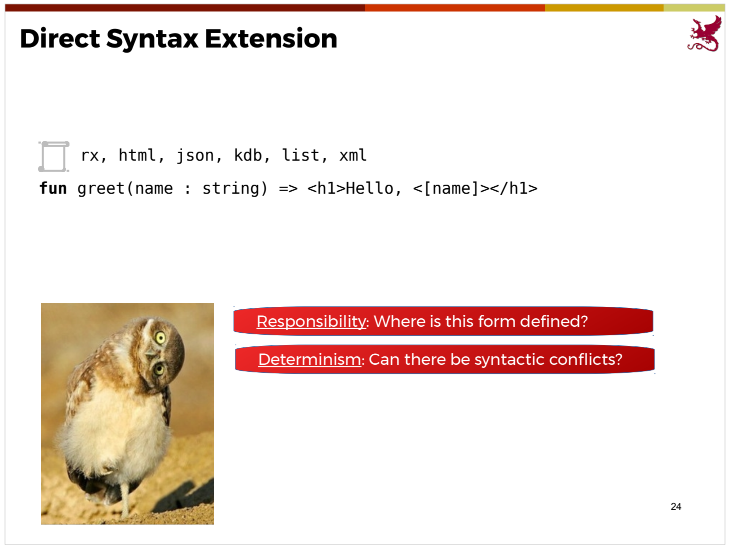
```
 rx, html, json, kdb, list, xml
fun greet(name : string) => <h1>Hello, <[name]></h1>
```

Responsibility: Where is this form defined?

23

how do we determine which library, and where within it, is responsible for this form?

In fact, there is no clear protocol for doing that.



Direct Syntax Extension

```
 rx, html, json, kdb, list, xml
fun greet(name : string) => <h1>Hello, <[name]></h1>
```

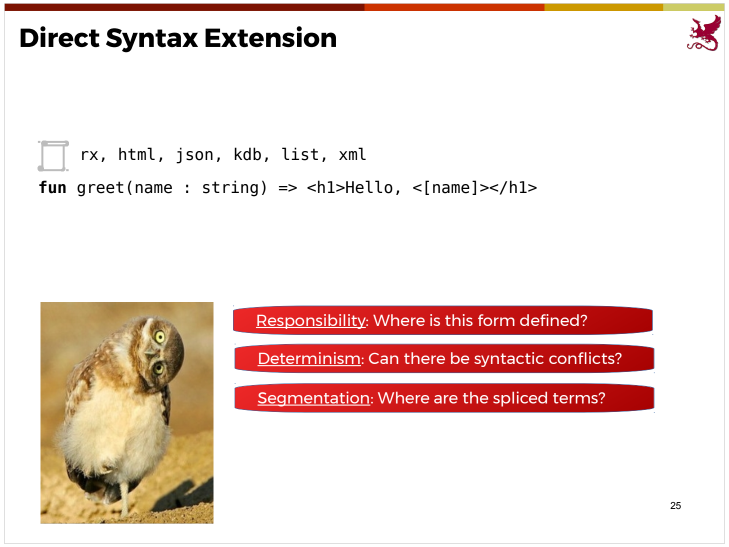
Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

24

Even worse, it may be the case that multiple libraries attempted to install this form, creating a conflict. Here you might have noticed that there is both an html and xml library that was imported, for whatever reason who cares.

So this is already quite problematic if you're trying to do "programming in the large", meaning that you're using a variety of independently developed libraries.



Direct Syntax Extension

```
rx, html, json, kdb, list, xml
fun greet(name : string) => <h1>Hello, <[name]></h1>
```

Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

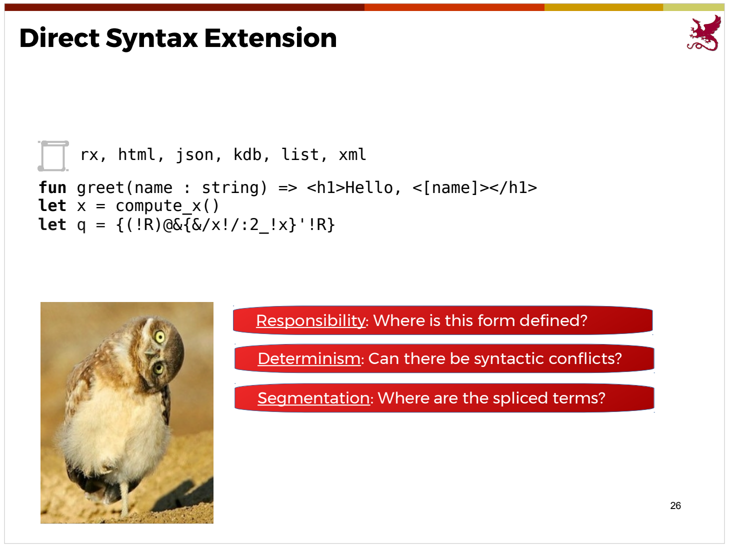
Segmentation: Where are the spliced terms?

25

But that’s not all.

Another question that you might have is “where are these spliced terms exactly?”

Here, I’ve primed you so you perhaps recall that name is a spliced term but h1 and Hello are not.



Direct Syntax Extension

```
rx, html, json, kdb, list, xml

fun greet(name : string) => <h1>Hello, <[name]></h1>
let x = compute_x()
let q = {(!R)@&{&/x!/:2_!x}'!R}
```

Responsibility: Where is this form defined?

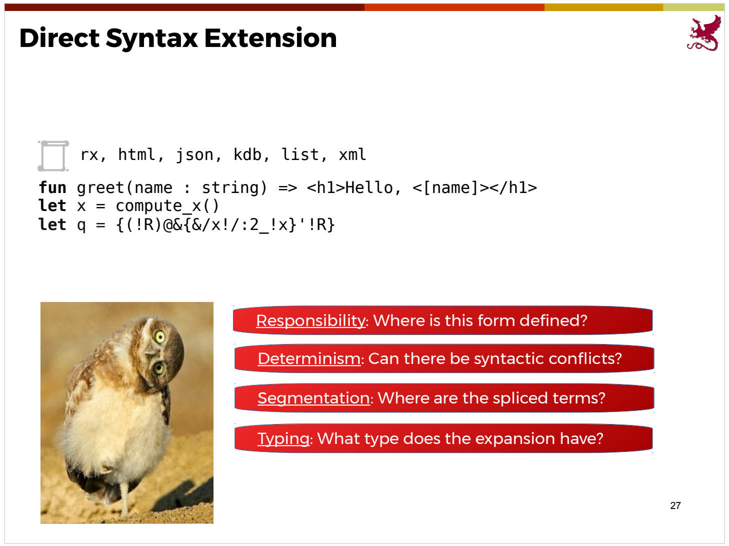
Determinism: Can there be syntactic conflicts?

Segmentation: Where are the spliced terms?

26

But consider another example,

we've done some computation, bound x,
And then used another piece of what is apparently user-defined syntax.
Again, it's not clear who is responsible, and if there might conflicts, but now its clearly unclear where the spliced terms are. Are those x's in there spliced terms? What about that R? Or that 2?



Direct Syntax Extension

```
rx, html, json, kdb, list, xml

fun greet(name : string) => <h1>Hello, <[name]></h1>
let x = compute_x()
let q = {(!R)@&{&/x!/:2_!x}'!R}
```

Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

Segmentation: Where are the spliced terms?

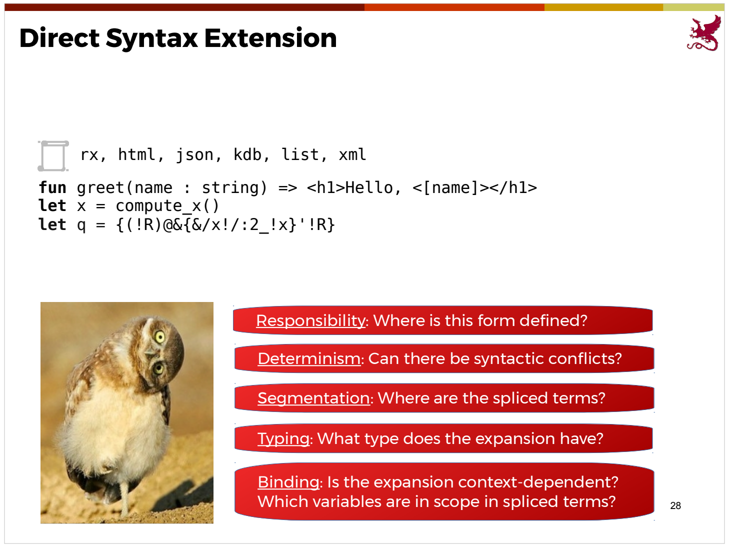
Typing: What type does the expansion have?

27

Alright well maybe we'll punt on that for a moment and consider another question: what type does the expansion have, i.e. here what type will q have?

Unclear without looking at the expansion.

Compare that to the situation where you don't know what type x has. Well, there is a clear protocol, you go follow the binding structure of the language and find the type of compute_x and that's all you need to know. You don't need to look at the body of the function.



Direct Syntax Extension

```
rx, html, json, kdb, list, xml

fun greet(name : string) => <h1>Hello, <[name]></h1>
let x = compute_x()
let q = {(!R)@&{&/x!/:2_!x}'!R}
```

Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

Segmentation: Where are the spliced terms?

Typing: What type does the expansion have?

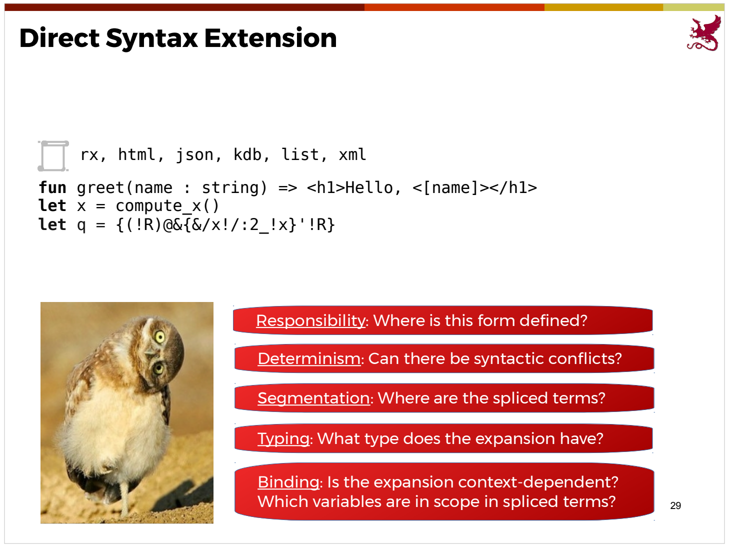
Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

28

And speaking of binding structure, that again is quite important, particularly in large programs where you have a large number of bindings.

Let’s consider again this HTML form up here. Is the expansion of this form context independent, or might itmake some assumptions about what’s bound? For example, might it assume helper functions are in scope that we don’t otherwise use?

Similarly, what about in spliced terms? For example, can we be sure that the variable in this example actually refers to the function argument? Or might it capture another binding that uses the same identifier from somewhere in the expansion? That would certainly obscure the binding structure of the language.



Direct Syntax Extension

```
rx, html, json, kdb, list, xml

fun greet(name : string) => <h1>Hello, <[name]></h1>
let x = compute_x()
let q = {(!R)@&{&/x!/:2_!x}'!R}
```

Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

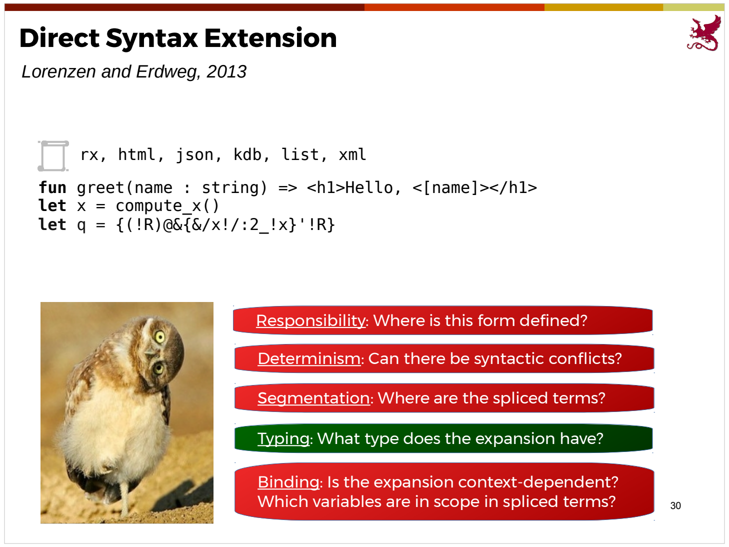
Segmentation: Where are the spliced terms?

Typing: What type does the expansion have?

Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

29

So to summarize here, the problem is we can’t hold the expansion and the logic that computes that expansion abstract if we want to reason about basic things like this, answer basic questions like this. We’re missing abstract reasoning principles.



Direct Syntax Extension

Lorenzen and Erdweg, 2013

```
rx, html, json, kdb, list, xml
fun greet(name : string) => <h1>Hello, <[name]></h1>
let x = compute_x()
let q = {(!R)@&{&/x!/:2_!x}'!R}
```

Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

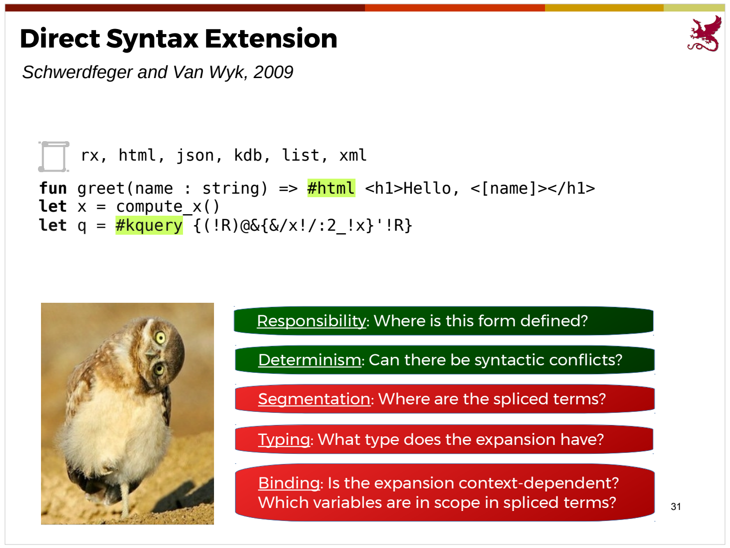
Segmentation: Where are the spliced terms?

Typing: What type does the expansion have?

Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

30

There has been some work on addressing some of these problems in clever ways. For example, in 2013 Lorenzen and Erdweg came up with an interesting system where each new derived form comes equipped a derived typing rule and the system attempts to automatically prove the expansion logic sound, so that ends up making it easier to reason about typing, at least if you're able to read a full typing derivation – it's not always as simple as just reading off an annotation.



Direct Syntax Extension

Schwerdfeger and Van Wyk, 2009

```
rx, html, json, kdb, list, xml
fun greet(name : string) => #html <h1>Hello, <[name]></h1>
let x = compute_x()
let q = #kquery {(!R)@&{&/x!/:2_!x}'!R}
```

Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

Segmentation: Where are the spliced terms?

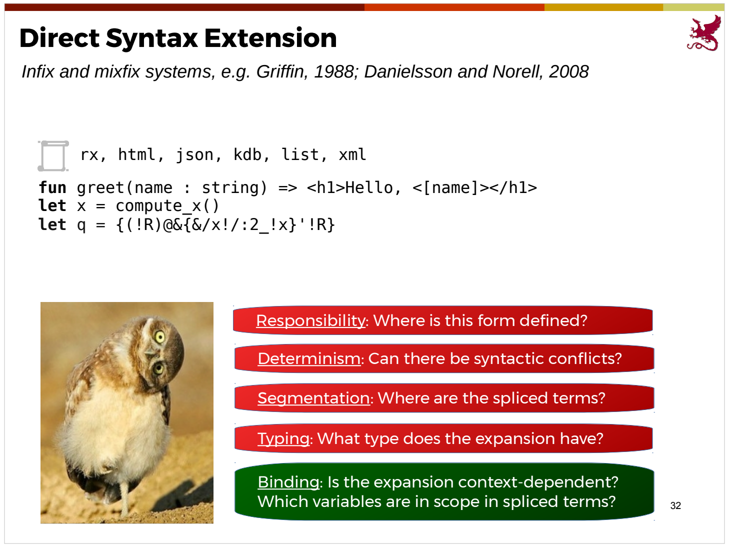
Typing: What type does the expansion have?

Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

31

Work by Scwerdfeger and Van Wyk has looked at the problem of determinsm and come up with a nice set of constraints on a class of context-free grammars that allow you to modularly prove determinism. The main constraint is that you have prefix each new form with this marking terminal, and that they all be distinct, and there are various clever mechanisms for dealing with that.

This maintains determinism and also helps you determinism responsibility because each marking terminal is uniquely affiliated with a syntax extension, but this mechanism doesn't address these other questions.



Direct Syntax Extension

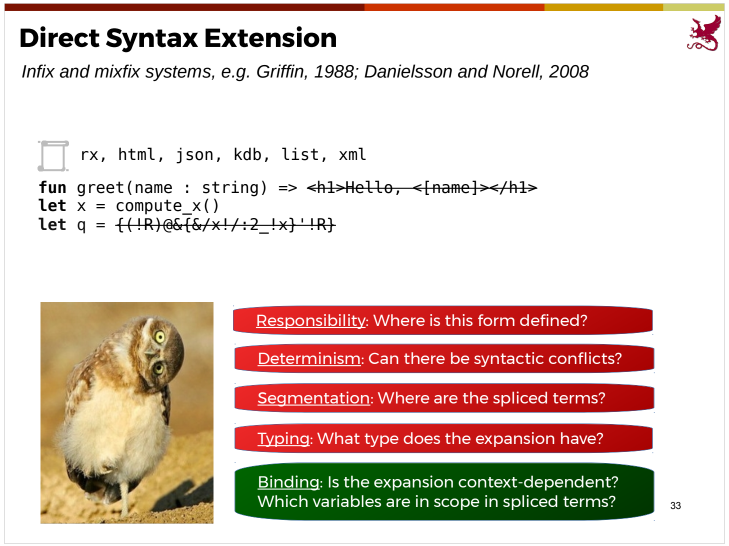
Infix and mixfix systems, e.g. Griffin, 1988; Danielsson and Norell, 2008

```
rx, html, json, kdb, list, xml

fun greet(name : string) => <h1>Hello, <[name]></h1>
let x = compute_x()
let q = {(!R)@&{&/x!/:2_!x}'!R}
```

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?
- Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

32

Another class of systems are ...



Direct Syntax Extension

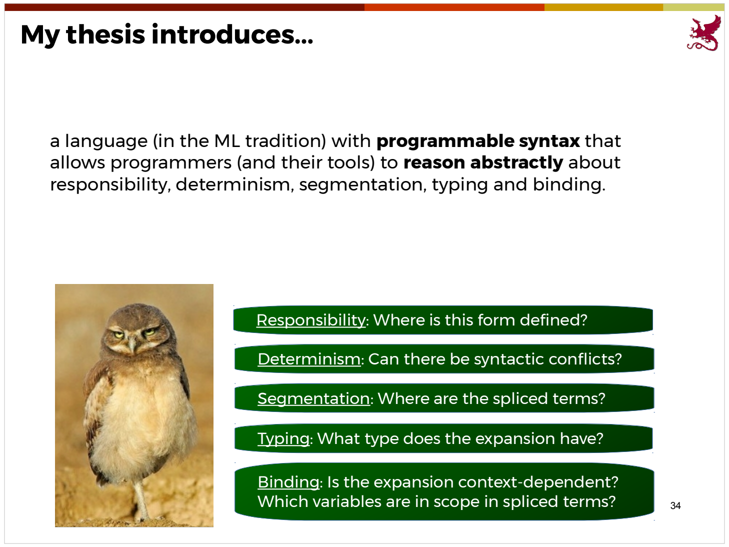
Infix and mixfix systems, e.g. Griffin, 1988; Danielsson and Norell, 2008

```
rx, html, json, kdb, list, xml

fun greet(name : string) => ~~<h1>Hello, <[name]></h1>~~
let x = compute_x()
let q = ~~{(!R)@&{&/x!/:2_!x}'!R}~~
```

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?
- Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

33

There are a number of other systems



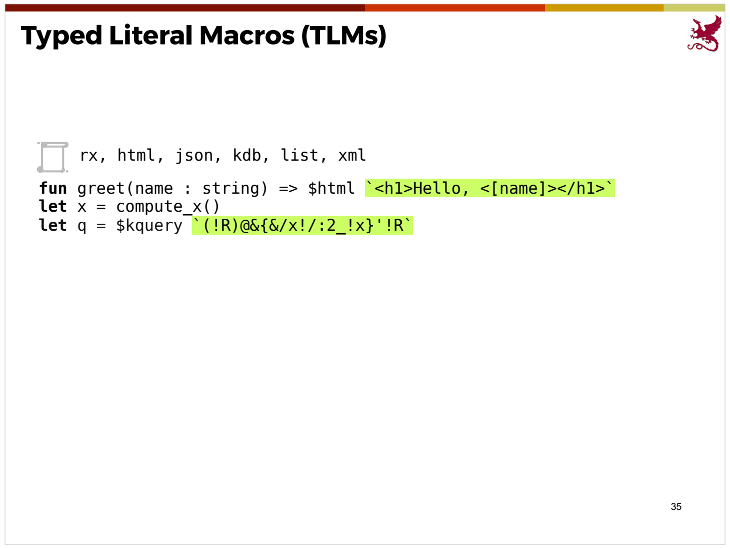
My thesis introduces…

a language (in the ML tradition) with **programmable syntax** that allows programmers (and their tools) to **reason abstractly** about responsibility, determinism, segmentation, typing and binding.

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?
- Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

34

So that motivates the work that I'm presenting here. My thesis introduces a … The level of syntactic control is comparable to direct syntax extension systems.

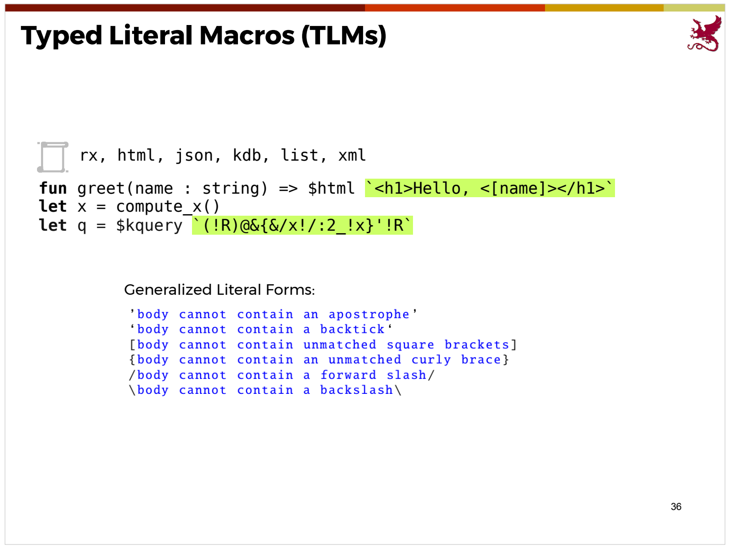


Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

35

Here’s how it works (next slide)



Typed Literal Macros (TLMs)

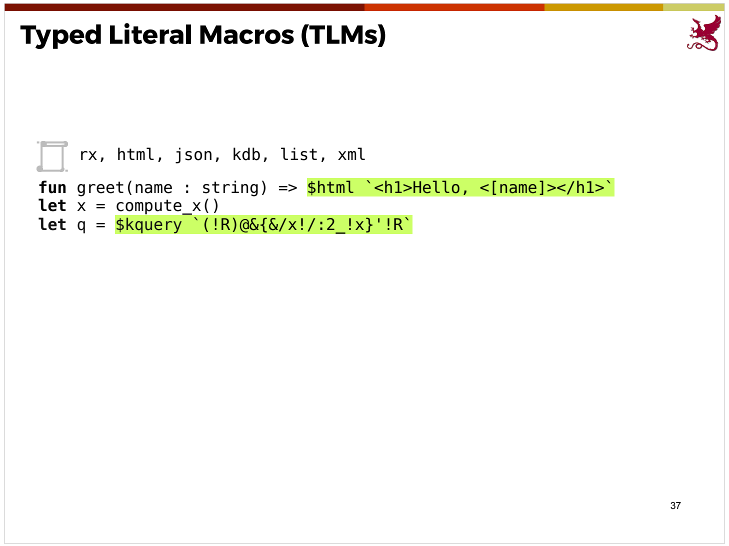
```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

Generalized Literal Forms:

```
'body cannot contain an apostrophe'
‘body cannot contain a backtick‘
[body cannot contain unmatched square brackets]
{body cannot contain an unmatched curly brace}
/body cannot contain a forward slash/
\body cannot contain a backslash\
```

36

So the first thing we do is give up on the idea of actually extending the grammar of the language. The grammar is fixed. However, in that grammar are these forms that we call generalized literal forms, which are syntactically very flexible.



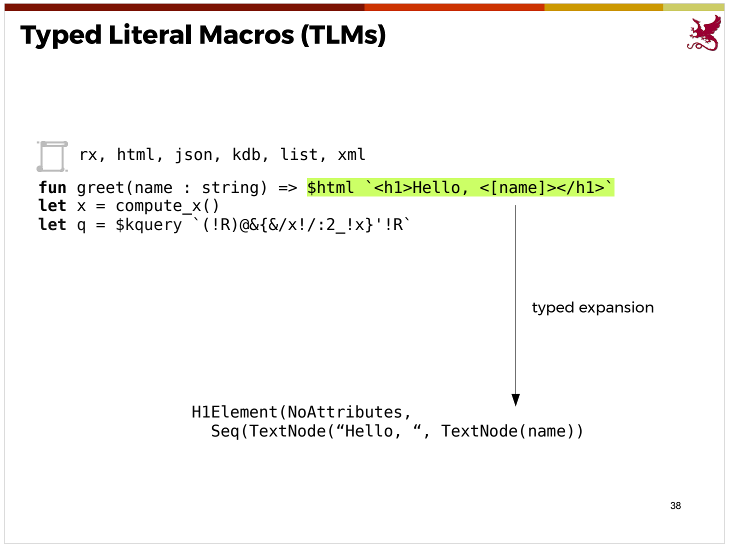
Typed Literal Macros (TLMs)

rx, html, json, kdb, list, xml

```
fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

37

Generalized literal form are given meaning by expansion at compile-time, actually during the typing process, by the applied typed literal macro (or TLM). TLM names are prefixed by $.



Typed Literal Macros (TLMs)

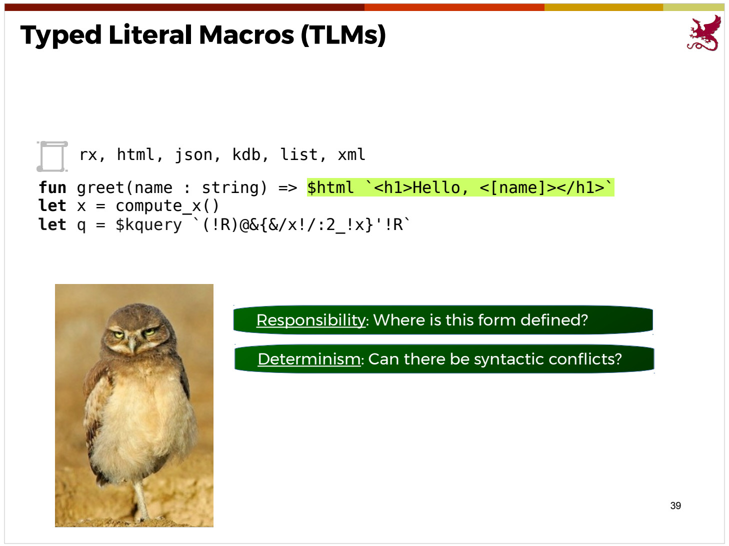
```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

typed expansion

```
H1Element(NoAttributes,
  Seq(TextNode("Hello, ", TextNode(name))
```

38

Let’s take a look at the html example first.



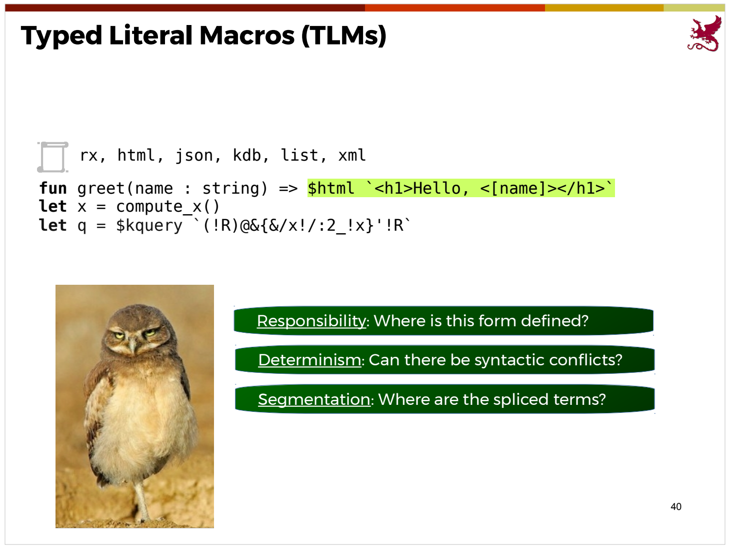
Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

Now immediately that addresses a couple of problems. The applied TLM is responsible. And there are no conflicts because the CFG is not modified. Can reason modularly about syntactic determinism.



Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kqwery `(!R)@&{&/x!/:2_!x}'!R`
```

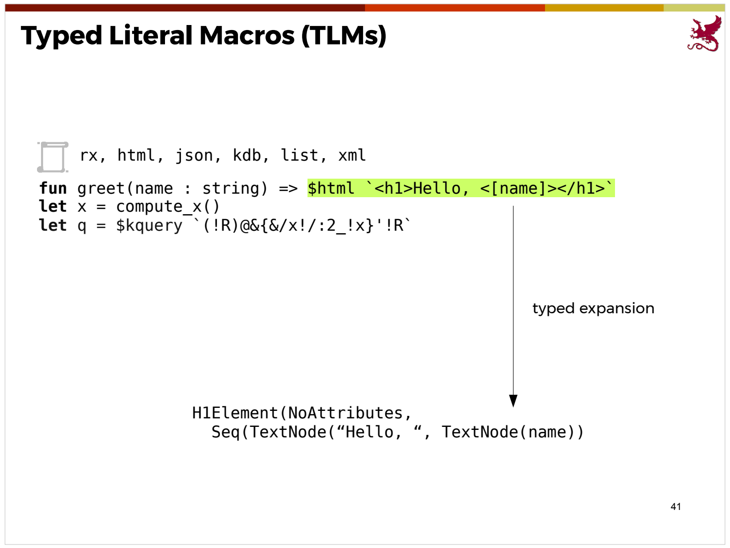
Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

Segmentation: Where are the spliced terms?

40

What about the issue of reasoning about segmentation? That’s actually perhaps the most interesting bit of all this.



Typed Literal Macros (TLMs)

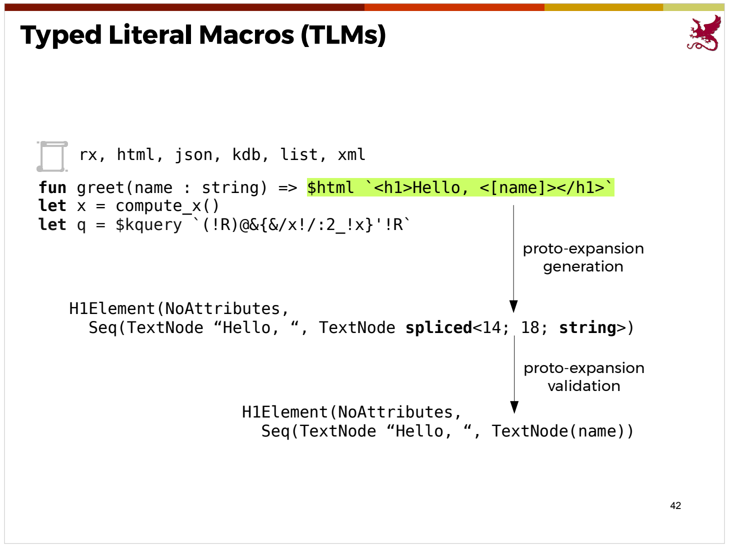
```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

typed expansion

```
H1Element(NoAttributes,
  Seq(TextNode(“Hello, “, TextNode(name))
```

41

That requires us to consider typed expansion in more detail. In fact, when performing typed expansion for a TSM application, we proceed in two steps.



Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

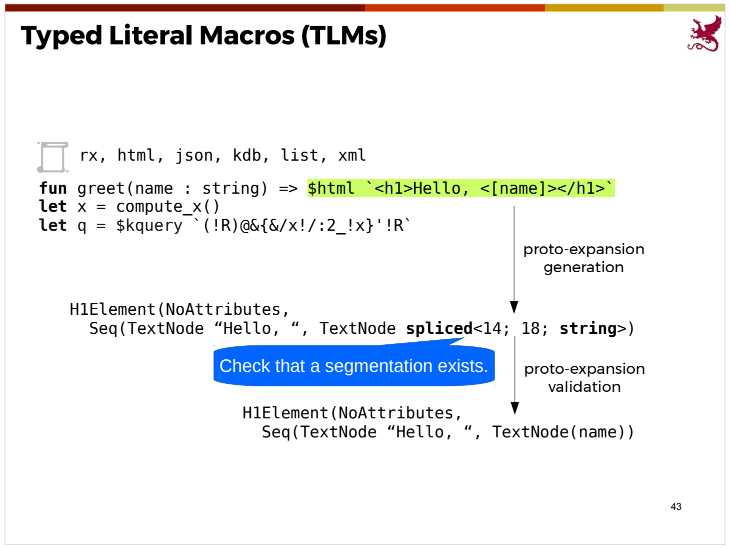
```
H1Element(NoAttributes,
  Seq(TextNode “Hello, “, TextNode spliced<14; 18; string>)
```

```
H1Element(NoAttributes,
  Seq(TextNode “Hello, “, TextNode(name))
```

42

First the TSM generates a “proto-expansion”. Then we validate that.

What’s a proto-expansion?



Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

proto-expansion generation

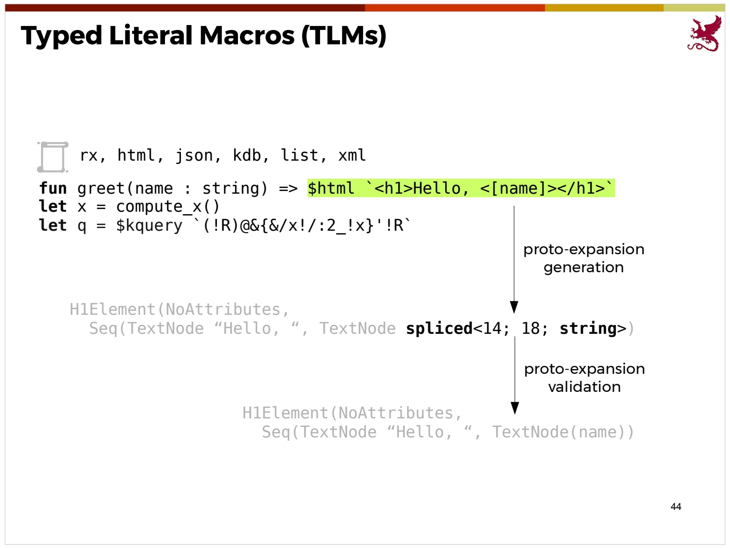
```
    H1Element(NoAttributes,
      Seq(TextNode “Hello, “, TextNode spliced<14; 18; string>)
```

Check that a segmentation exists.
proto-expansion validation

```
H1Element(NoAttributes,
  Seq(TextNode “Hello, “, TextNode(name))
```

43

An expansion, but with spliced segments represented abstractly by location, rather than inserted directly. They must be disjoint.



Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kqurey `(!R)@&{&/x!/:2_!x}'!R`
```

proto-expansion generation

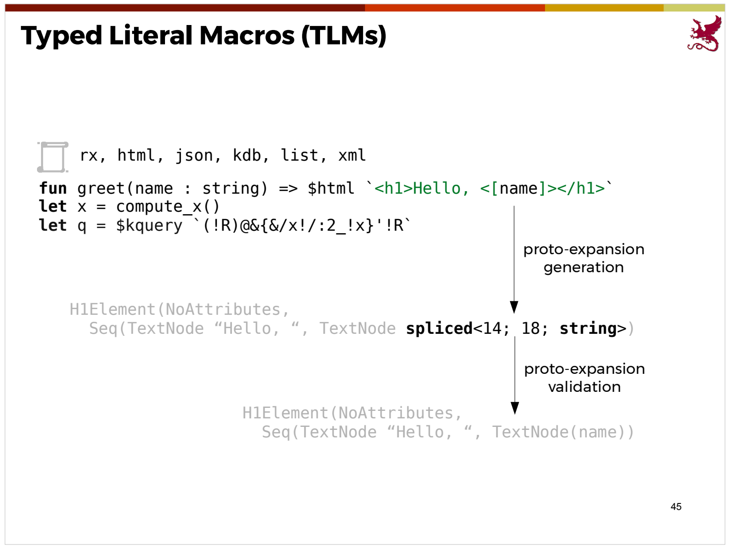
```
H1Element(NoAttributes,
  Seq(TextNode "Hello, ", TextNode spliced<14; 18; string>)
```

proto-expansion validation

```
H1Element(NoAttributes,
  Seq(TextNode "Hello, ", TextNode(name))
```

44

To communicate the segmentation itself, we need only reveal the segmentation.



Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

proto-expansion generation

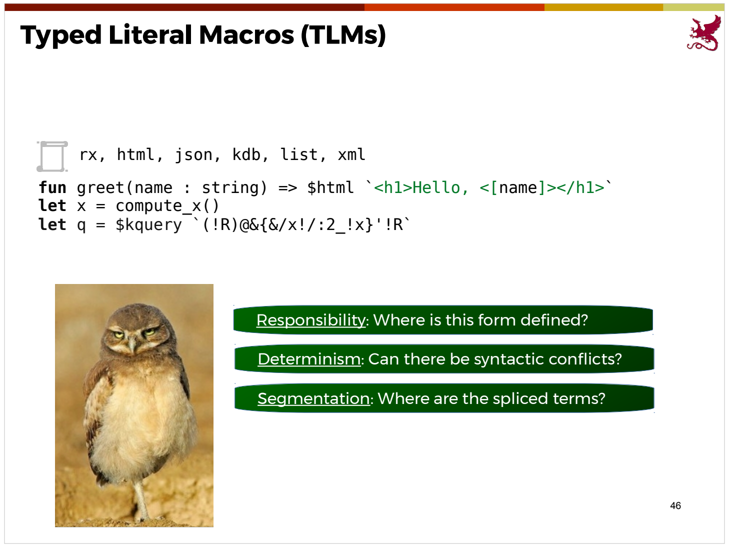
```
H1Element(NoAttributes,
  Seq(TextNode “Hello, “, TextNode spliced<14; 18; string>)
```

proto-expansion validation

```
H1Element(NoAttributes,
  Seq(TextNode “Hello, “, TextNode(name))
```

45

Using colors.



Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

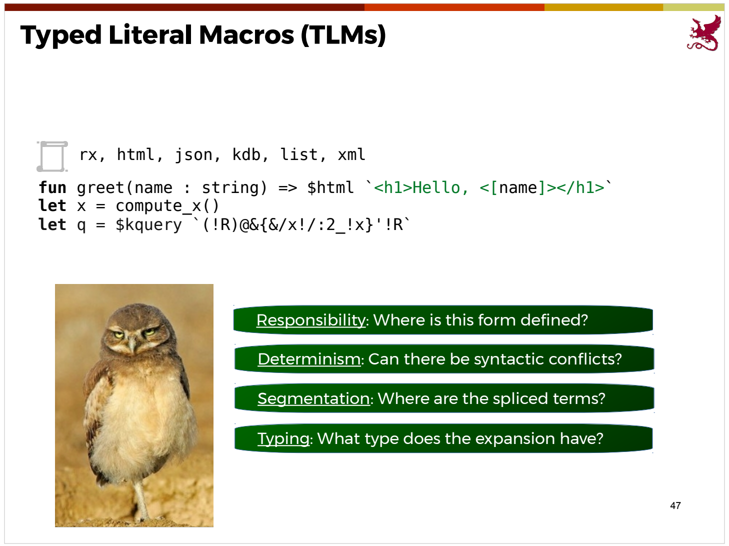
Responsibility: Where is this form defined?

Determinism: Can there be syntactic conflicts?

Segmentation: Where are the spliced terms?

46

So that addresses this problem of segmentation. Now...



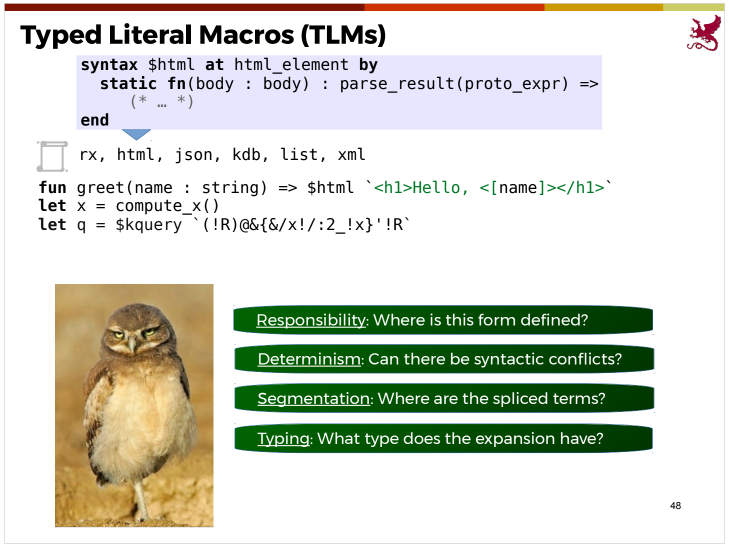
Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?

47

What about typing?



Typed Literal Macros (TLMs)

```
syntax $html at html_element by
  static fn(body : body) : parse_result(proto_expr) =>
     (* … *)
end
```

```
rx, html, json, kdb, list, xml
```

```
fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

Responsibility: Where is this form defined?

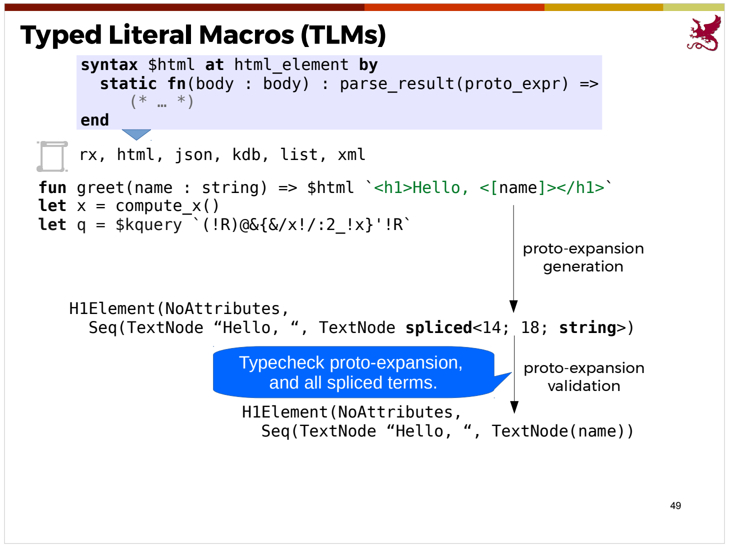
Determinism: Can there be syntactic conflicts?

Segmentation: Where are the spliced terms?

Typing: What type does the expansion have?

48

For that, let’s actually look at a TLM definition. Notice that it has a type annotation!



Typed Literal Macros (TLMs)

```
syntax $html at html_element by
  static fn(body : body) : parse_result(proto_expr) =>
     (* … *)
end
```

```
rx, html, json, kdb, list, xml
```

```
fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

proto-expansion generation

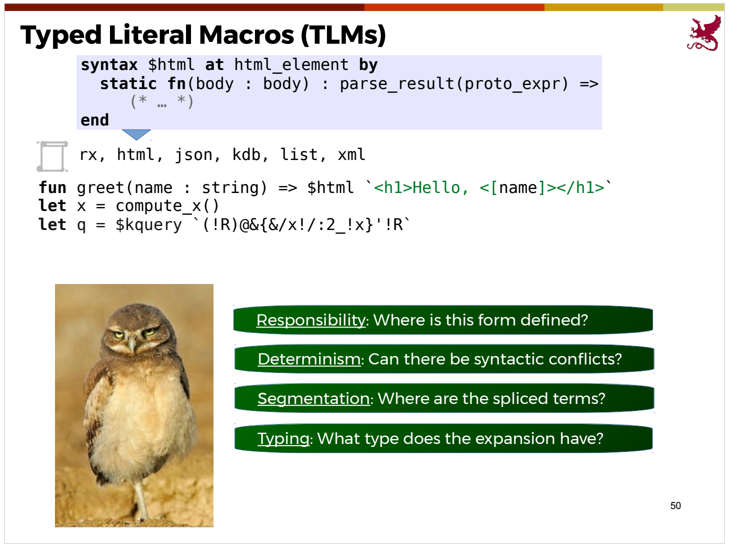
```
   H1Element(NoAttributes,
     Seq(TextNode “Hello, “, TextNode spliced<14; 18; string>)
```

Typecheck proto-expansion, and all spliced terms.
proto-expansion validation

```
H1Element(NoAttributes,
  Seq(TextNode “Hello, “, TextNode(name))
```

49

Proto-expansion validation checks the expansion against that annotation, and also checks spliced segments against corresponding type annotation. That means you can reason abstractly about types – you need not examine the full expansion, but rather only the annotations.



Typed Literal Macros (TLMs)

```
syntax $html at html_element by
  static fn(body : body) : parse_result(proto_expr) =>
     (* … *)
end
```

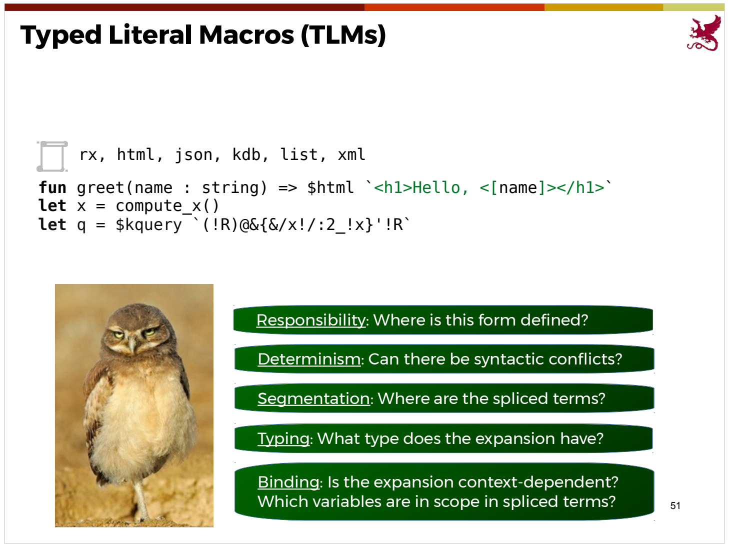
```
rx, html, json, kdb, list, xml
```

```
fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?

50

Finally, ...



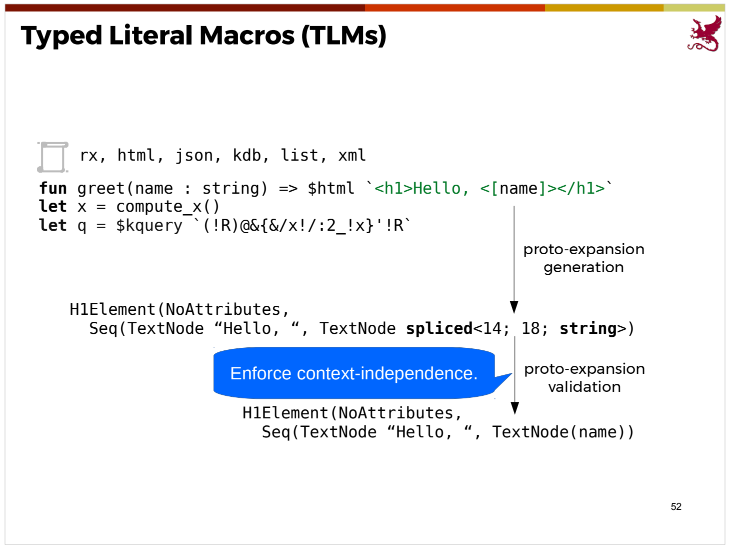
Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?
- Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

51

What about binding?



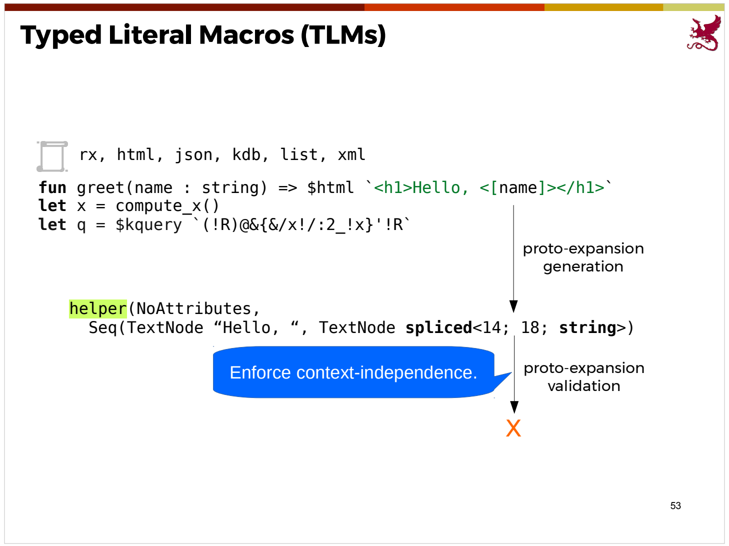
Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`

    H1Element(NoAttributes,
      Seq(TextNode "Hello, ", TextNode spliced<14; 18; string>)

         H1Element(NoAttributes,
           Seq(TextNode "Hello, ", TextNode(name))
```

proto-expansion generation
Enforce context-independence.
proto-expansion validation

52

Again, proto-expansion validation to the rescue. Here, we enforce a very strong **hygienic binding discipline.**



Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

proto-expansion generation

```
    helper(NoAttributes,
      Seq(TextNode “Hello, “, TextNode spliced<14; 18; string>)
```

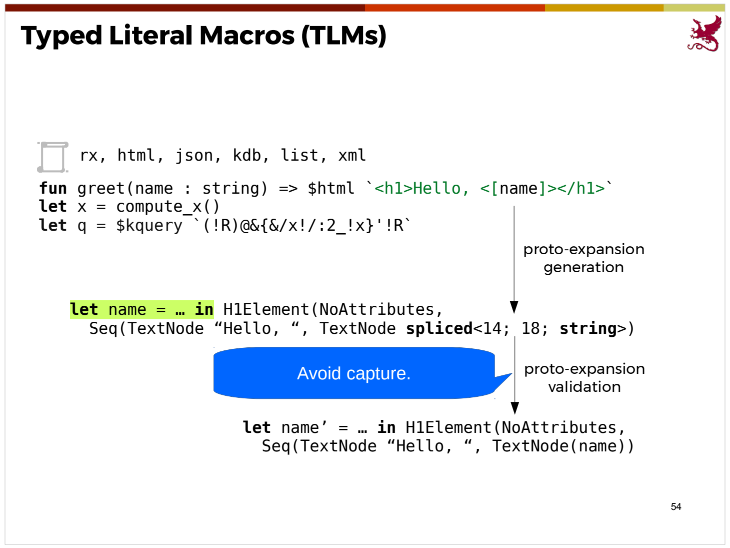
Enforce context-independence.

proto-expansion validation

X

53

If you aren’t context-independent, validation fails. (Parametric TLMs, introduced shortly, allow you to use helpers in a hygienic manner.)



Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

proto-expansion generation

```
let name = … in H1Element(NoAttributes,
  Seq(TextNode “Hello, “, TextNode spliced<14; 18; string>)
```

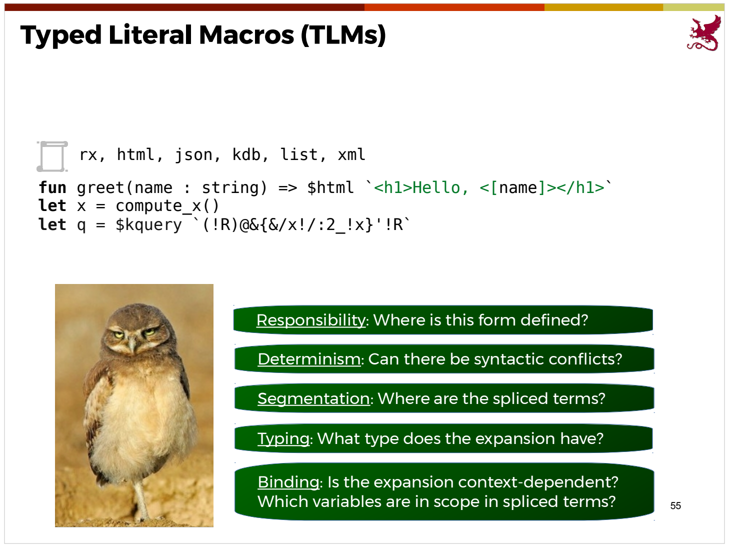
Avoid capture.

proto-expansion validation

```
let name’ = … in H1Element(NoAttributes,
  Seq(TextNode “Hello, “, TextNode(name))
```

54

It also avoids capture of bindings in the expansion by the spliced segments.



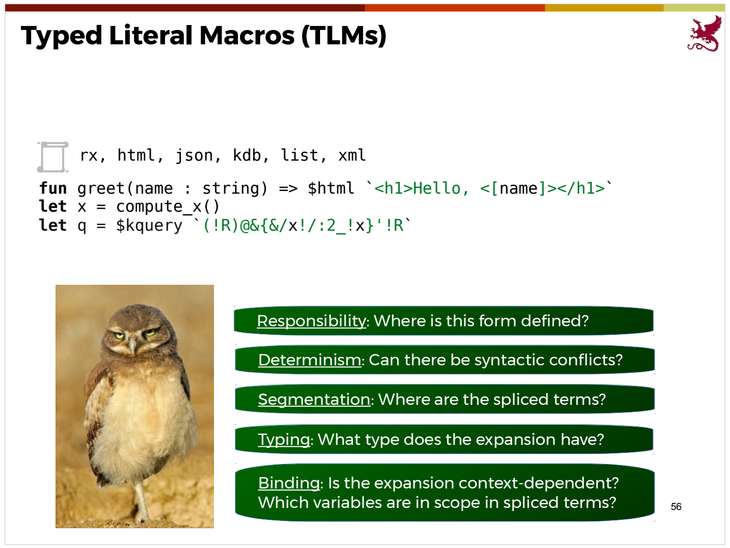
Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?
- Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

55

So that's it, we've recovered all these nice reasoning principles in a setting with high syntactic control.



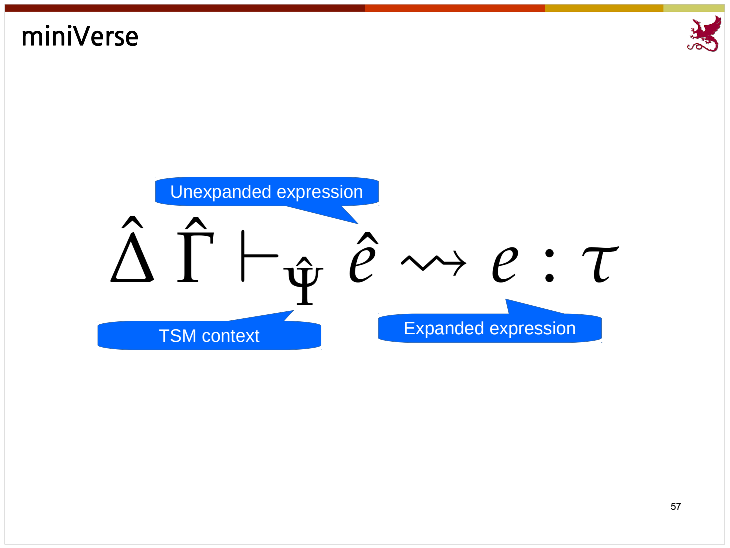
Typed Literal Macros (TLMs)

```
rx, html, json, kdb, list, xml

fun greet(name : string) => $html `<h1>Hello, <[name]></h1>`
let x = compute_x()
let q = $kquery `(!R)@&{&/x!/:2_!x}'!R`
```

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?
- Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

56

We can communicate the necessary information using secondary notation in a straightforward way. The full expansion can be held abstract.

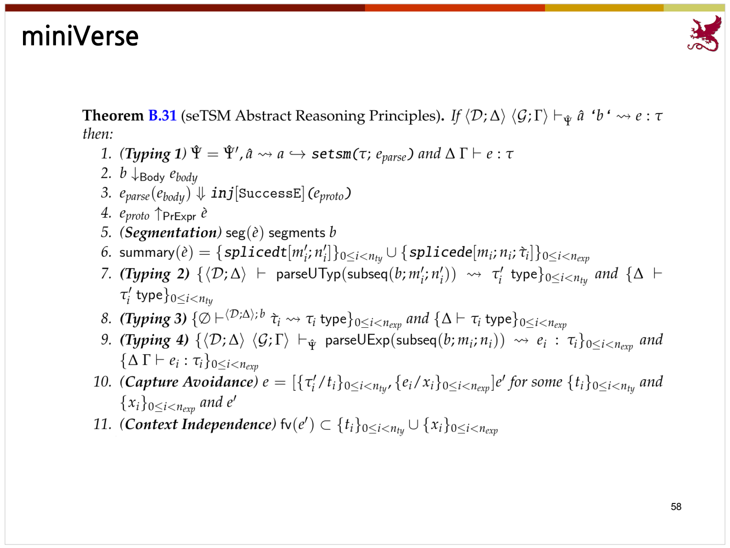


miniVerse

$$\hat{\Delta}\ \hat{\Gamma} \vdash_{\hat{\Psi}} \hat{e} \rightsquigarrow e : \tau$$

Unexpanded expression
TSM context
Expanded expression

Here is just a taste of the semantics. Typed expansion from an unexpanded language (where literal bodies remain unparsed) to an expanded language, where no literals remain.



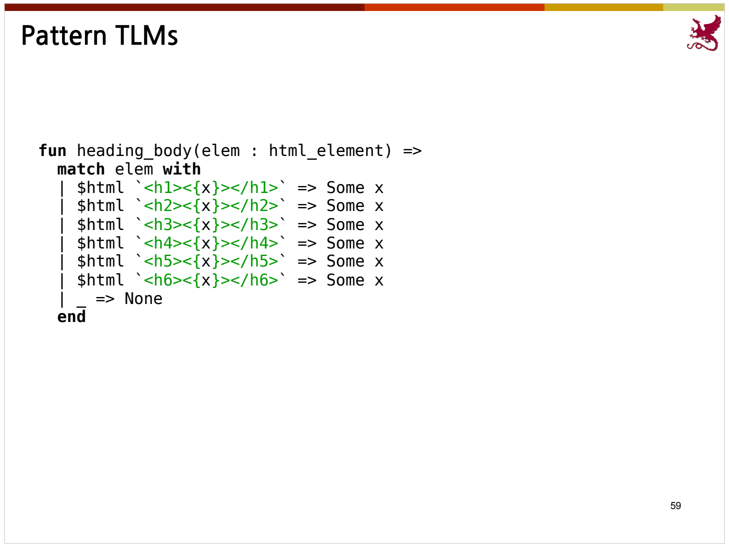
miniVerse

Theorem B.31 (seTSM Abstract Reasoning Principles). *If* $\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \hat{a}\ \text{‘}b\text{‘} \rightsquigarrow e : \tau$ *then:*

1. *(**Typing 1**)* $\hat{\Psi} = \hat{\Psi}', \hat{a} \rightsquigarrow a \hookrightarrow \texttt{setsm}(\tau; e_{parse})$ *and* $\Delta\ \Gamma \vdash e : \tau$
2. $b \downarrow_{\mathsf{Body}} e_{body}$
3. $e_{parse}(e_{body}) \Downarrow \texttt{inj}[\texttt{SuccessE}](e_{proto})$
4. $e_{proto} \uparrow_{\mathsf{PrExpr}} \grave{e}$
5. *(**Segmentation**)* $\mathsf{seg}(\grave{e})$ segments b
6. $\mathsf{summary}(\grave{e}) = \{\texttt{splicedt}[m'_i; n'_i]\}_{0 \le i < n_{ty}} \cup \{\texttt{splicede}[m_i; n_i; \grave{\tau}_i]\}_{0 \le i < n_{exp}}$
7. *(**Typing 2**)* $\{\langle \mathcal{D}; \Delta \rangle \vdash \mathsf{parseUTyp}(\mathsf{subseq}(b; m'_i; n'_i)) \rightsquigarrow \tau'_i\ \mathsf{type}\}_{0 \le i < n_{ty}}$ *and* $\{\Delta \vdash \tau'_i\ \mathsf{type}\}_{0 \le i < n_{ty}}$
8. *(**Typing 3**)* $\{\emptyset \vdash^{\langle \mathcal{D}; \Delta \rangle; b} \grave{\tau}_i \rightsquigarrow \tau_i\ \mathsf{type}\}_{0 \le i < n_{exp}}$ *and* $\{\Delta \vdash \tau_i\ \mathsf{type}\}_{0 \le i < n_{exp}}$
9. *(**Typing 4**)* $\{\langle \mathcal{D}; \Delta \rangle\ \langle \mathcal{G}; \Gamma \rangle \vdash_{\hat{\Psi}} \mathsf{parseUExp}(\mathsf{subseq}(b; m_i; n_i)) \rightsquigarrow e_i : \tau_i\}_{0 \le i < n_{exp}}$ *and* $\{\Delta\ \Gamma \vdash e_i : \tau_i\}_{0 \le i < n_{exp}}$
10. *(**Capture Avoidance**)* $e = [\{\tau'_i / t_i\}_{0 \le i < n_{ty}}, \{e_i / x_i\}_{0 \le i < n_{exp}}]e'$ *for some* $\{t_i\}_{0 \le i < n_{ty}}$ *and* $\{x_i\}_{0 \le i < n_{exp}}$ *and* e'
11. *(**Context Independence**)* $\mathsf{fv}(e') \subset \{t_i\}_{0 \le i < n_{ty}} \cup \{x_i\}_{0 \le i < n_{exp}}$

58

All of those reasoning principles – the green bubbles – I talked about are formally established.

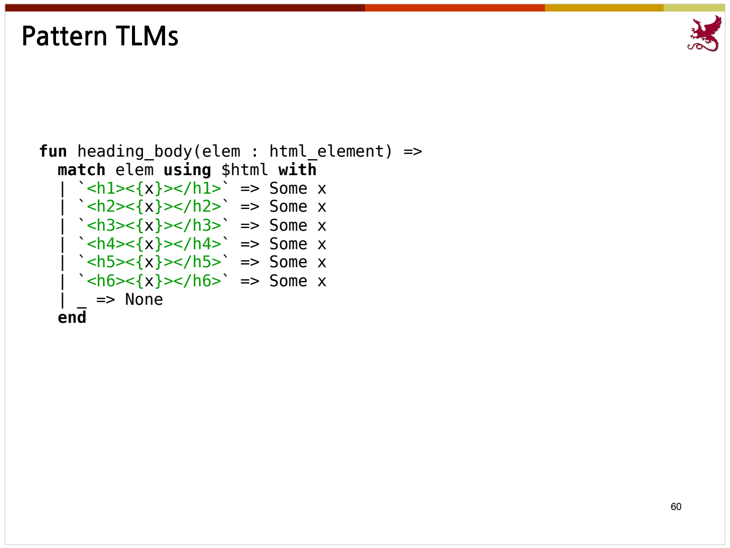


Pattern TLMs

```
fun heading_body(elem : html_element) =>
  match elem with
  | $html `<h1><{x}></h1>` => Some x
  | $html `<h2><{x}></h2>` => Some x
  | $html `<h3><{x}></h3>` => Some x
  | $html `<h4><{x}></h4>` => Some x
  | $html `<h5><{x}></h5>` => Some x
  | $html `<h6><{x}></h6>` => Some x
  | _ => None
  end
```

59

The thesis talks not just about expression TLMs, but also pattern TLMs.

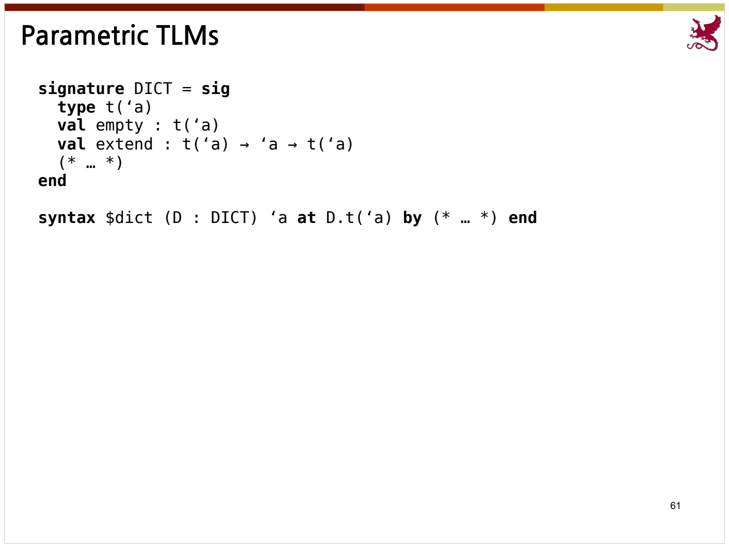


Pattern TLMs

```
fun heading_body(elem : html_element) =>
  match elem using $html with
  | `<h1><{x}></h1>` => Some x
  | `<h2><{x}></h2>` => Some x
  | `<h3><{x}></h3>` => Some x
  | `<h4><{x}></h4>` => Some x
  | `<h5><{x}></h5>` => Some x
  | `<h6><{x}></h6>` => Some x
  | _ => None
  end
```

60

More conveniently..



Parametric TLMs

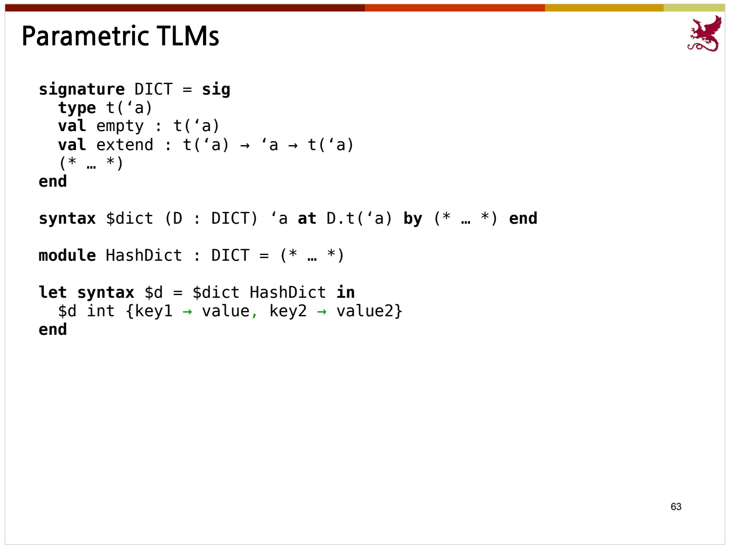
```
signature DICT = sig
  type t(‘a)
  val empty : t(‘a)
  val extend : t(‘a) → ‘a → t(‘a)
  (* … *)
end

syntax $dict (D : DICT) ‘a at D.t(‘a) by (* … *) end
```

61

And also addresses the problem of defining TLMs not just at one type but over type- and module-parameterized families of types, like you might have in ML. This also makes it easier to deal with the context-independence constraint – you can pass in helper functions via modules.

Parametric TLMs

```
signature DICT = sig
  type t('a)
  val empty : t('a)
  val extend : t('a) → 'a → t('a)
  (* … *)
end

syntax $dict (D : DICT) 'a at D.t('a) by (* … *) end

module HashDict : DICT = (* … *)

$dict HashDict int {key1 → value, key2 → value2}
```

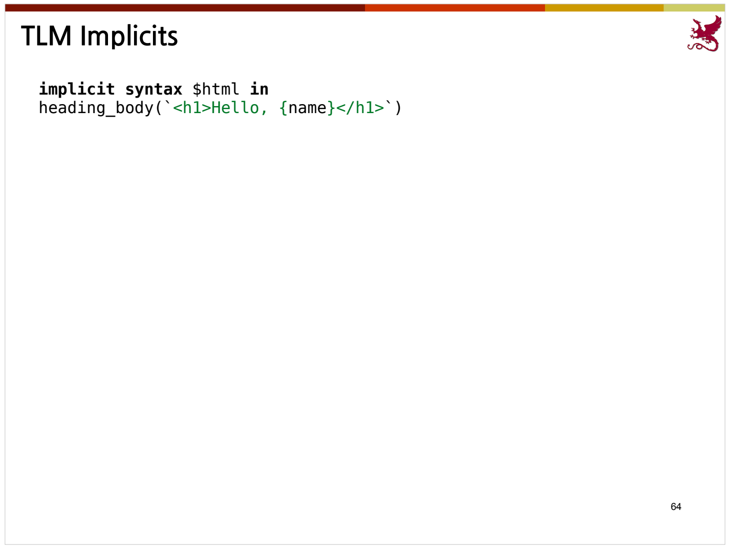


Parametric TLMs

```
signature DICT = sig
  type t(‘a)
  val empty : t(‘a)
  val extend : t(‘a) → ‘a → t(‘a)
  (* … *)
end

syntax $dict (D : DICT) ‘a at D.t(‘a) by (* … *) end

module HashDict : DICT = (* … *)

let syntax $d = $dict HashDict in
  $d int {key1 → value, key2 → value2}
end
```

63

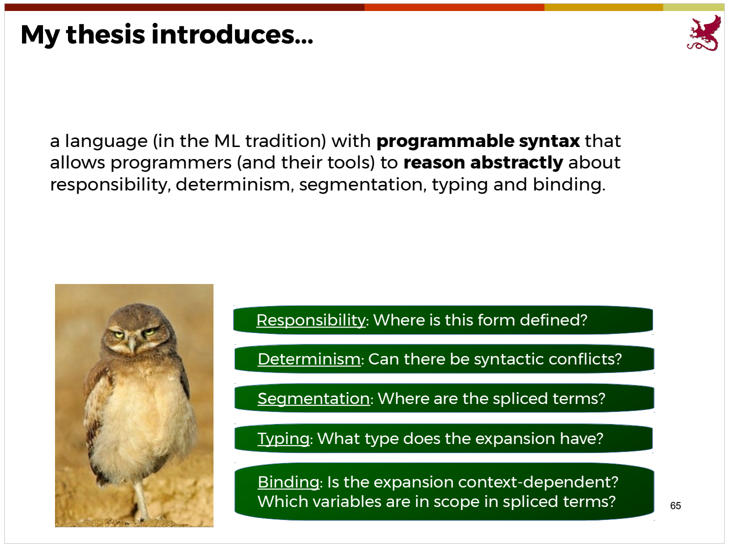
You can partially apply parameters to make things more convenient.



TLM Implicits

```
implicit syntax $html in
heading_body(`<h1>Hello, {name}</h1>`)
```

Finally, for small literal bodies and frequently applied TLMs, we can use a mechanism of TLM implicits defined in the dissertation to further reduce syntactic cost.



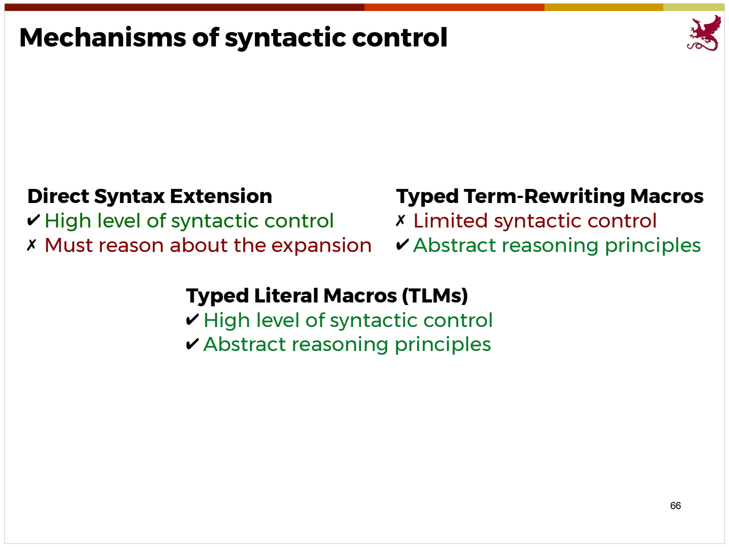
My thesis introduces…

a language (in the ML tradition) with **programmable syntax** that allows programmers (and their tools) to **reason abstractly** about responsibility, determinism, segmentation, typing and binding.

- Responsibility: Where is this form defined?
- Determinism: Can there be syntactic conflicts?
- Segmentation: Where are the spliced terms?
- Typing: What type does the expansion have?
- Binding: Is the expansion context-dependent? Which variables are in scope in spliced terms?

65

So my thesis introduces a mechanism that allows programmers to define new syntactic sugar while maintaing the ability to reason abstractly, meaning without examining the expansion itself, or the expansion logic, about these things: responsibility, determinism, segmentation, typing and binding.



Mechanisms of syntactic control

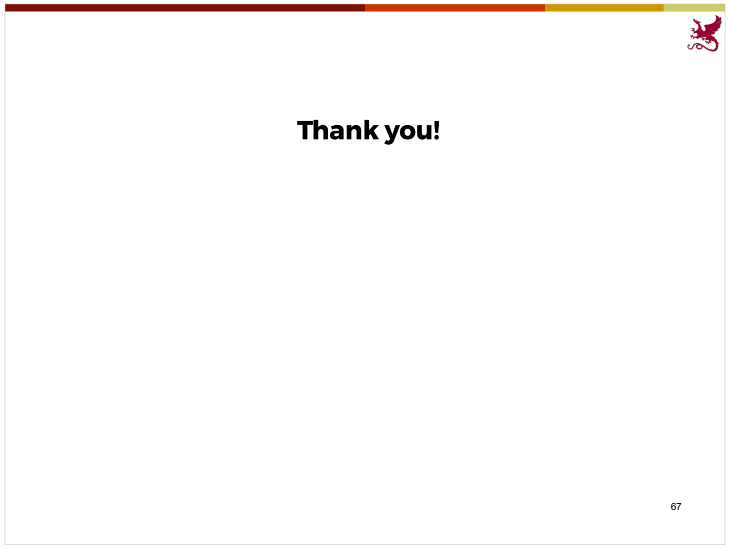
Direct Syntax Extension
✔ High level of syntactic control
✗ Must reason about the expansion

Typed Term-Rewriting Macros
✗ Limited syntactic control
✔ Abstract reasoning principles

Typed Literal Macros (TLMs)
✔ High level of syntactic control
✔ Abstract reasoning principles

66

Here is again the comparison to the other two approaches.



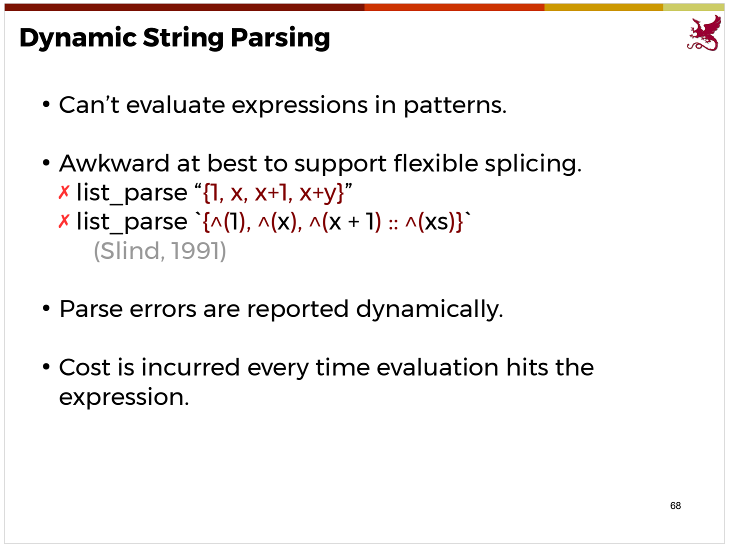
Thank you!

67

<3

See the acknowledgments section of my dissertation itself for a lot of nice words about a lot of nice people:

http://www.cs.cmu.edu/~comar/omar-thesis.pdf.


Dynamic String Parsing

- Can't evaluate expressions in patterns.
- Awkward at best to support flexible splicing.
  ✗ list_parse "{1, x, x+1, x+y}"
  ✗ list_parse `{^(1), ^(x), ^(x + 1) :: ^(xs)}`
  (Slind, 1991)
- Parse errors are reported dynamically.
- Cost is incurred every time evaluation hits the expression.

68

Bonus slide! Why can't we just parse strings at run-time????