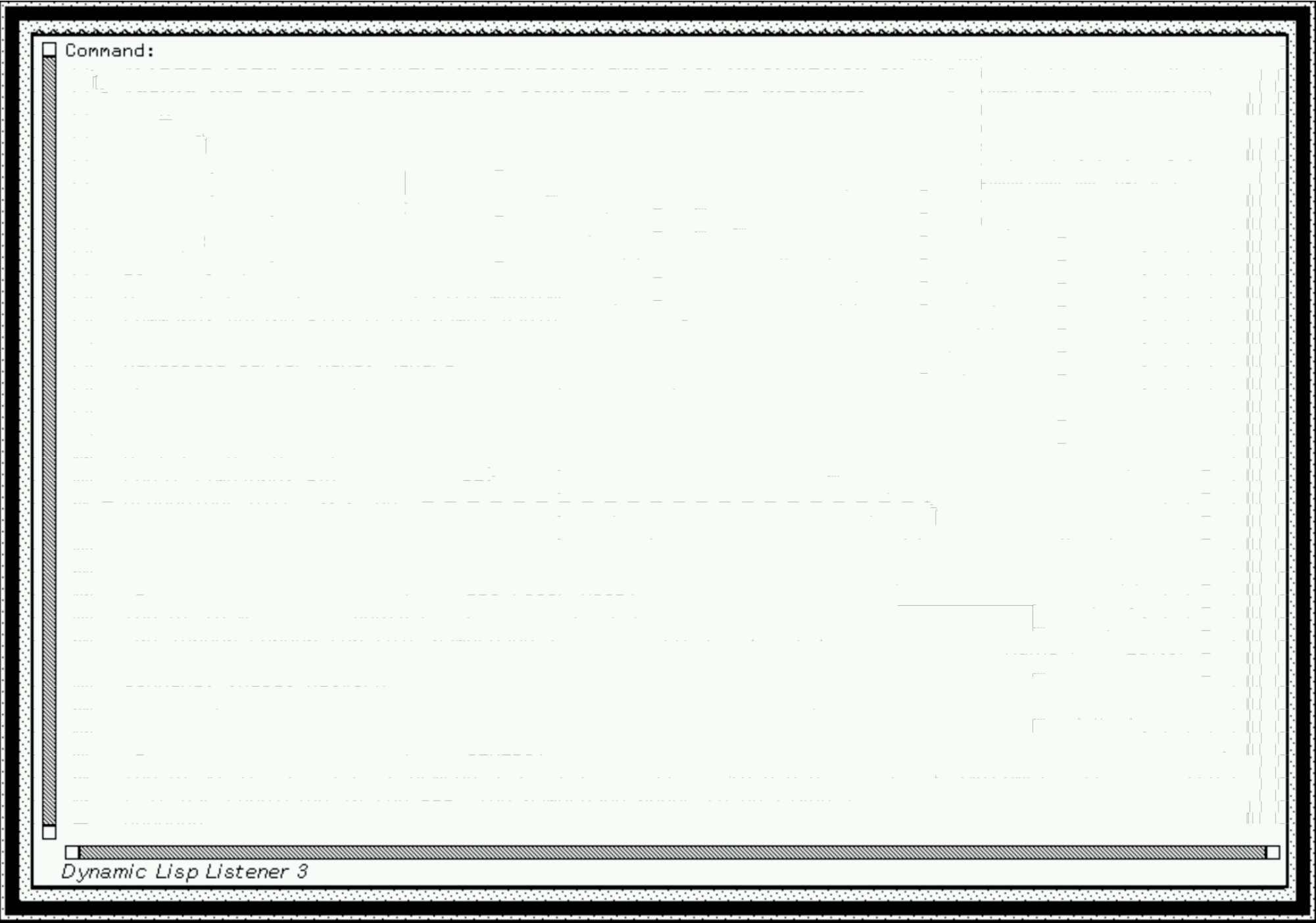
A Pretty Printer that Says what it Means

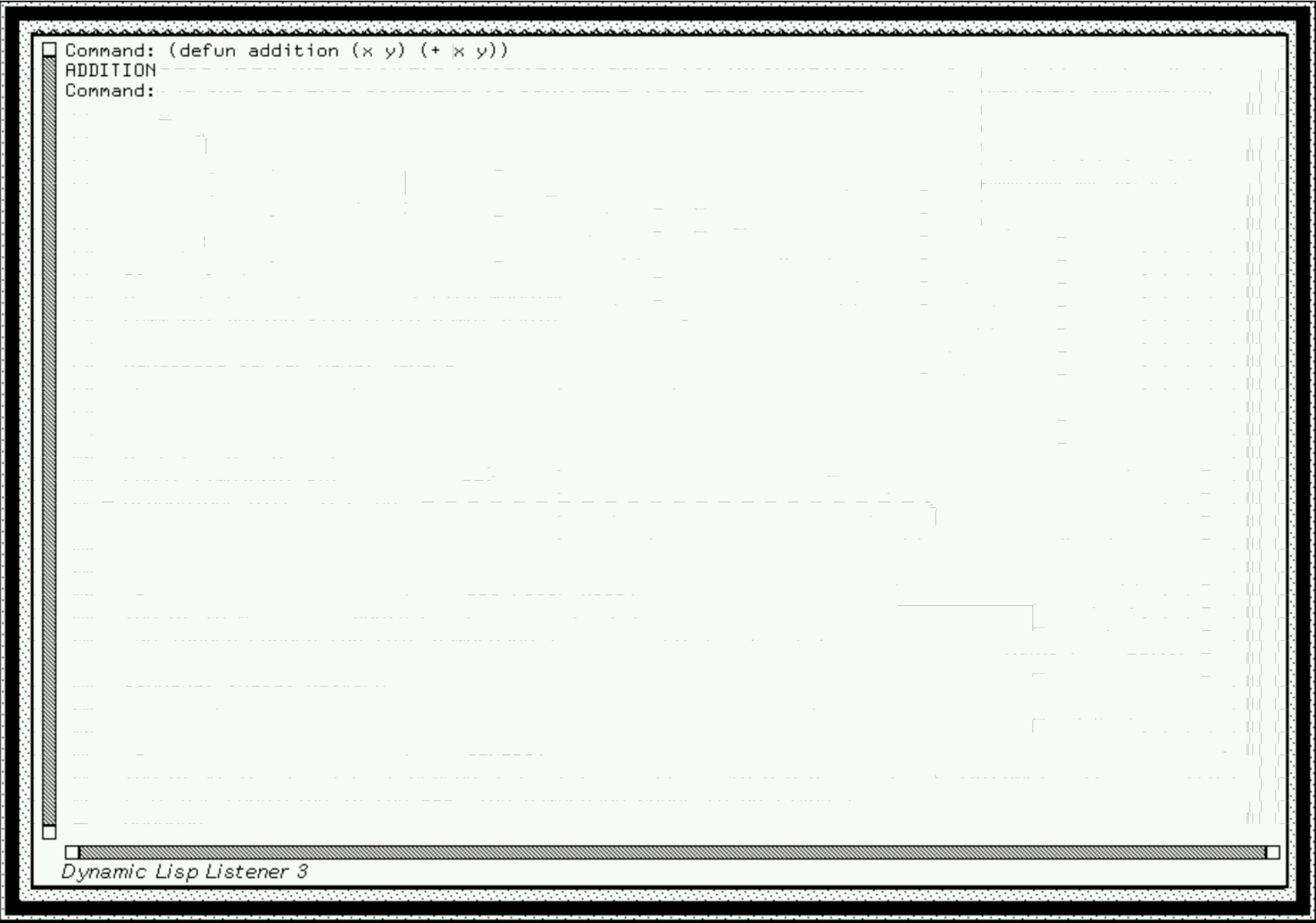
David Raymond Christiansen[1]

[1]Supported by the Danish Advanced Technology Foundation (*Højteknologifonden*) grant 017-2010-3

IT University of Copenhagen



Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.



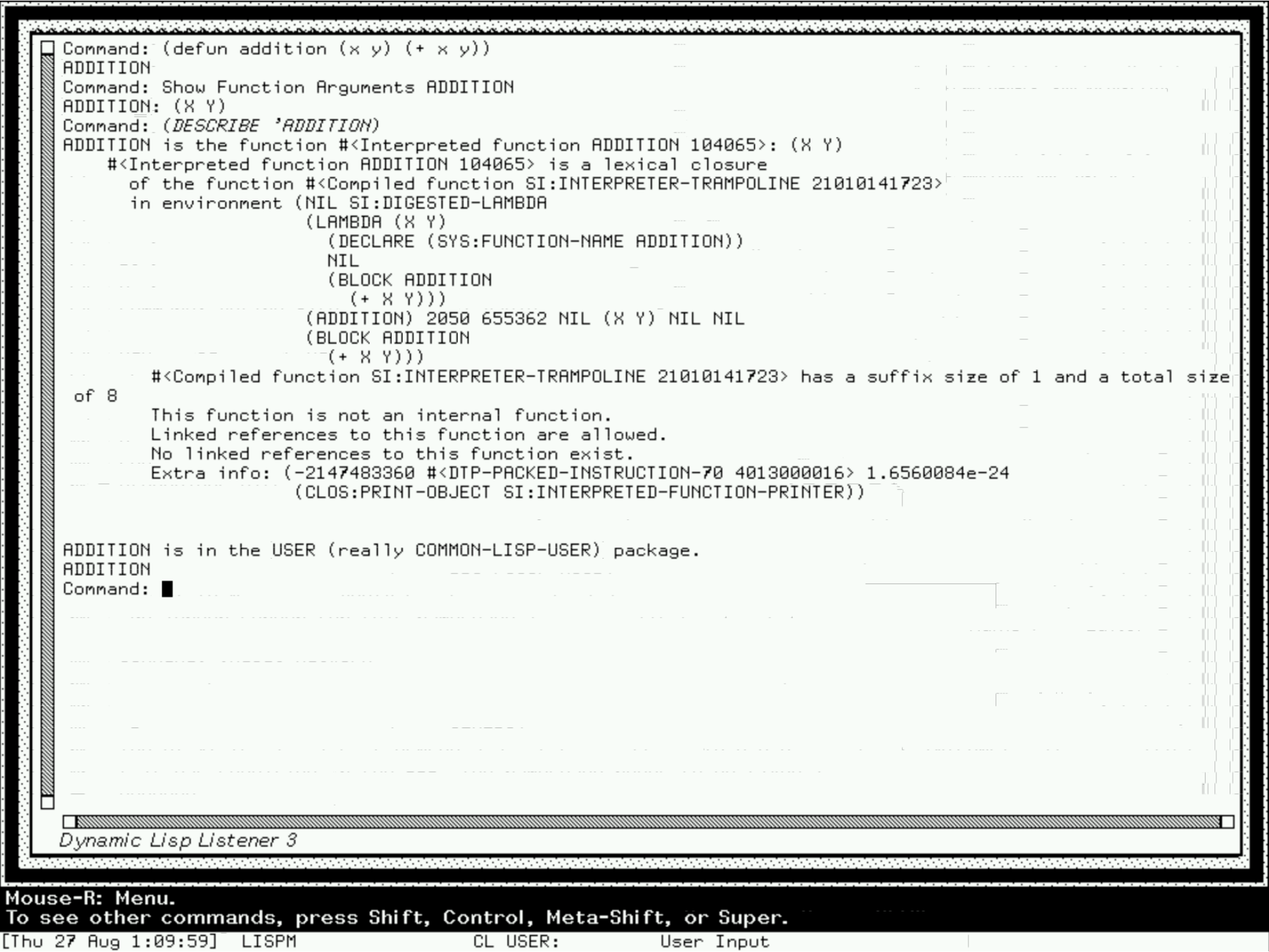
Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.

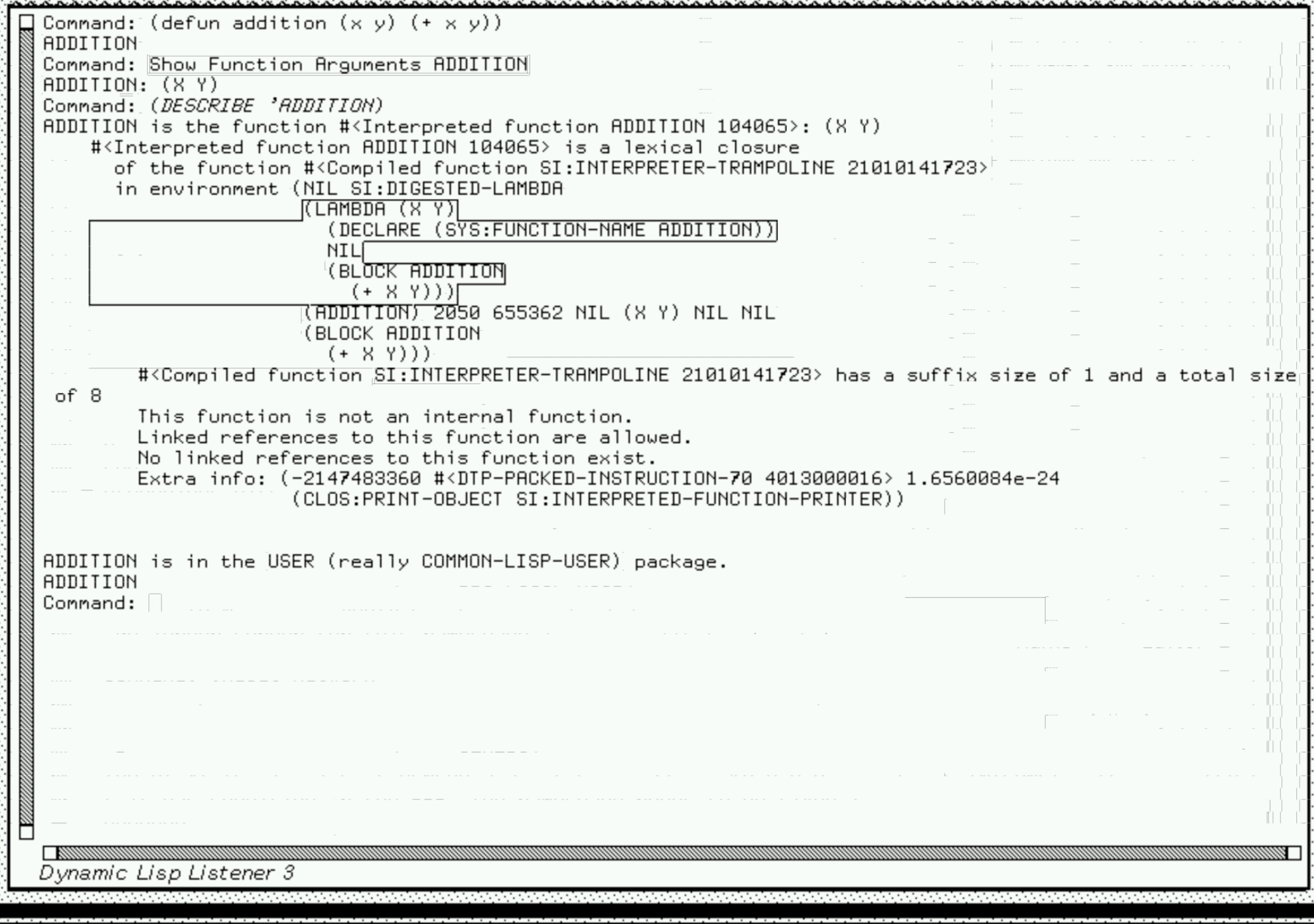

Command: (defun addition (x y) (+ x y))
ADDITION
Command:
Dynamic Lisp Listener 3
Mouse-L: 'ADDITION; Mouse-M: (DESCRIBE 'ADDITION); Mouse-R: Menu.
To see other commands, press Shift, Control, Meta, Meta-Shift, Super, or Super-Shift.
[Thu 27 Aug 1:07:30] LISPM CL USER: User Input


Operation on ADDITION:
'ADDITION
(DESCRIBE 'ADDITION)
(INSPECT 'ADDITION)
ADDITION
ADDITION [will not activate]
Edit Definition (name) ADDITION (type) Function
Edit viewspecs
Show this function's argument list
Marking and yanking menu
Presentation debugging menu
System menu
Window operation menu
Command:
ADDITION
Command:
Dynamic Lisp Listener 3
Mouse-L, -M, -R: Show this function's argument list.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Thu 27 Aug 1:07:52] LISPM CL USER: User Input


Command: (defun addition (x y) (+ x y))
ADDITION
Command: Show Function Arguments ADDITION
ADDITION: (X Y)
Command:
Dynamic Lisp Listener 3
Mouse-L: Show Function Arguments (function s; Mouse-M: Re-execute command after modifyi; Mouse-R: Menu
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Thu 27 Aug 1:09:17] LISPM CL USER: User Input


Operation on ADDITION:
'ADDITION
(DESCRIBE 'ADDITION)
(INSPECT 'ADDITION)
ADDITION
ADDITION [will not activate]
Edit Definition (name) ADDITION (type) Function
Edit viewspecs
Show this function's argument list
Marking and yanking menu
Presentation debugging menu
System menu
Window operation menu
Command:
ADDITION
Command:
ADDITION
Command:
Dynamic Lisp Listener 3
Mouse-L, -M, -R: (DESCRIBE 'ADDITION).
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Thu 27 Aug 1:09:37] LISPM CL USER: User Input


Command: (defun addition (x y) (+ x y))
ADDITION
Command: Show Function Arguments ADDITION
ADDITION: (X Y)
Command: (DESCRIBE 'ADDITION)
ADDITION is the function #<Interpreted function ADDITION 104065>: (X Y)
#<Interpreted function ADDITION 104065> is a lexical closure
of the function #<Compiled function SI:INTERPRETER-TRAMPOLINE 21010141723>
in environment (NIL SI:DIGESTED-LAMBDA
(LAMBDA (X Y)
(DECLARE (SYS:FUNCTION-NAME ADDITION))
NIL
(BLOCK ADDITION
(+ X Y)))
(ADDITION) 2050 655362 NIL (X Y) NIL NIL
(BLOCK ADDITION
(+ X Y)))
#<Compiled function SI:INTERPRETER-TRAMPOLINE 21010141723> has a suffix size of 1 and a total size
of 8
This function is not an internal function.
Linked references to this function are allowed.
No linked references to this function exist.
Extra info: (-2147483360 #<DTP-PACKED-INSTRUCTION-70 4013000016> 1.6560084e-24
(CLOS:PRINT-OBJECT SI:INTERPRETED-FUNCTION-PRINTER))
ADDITION is in the USER (really COMMON-LISP-USER) package.
ADDITION
Command:
Dynamic Lisp Listener 3
Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Thu 27 Aug 1:09:59] LISPM CL USER: User Input



Mouse-L: '(LAMBDA (X Y) (DECLARE #) ...); Mouse-M: (DESCRIBE '(LAMBDA # # ...)); Mouse-R: Menu.
To see other commands, press Shift, Control, Meta, Meta-Shift, Control-Meta, Super, or Super-Shift.

[Thu 27 Aug 1:12:27] LISPM CL USER: User Input

The Legend

Presentations: a connection between screen output and the underlying objects.

The world, today

- Programming languages can no longer live apart in their own world

The world, today

- Programming languages can no longer live apart in their own world
- We must support a variety of interfaces: console, Web, editors, and more

The world, today

- Programming languages can no longer live apart in their own world
- We must support a variety of interfaces: console, Web, editors, and more
- It is useful to support clients other than development environments: IRC bots, search engines, and unknown products of ingenuity

This talk

How did we implement presentations in Idris? Can Haskell make use of the same technique?

What is Idris?

What is Idris?

- Pure functional programming language

What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax

What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax
- Full dependent types

What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax
- Full dependent types
- Primarily developed by Edwin Brady at St Andrews, contributors around the world

What is Idris?

- Pure functional programming language
- Resembles Haskell: type classes, do-notation, monadic IO, layout syntax
- Full dependent types
- Primarily developed by Edwin Brady at St Andrews, contributors around the world
- **Written in Haskell**

Demo!

How does this work?

- The pretty printing library supports **semantic annotations**

How does this work?

- The pretty printing library supports **semantic annotations**
- Annotations describe the **meaning** of sub-documents

How does this work?

- The pretty printing library supports **semantic annotations**
- Annotations describe the **meaning** of sub-documents
- Provide in-band and out-of-band communication of annotations

How does this work?

- The pretty printing library supports **semantic annotations**
- Annotations describe the **meaning** of sub-documents
- Provide in-band and out-of-band communication of annotations
- REPL commands take annotations as arguments

Concretely

Begin with a pretty-printing library:

- Hughes-Peyton Jones style
- Wadler-Leijen style

What's in a pretty printer?

- Type `Doc`, representing sets of strings

What's in a pretty printer?

- Type `Doc`, representing sets of strings
- Combinators for constructing a `Doc`

What's in a pretty printer?

- Type `Doc`, representing sets of strings
- Combinators for constructing a `Doc`
- Renderers that convert a `Doc` into a `String`, or to write it to a handle

Make it say what it means!

1. Add a type parameter to `Doc`, representing the type of **annotations**

Make it say what it means!

1. Add a type parameter to `Doc`, representing the type of **annotations**
2. Add the combinator

```
annotate :: a → Doc a → Doc a
```

Make it say what it means!

1. Add a type parameter to `Doc`, representing the type of **annotations**
2. Add the combinator

```
annotate :: a → Doc a → Doc a
```

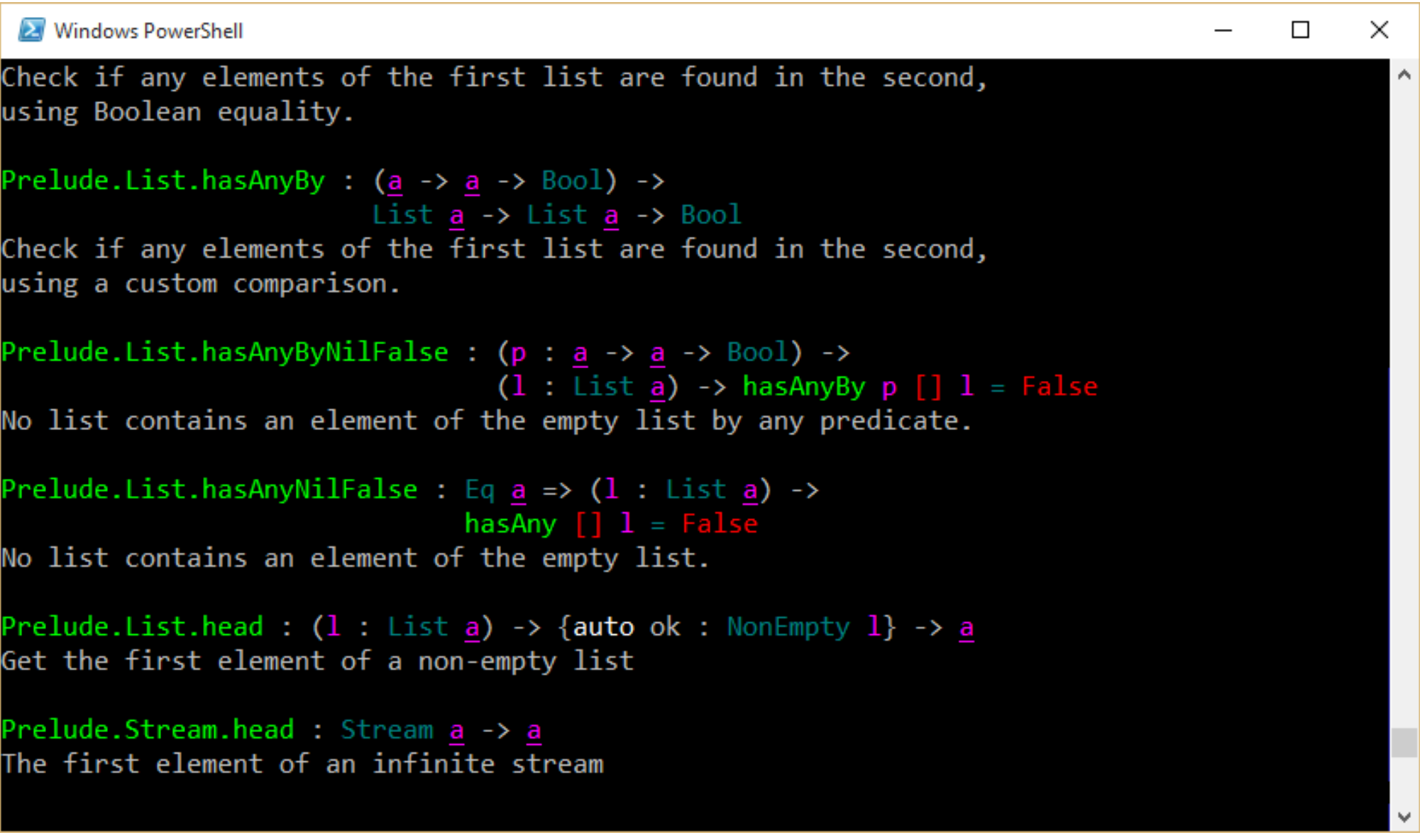
3. Add output methods:

```
outputSpans :: Doc a → (String, [(Int, Int, a)])

displayDecorated :: (a → String → String)
                 → Doc a → String
```

What About Windows?

Windows needs side effects to change console colors: ANSI codes don’t work!


Windows PowerShell
Check if any elements of the first list are found in the second,
using Boolean equality.
Prelude.List.hasAnyBy : (a -> a -> Bool) ->
List a -> List a -> Bool
Check if any elements of the first list are found in the second,
using a custom comparison.
Prelude.List.hasAnyByNilFalse : (p : a -> a -> Bool) ->
(l : List a) -> hasAnyBy p [] l = False
No list contains an element of the empty list by any predicate.
Prelude.List.hasAnyNilFalse : Eq a => (l : List a) ->
hasAny [] l = False
No list contains an element of the empty list.
Prelude.List.head : (l : List a) -> {auto ok : NonEmpty l} -> a
Get the first element of a non-empty list
Prelude.Stream.head : Stream a -> a
The first element of an infinite stream

More generality, please!

```
displayDecoratedA :: (Applicative f, Monoid b)
                  ⇒ (String → f b)
                  → (a → f b) → (a → f b)
                  → Doc a → f b
```

More generality, please!

```
displayDecoratedA :: (Applicative f, Monoid b)
                  ⇒ (String → f b)
                  → (a → f b) → (a → f b)
                  → Doc a → f b
```

For Windows output, let `f` be `IO` and let `b` be `()`

Doc is a Functor

Because Doc is a Functor, we can transform or decorate annotations.

Doc is a Functor

Because Doc is a Functor, we can transform or decorate annotations. Uses:

- Add additional type information without imposing dependencies on pretty-printer
- Convert annotations to the IDE protocol
- Type check terms that occur inside of docstrings

Implementations

`annotated-wl-pprint`

Idris's pretty-printing library — a Wadler-Leijen derivative

`pretty`

Trevor Elliott at Galois implemented annotations for it in 2014

Similar libraries

`wl-pprint-extras`

A free monad based on `wl-pprint` where effects can be embedded in documents

Thanks for listening!

- When pretty printers say what they mean, listeners don't need to guess

Thanks for listening!

- When pretty printers say what they mean, listeners don't need to guess (or, even worse — parse)

Thanks for listening!

- When pretty printers say what they mean, listeners don't need to guess (or, even worse — parse)
- Editor commands can get ahold of references directly

Thanks for listening!

- When pretty printers say what they mean, listeners don't need to guess (or, even worse — parse)
- Editor commands can get ahold of references directly
- Can be implemented incrementally:

```
type Doc = Annotated.Doc ()
```

Thanks for listening!

- When pretty printers say what they mean, listeners don't need to guess (or, even worse — parse)
- Editor commands can get ahold of references directly
- Can be implemented incrementally:

```
type Doc = Annotated.Doc ()
```

- Let's take good ideas from Lisp and Smalltalk UIs, and make them **even better** with types!