Lispz

An old language for a new paradigm

http://lispz.marrington.net

Contents of this Talk

Why Another Language Implementation

Lisp History

Lispz as a Language

The Future

Where to Learn More

Why Another Language Implementation

Compile to JavaScript

Compile on the Browser

Functional Language

Expose JavaScript Goodies

Use JavaScript Libraries

History

1956-8 - Invented at MIT

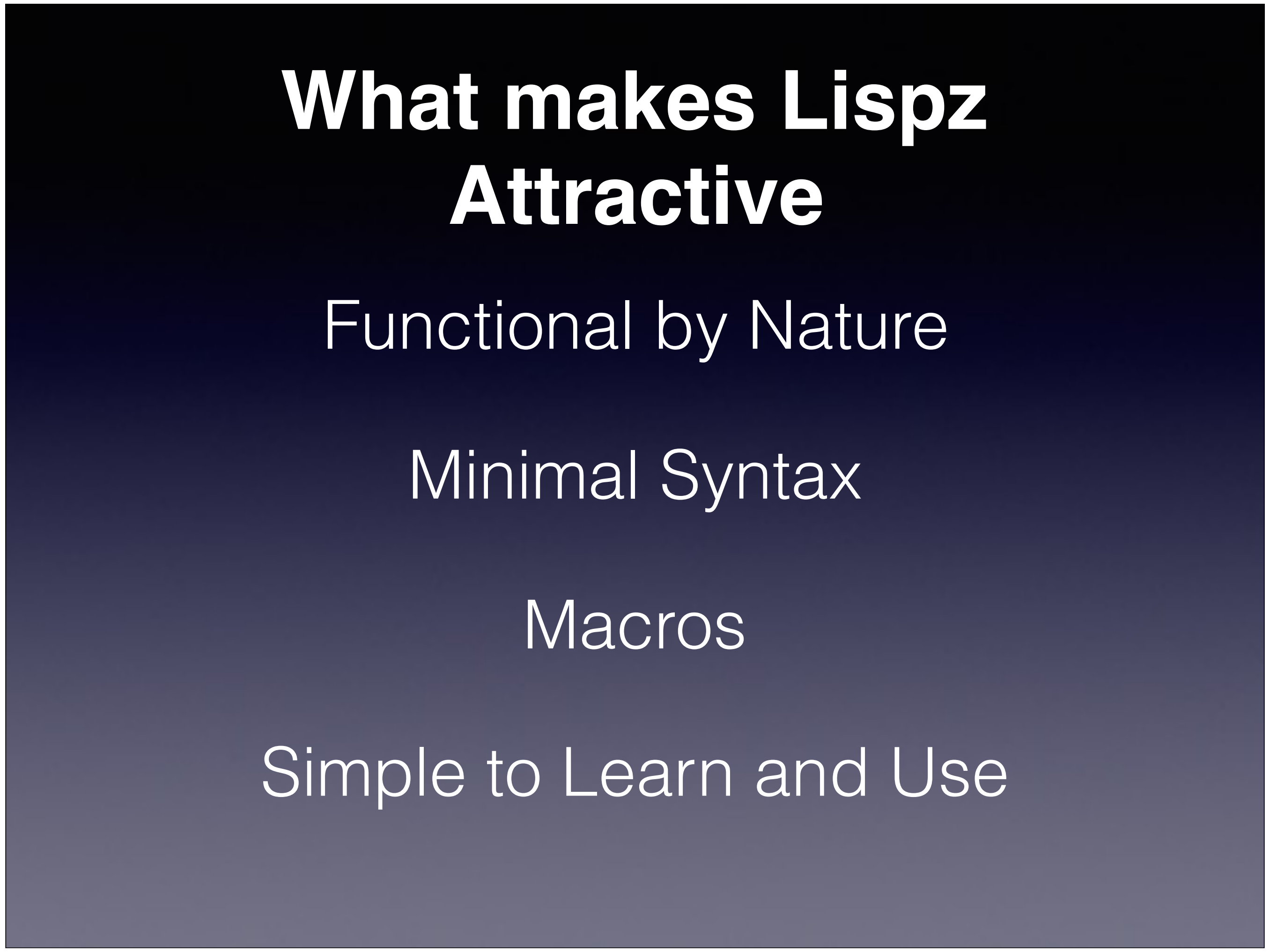
John McCarthy

Lambda Calculus, Recursion, Referential Integrity

Lead AI Language for Two Decades

Survivors: Scheme, Common Lisp, Racket & friends

Clojure



What makes Lispz Attractive

Functional by Nature

Minimal Syntax

Macros

Simple to Learn and Use

Goals

Simplicity

Ease of Use

Support Functional Programming

Support Referential Integrity

Lispz as a Language

Minimal Syntax

```
(action p1 p2 p3)

[raw list]

[[list]]

{associative: list}
```

Lispz as a Language

Minimal Syntax

```
(function-reference p1 p2 p3)

[raw list]

[[list]]

{associative: list}
```

Lispz as a Language

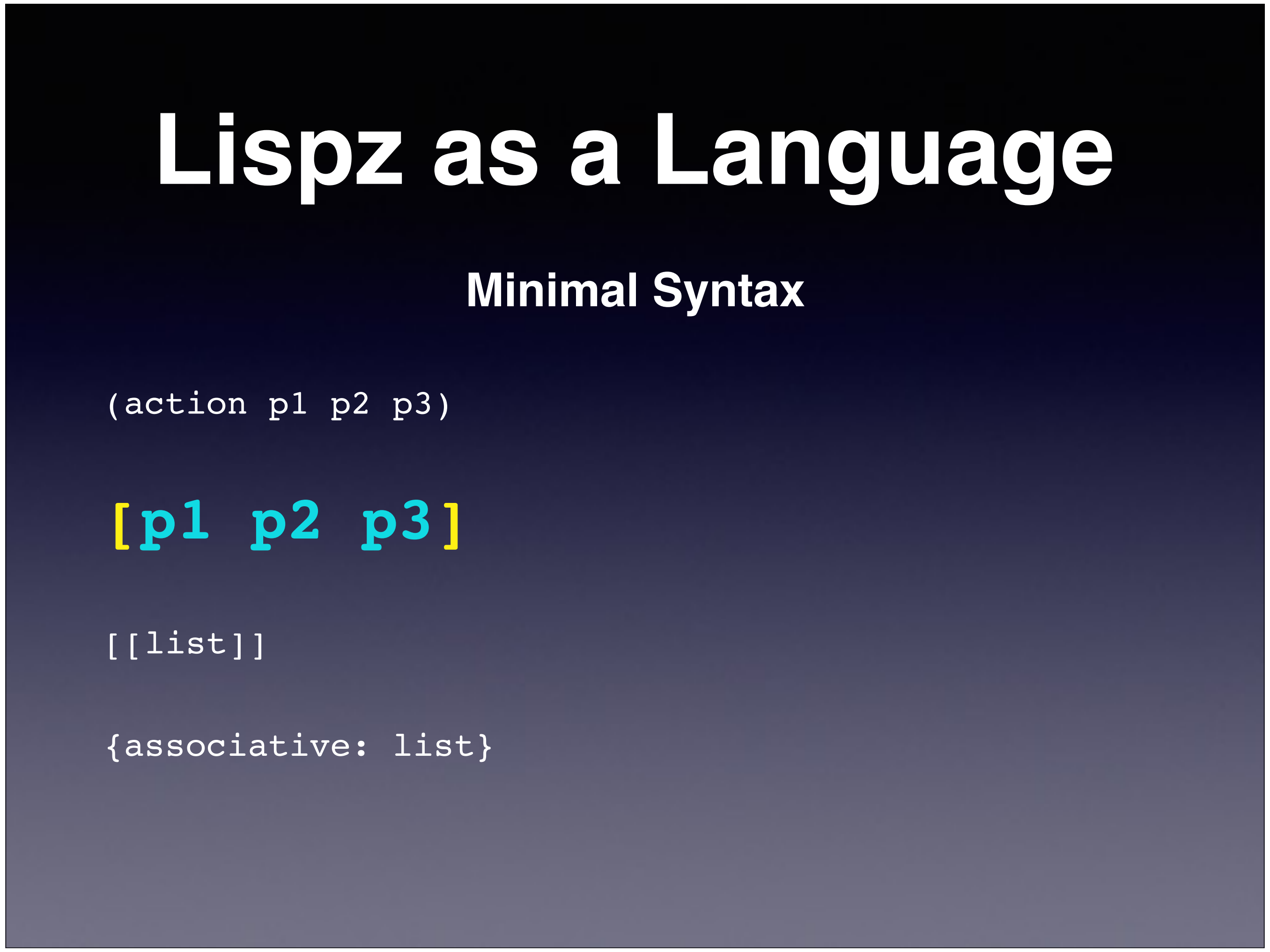
Minimal Syntax

```
(macro p1 p2 p3)

[raw list]

[[list]]

{associative: list}
```



Lispz as a Language

Minimal Syntax

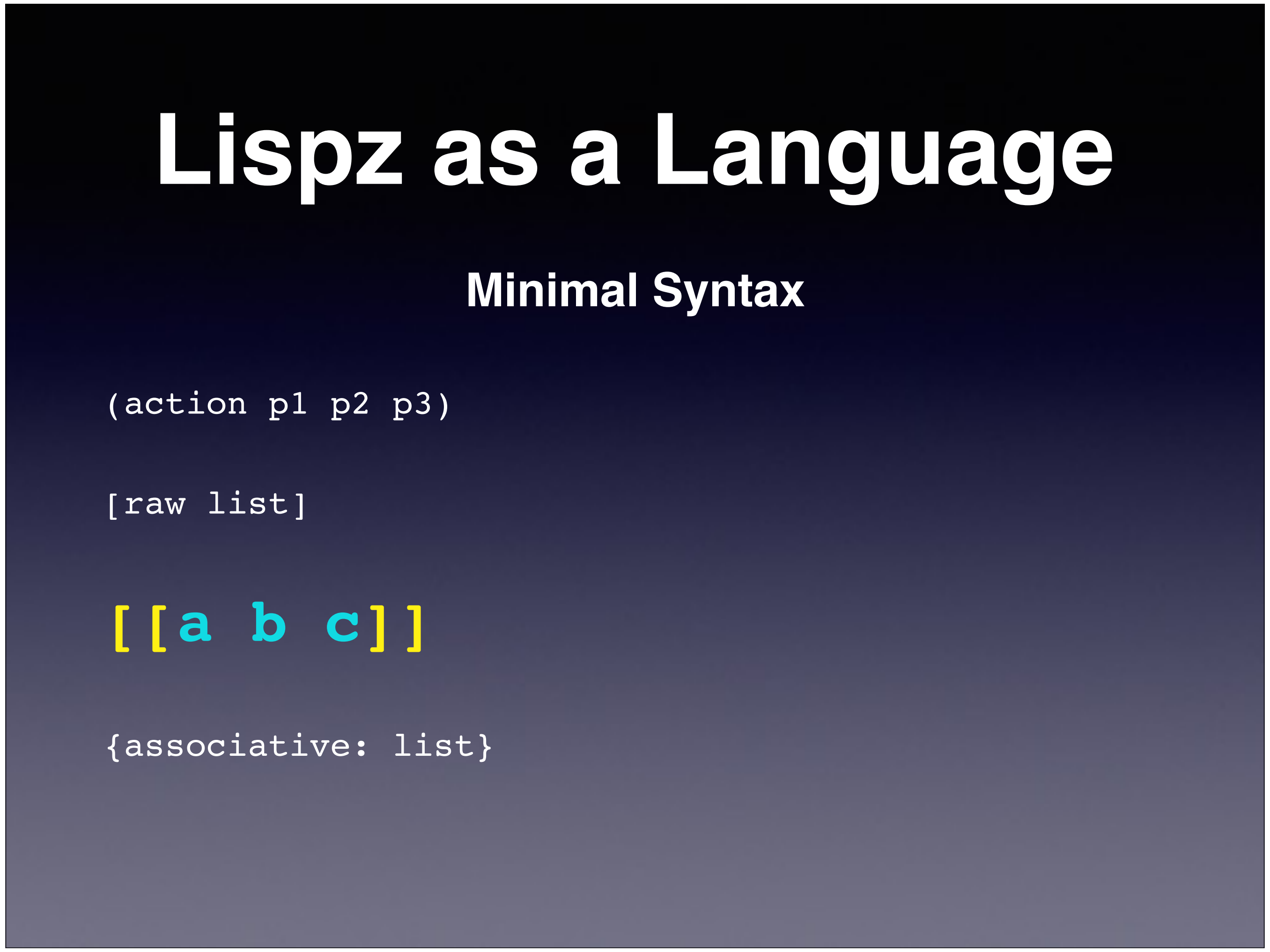
```
(action p1 p2 p3)

[p1 p2 p3]

[[list]]

{associative: list}
```

Raw list - just like a list of parameters without the function reference.



Lispz as a Language

Minimal Syntax

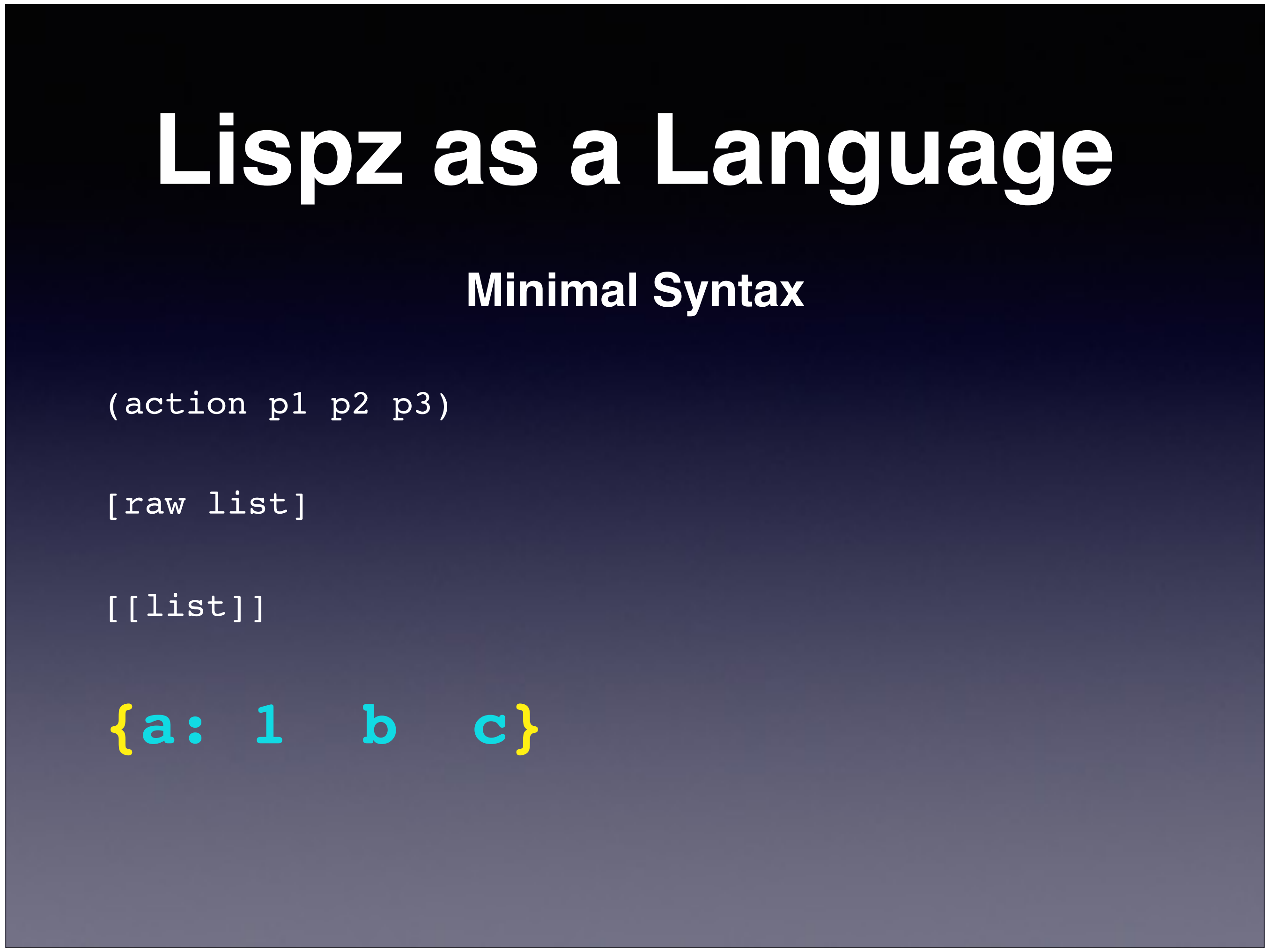
```
(action p1 p2 p3)

[raw list]

[[a b c]]

{associative: list}
```

Analogous to an array in almost any language.



Lispz as a Language

Minimal Syntax

```
(action p1 p2 p3)

[raw list]

[[list]]

{a: 1  b  c}
```

Also called maps or dictionaries.
Either the key ends in a colon or it uses the symbol as key and the reference as value.

Referential Integrity

Flesh is the juicy pure functions.

Rind deals with the outside world.

Pips are contained stateful objects - caches, etc

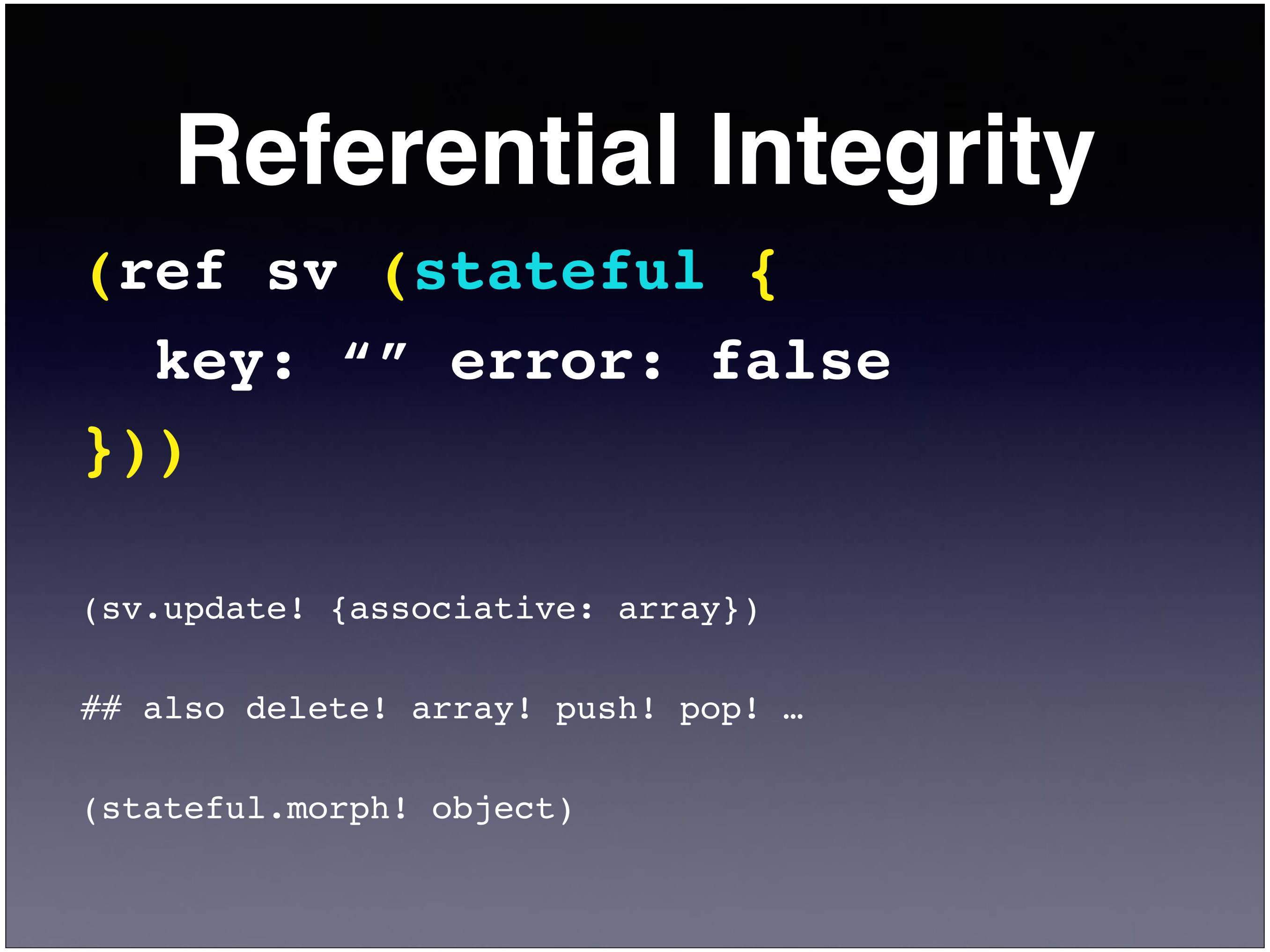
Referential Integrity

```
(ref sv (stateful {seed}))

(sv.update! {associative: array})

## also delete! array! push! pop! …

(stateful.morph! object)
```



Referential Integrity

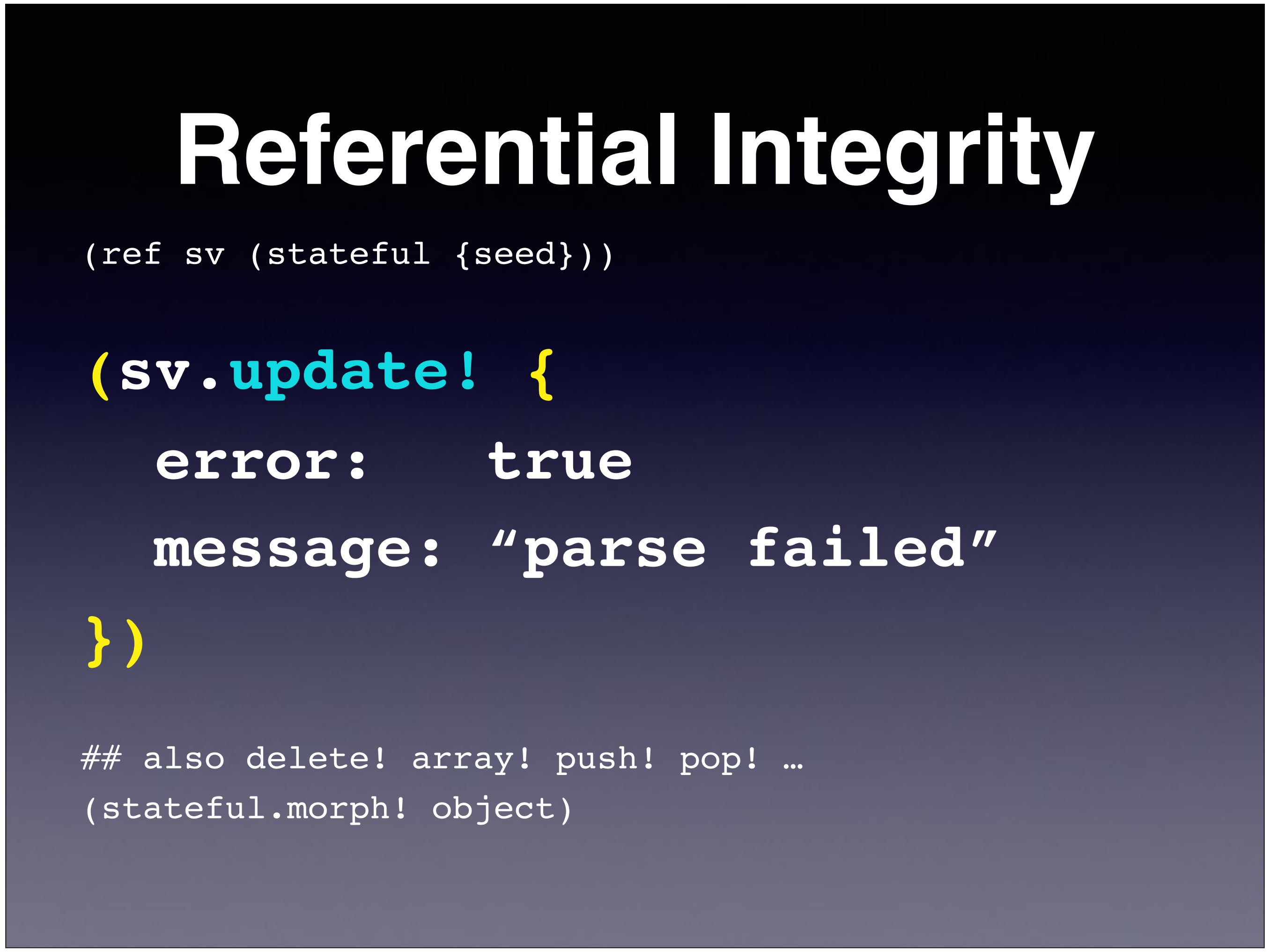
```
(ref sv (stateful {
  key: “” error: false
}))
```

```
(sv.update! {associative: array})

## also delete! array! push! pop! …

(stateful.morph! object)
```

A stateful object is an associative array with benefits.



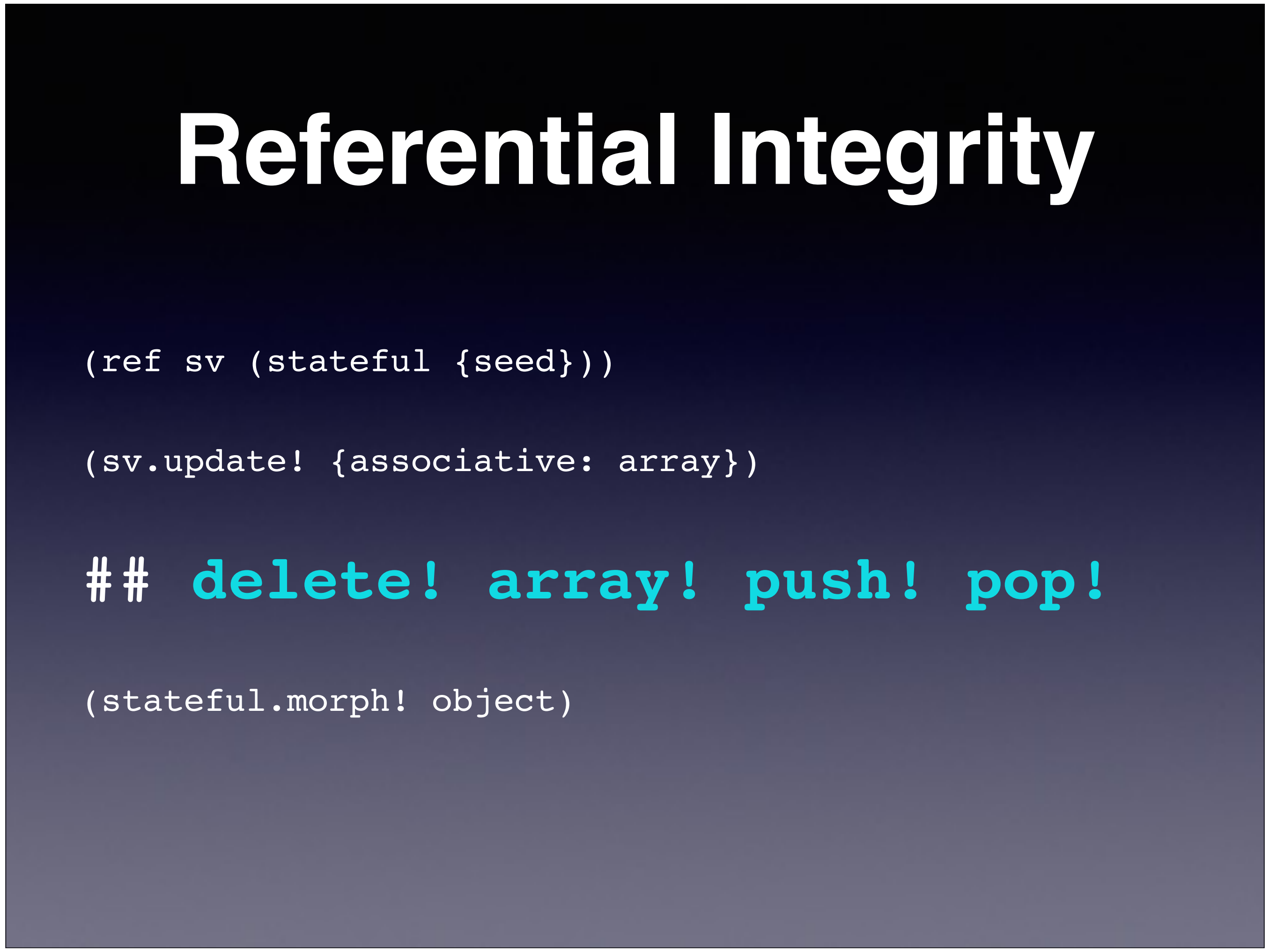
Referential Integrity

```
(ref sv (stateful {seed}))

(sv.update! {
  error:   true
  message: “parse failed”
})

## also delete! array! push! pop! …
(stateful.morph! object)
```

We can change and add members.



Referential Integrity

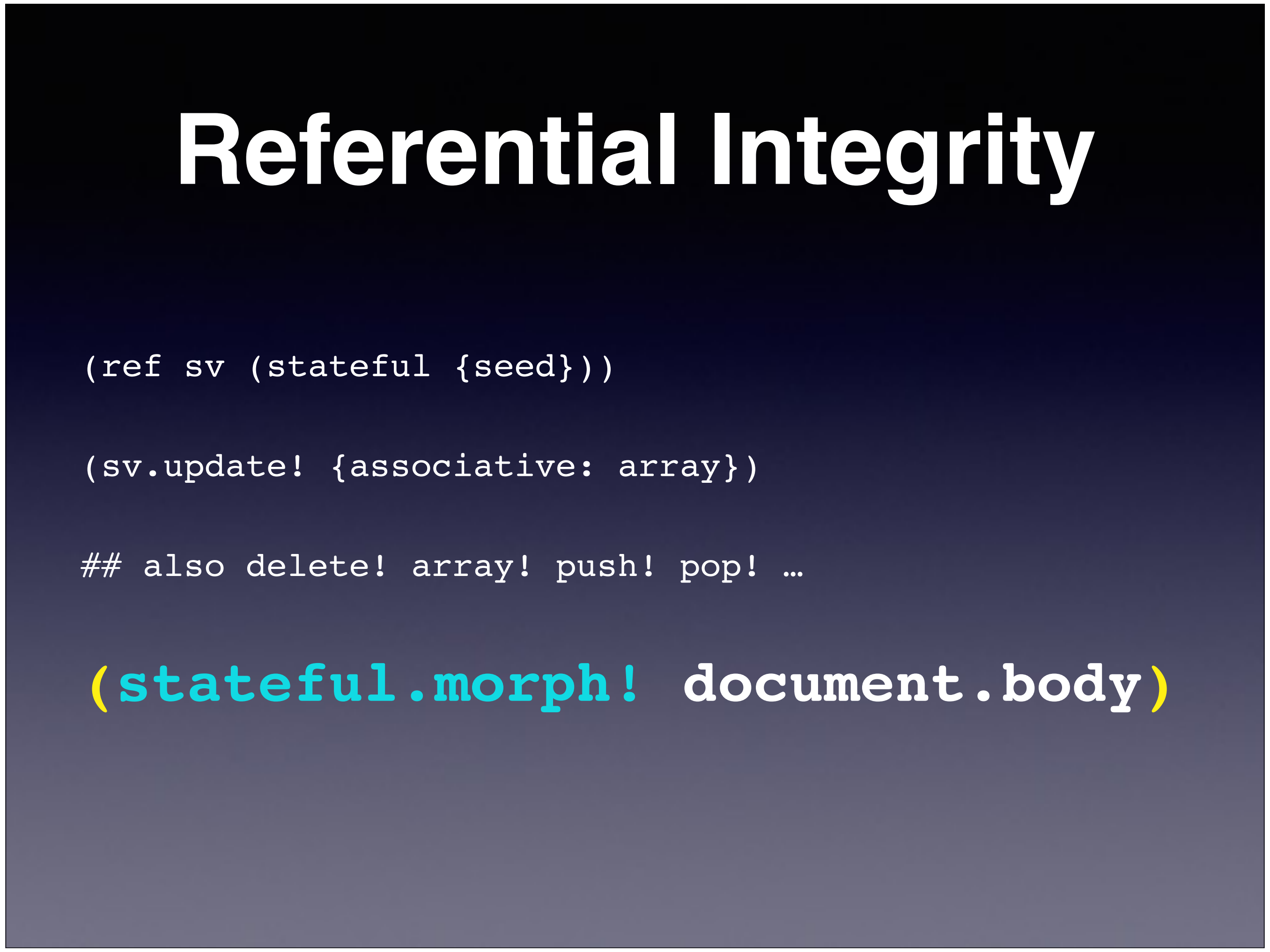
```
(ref sv (stateful {seed}))

(sv.update! {associative: array})

## delete! array! push! pop!

(stateful.morph! object)
```

As well as delete and deal with changing arrays.



Referential Integrity

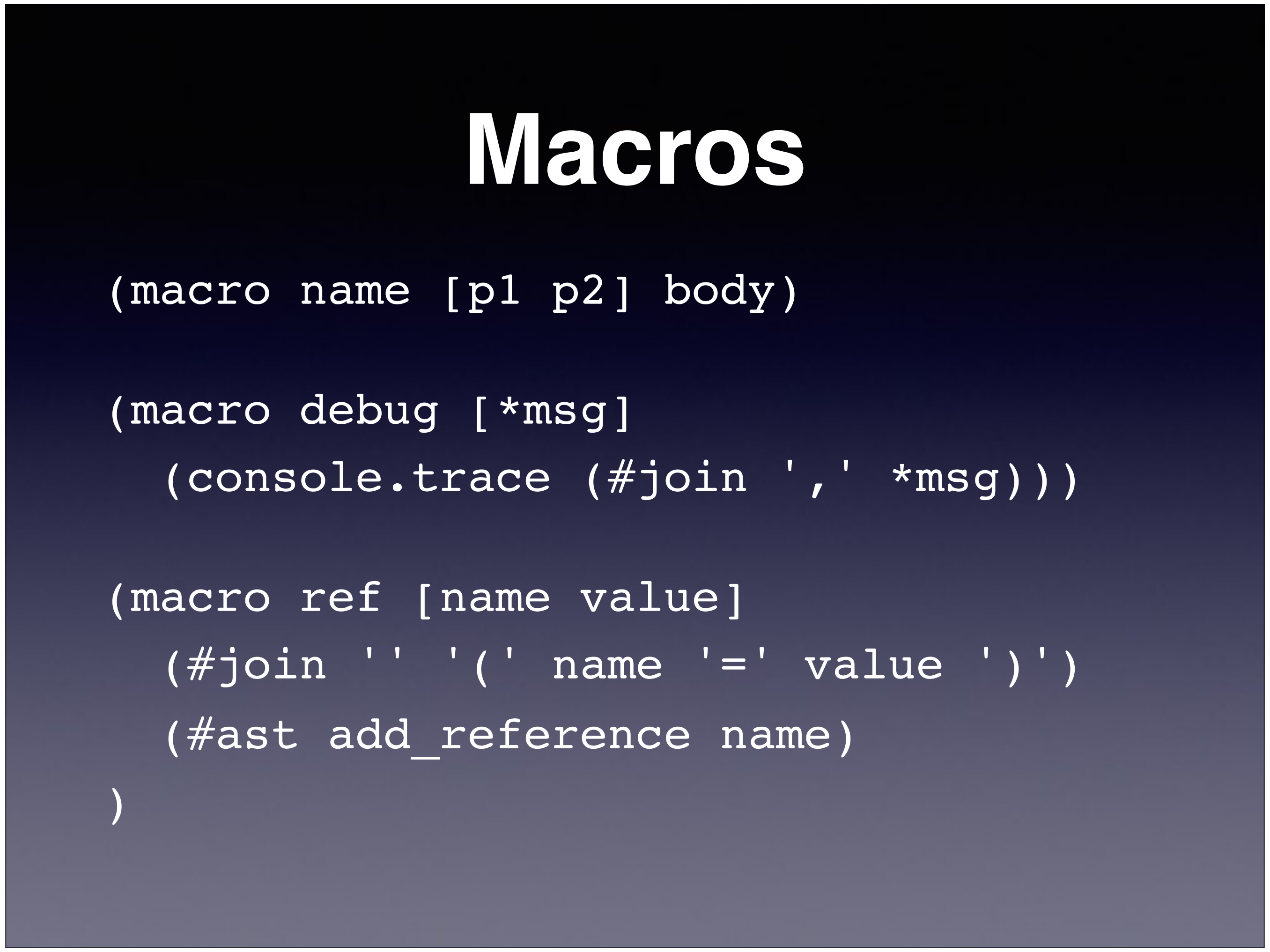
```
(ref sv (stateful {seed}))

(sv.update! {associative: array})

## also delete! array! push! pop! …

(stateful.morph! document.body)
```

To deal with the outside world.



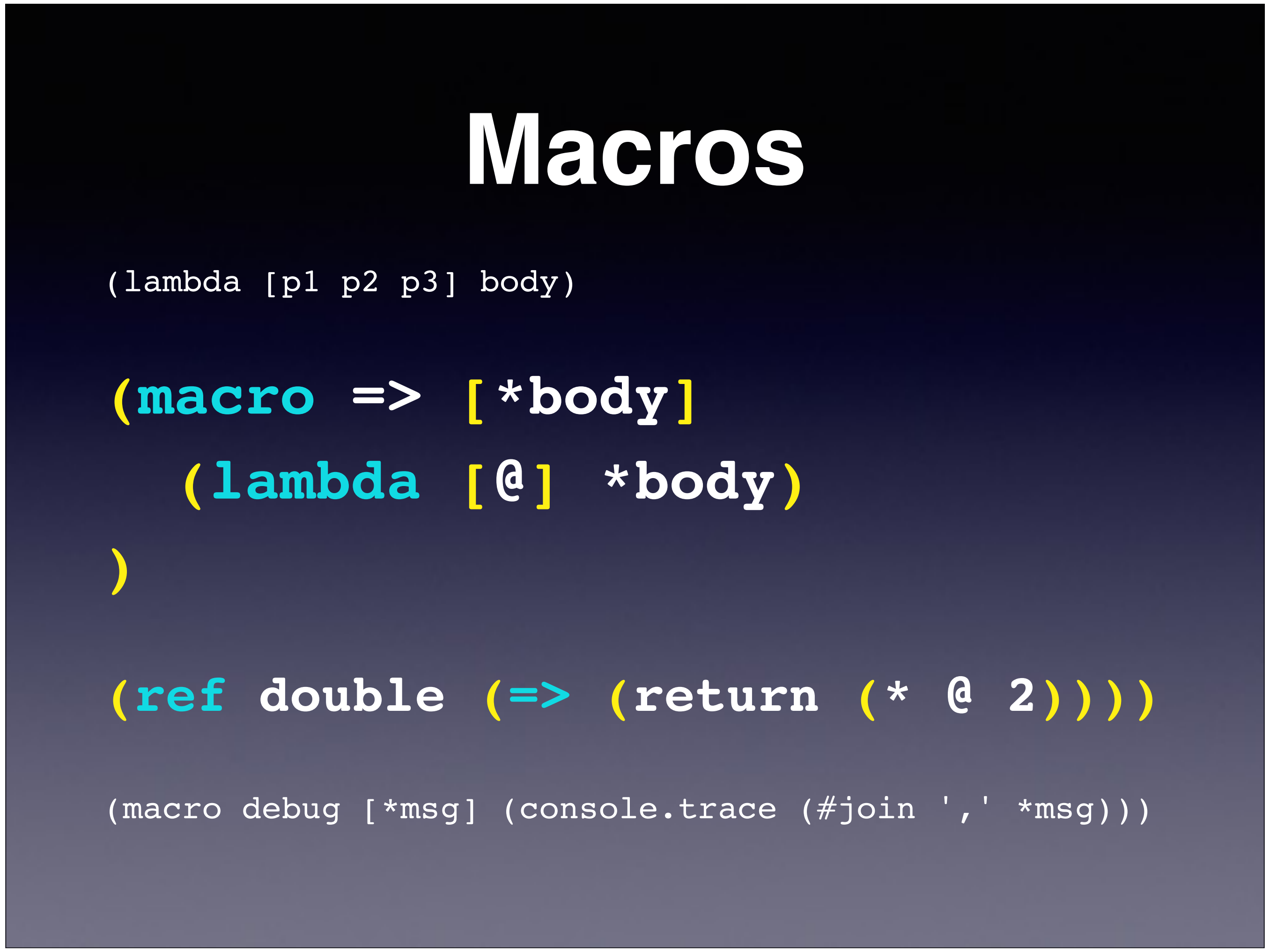
Macros

```
(macro name [p1 p2] body)

(macro debug [*msg]
  (console.trace (#join ',' *msg)))

(macro ref [name value]
  (#join '' '(' name '=' value ')')
  (#ast add_reference name)
)
```

Macros are the magic core that makes lisp shine.



Macros

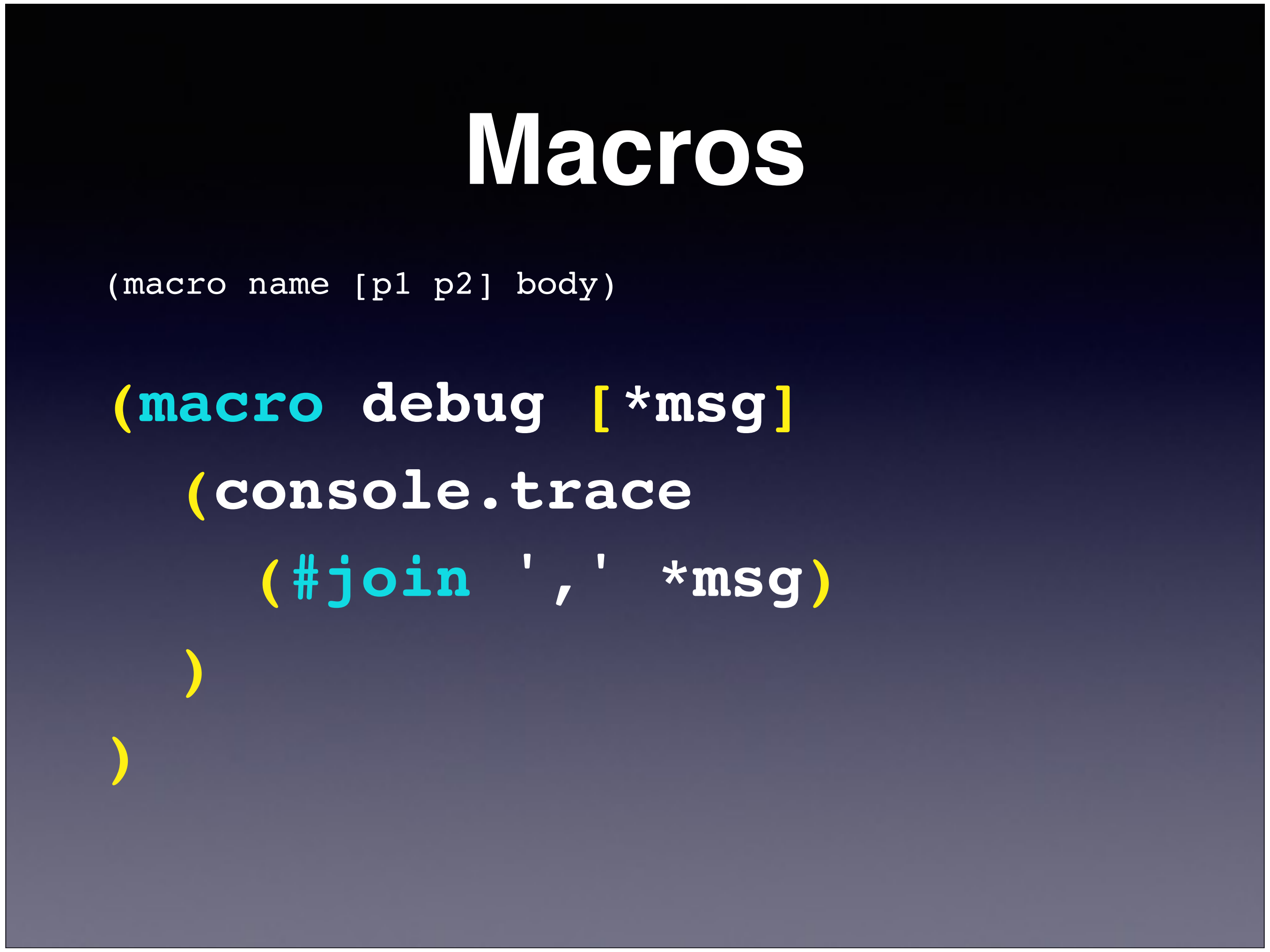
```
(lambda [p1 p2 p3] body)

(macro => [*body]
  (lambda [@] *body)
)

(ref double (=> (return (* @ 2))))

(macro debug [*msg] (console.trace (#join ',' *msg)))
```

Here we create a fat arrow function definition - basic text replacement similar to C.

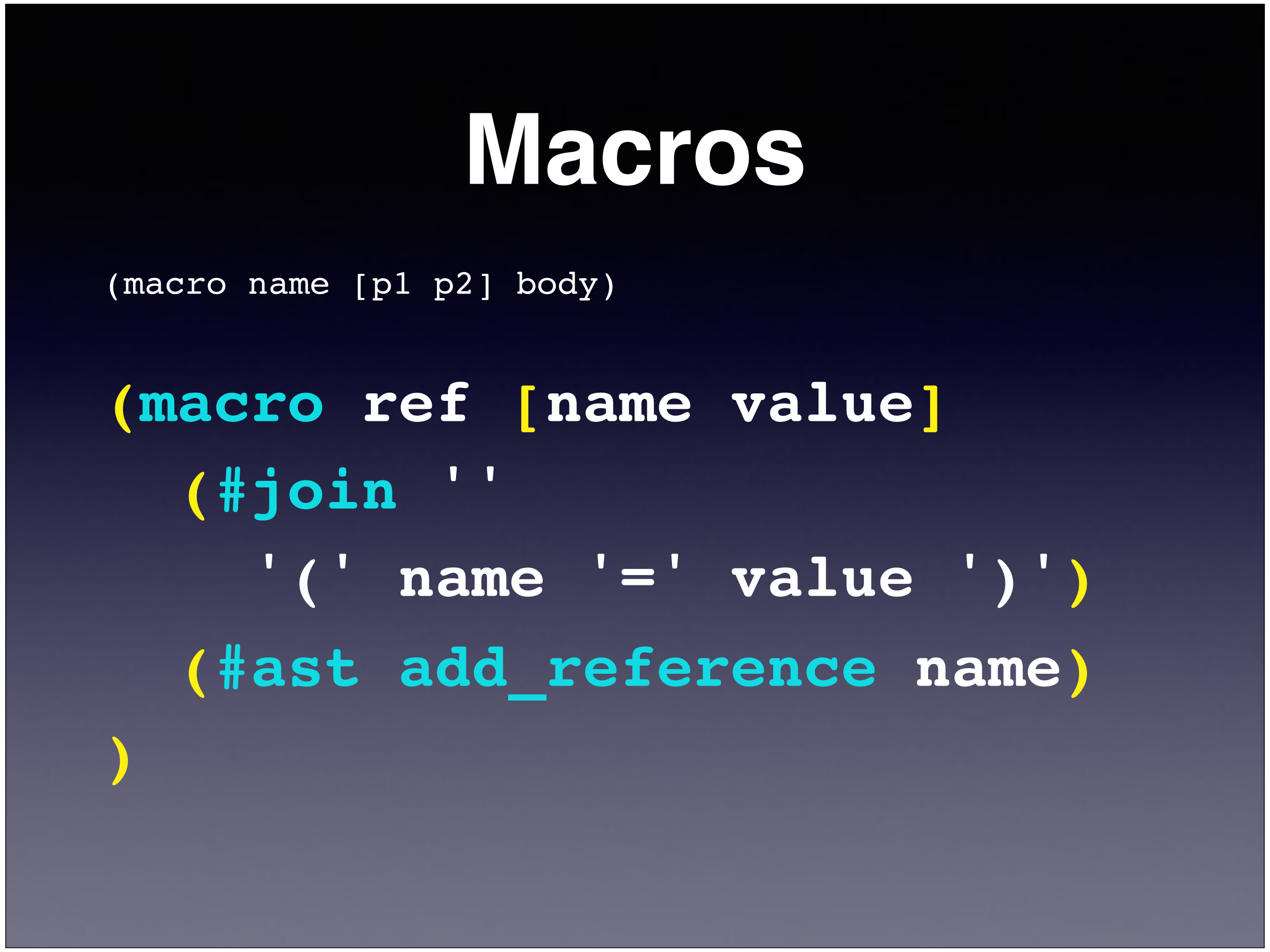


Macros

```
(macro name [p1 p2] body)
```

```
(macro debug [*msg]
  (console.trace
    (#join ',' *msg)
  )
)
```

#join is an immediate function that acts at compile time.

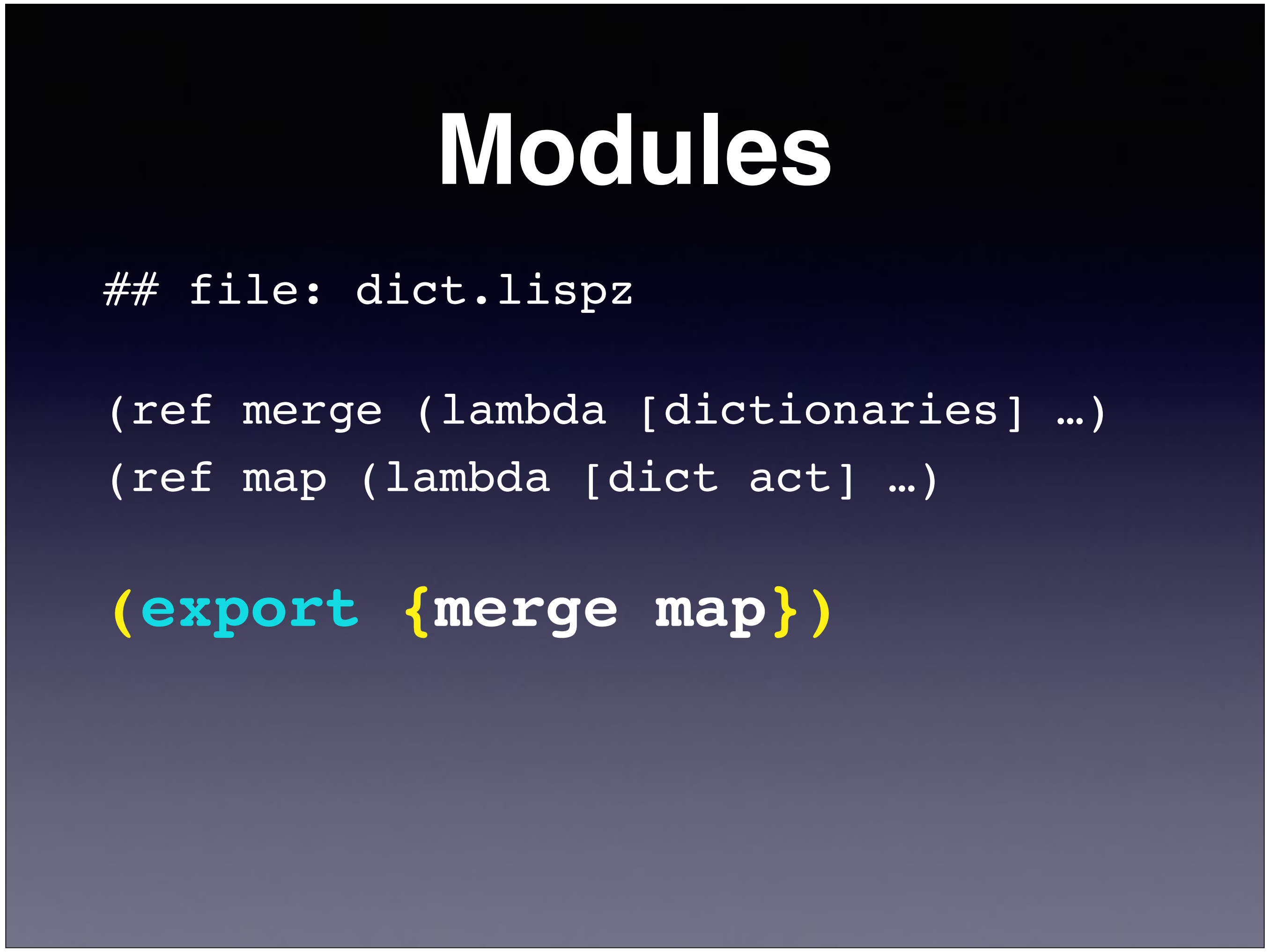


Macros

```
(macro name [p1 p2] body)
```

```
(macro ref [name value]
  (#join ''
    '(' name '=' value ')')
  (#ast add_reference name)
)
```

Here we have two immediate functions acting in concert to create a JavaScript var definition.



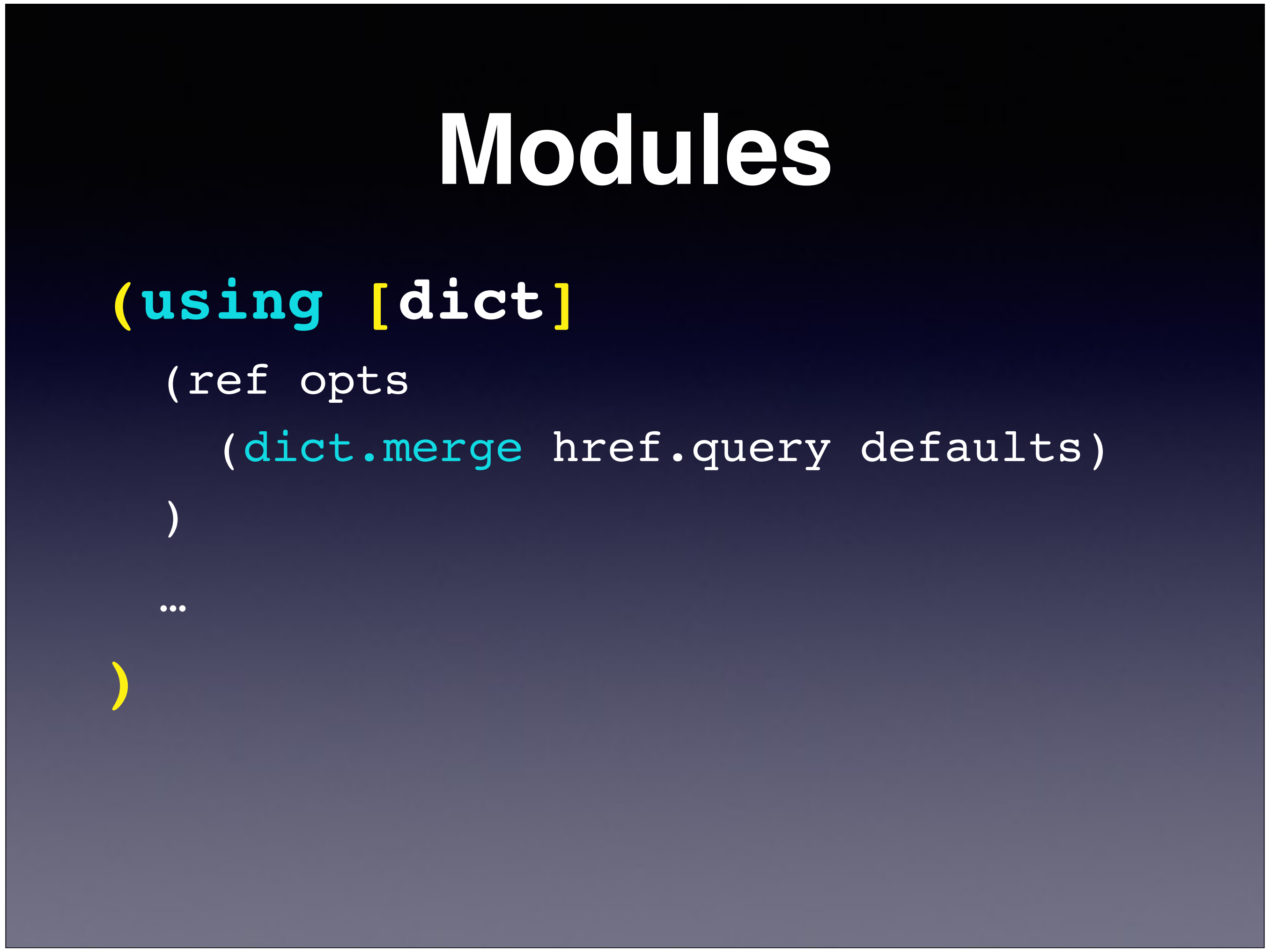
Modules

```
## file: dict.lispz

(ref merge (lambda [dictionaries] …)
(ref map (lambda [dict act] …)

(export {merge map})
```

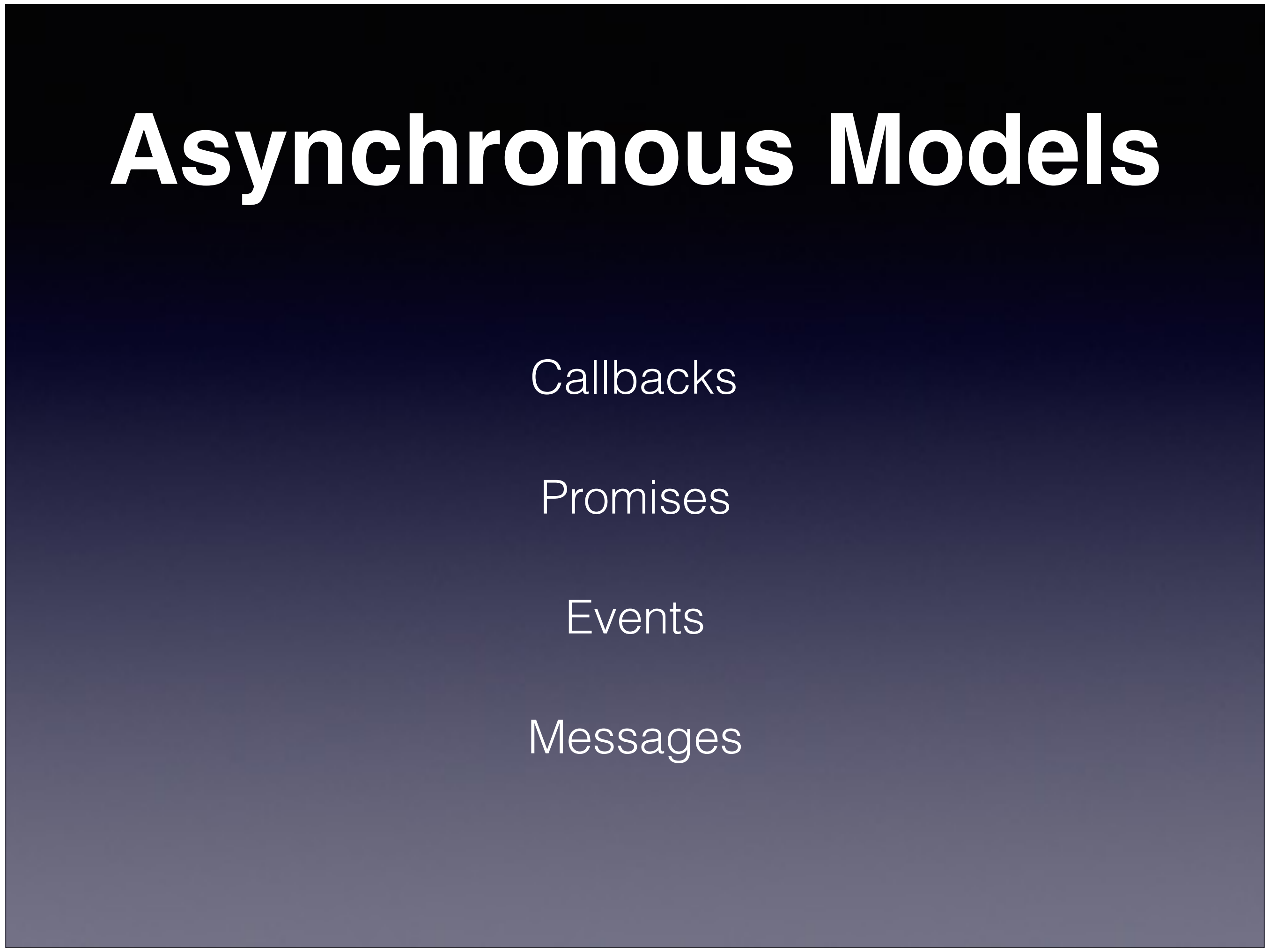
Each file is a module. Modules are loaded separately during development and pushed together for production.



Modules

```
(using [dict]
  (ref opts
    (dict.merge href.query defaults)
  )
  …
)
```

Wrap a using statement to access module exports. It is asynchronous when loading a module, immediate otherwise.



Asynchronous Models

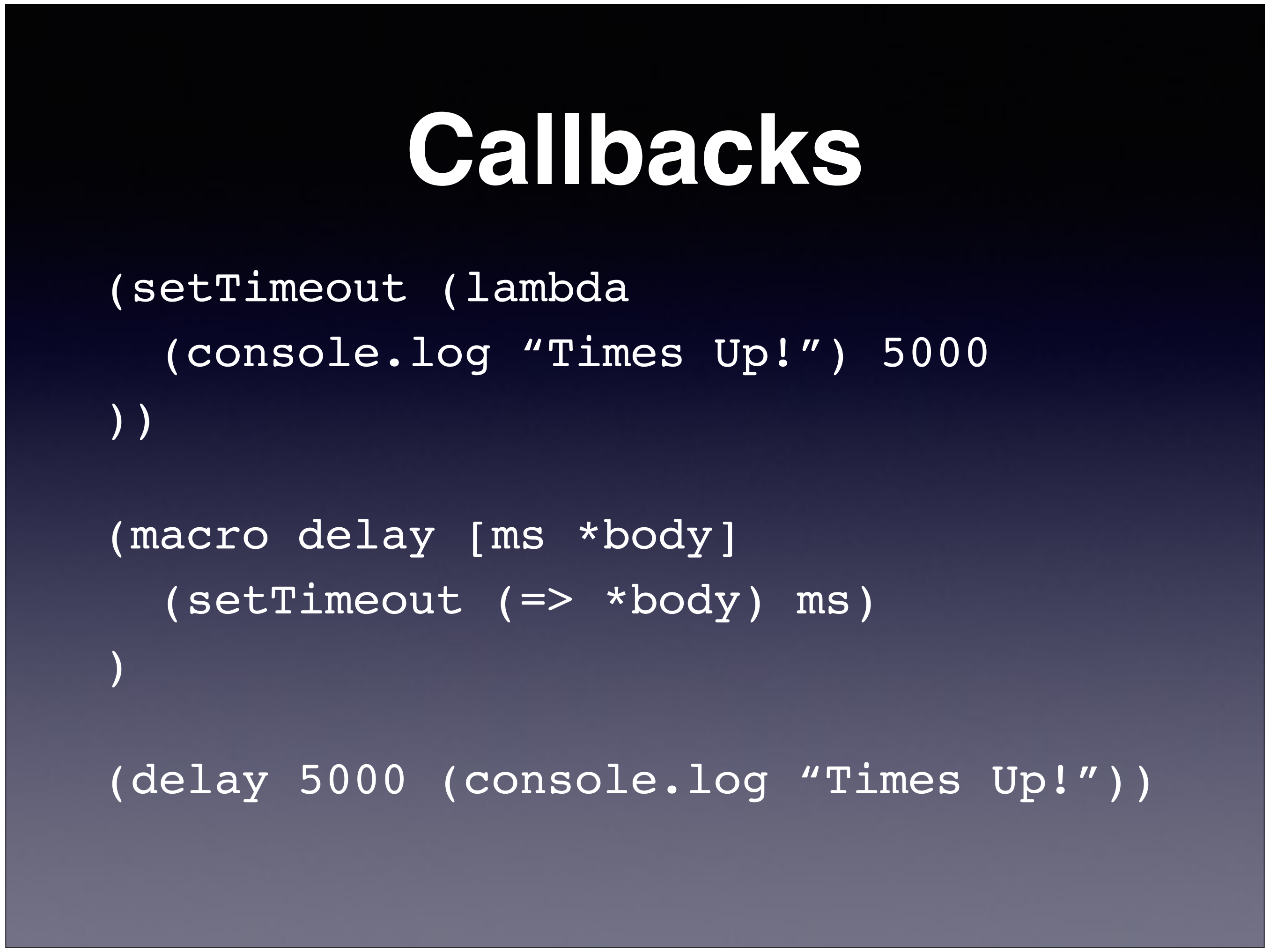
Callbacks

Promises

Events

Messages

JavaScript works using engines with an asynchronous model - Windows 1 to 3 called it cooperative multi-tasking.



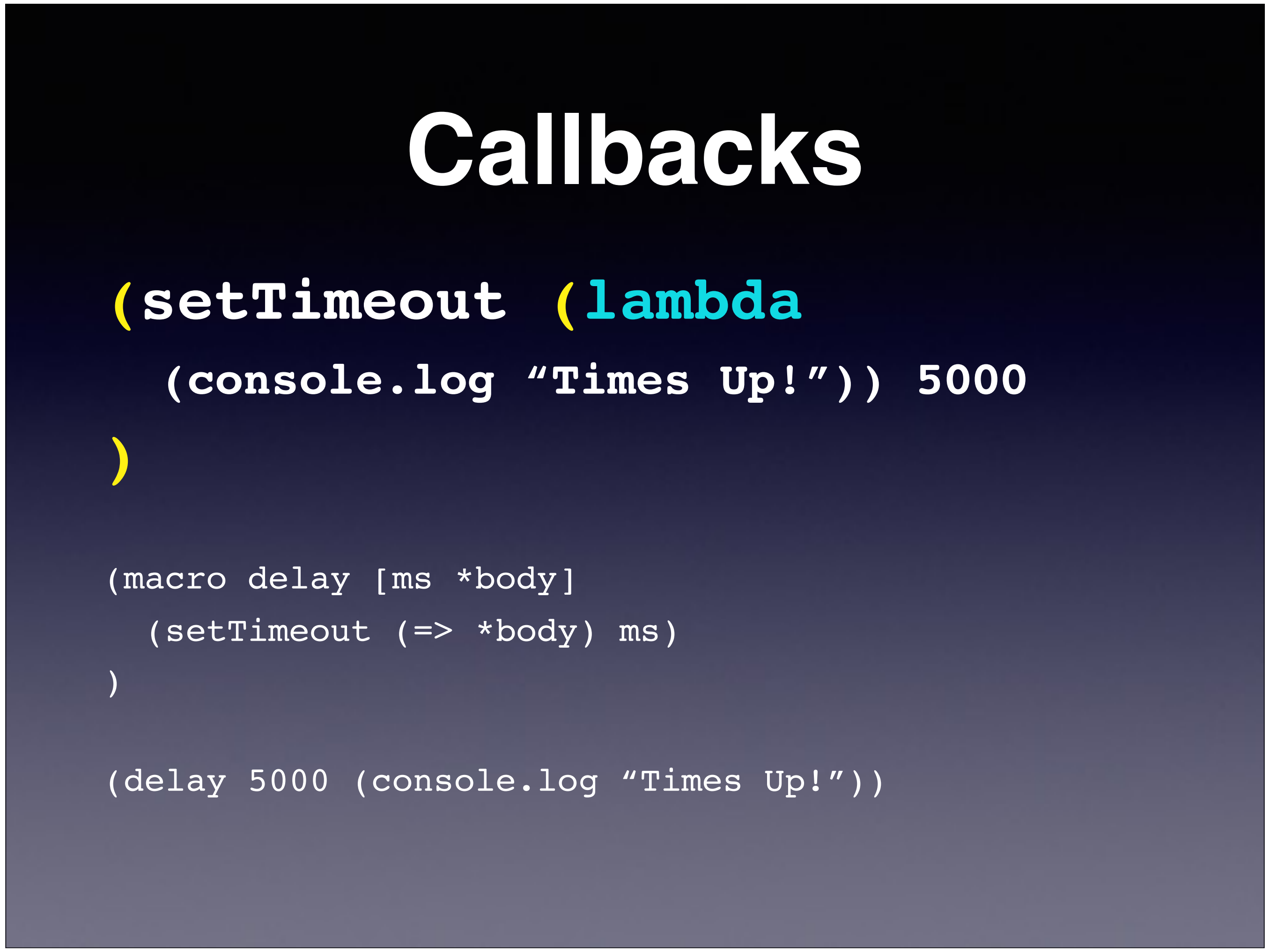
Callbacks

```
(setTimeout (lambda
  (console.log “Times Up!”) 5000
))

(macro delay [ms *body]
  (setTimeout (=> *body) ms)
)

(delay 5000 (console.log “Times Up!”))
```

Javascript traditionally used callbacks to support asynchronous operations.
The have the least overhead of any asynchronous functionality.
Lispz macros can make them a little more palatable.



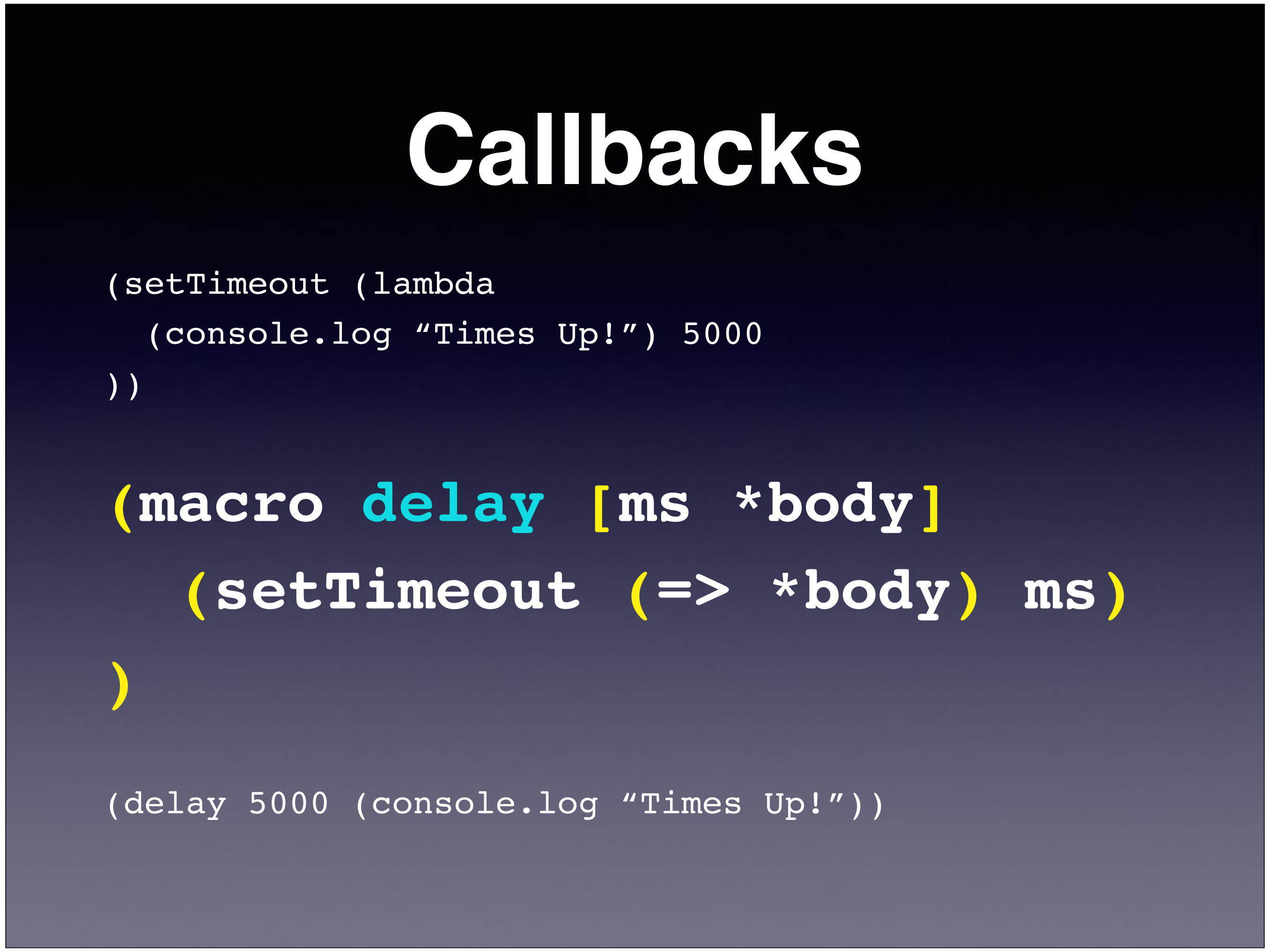
Callbacks

```
(setTimeout (lambda
  (console.log “Times Up!”)) 5000
)
```

```
(macro delay [ms *body]
  (setTimeout (=> *body) ms)
)

(delay 5000 (console.log “Times Up!”))
```

A classic callback translated directly from JavaScript - warts and all.



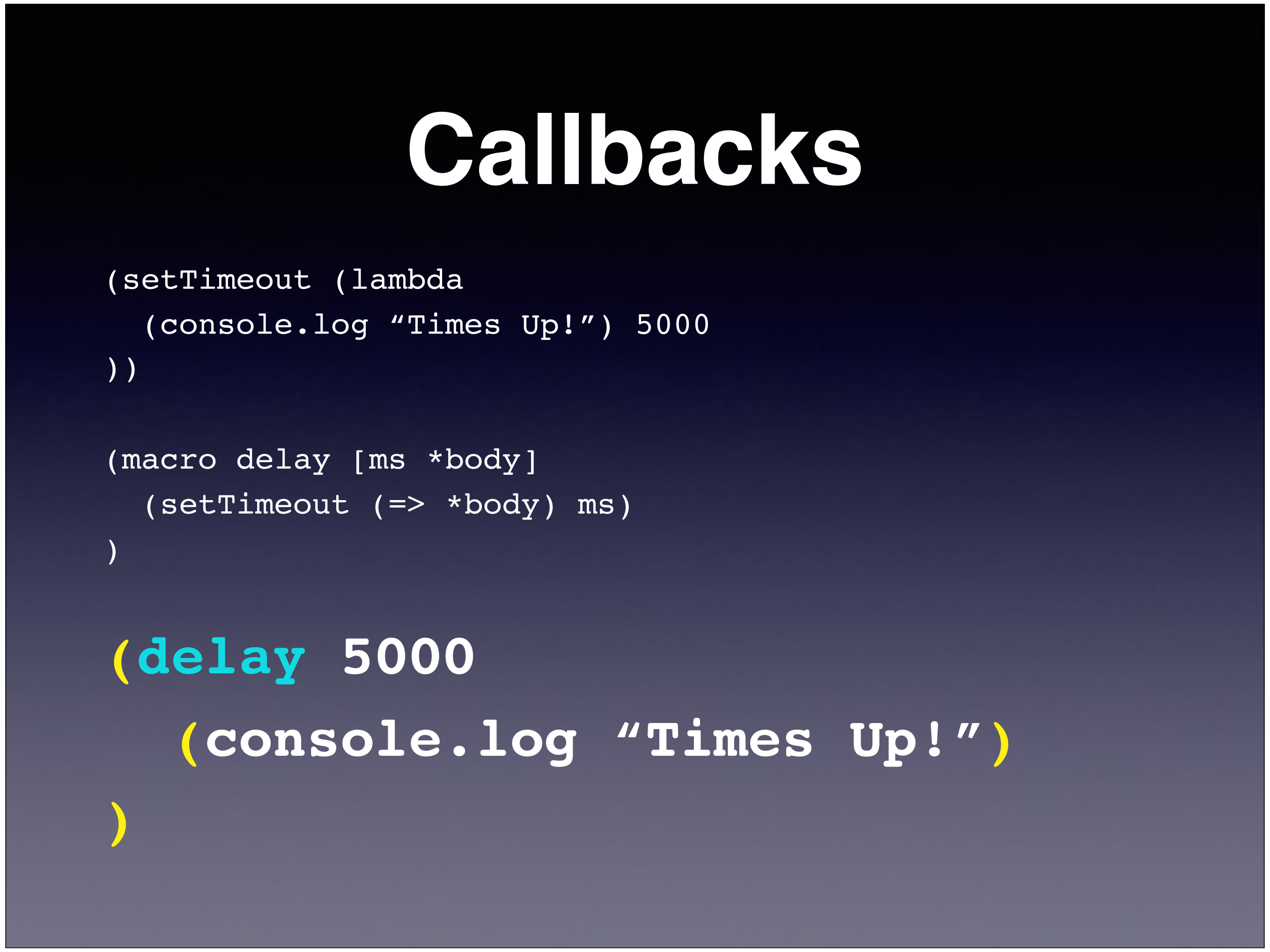
Callbacks

```
(setTimeout (lambda
  (console.log “Times Up!”) 5000
))

(macro delay [ms *body]
  (setTimeout (=> *body) ms)
)

(delay 5000 (console.log “Times Up!”))
```

Let’s use a Lispz macro to make it more palatable.



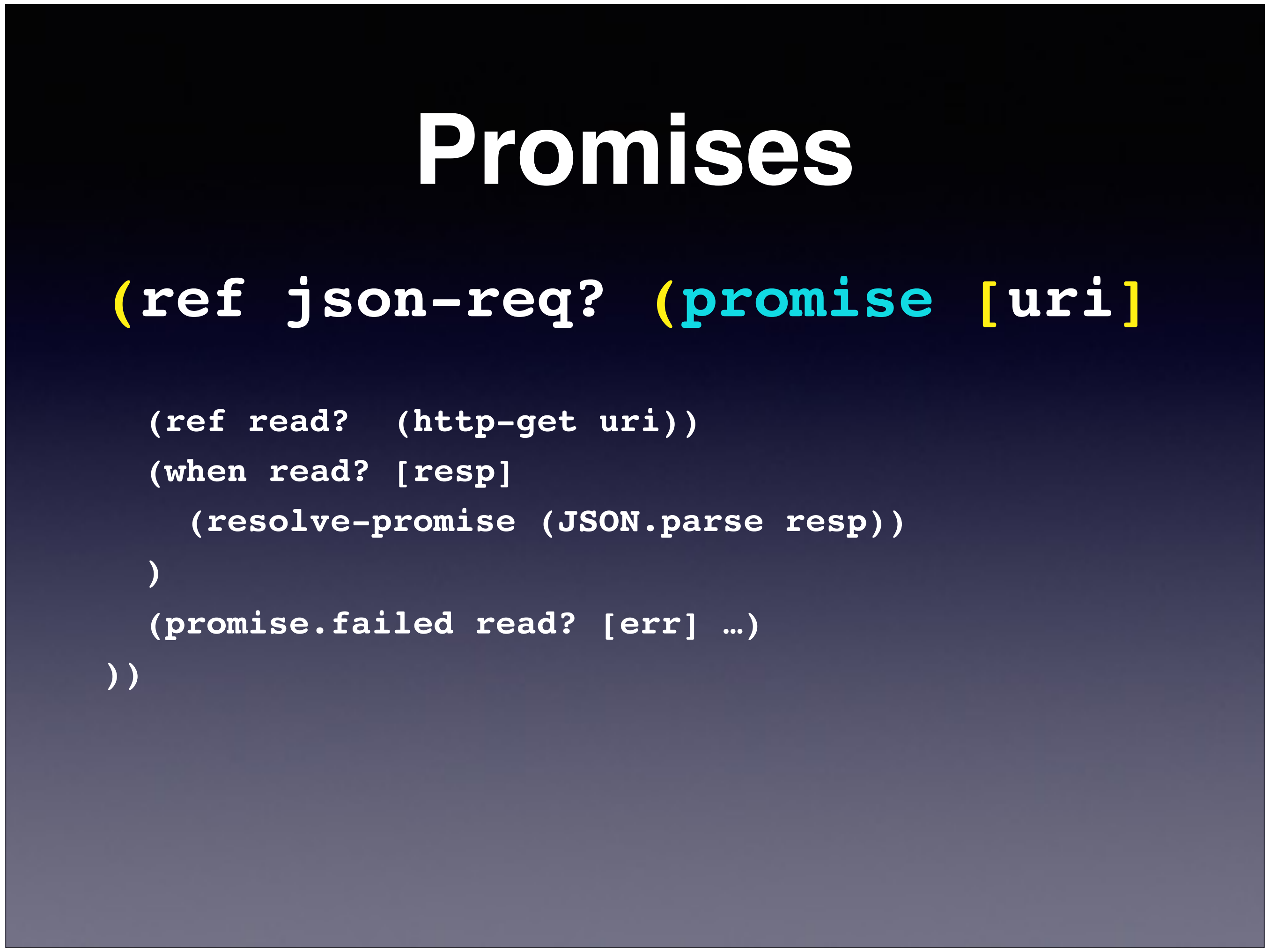
Callbacks

```
(setTimeout (lambda
  (console.log “Times Up!”) 5000
))

(macro delay [ms *body]
  (setTimeout (=> *body) ms)
)
```

```
(delay 5000
  (console.log “Times Up!”)
)
```

The result is more readable with less scaffolding.



Promises

```
(ref json-req? (promise [uri]

  (ref read?  (http-get uri))
  (when read? [resp]
    (resolve-promise (JSON.parse resp))
  )
  (promise.failed read? [err] …)
))
```

ES6 has promises, but Lispz can make them more palatable.
Replace lambda with promise and omit a return.

Promises

```
(ref json-request (promise [uri]

  (ref read? (http-get uri))

  (when read? [resp]
    (resolve-promise (JSON.parse resp))
  )
  (promise.failed read? [err] …)
))
```

Promises

```
(ref json-request (promise [uri]
  (ref read?  (http-get uri))

  (when read? [resp]

    (resolve-promise (JSON.parse resp))
  )
  (promise.failed read? [err] …)
))
```

Promises

```
(ref json-request (promise [uri]
  (ref read?  (http-get uri))
  (when read? [resp]

    (resolve-promise
      (JSON.parse resp)
    )

  )
  (promise.failed read? [err] …)
))
```

Promises

```
(ref json-request (promise [uri]
  (ref read?  (http-get uri))
  (when read? [resp]
    (resolve-promise (JSON.parse resp))
  )
  (promise.failed read? [err]
     ...
  )
))
```

Promises

```
(ref read
 (promise.callback [path]
   (github.read branch path callback)
  )
)
```

Promises

```
(ref file {
  read?: (promise.deferred [url]
    ## Only run on first when
  )
})
```

Promises

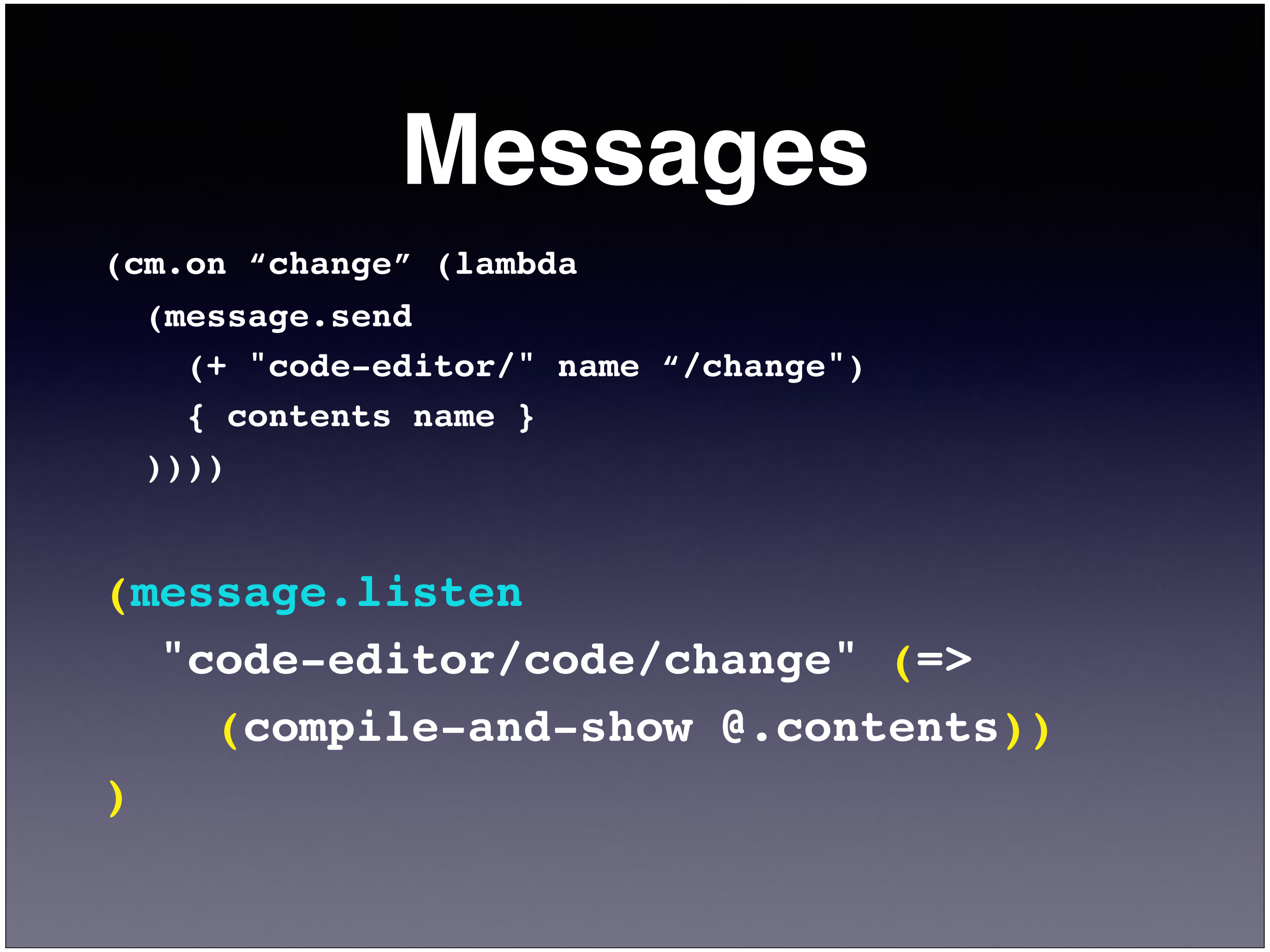
```
(promise.resolved false)

(promised result.contents)
```

Messages

```
(cm.on “change” (lambda
  (message.send
    (+ "code-editor/" name “/change")
    { contents name }
  )
)))
```

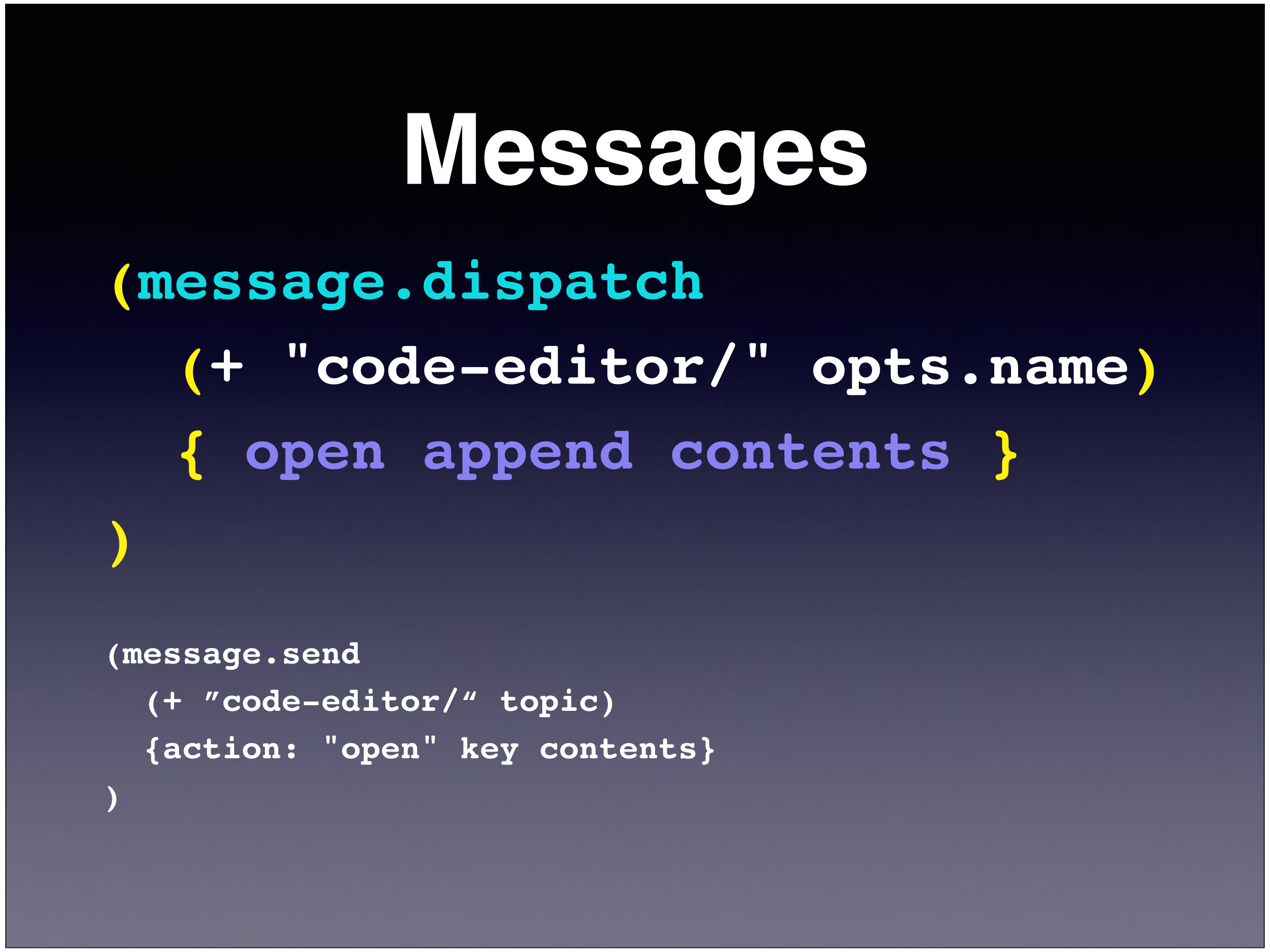
```
(message.listen "code-editor/code/change" (lambda
  (compile-and-show @.contents)))
```



Messages

```
(cm.on “change” (lambda
  (message.send
    (+ "code-editor/" name “/change")
    { contents name }
  ))))
```

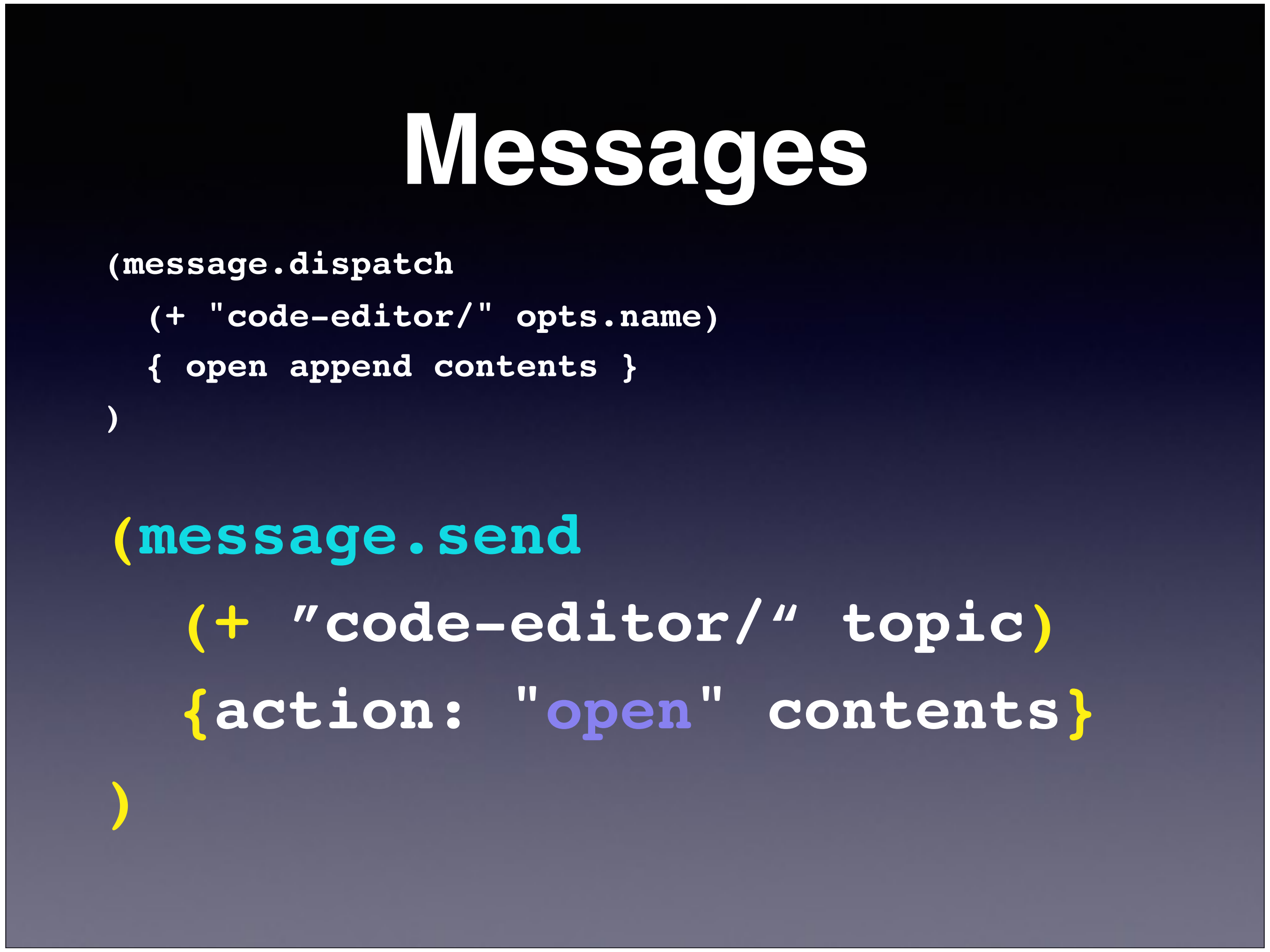
```
(message.listen
   "code-editor/code/change" (=>
      (compile-and-show @.contents))
)
```



Messages

```
(message.dispatch
  (+ "code-editor/" opts.name)
  { open append contents }
)
```

```
(message.send
  (+ ″code-editor/“ topic)
  {action: "open" key contents}
)
```



Messages

```
(message.dispatch
  (+ "code-editor/" opts.name)
  { open append contents }
)
```

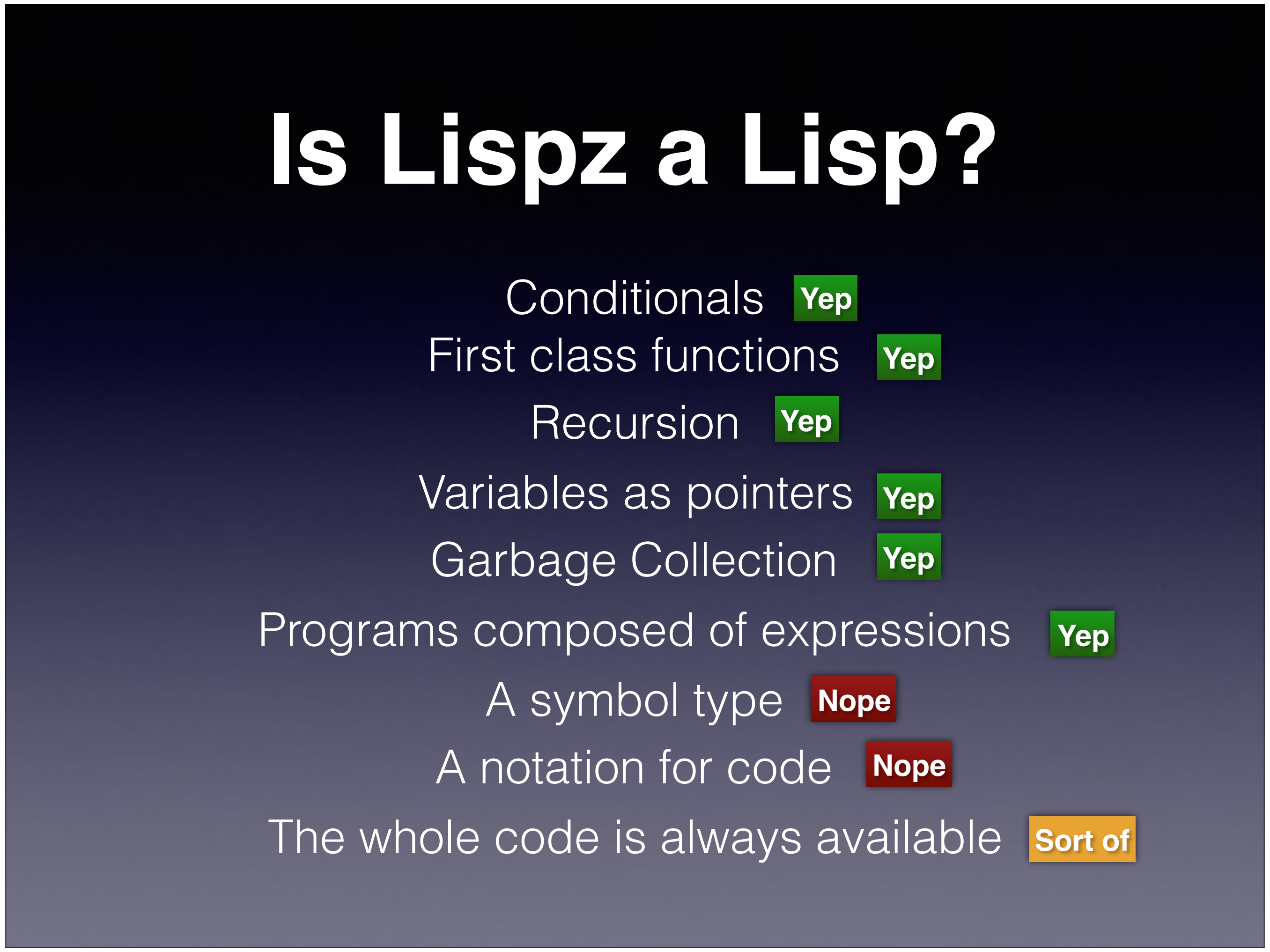
```
(message.send
  (+ ”code-editor/“ topic)
  {action: "open" contents}
)
```

Lispz - Now and the Future

Is Lispz a Lisp?

Is Lispz Functional?

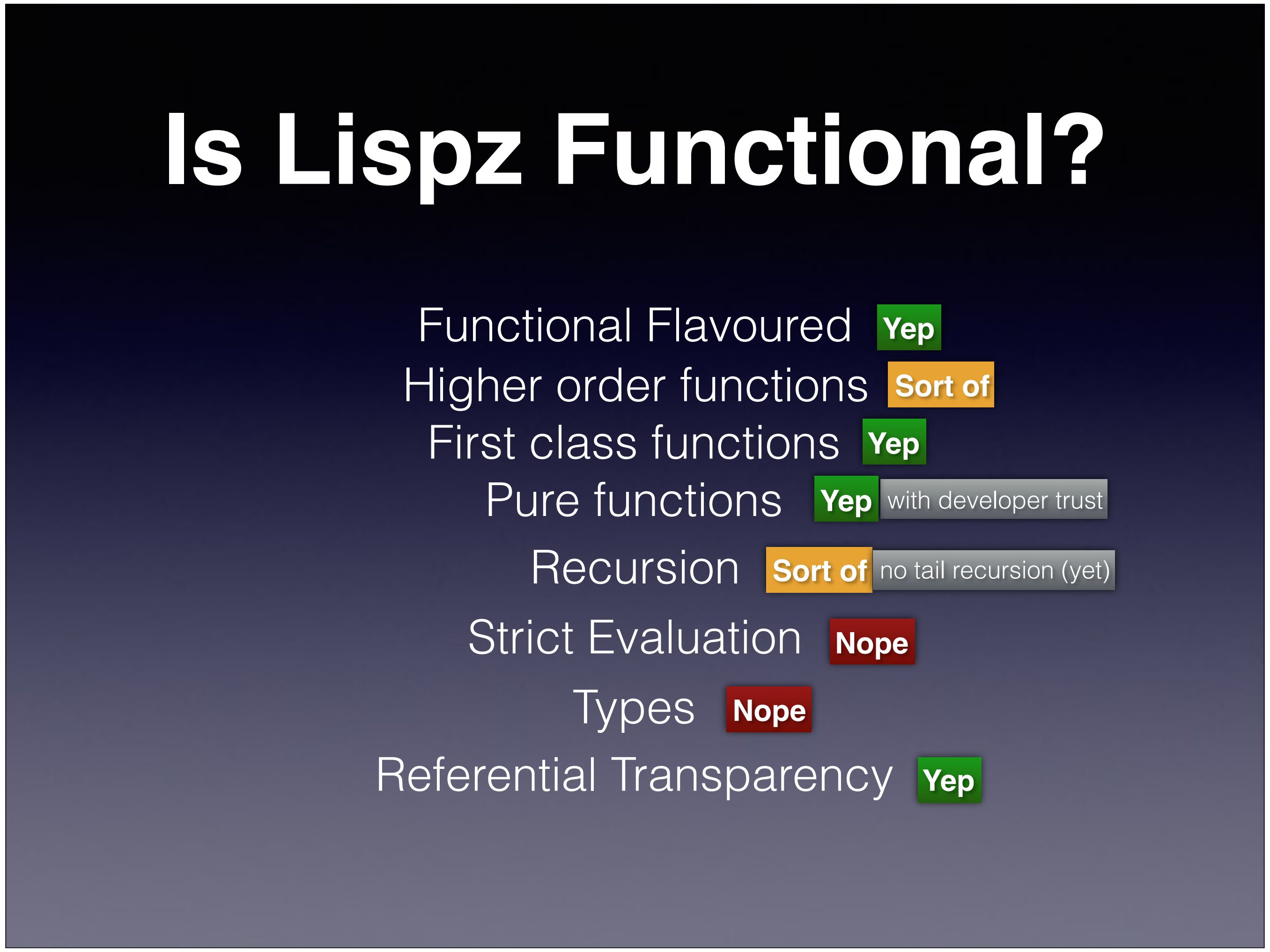
Adding Static Types



Is Lispz a Lisp?

Conditionals	Yep
First class functions	Yep
Recursion	Yep
Variables as pointers	Yep
Garbage Collection	Yep
Programs composed of expressions	Yep
A symbol type	Nope
A notation for code	Nope
The whole code is always available	Sort of

http://www.paulgraham.com/diff.html
When McCarthy designed Lisp in the late 1950s



Is Lispz Functional?

Functional Flavoured	Yep	
Higher order functions	Sort of	
First class functions	Yep	
Pure functions	Yep	with developer trust
Recursion	Sort of	no tail recursion (yet)
Strict Evaluation	Nope	
Types	Nope	
Referential Transparency	Yep	

Adding Static Types

Algebraic Data Type
Compile-time Type Checking
Pattern Matching

Compile to TypeScript

or

Roll-my-own

Lispz

http://lispz.marrington.net

https://github.com/paulmarrington/lispz

twitter: @paulmarrington