# A Trusted Mechananised Specification of the JavaScript Standard

Philippa Gardner

http://jscert.org

Imperial College London

UK Research Institute for Automatic Program Analysis and Verification, funded by GCHQ with EPSRC

# People

## IN IA

- Martin Bodin
- Arthur Charguéraud
- Alan Schmitt

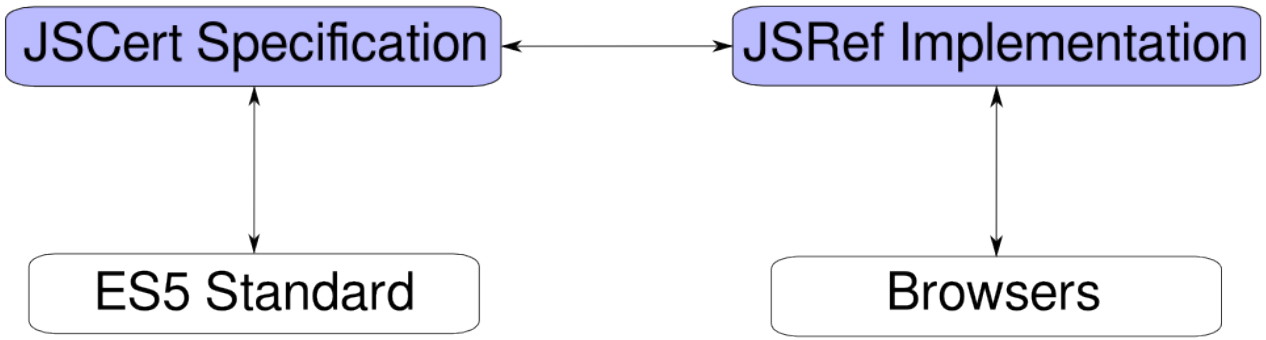
## Imperial College

- Daniele Filaretti
- Philippa Gardner
- Sergio Maffeis
- Daiva Naudžiūnienė
- Gareth Smith
- Adam Wright

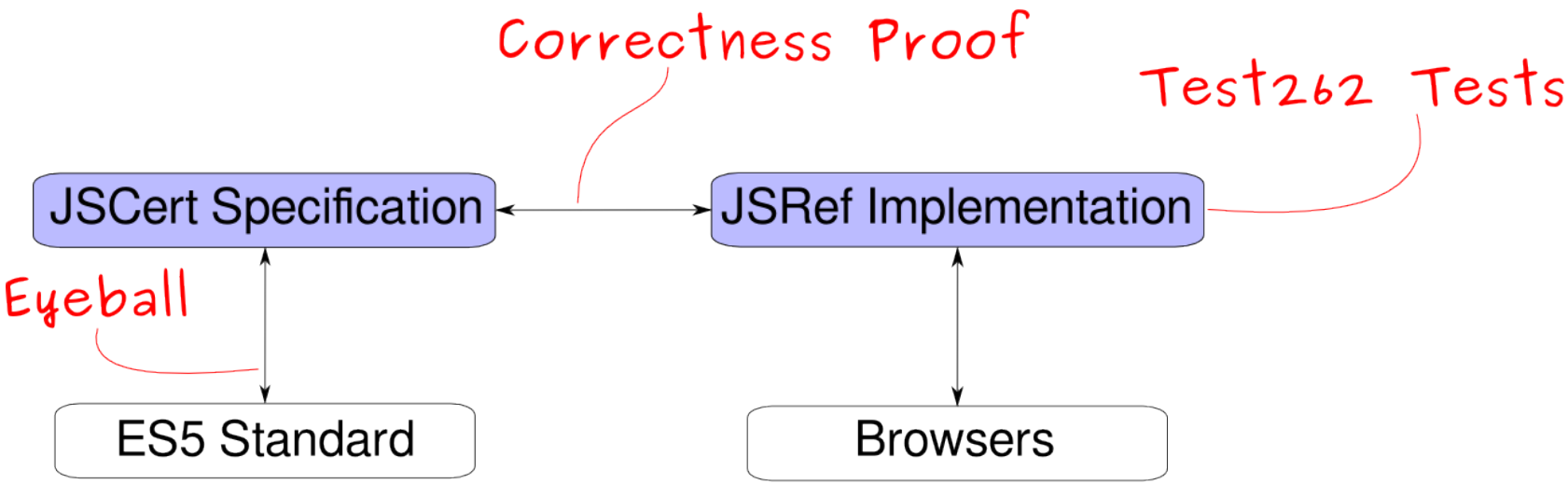
# JavaScript Specifications

- Initial Implementation (Netscape Navigator 1995)
- ECMAScript 3 standard (1999)
- Formal definition for the full language, justification via closeness to specification and proofs of safety properties (APLAS'08)
- ECMAScript 5 (2009)
- $\lambda_{JS}$: translation into a $\lambda$-calculus with references, justification via testing (ECOOP'10)
- Program logic for a core part of the language (POPL'12)
- $\lambda$S5: like $\lambda_{JS}$ for ES5 strict mode (DLS'12)
- $F^*$ to JavaScript via $\lambda_{JS}$, a full abstraction result (POPL'13)

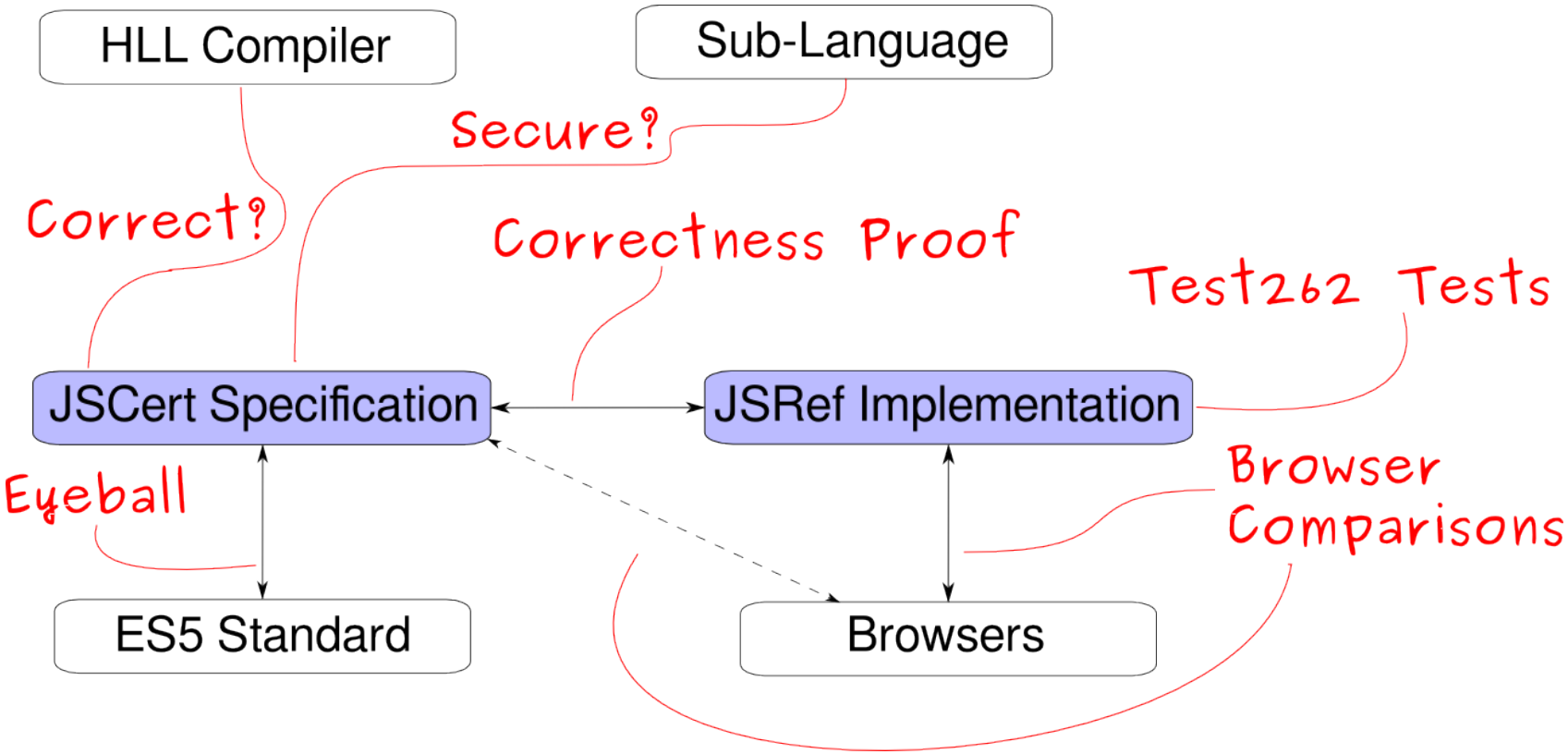
# The Talk


JSCert Specification
JSRef Implementation
ES5 Standard
Browsers

# The Talk: Trust



# The Talk: Applications



# JSCert

- A Coq specification of the ES5 standard (strict and non-strict)

# Coq

- An interactive formal proof assistant
- Developed mainly in several INRIAs, France
- Widely used
  - correct C compilation LeRoy INRIA, Appel Princeton
  - mechanised proof of the four-colour theorem Gonthier Microsoft
  - undergraduate teaching Pierce Penn, Morrisett Harvard

# Reliability of proof assistants.

Why should we trust proof assistants?
De Bruijn criterion for reliability (for all proof assistants):

> *Proof(term)s may be created by programs of arbitrary complexity, but there should be a very small and manually verifiable part of the program (kernel) to check those proof(term)s.*

One still may ask:

- What if the hardware is flawed?
  test on many different architectures
- What if the compiler used to build PA is flawed?
  compilers are about the most thoroughly tested pieces of software we have
- What if what you are formalizing is different that what you have in mind (and want to prove)?
  definitions are much easier (and shorter) than proofs; some experience required

So we will never have absolute certainty but it seems that PAs are as close as we can get

# JSCert Progress

Subset of JavaScript formalized so far:

- variables: scopes, prototype chains, assignment
- functions: declare, call, new
- objects: delete, access, get, set
- operators: unary and binary
- control flow: sequence, conditional, while loop, if, break, continue, switch, etc
- with construct, this construct
- exceptions: throw, try-catch-finally
- type conversions
- eval (parameterised by any trusted parser)
- libraries: Object, Function, Errors; some of Boolean, Number

Main missing features:

- control flow: for loops (interesting, on-going)
- parsing (affects eval)
- native libraries: Arrays, Regexp, Date, Math, ...

# JSCert

- A Coq specification of the ES5 standard (strict and non-strict)
- Eyeball-closeness to ES5 standard

# While

The production *IterationStatement*
`while` ( *Expression* ) *Statement* is evaluated as follows:

1. Let $V$ = `empty`.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If `To oolean(GetValue(`*exprRef*`))` is `false`, return (`normal`, $V$, `empty`).
   c. Let *stmt* be the result of evaluating *Statement*.
   d. If *stmt*.value is not empty, let V = *stmt*.value.
   e. If *stmt*.type is not `continue` || *stmt*.target is not in the current label set, then
      i. If *stmt*.type is `break` and *stmt*.target is in the current label set, then return (`normal`, $V$, `empty`).
      ii. If *stmt* is an abrupt completion, return *stmt*.

# Eyeball-closeness of While

## ES5

**12.6.2 The `while` Statement**

The production *IterationStatement* : **`while`** **`(`** *Expression* **`)`** *Statement* is evaluated as follows:

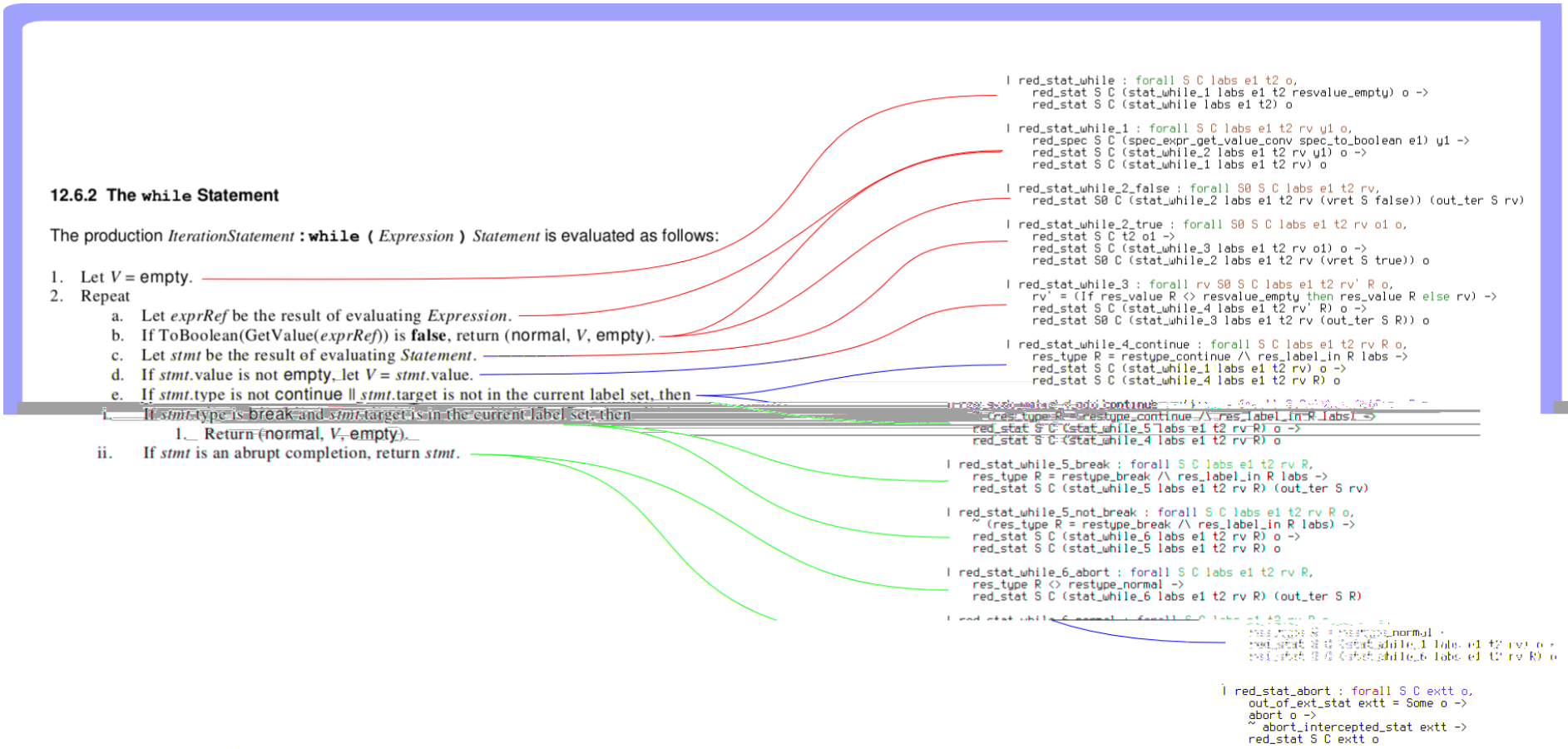
1. Let $V$ = empty.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, $V$, empty).
   c. Let *stmt* be the result of evaluating *Statement*.
   d. If *stmt*.value is not empty, let $V$ = *stmt*.value.
   e. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
      i. If *stmt*.type is break and *stmt*.target is in the current label set, then
         1. Return (normal, $V$, empty).
      ii. If *stmt* is an abrupt completion, return *stmt*.

## JSCert

```
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)

| red_stat_while_2_true : forall S0 S C labs e1 t2 rv o1 o,
    red_stat S C t2 o1 ->
    red_stat S C (stat_while_3 labs e1 t2 rv o1) o ->
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S true)) o

| red_stat_while_3 : forall rv S0 S C labs e1 t2 rv' R o,
    rv' = (If res_value R <> resvalue_empty then res_value R else rv) ->
    red_stat S C (stat_while_4 labs e1 t2 rv' R) o ->
    red_stat S0 C (stat_while_3 labs e1 t2 rv (out_ter S R)) o

| red_stat_while_4_continue : forall S C labs e1 t2 rv R o,
    res_type R = restype_continue /\ res_label_in R labs ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_4_not_continue : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_continue /\ res_label_in R labs) ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_5_break : forall S C labs e1 t2 rv R,
    res_type R = restype_break /\ res_label_in R labs ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) (out_ter S rv)

| red_stat_while_5_not_break : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_break /\ res_label_in R labs) ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o

| red_stat_while_6_abort : forall S C labs e1 t2 rv R,
    res_type R <> restype_normal ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) (out_ter S R)

| red_stat_while_6_normal : forall S C labs e1 t2 rv R o,
    res_type R = restype_normal ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o
```

```
| red_stat_abort : forall S C extt o,
    out_of_ext_stat extt = Some o ->
    abort o ->
    ~ abort_intercepted_stat extt ->
    red_stat S C extt o
```

# Eyeball-closeness of While



**12.6.2 The `while` Statement**

The production *IterationStatement* **`:while (`** *Expression* **`)`** *Statement* is evaluated as follows:

1. Let $V$ = empty.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, $V$, empty).
   c. Let *stmt* be the result of evaluating *Statement*.
   d. If *stmt*.value is not empty, let $V$ = *stmt*.value.
   e. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
      i. If *stmt*.type is break and *stmt*.target is in the current label set, then
         1. Return (normal, $V$, empty).
      ii. If *stmt* is an abrupt completion, return *stmt*.

```
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)

| red_stat_while_2_true : forall S0 S C labs e1 t2 rv o1 o,
    red_stat S C t2 o1 ->
    red_stat S C (stat_while_3 labs e1 t2 rv o1) o ->
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S true)) o

| red_stat_while_3 : forall rv S0 S C labs e1 t2 rv' R o,
    rv' = (If res_value R <> resvalue_empty then res_value R else rv) ->
    red_stat S C (stat_while_4 labs e1 t2 rv' R) o ->
    red_stat S0 C (stat_while_3 labs e1 t2 rv (out_ter S R)) o

| red_stat_while_4_continue : forall S C labs e1 t2 rv R o,
    res_type R = restype_continue /\ res_label_in R labs ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_5_break : forall S C labs e1 t2 rv R,
    res_type R = restype_break /\ res_label_in R labs ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) (out_ter S rv)

| red_stat_while_5_not_break : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_break /\ res_label_in R labs) ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o

| red_stat_while_6_abort : forall S C labs e1 t2 rv R,
    res_type R <> restype_normal ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) (out_ter S R)

| red_stat_abort : forall S C extt o,
    out_of_ext_stat extt = Some o ->
    abort o ->
    ~ abort_intercepted_stat extt ->
    red_stat S C extt o
```

# Eyeball-closeness of While

## ES5

**12.6.2 The `while` Statement**

The production *IterationStatement* : **`while`** **`(`** *Expression* **`)`** *Statement* is evaluated as follows:

1. Let $V$ = empty.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, $V$, empty).
   c. Let *stmt* be the result of evaluating *Statement*.
   d. If *stmt*.value is not empty, let $V$ = *stmt*.value.
   e. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
      i. If *stmt*.type is break and *stmt*.target is in the current label set, then
         1. Return (normal, $V$, empty).
      ii. If *stmt* is an abrupt completion, return *stmt*.

## JSCert

```
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)

| red_stat_while_2_true : forall S0 S C labs e1 t2 rv o1 o,
    red_stat S C t2 o1 ->
    red_stat S C (stat_while_3 labs e1 t2 rv o1) o ->
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S true)) o

| red_stat_while_3 : forall rv S0 S C labs e1 t2 rv' R o,
    rv' = (If res_value R <> resvalue_empty then res_value R else rv) ->
    red_stat S C (stat_while_4 labs e1 t2 rv' R) o ->
    red_stat S0 C (stat_while_3 labs e1 t2 rv (out_ter S R)) o

| red_stat_while_4_continue : forall S C labs e1 t2 rv R o,
    res_type R = restype_continue /\ res_label_in R labs ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_4_not_continue : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_continue /\ res_label_in R labs) ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_5_break : forall S C labs e1 t2 rv R,
    res_type R = restype_break /\ res_label_in R labs ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) (out_ter S rv)

| red_stat_while_5_not_break : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_break /\ res_label_in R labs) ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o

| red_stat_while_6_abort : forall S C labs e1 t2 rv R,
    res_type R <> restype_normal ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) (out_ter S R)

| red_stat_while_6_normal : forall S C labs e1 t2 rv R o,
    res_type R = restype_normal ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o
```

```
| red_stat_abort : forall S C extt o,
    out_of_ext_stat extt = Some o ->
    abort o ->
    ~ abort_intercepted_stat extt ->
    red_stat S C extt o
```

# Eyeball-closeness of While

1. Let $V$ = empty.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, $V$, empty).

```
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)
```

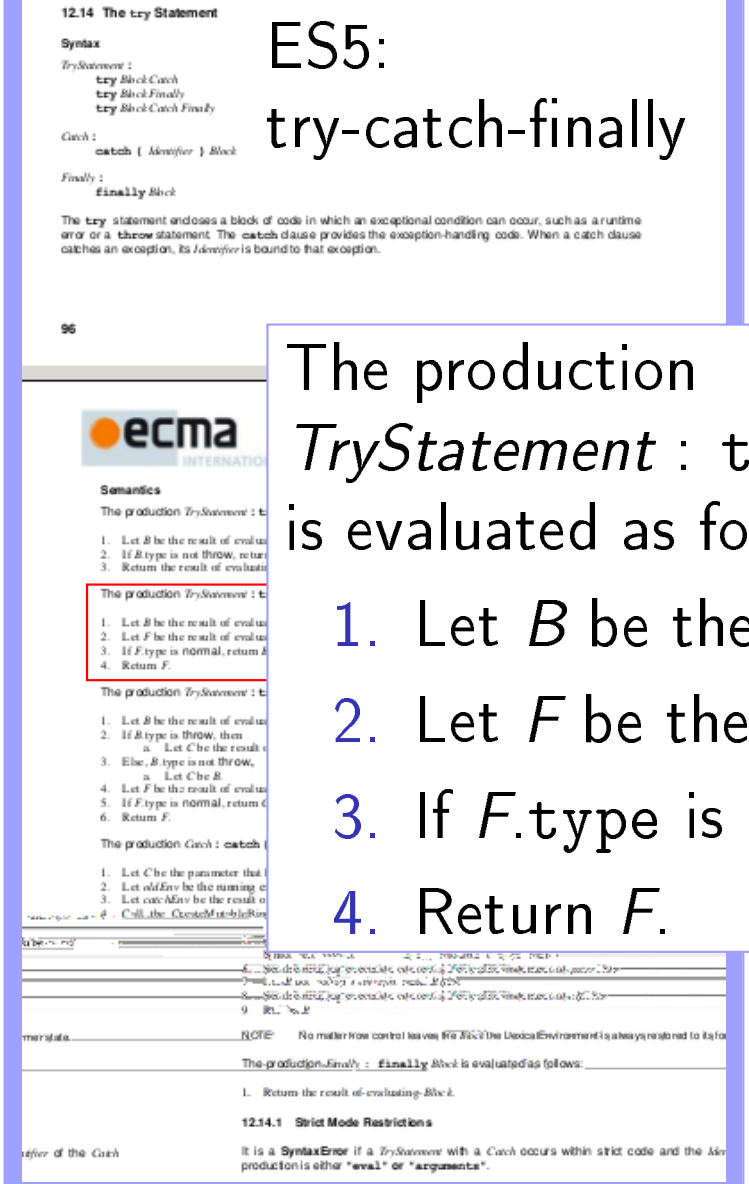
. . . using pretty-big step semantics (Charguéraud).

# JSCert

- A Coq specification of the ES5 standard (strict and non-strict)
- Eyeball-closeness to ES5 standard
- Comparisons with the browser implementations

# Implementation Comparison for Try

ES5:
try-catch-finally


12.14 The try Statement

Syntax

*TryStatement* :
try *Block Catch*
try *Block Finally*
try *Block Catch Finally*

*Catch* :
catch ( *Identifier* ) *Block*

*Finally* :
finally *Block*

The try statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a throw statement. The catch clause provides the exception-handling code. When a catch clause catches an exception, its *Identifier* is bound to that exception.

96

ecma

Semantics

The production *TryStatement* : t

1. Let *B* be the result of evalu
2. If *B*.type is not throw, retur
3. Return the result of evaluati

The production *TryStatement* : t

1. Let *B* be the result of evalu
2. Let *F* be the result of evalu
3. If *F*.type is normal, return *B*
4. Return *F*.

The production *TryStatement* : t

1. Let *B* be the result of evalu
2. If *B*.type is throw, then
   a. Let *C* be the result
3. Else, *B*.type is not throw,
   a. Let *C* be *B*.
4. Let *F* be the result of evalu
5. If *F*.type is normal, return
6. Return *F*.

The production *Catch* : catch (

1. Let *C* be the parameter that
2. Let *oldEnv* be the running e
3. Let *catchEnv* be the result o
4. Call the CreateMutableBin

NOTE No matter how control leaves the *Block* the LexicalEnvironment is always restored to its for

The production *Finally* : finally *Block* is evaluated as follows:

1. Return the result of evaluating *Block*.

12.14.1 Strict Mode Restrictions

It is a SyntaxError if a *TryStatement* with a *Catch* occurs within strict code and the *Ident* production is either "eval" or "arguments".

JSCert

```
1 o,

irrelevant) o1 ->

    red_stat S C (stat_try_4 R None) (out_ter S R)

| red_stat_try_4_finally : forall S C R t1 o o1,
    red_stat S C t1 o1 ->
    red_stat S C (stat_try_5 R o1) o ->
    red_stat S C (stat_try_4 R (Some t1)) o

| red_stat_try_5_finally_result : forall S0 S C R rv,
    red_stat S0 C (stat_try_5 R (out_ter S rv)) (out_ter S R)
```

The production
*TryStatement* : `try` *Block Finally*
is evaluated as follows:

1. Let *B* be the result of evaluating *Block*.
2. Let *F* be the result of evaluating *Finally*.
3. If *F*.`type` is normal, return *B*.
4. Return *F*.

# What do Browsers do?

```
try { "try" } finally { "finally" }
```

ES5, Opera: (normal, "try")

Chrome, FF, IE, Safari: (normal, "finally")

```
try { "try" ; break } finally { "finally" }
```

ES5, Opera, Safari: (break, "try")

Chrome, FF, IE: (break, "finally")

```
try { "try" } finally { "finally" ; break }
```

ES5, Chrome, FF, IE, Safari: (break, "finally")

Opera: (break, "try")

# What do Browsers do?

```
while(true) {
    try { "try" ; break }
    finally { "finally" }
}
```

Chrome: (break, "finally")

```
while(true) {
    try { "try" ; break }
    finally { "finally" }
    y = "done"
}
```

Chrome: (break, "try")

# What do Browsers do?

```
while(true) {
    try { "try" ; break }
    finally { "finally" }
    if(true) {2} else {var x = 3}
}
```

Chrome: (break, "finally")

```
while(true) {
    try { "try" ; break }
    finally { "finally" }
    if(true) {2} else {3}
}
```

Chrome: (break, "try")

# JSCert

- A Coq specification of the ES5 standard (strict and non-strict)
- Eyeball-closeness to ES5 standard
- Comparisons with the browser implementations
- Rigorous assessment of ES5: only issues with data attributes and for-in.

# ES5 assessment using For

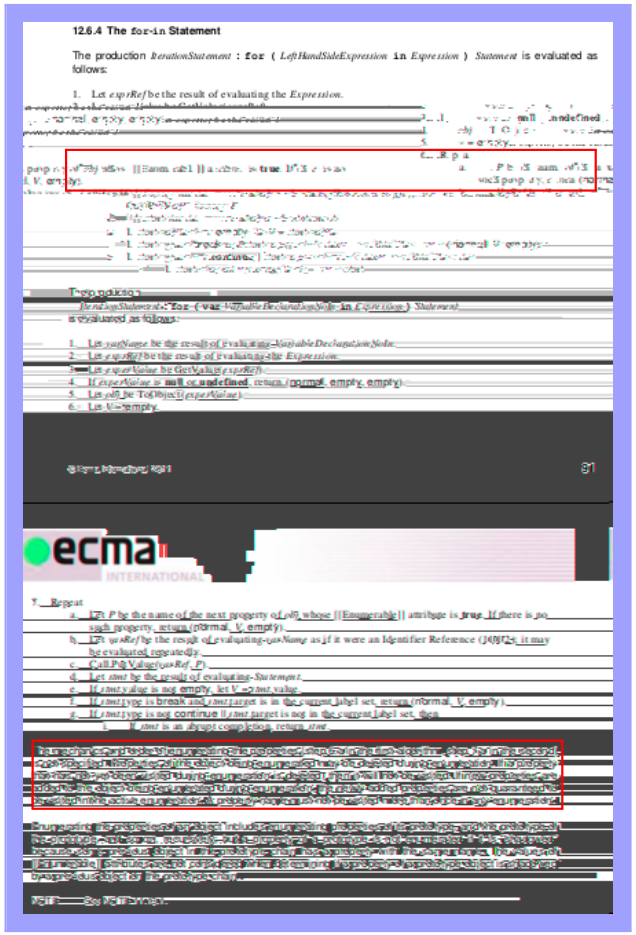
one of the darkest corners of the ES spec

intentionally vague

I wouldn't even assume that for-in semantics is deterministic for any given VM – it can change depending on dynamic optimisations and representation changes.

— helpful people on es-discuss

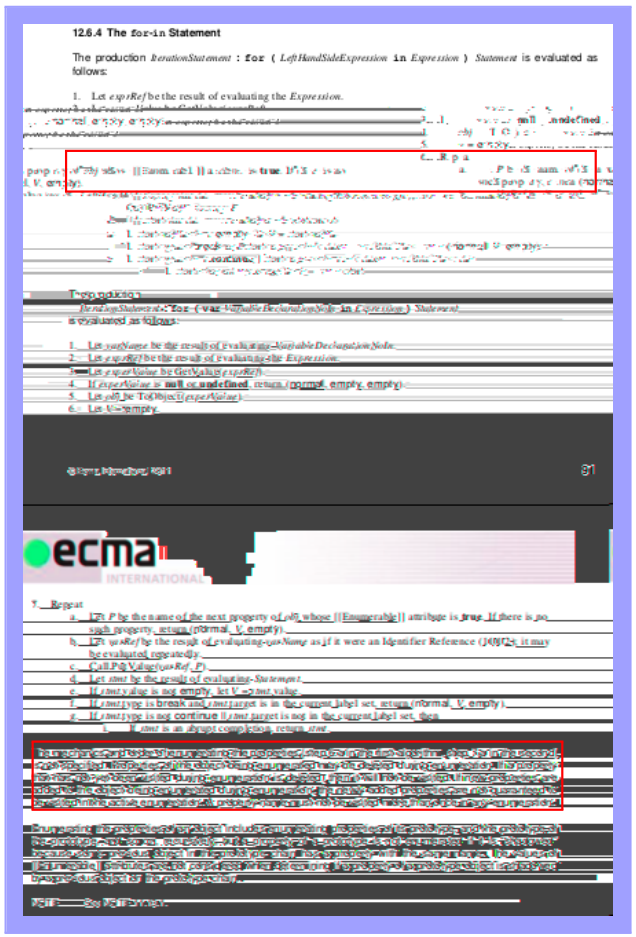
# ES5 assessment using For



6. Repeat
    a. Let P be the name of the next property of obj whose [[Enumerable]] attribute is true. If there is no such property, return (normal, V, empty).

The mechanics and order of enumerating the properties (step 6.a in the first algorithm, step 7.a in the second) is not specified. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited...

# ES5 assessment using For




6. Repeat
    a. Let P be the name of the next property of obj whose [[Enumerable]] attribute is true. If there is no such property, return (normal, V, empty).

...If new properties are added to the object being enumerated during enumeration, the newly added properties are not guaranteed to be visited in the active enumeration. A property name must not be visited more than once in any enumeration.

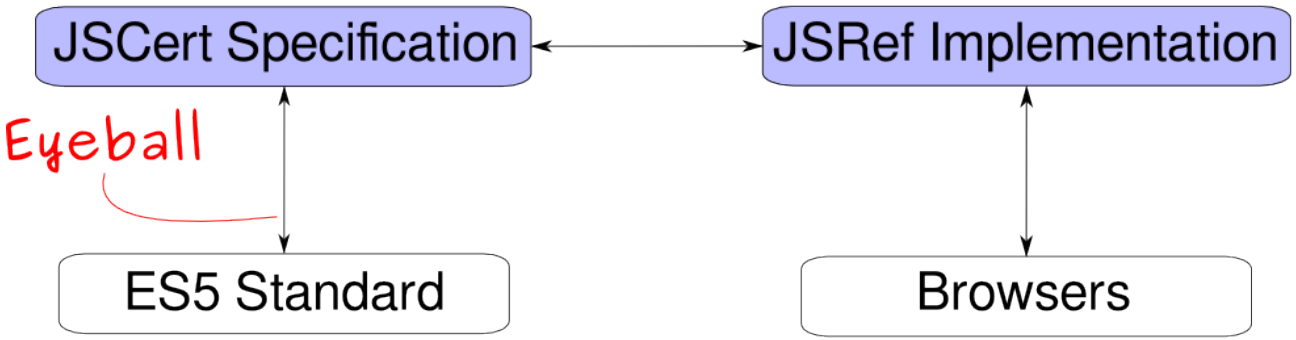
# For-related bugs

- Firefox bug 862771 (Unconfirmed)
- V8 Issue 705 (Miller 2010: "New")
- Webkit bug 38970 (Miller 2010: "New")
- ES6 Bugs 1442, 1443, 1444 (Fixed, Confirmed, Confirmed)
- Test262 Bug 1445 (In Progress)

# JSCert

- A Coq specification of the ES5 standard (strict and non-strict)
- Eyeball-closeness to ES5 standard
- Comparisons with the browser implementations
- Rigorous assessment of ES5: bugs reported in Firefox, Chrome, Safari, ES6 draft standard, Test262
- Future: prove safety properties
  - No well-formed program 'gets stuck'
  - The heap is always well-formed

# The Talk


JSCert Specification
JSRef Implementation
Eyeball
ES5 Standard
Browsers

# JSRef

- An executable reference interpreter in OCaml

JSCert: while

```
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)

| red_stat_while_2_true : forall S0 S C labs e1 t2 rv o1 o,
    red_stat S C t2 o1 ->
    red_stat S C (stat_while_3 labs e1 t2 rv o1) o ->
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S true)) o

| red_stat_while_3 : forall rv S0 S C labs e1 t2 rv' R o,
    rv' = (If res_value R <> resvalue_empty then res_value R else rv) ->
    red_stat S C (stat_while_4 labs e1 t2 rv' R) o ->
    red_stat S0 C (stat_while_3 labs e1 t2 rv (out_ter S R)) o

| red_stat_while_4_continue : forall S C labs e1 t2 rv R o,
    res_type R = restype_continue /\ res_label_in R labs ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_4_not_continue : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_continue /\ res_label_in R labs) ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_5_break : forall S C labs e1 t2 rv R,
    res_type R = restype_break /\ res_label_in R labs ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) (out_ter S rv)

| red_stat_while_5_not_break : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_break /\ res_label_in R labs) ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o

| red_stat_while_6_abort : forall S C labs e1 t2 rv R,
    res_type R <> restype_normal ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) (out_ter S R)

| red_stat_while_6_normal : forall S C labs e1 t2 rv R o,
    res_type R = restype_normal ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o


| red_stat_abort : forall S C extt o,
    out_of_ext_stat extt = Some o ->
    abort o ->
    ~ abort_intercepted_stat extt ->
    red_stat S C extt o
```

JSRef Coq: while

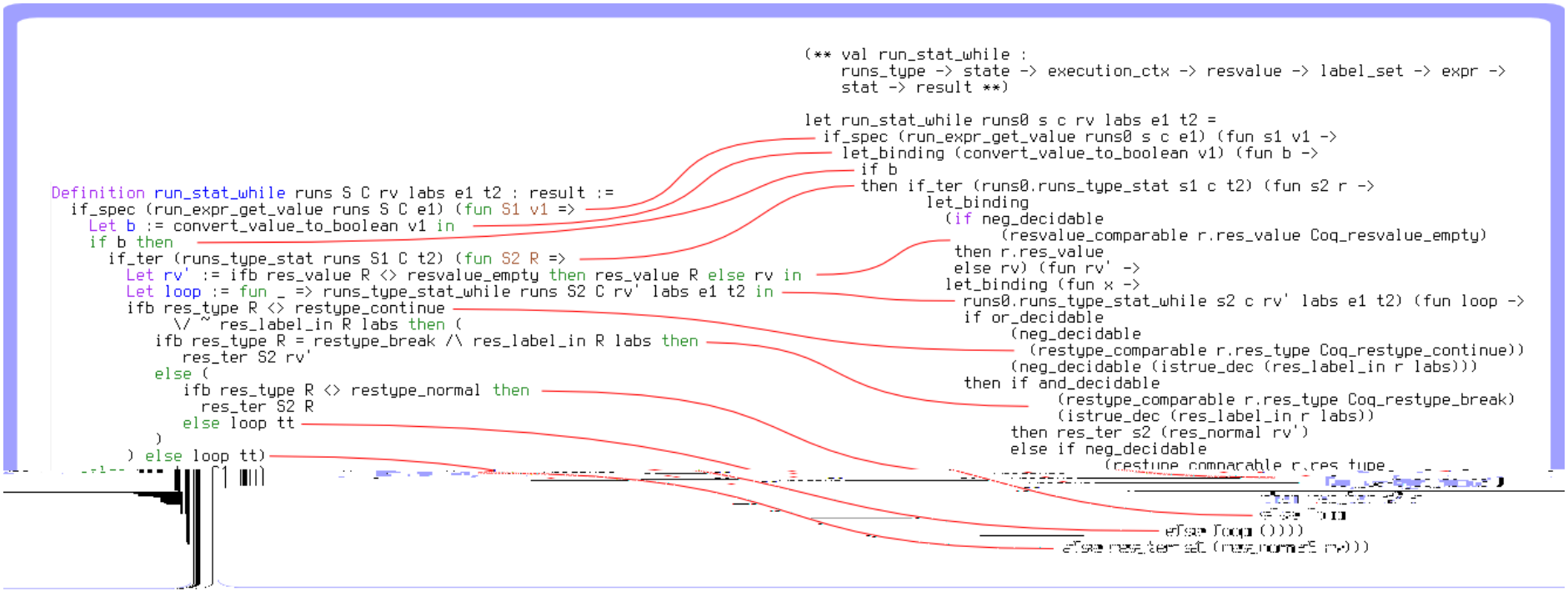
```
Definition run_stat_while runs S C rv labs e1 t2 : result :=
  if_spec (run_expr_get_value runs S C e1) (fun S1 v1 =>
    Let b := convert_value_to_boolean v1 in
    if b then
      if_ter (runs_type_stat runs S1 C t2) (fun S2 R =>
        Let rv' := ifb res_value R <> resvalue_empty then res_value R else rv in
        Let loop := fun _ => runs_type_stat_while runs S2 C rv' labs e1 t2 in
        ifb res_type R <> restype_continue
            \/ ~ res_label_in R labs then (
          ifb res_type R = restype_break /\ res_label_in R labs then
            res_ter S2 rv'
          else (
            ifb res_type R <> restype_normal then
              res_ter S2 R
            else loop tt
          )
        ) else loop tt)
    else res_ter S1 rv).
```

# JSRef

- An executable reference interpreter in OCaml

JSRef Coq: while

JSRef OCaml: while



```
Definition run_stat_while runs S C rv labs e1 t2 : result :=
  if_spec (run_expr_get_value runs S C e1) (fun S1 v1 =>
    Let b := convert_value_to_boolean v1 in
    if b then
      if_ter (runs_type_stat runs S1 C t2) (fun S2 R =>
        Let rv' := ifb res_value R <> resvalue_empty then res_value R else rv in
        Let loop := fun _ => runs_type_stat_while runs S2 C rv' labs e1 t2 in
        ifb res_type R <> restype_continue
             \/ ~ res_label_in R labs then (
           ifb res_type R = restype_break /\ res_label_in R labs then
              res_ter S2 rv'
           else (
              ifb res_type R <> restype_normal then
                res_ter S2 R
              else loop tt
           )
        ) else loop tt)
```

```
(** val run_stat_while :
    runs_type -> state -> execution_ctx -> resvalue -> label_set -> expr ->
    stat -> result **)

let run_stat_while runs0 s c rv labs e1 t2 =
  if_spec (run_expr_get_value runs0 s c e1) (fun s1 v1 ->
  let_binding (convert_value_to_boolean v1) (fun b ->
  if b
  then if_ter (runs0.runs_type_stat s1 c t2) (fun s2 r ->
         let_binding
           (if neg_decidable
               (resvalue_comparable r.res_value Coq_resvalue_empty)
            then r.res_value
            else rv) (fun rv' ->
           let_binding (fun x ->
             runs0.runs_type_stat_while s2 c rv' labs e1 t2) (fun loop ->
             if or_decidable
                  (neg_decidable
                     (restype_comparable r.res_type Coq_restype_continue))
                  (neg_decidable (istrue_dec (res_label_in r labs)))
             then if and_decidable
                       (restype_comparable r.res_type Coq_restype_break)
                       (istrue_dec (res_label_in r labs))
                  then res_ter s2 (res_normal rv')
                  else if neg_decidable
                            (restype_comparable r.res_type
```

# JSRef

- An executable reference interpreter in OCaml
- A Coq proof that JSRef Coq is correct with respect to JSCert

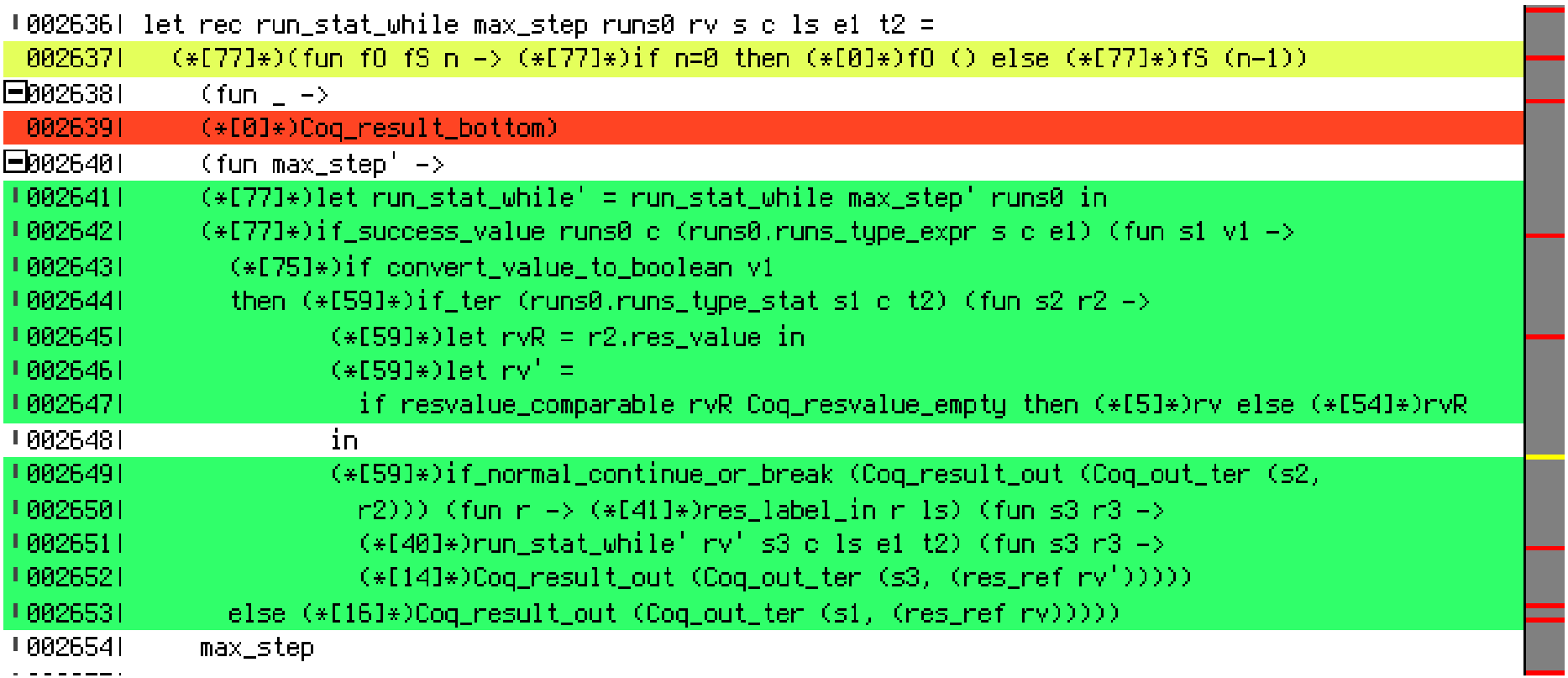
# JSRef

- An executable reference interpreter in OCaml
- A Coq proof that JSRef Coq is correct with respect to JSCert
- Tested using Test262 test suite

# Tested using Test262 test suite

- Pass 1993 interesting tests.
- Fail 195 interesting tests, corresponding to 35 bugs (on-going)
- Ignore:
  - things not implemented: Math, Date, Array, JSON, parse[Int|Float], [en|de]codeURI[Component], Regexp, String, ...
  - Buggy tests (Bug 1600)
  - Submitted bugs 1445, 1450, 1600 (In Progress, In Progress, Confirmed)

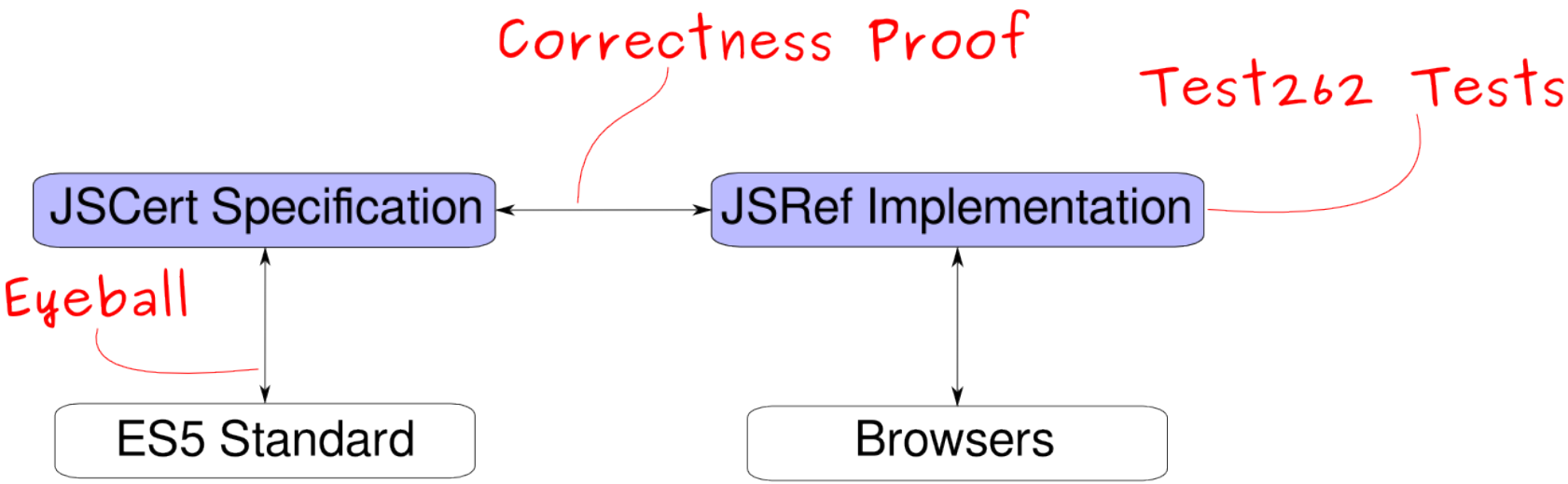
# Coverage for While using Bisect



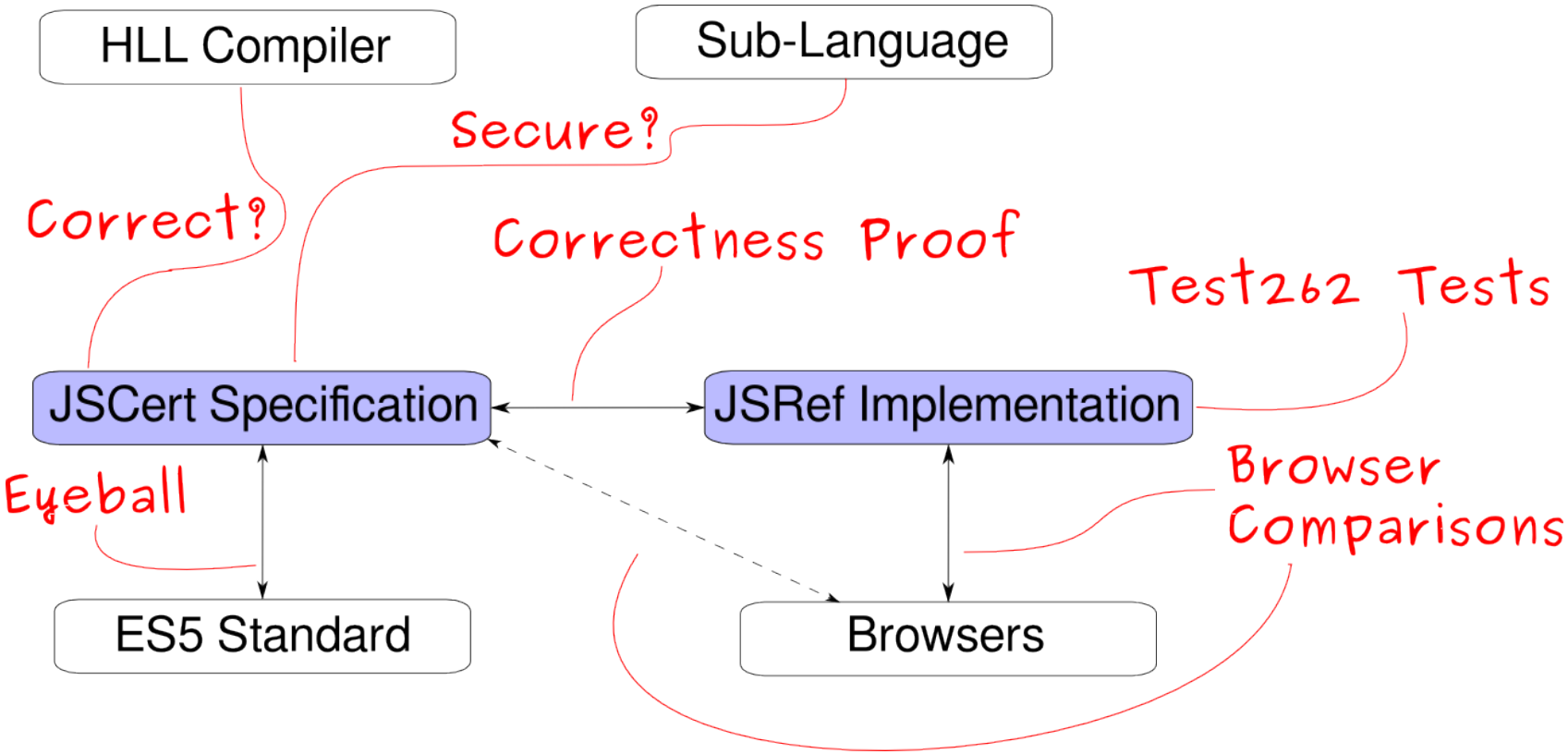
# JSRef

- An executable reference interpreter in OCaml
- A Coq proof that JSRef Coq is correct with respect to JSCert
- Tested using Test262 test suite
- Future: test with Firefox test suite, develop Firefox clone
- Future: more complete test coverage for ES5.

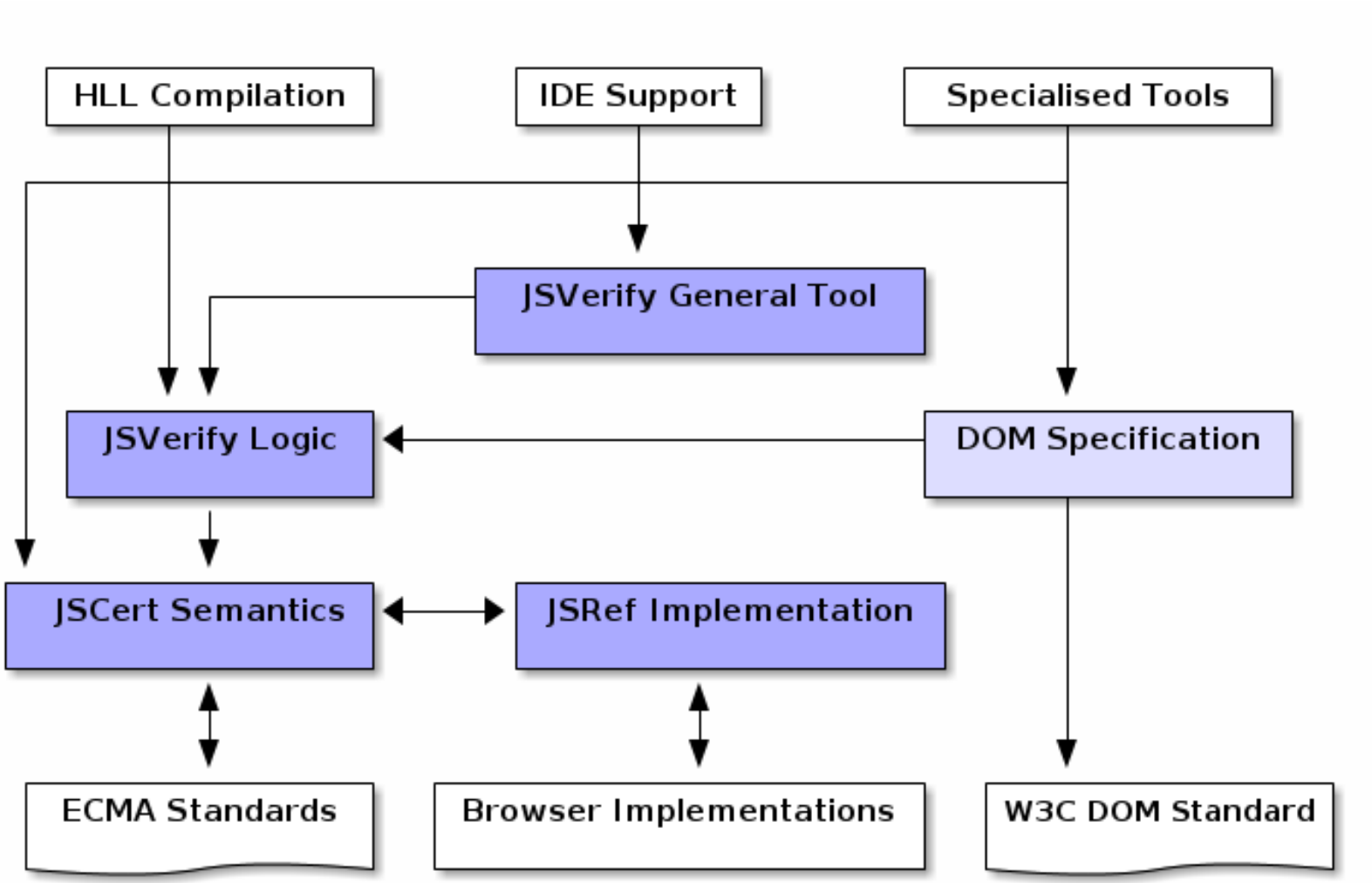
# The Talk


Correctness Proof
Test262 Tests
JSCert Specification
JSRef Implementation
Eyeball
ES5 Standard
Browsers

# Future


HLL Compiler
Sub-Language
Secure?
Correct?
Correctness Proof
Test262 Tests
JSCert Specification
JSRef Implementation
Eyeball
Browser
Comparisons
ES5 Standard
Browsers

# Future


HLL Compilation
IDE Support
Specialised Tools
JSVerify General Tool
JSVerify Logic
DOM Specification
JSCert Semantics
JSRef Implementation
ECMA Standards
Browser Implementations
W3C DOM Standard