building effective language parsers for penetration testers

columbus owasp
28 march 2019

john toterhi
cetfor@gmail.com

i like breaking software. i've done it for these organizations.

str SYSTEMS & TECHNOLOGY RESEARCH

FINITE STATE

BATTELLE

U.S. AIR FORCE

NATIONAL AIR & SPACE INTELLIGENCE CENTER

analyzing source code for bugs is hard for many reasons.

lexical ambiguity

variable state considerations

grammar complexity

variable scope

knowledge of source / sinks

effects of sanitization

before we jump into code parsing, let’s explore some fundamentals with english.

What does this sentence mean?

“I shot the elephant in my pajamas.”

What does this sentence mean?

“I shot the elephant in my pajamas.”

1. The shooter wore pajamas while shooting the elephant.

2. The elephant was *wearing* the shooter’s pajamas.

What does this sentence mean?

“He drove down the street in the car.”

What does this sentence mean?

“He drove down the street in the car.”

1. He was driving a car on the street.

2. The street he drove down was actually *in his car*.

What does this sentence mean?

“The complex houses married and single soldiers and their families.’”

What does this sentence mean?

“The complex houses married and single soldiers and their families.”

1. The housing complex contains both married and single soldiers, as well as their families.

What does this sentence mean?

“Buffalo buffalo buffalo buffalo buffalo buffalo buffalo buffalo.”

What does this sentence mean?

“Buffalo buffalo buffalo buffalo buffalo buffalo buffalo buffalo.”

1. Bison from Buffalo, New York, who are intimidated by other bison in Buffalo, New York also happen to intimidate other bison in Buffalo New York.

Some fun examples of lexical ambiguity

Kids make nutritious snacks

Milk drinkers are turning to powder

Drunk musician gets nine months in violin case

Man eating piranha mistakenly sold as pet fish

Include your children when baking cookies

Red tape holds up new bridge

linguistic principles: context-free grammars

formal definition of a context-free grammar

A **context-free grammar** can be defined as ***G = (N, Σ, R, S)*** where:

- ***N*** is a set of non-terminal symbols
- ***Σ*** is a set of terminal symbols
- ***R*** is a set of rules
- ***S*** ∈ ***N*** is a distinguished start symbol

A Simplified Context-free Grammar for English

***G* = (*N*, *Σ*, *R*, *S*)** where:

- ***N*** = { S, NP, VP, PP, DT, Vi, Vt, NN, IN, PR }
- ***Σ*** = { sleeps, saw, man, woman, telescope, the, with, in }
- ***R*** is the set of rules (or “derivations”) shown below
- ***S*** = S

R

S	→	NP VP
VP	→	Vi
VP	→	Vt NP
VP	→	VP NP
NP	→	PR
NP	→	DT NN
NP	→	NP PP
PP	→	IN NP

Σ

Vi	→	sleeps
Vt	→	saw
NN	→	man
NN	→	woman
NN	→	telescope
DT	→	the
IN	→	with
IN	→	in
PR	→	he

N

S	sentence
VP	verb phrase
NP	noun phrase
PP	prepositional phrase
DT	determiner
Vi	intransitive verb
Vt	transitive verb
NN	noun
IN	preposition
PR	pronoun

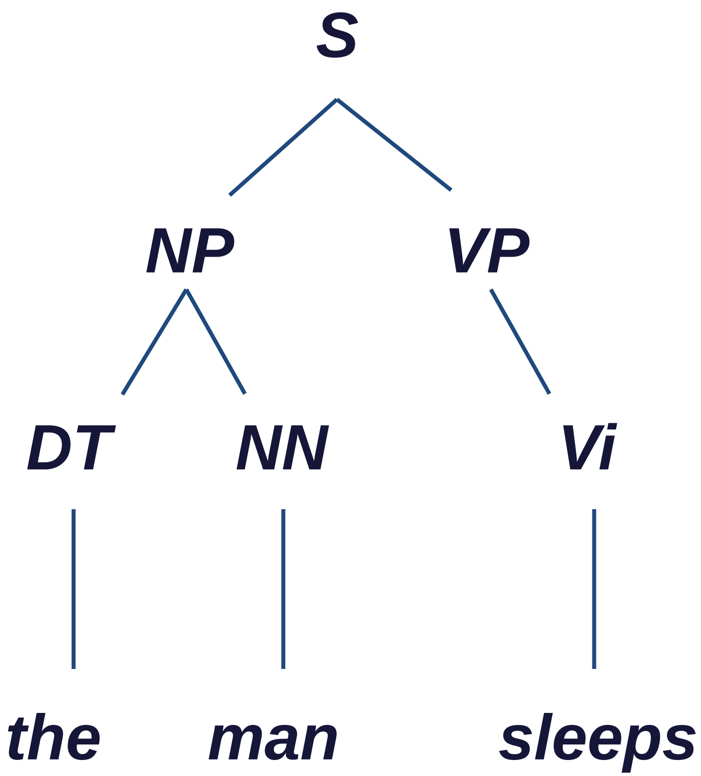
Parse Tree for this Simplified Grammar

S	→	NP VP
VP	→	Vi
VP	→	Vt NP
VP	→	VP NP
NP	→	PR
NP	→	DT NN
NP	→	NP PP
PP	→	IN NP

R

Vi	→	sleeps
Vt	→	saw
NN	→	man
NN	→	woman
NN	→	telescope
DT	→	the
IN	→	with
IN	→	in
PR	→	he

Σ



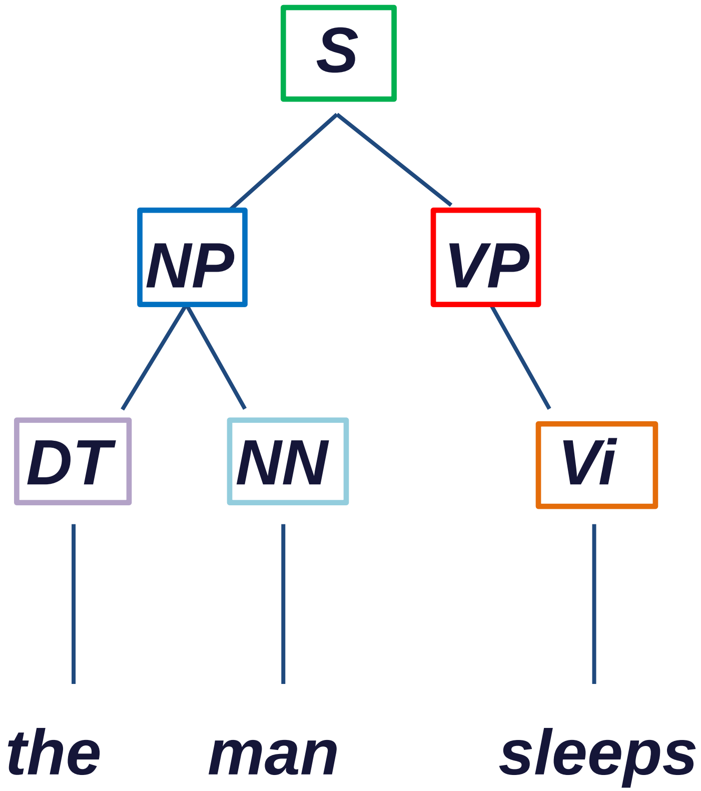
Parse Tree for this Simplified Grammar

R

S	→	NP VP
VP	→	Vi
VP	→	Vt NP
VP	→	VP NP
NP	→	PR
NP	→	DT NN
NP	→	NP PP
PP	→	IN NP

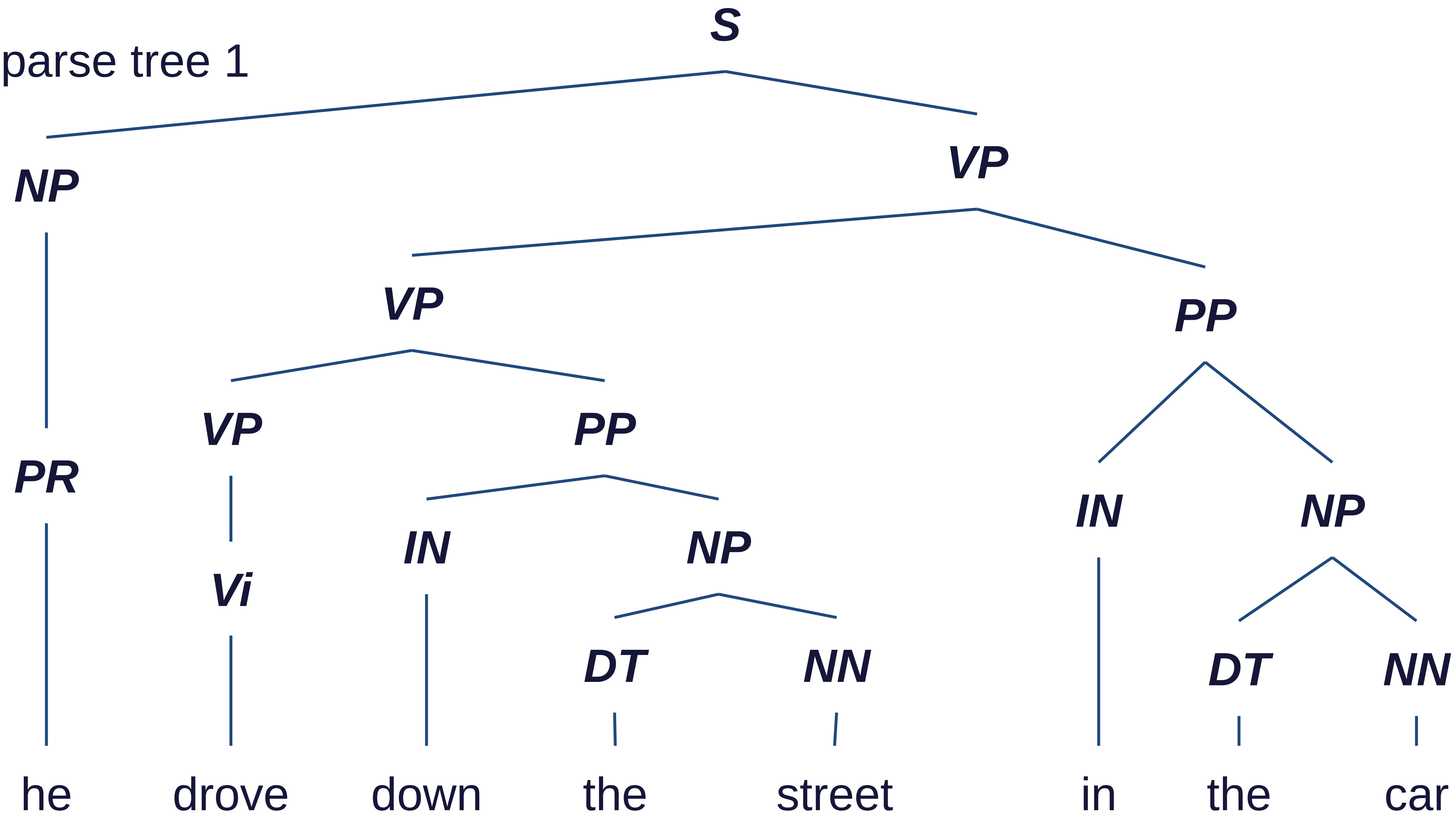
Σ

Vi	→	sleeps
Vt	→	saw
NN	→	man
NN	→	woman
NN	→	telescope
DT	→	the
IN	→	with
IN	→	in
PR	→	he



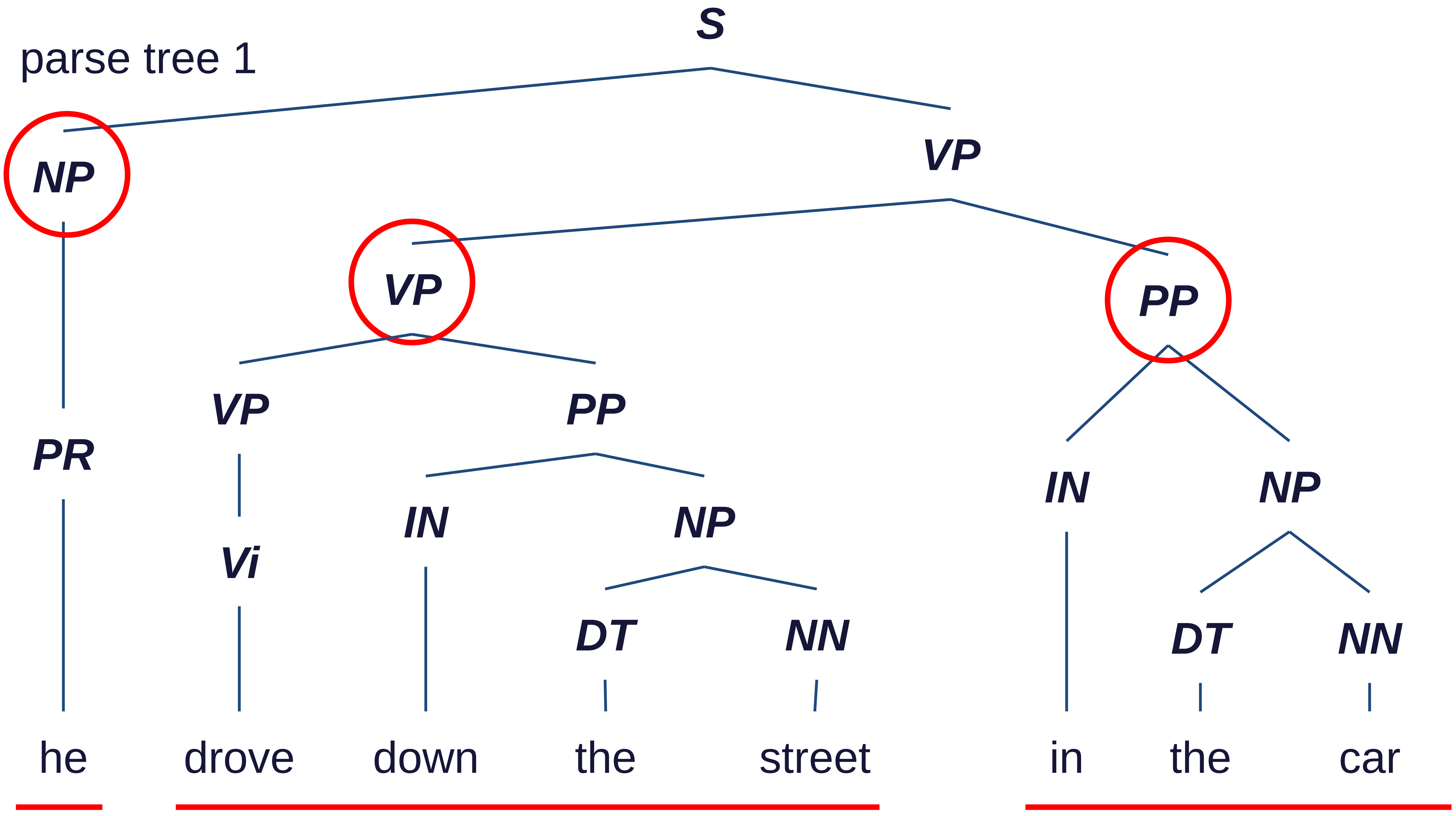
ambiguity - “he drove down the street in the car”

parse tree 1



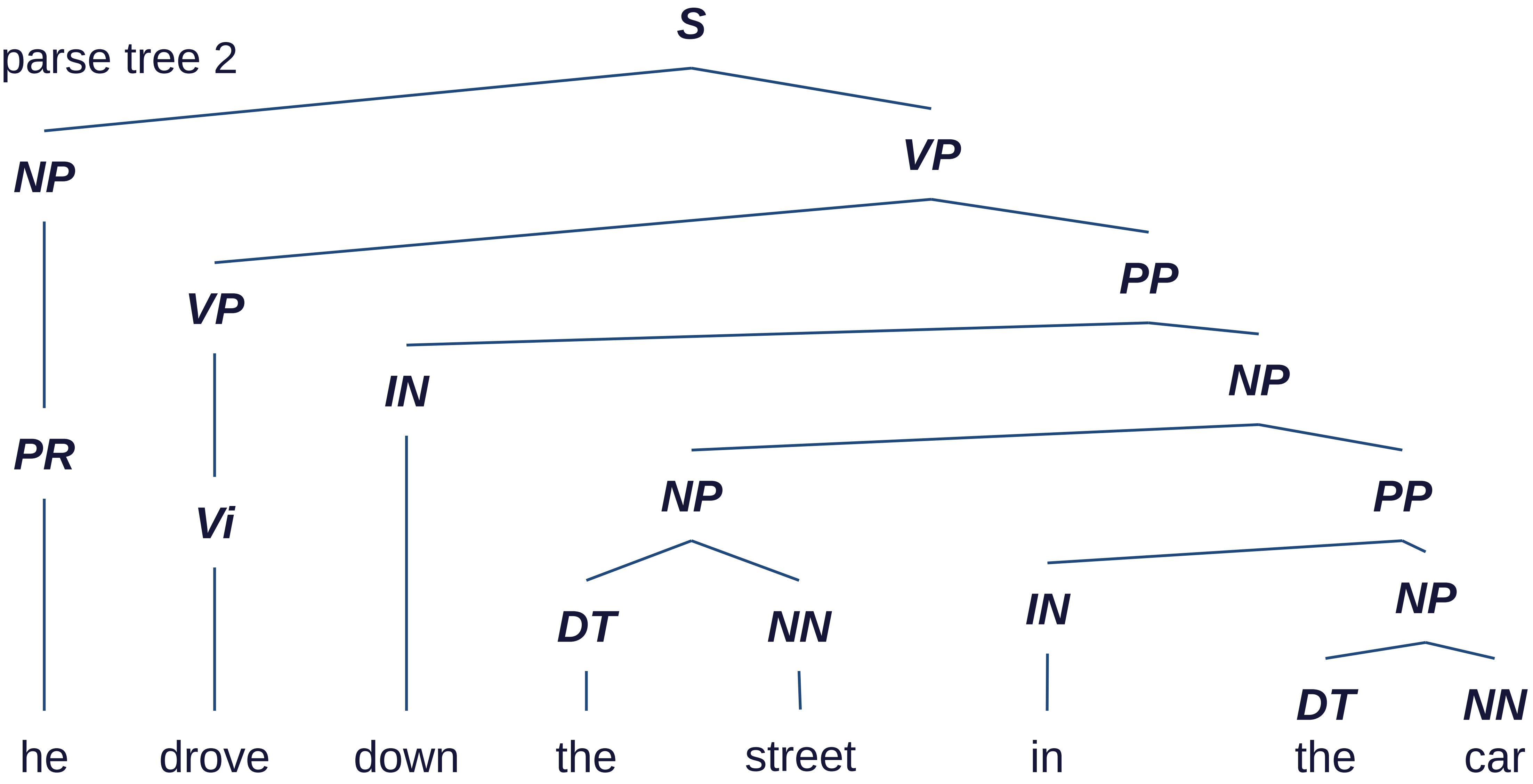
ambiguity - “he drove down the street in the car”

parse tree 1



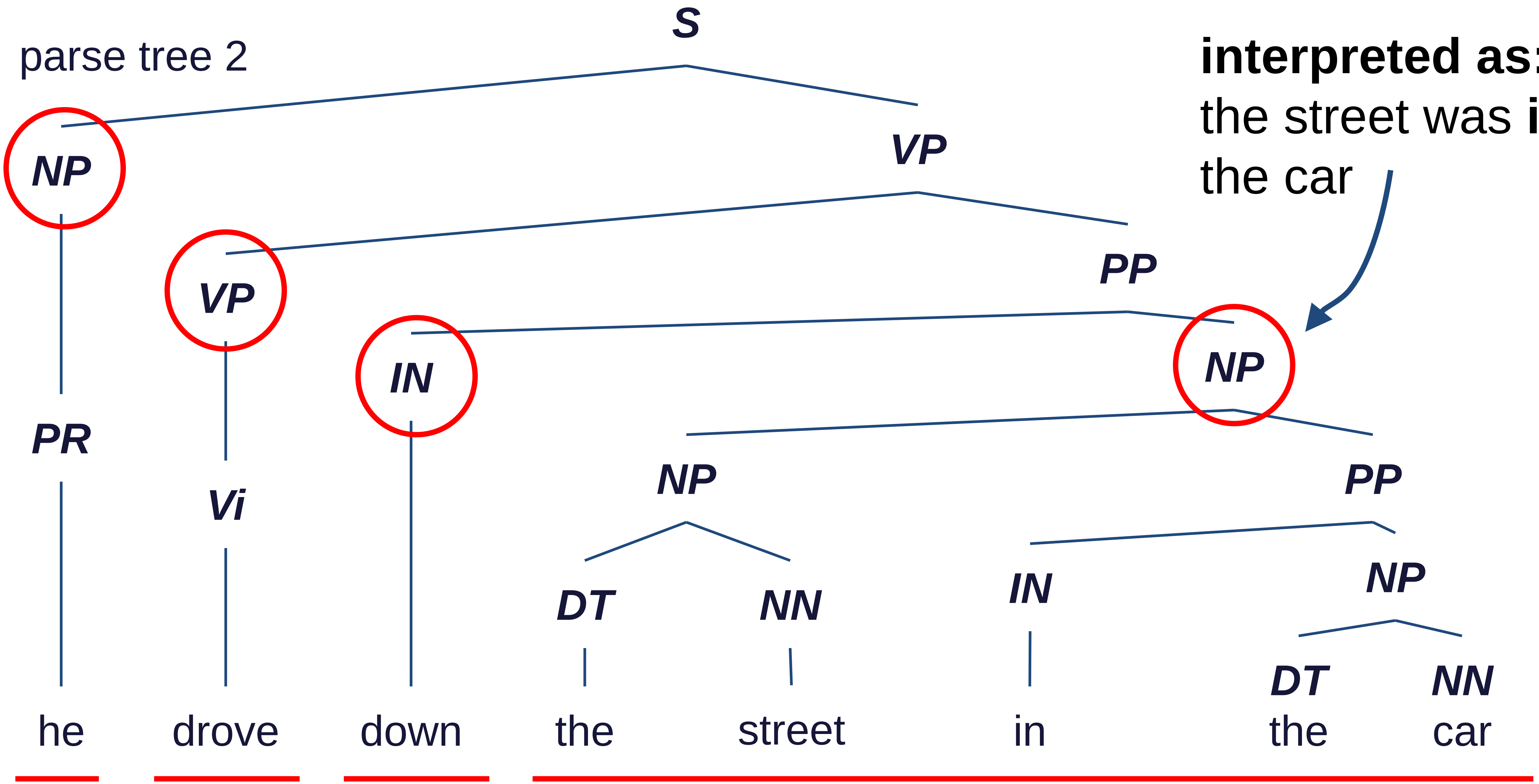
ambiguity - “he drove down the street in the car”

parse tree 2



ambiguity - “he drove down the street in the car”

parse tree 2



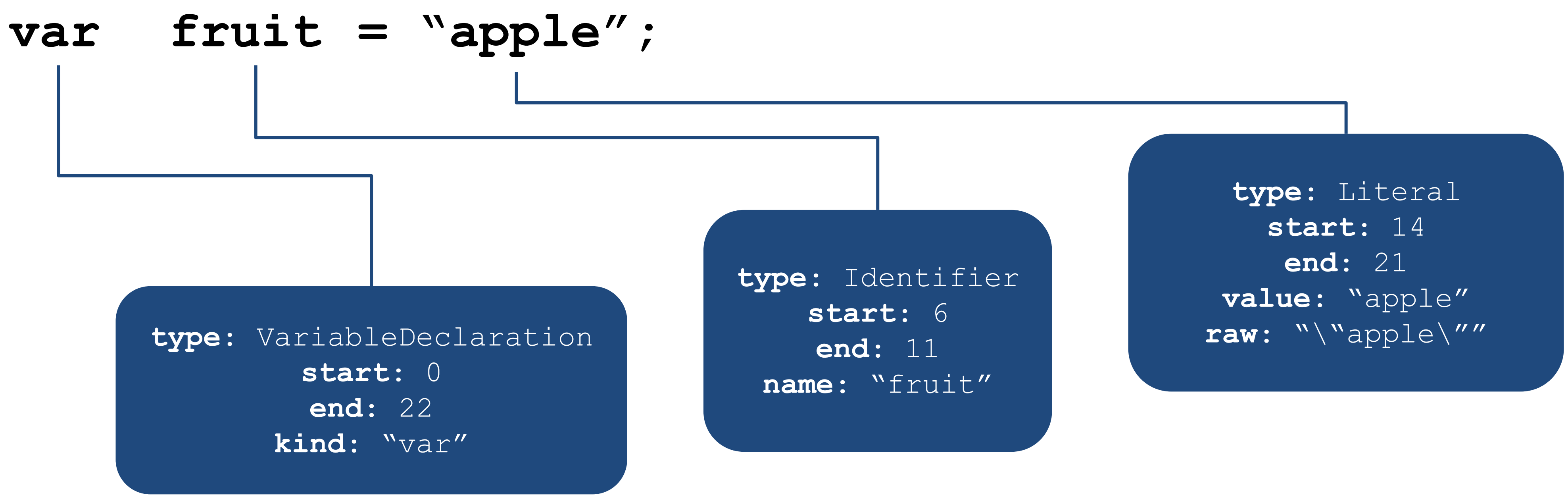
interpreted as:
the street was **in** the car

lexing & parsing using context-free grammars

lexing / tokenization

lexing is the process of breaking an input stream into discrete components (lexemes) and applying defining characteristic information to them.

lexing is a fancy word for tokenization, lexeme is a fancy word for token



```
var  fruit = “apple”;
```

type: VariableDeclaration
start: 0
end: 22
kind: “var”

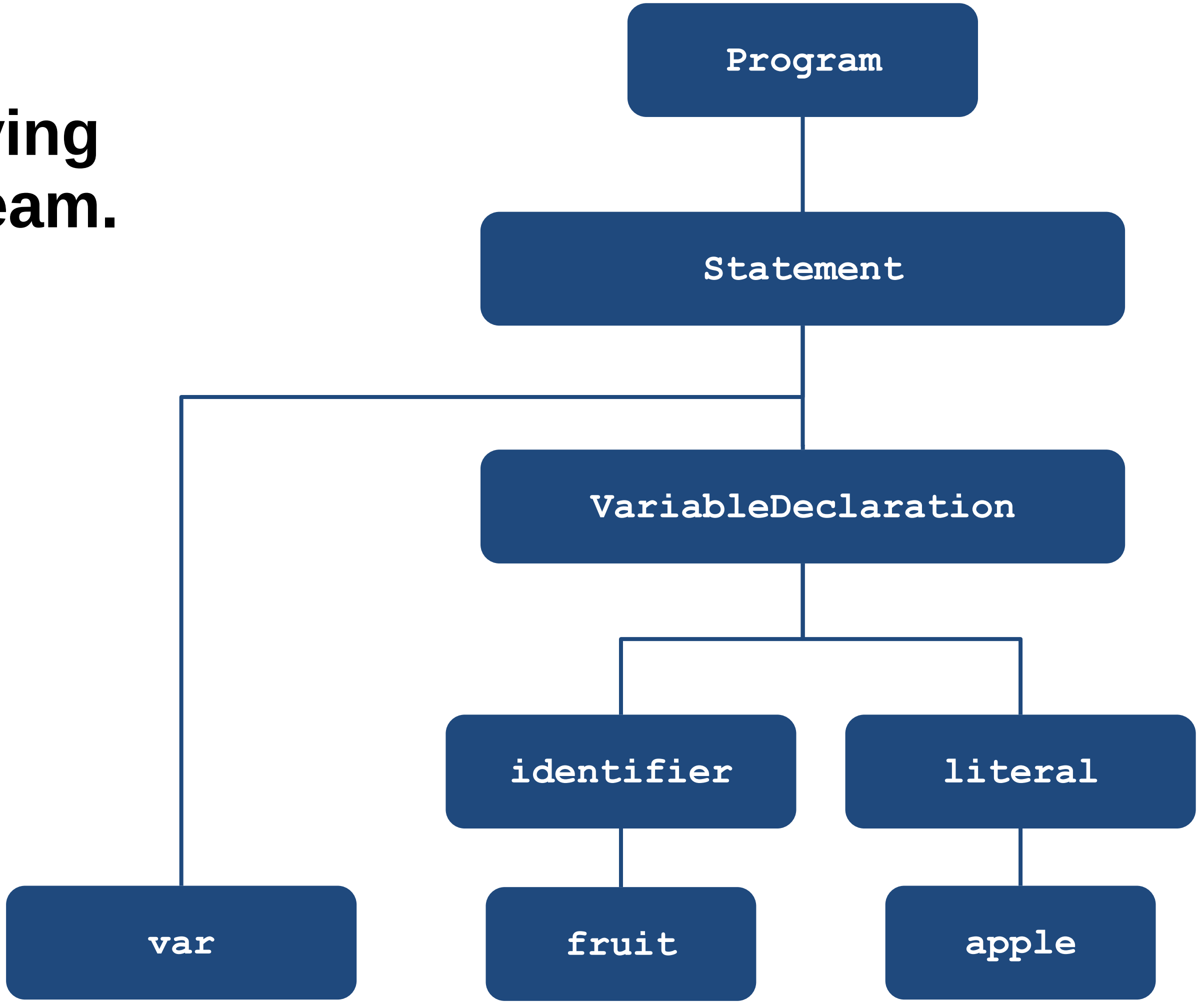
type: Identifier
start: 6
end: 11
name: “fruit”

type: Literal
start: 14
end: 21
value: “apple”
raw: “\“apple\””

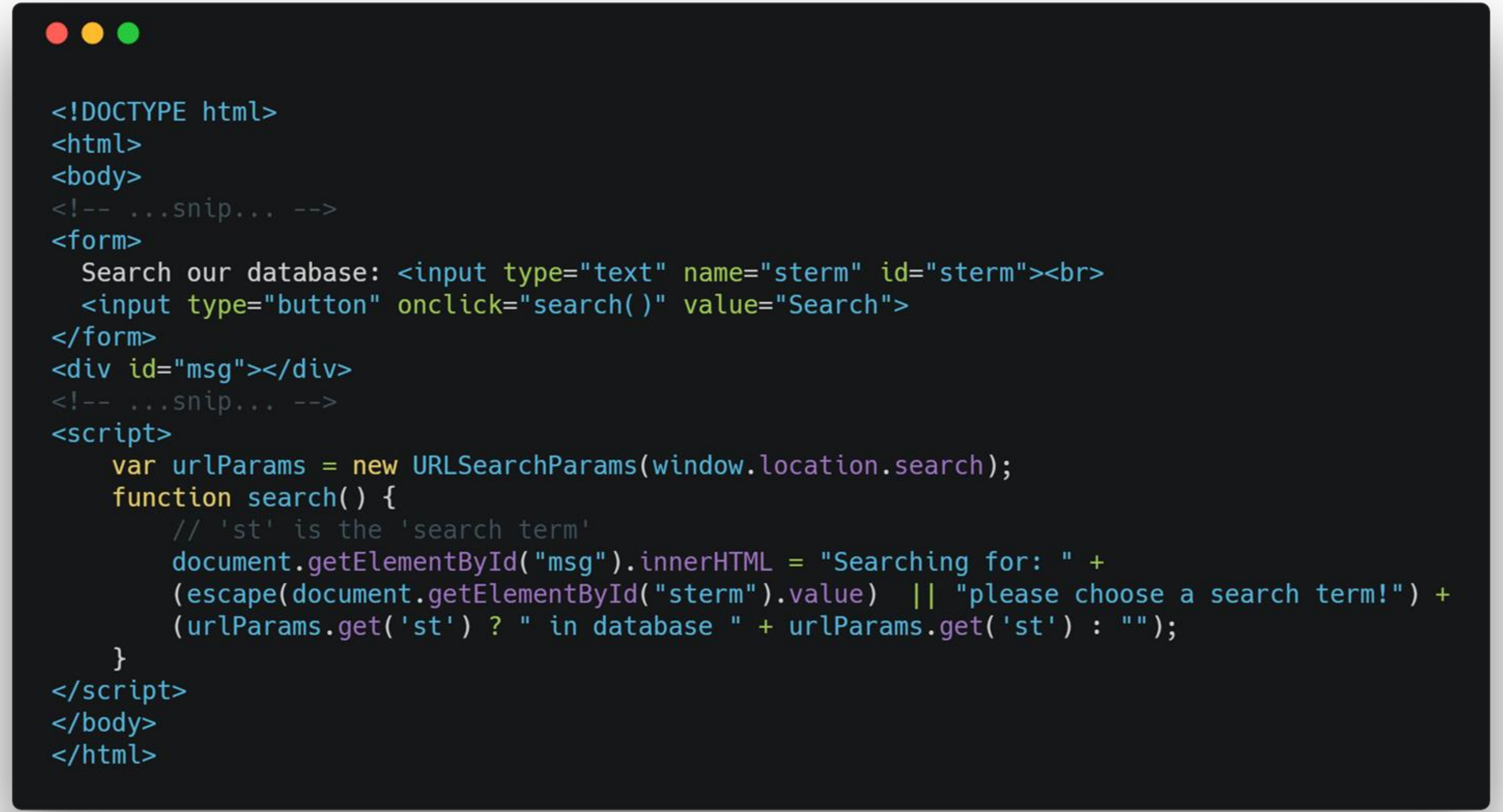
parsing

parsing is the process of applying structure to an input token stream.

```
var fruit = “apple”;
```



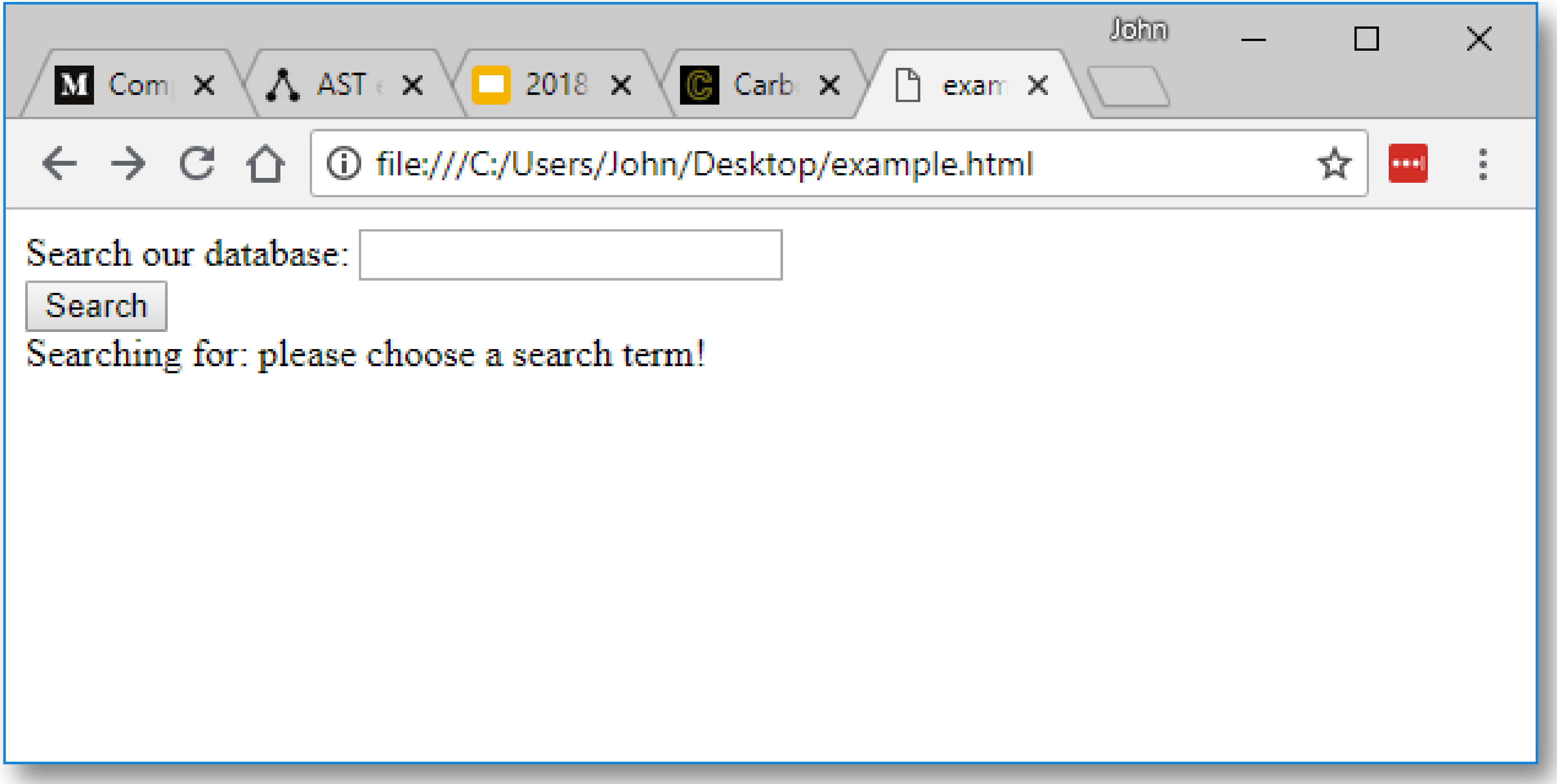
question:
can you spot the vulnerability in this javascript code?



```
<!DOCTYPE html>
<html>
<body>
<!-- ...snip... -->
<form>
  Search our database: <input type="text" name="sterm" id="sterm"><br>
  <input type="button" onclick="search()" value="Search">
</form>
<div id="msg"></div>
<!-- ...snip... -->
<script>
    var urlParams = new URLSearchParams(window.location.search);
    function search() {
        // 'st' is the 'search term'
        document.getElementById("msg").innerHTML = "Searching for: " +
        (escape(document.getElementById("sterm").value)  || "please choose a search term!") +
        (urlParams.get('st') ? " in database " + urlParams.get('st') : "");
    }
</script>
</body>
</html>
```

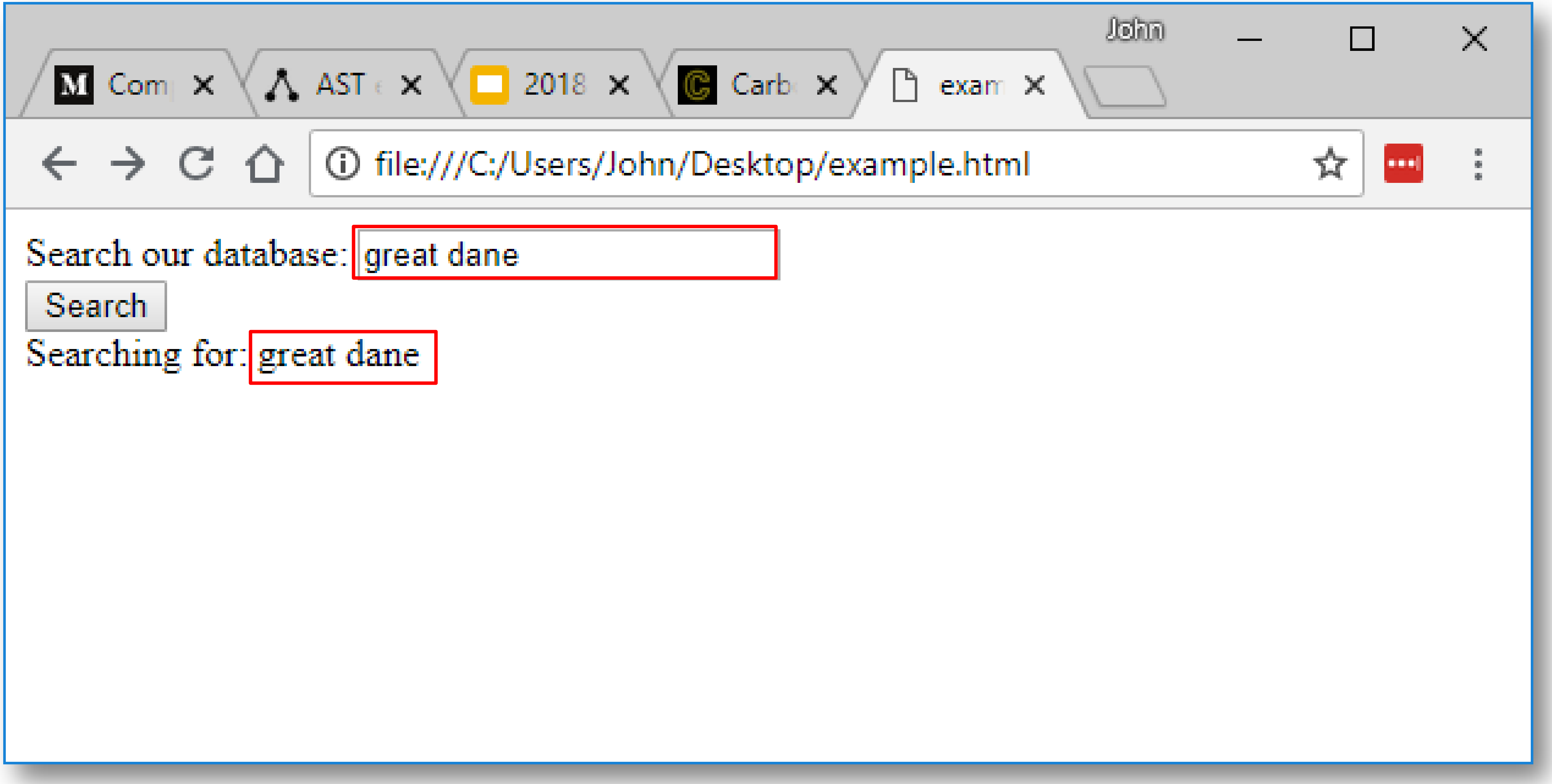
```
var urlParams = new URLSearchParams(window.location.search);

function search() {
     // 'st' is the 'search term'
     document.getElementById("msg").innerHTML = "Searching for: " +
     (escape(document.getElementById("sterm").value) || "please
     choose a search term!") + (urlParams.get('st') ? " in database "
     + urlParams.get('st') : "");
}
```



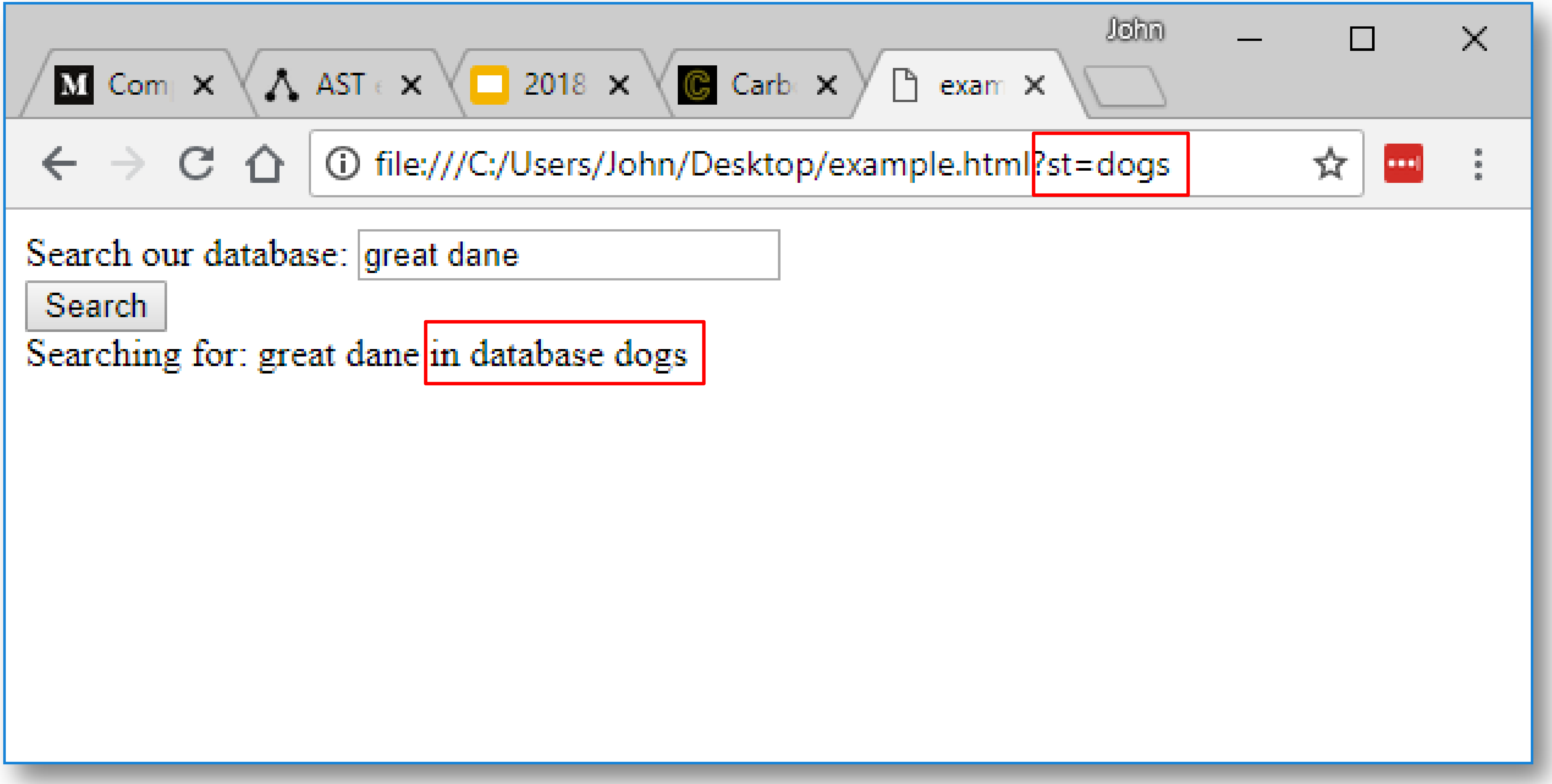
Search our database:

Searching for: please choose a search term!



Search our database: great dane

Searching for: great dane

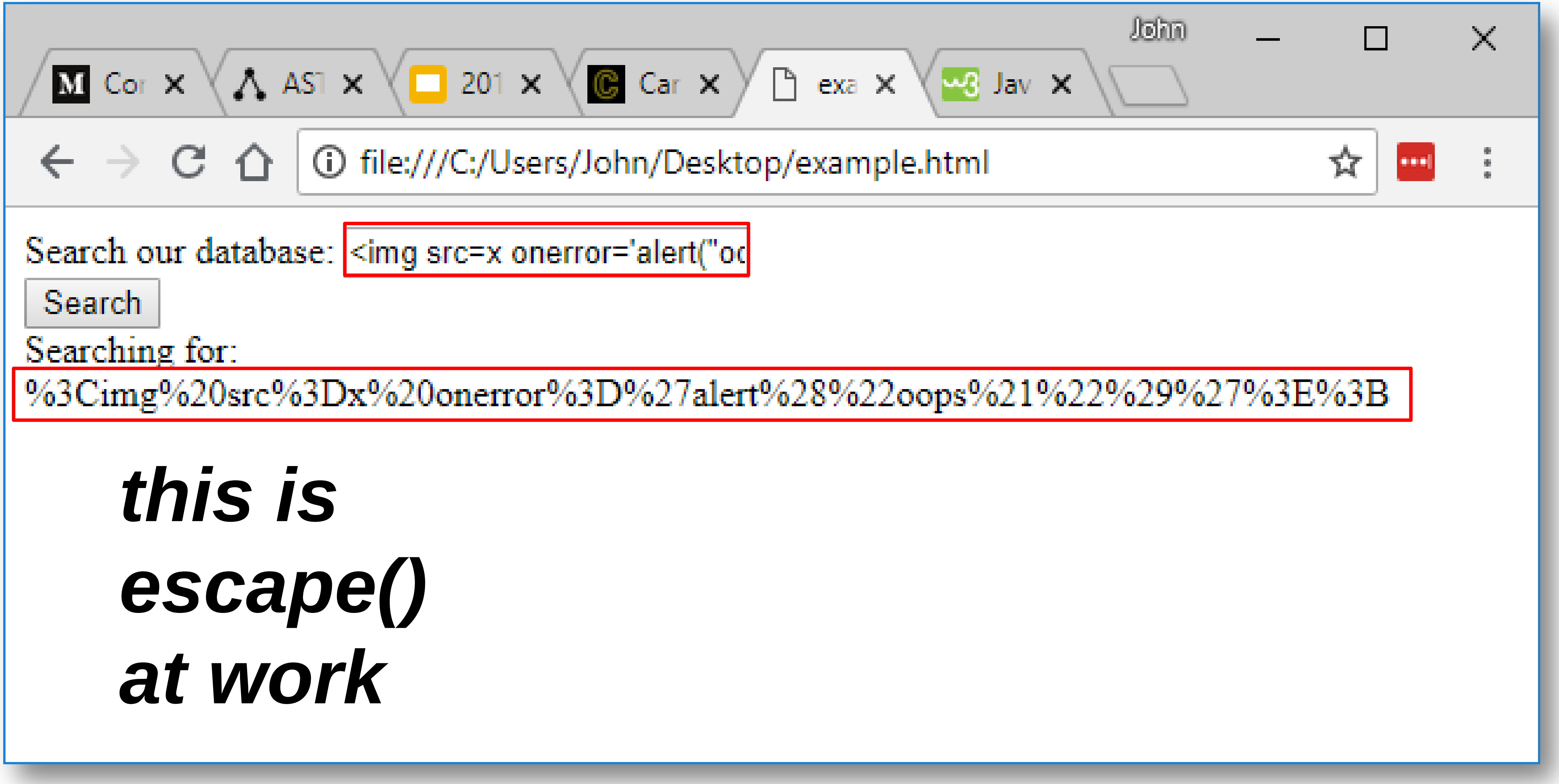


file:///C:/Users/John/Desktop/example.html?st=dogs

Search our database: great dane

Search

Searching for: great dane in database dogs

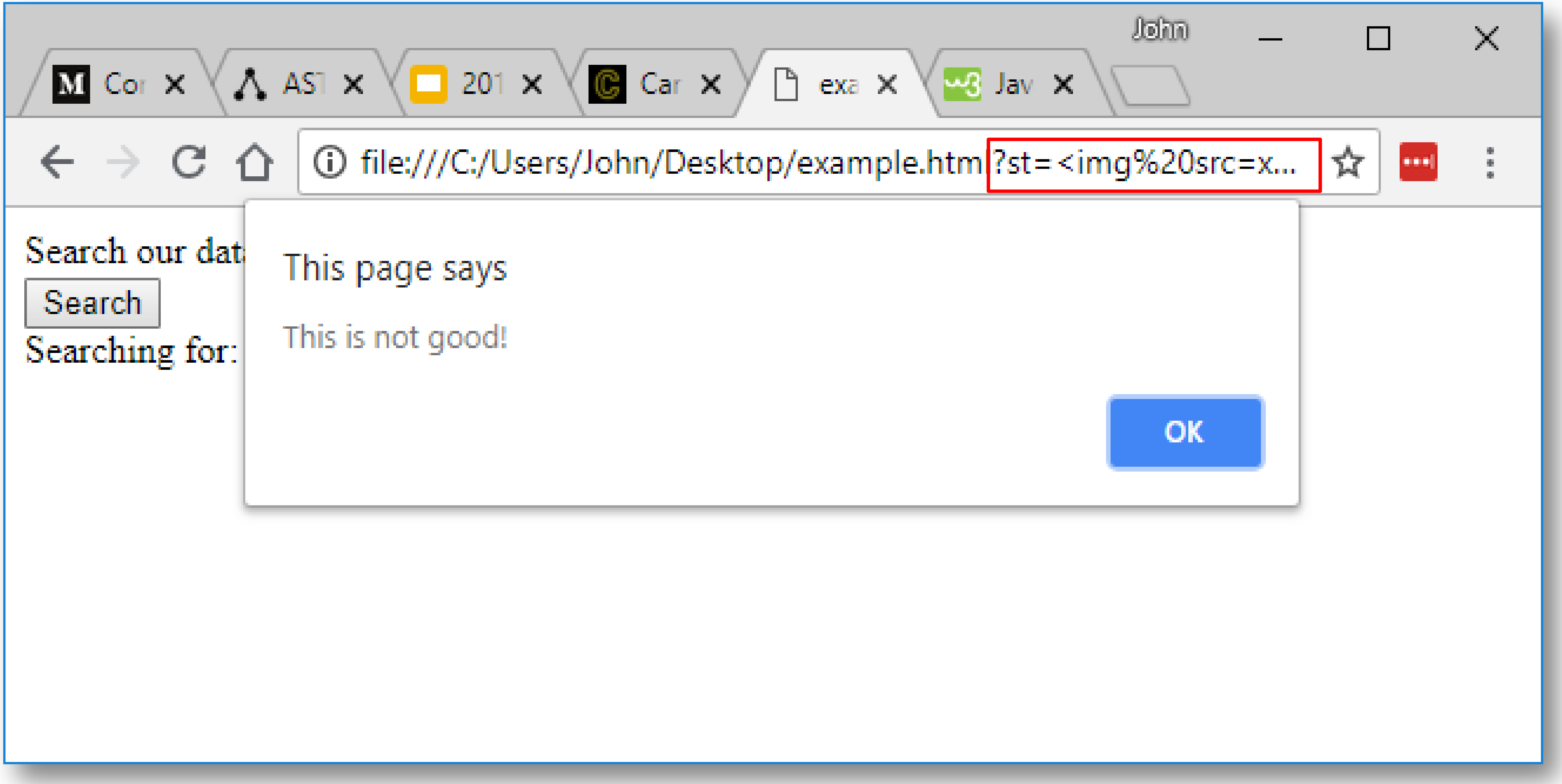


Search our database: <img src=x onerror='alert("oc

Search

Searching for:
%3Cimg%20src%3Dx%20onerror%3D%27alert%28%22oops%21%22%29%27%3E%3B

this is escape() at work



file:///C:/Users/John/Desktop/example.html?st=<img%20src=x...

Search our dat

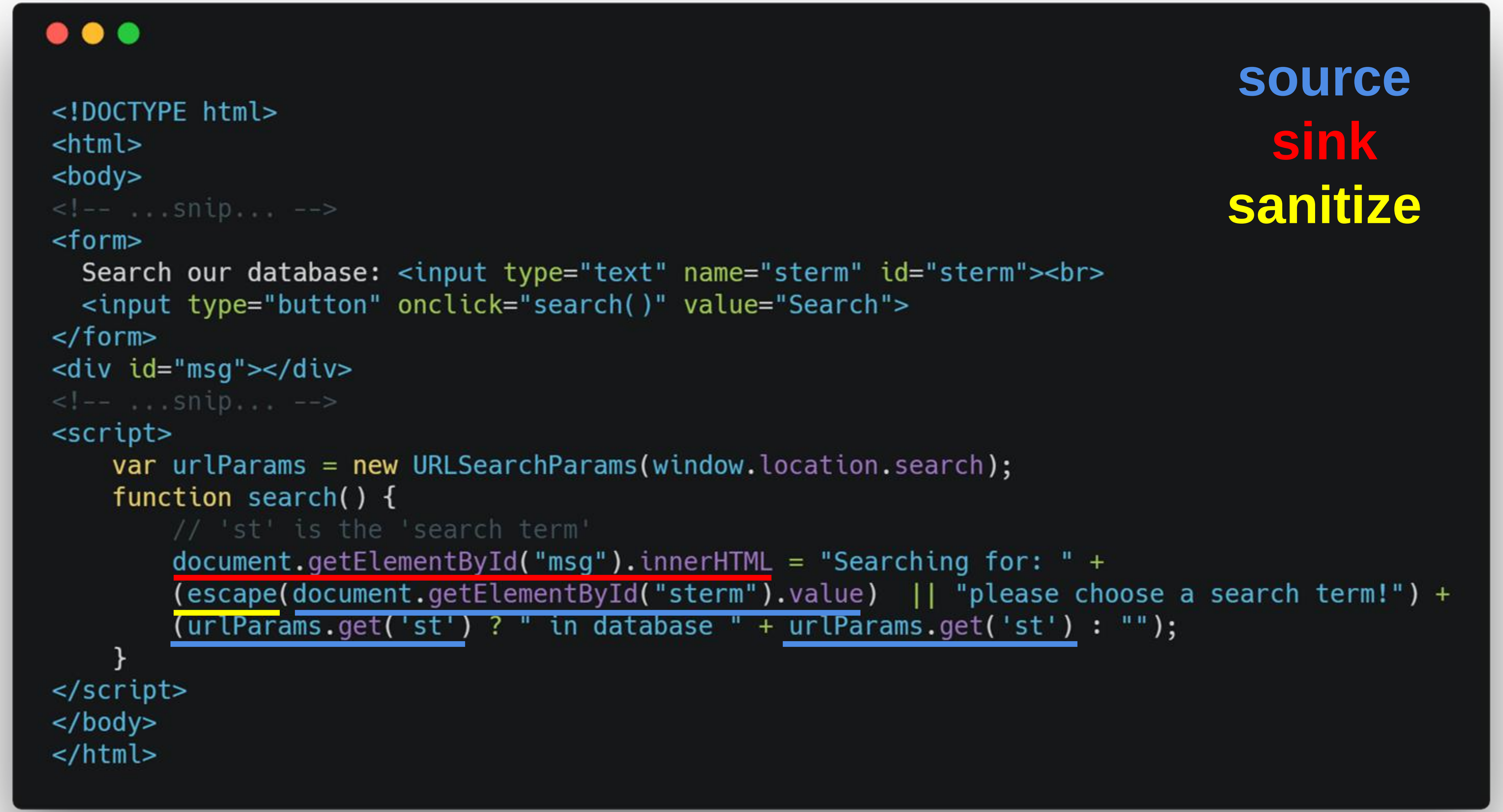
Search

Searching for:

This page says

This is not good!

OK



source

sink

sanitize

```
<!DOCTYPE html>
<html>
<body>
<!-- ...snip... -->
<form>
  Search our database: <input type="text" name="sterm" id="sterm"><br>
  <input type="button" onclick="search()" value="Search">
</form>
<div id="msg"></div>
<!-- ...snip... -->
<script>
    var urlParams = new URLSearchParams(window.location.search);
    function search() {
        // 'st' is the 'search term'
        document.getElementById("msg").innerHTML = "Searching for: " +
        (escape(document.getElementById("sterm").value)  || "please choose a search term!") +
        (urlParams.get('st') ? " in database " + urlParams.get('st') : "");
    }
</script>
</body>
</html>
```

```
var urlParams = new URLSearchParams(window.location.search);

function search() {
      // 'st' is the 'search term'
      document.getElementById("msg").innerHTML = "Searching for: " +
      (escape(document.getElementById("sterm").value) || "please
      choose a search term!") + (urlParams.get('st') ? " in database "
      + urlParams.get('st') : "");
}
```

source

sink

sanitize

can you automate this process using *grep* or *regex*?

can you automate this process using *grep* or *regex*?

you can:

- find sources of user-controlled data
- find vulnerable sinks
- look for sanitization functions

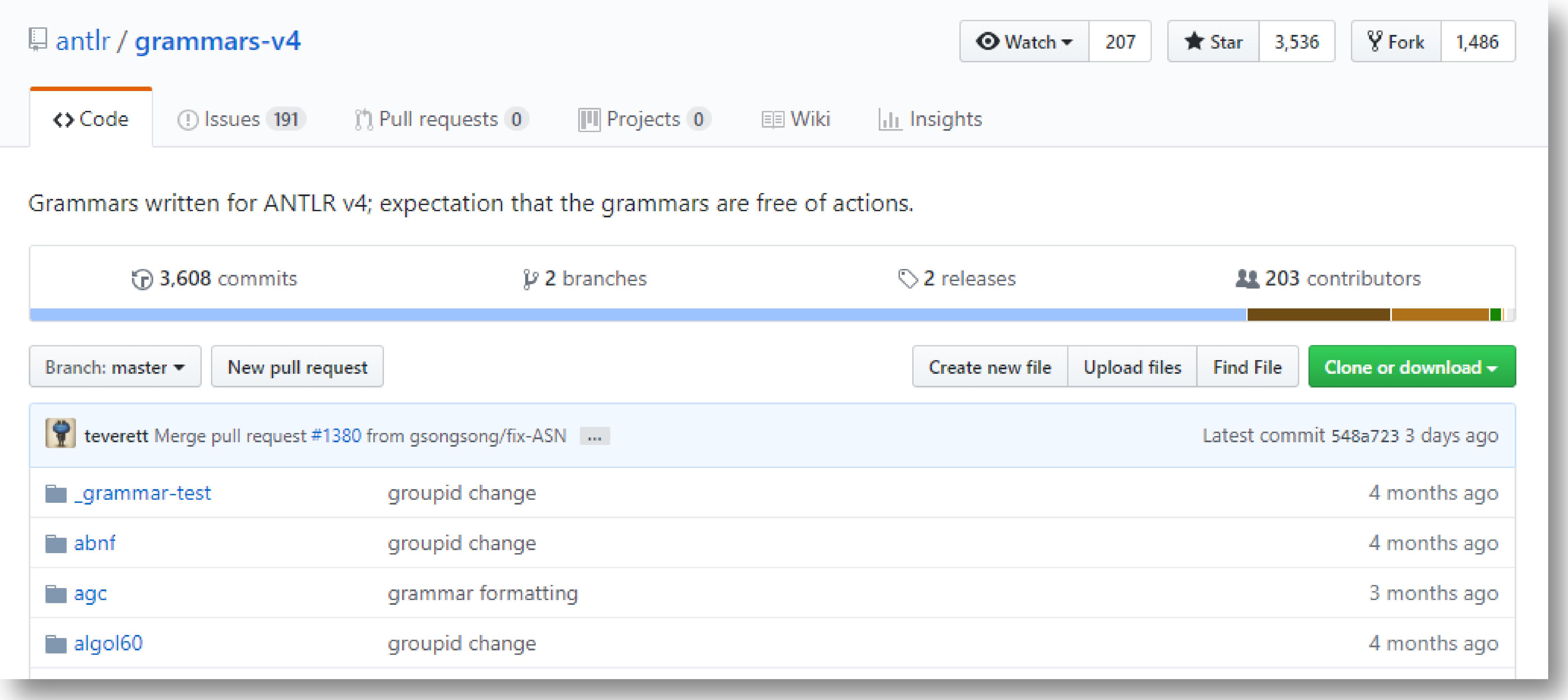
you cannot:

- determine the relationships between sources, sinks, and sanitizers
- resolve variable scope easily
- deconflict ambiguous variable names

automating this process using parsing strategies.

let’s use ANTLR as our lexer / parser generator, to build parse trees and walk them looking for issues.

currently 190 grammars available

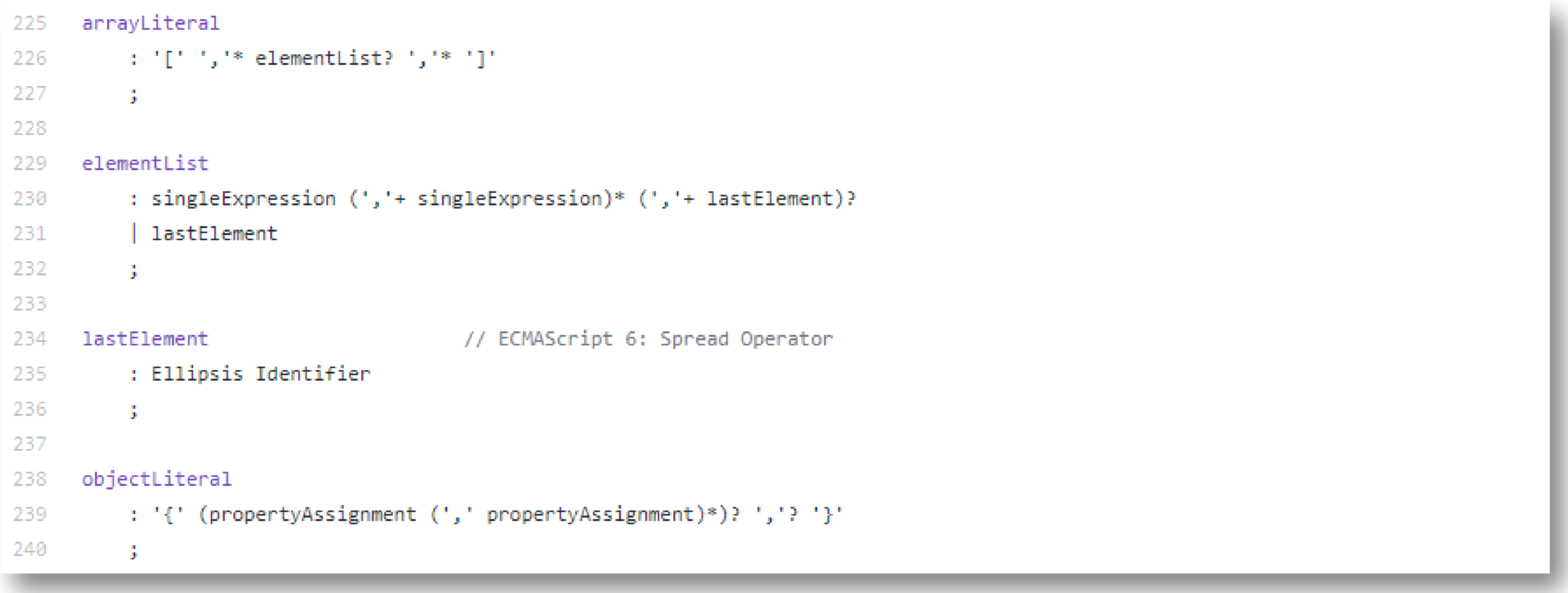


Grammars written for ANTLR v4; expectation that the grammars are free of actions.

teverett Merge pull request #1380 from gsongsong/fix-ASN … Latest commit 548a723 3 days ago

_grammar-test	groupid change	4 months ago
abnf	groupid change	4 months ago
agc	grammar formatting	3 months ago
algol60	groupid change	4 months ago

extended backus-naur form (EBNF)

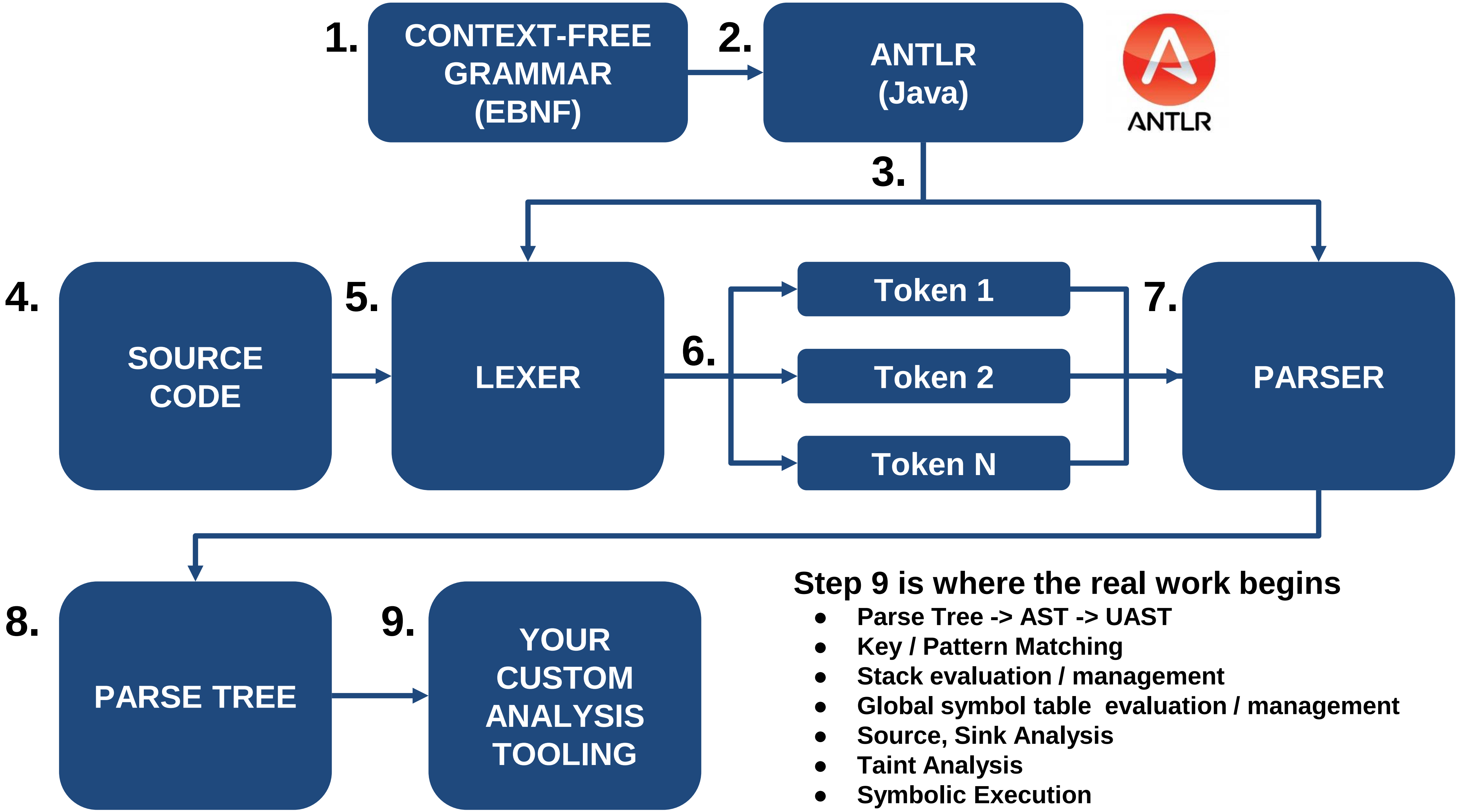


```
arrayLiteral
    : '[' ','* elementList? ','* ']'
    ;

elementList
    : singleExpression (','+ singleExpression)* (','+ lastElement)?
    | lastElement
    ;

lastElement                      // ECMAScript 6: Spread Operator
    : Ellipsis Identifier
    ;

objectLiteral
    : '{' (propertyAssignment (',' propertyAssignment)*)? ','? '}'
    ;
```



1. CONTEXT-FREE GRAMMAR (EBNF)

2. ANTLR (Java)

ANTLR

3.

4. SOURCE CODE

5. LEXER

6.

Token 1

Token 2

Token N

7. PARSER

8. PARSE TREE

9. YOUR CUSTOM ANALYSIS TOOLING

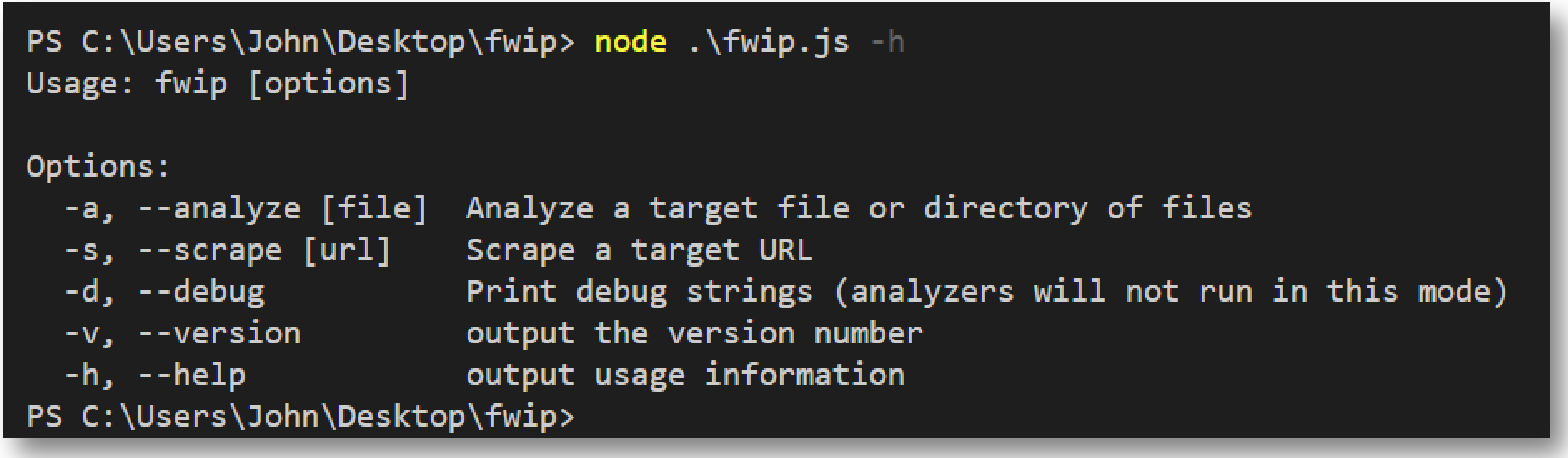
Step 9 is where the real work begins

- Parse Tree -> AST -> UAST
- Key / Pattern Matching
- Stack evaluation / management
- Global symbol table evaluation / management
- Source, Sink Analysis
- Taint Analysis
- Symbolic Execution
- Constraint Solving

demo:
fun with improvisational parsers (fwip)

github.com/cetfor/fwip

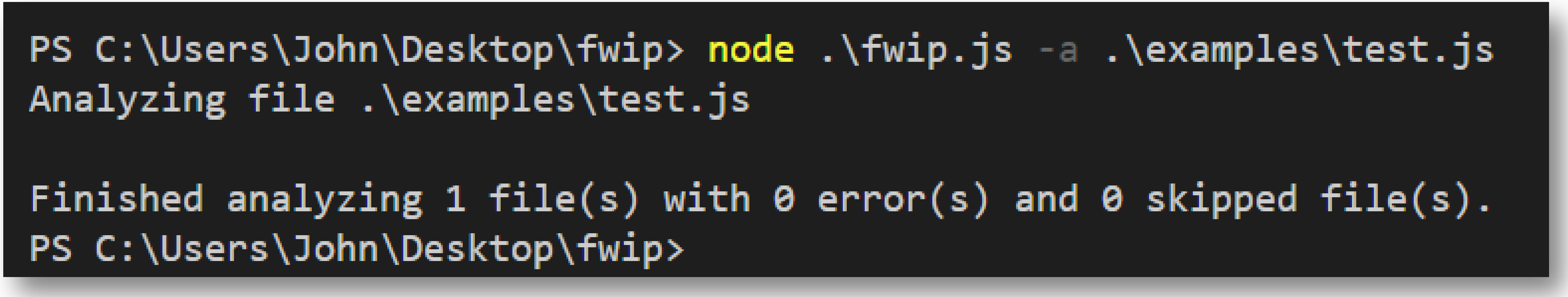
using fwip: help



```
PS C:\Users\John\Desktop\fwip> node .\fwip.js -h
Usage: fwip [options]

Options:
  -a, --analyze [file]  Analyze a target file or directory of files
  -s, --scrape [url]    Scrape a target URL
  -d, --debug           Print debug strings (analyzers will not run in this mode)
  -v, --version         output the version number
  -h, --help            output usage information
PS C:\Users\John\Desktop\fwip>
```

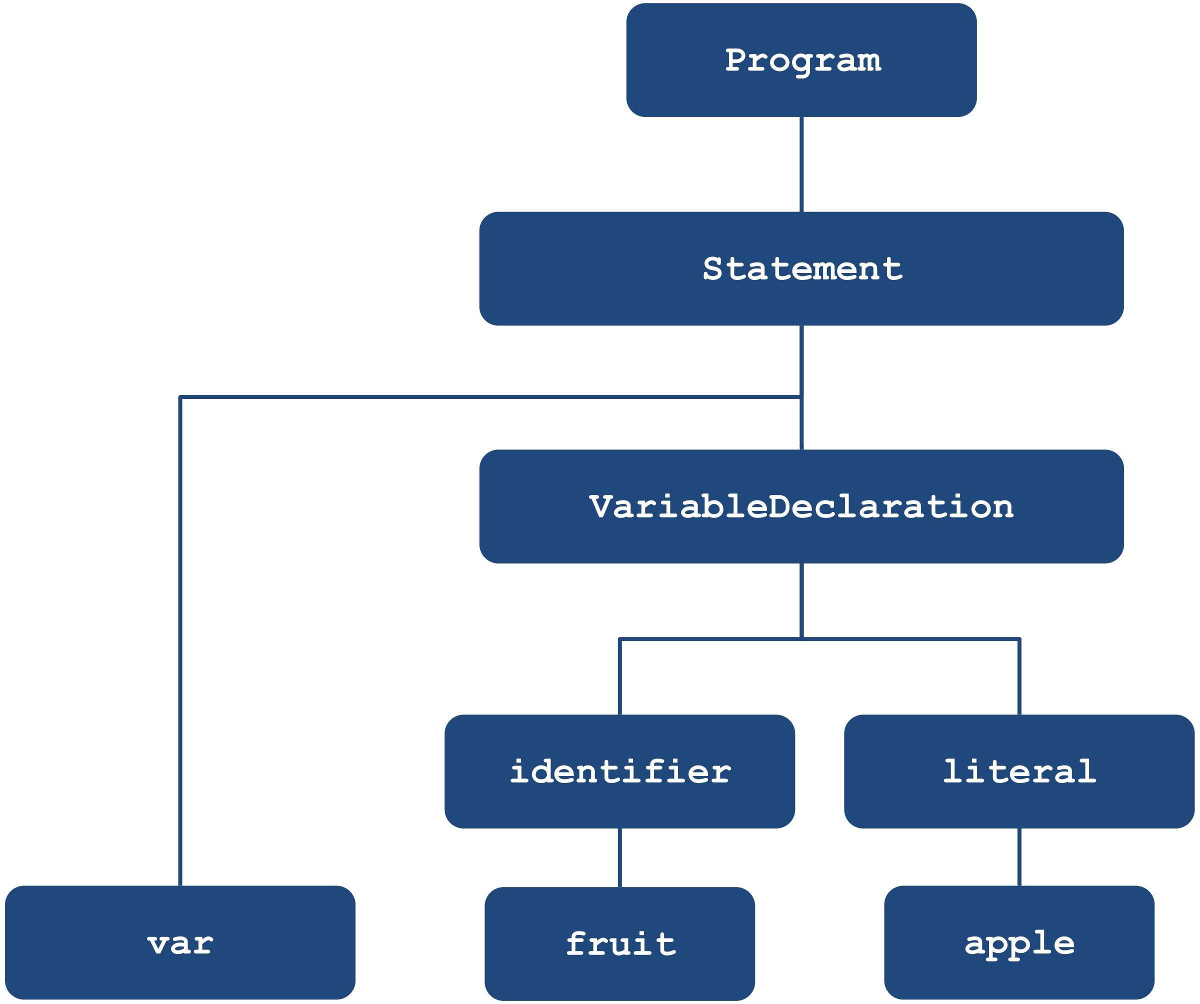
using fwip: analyzing a file

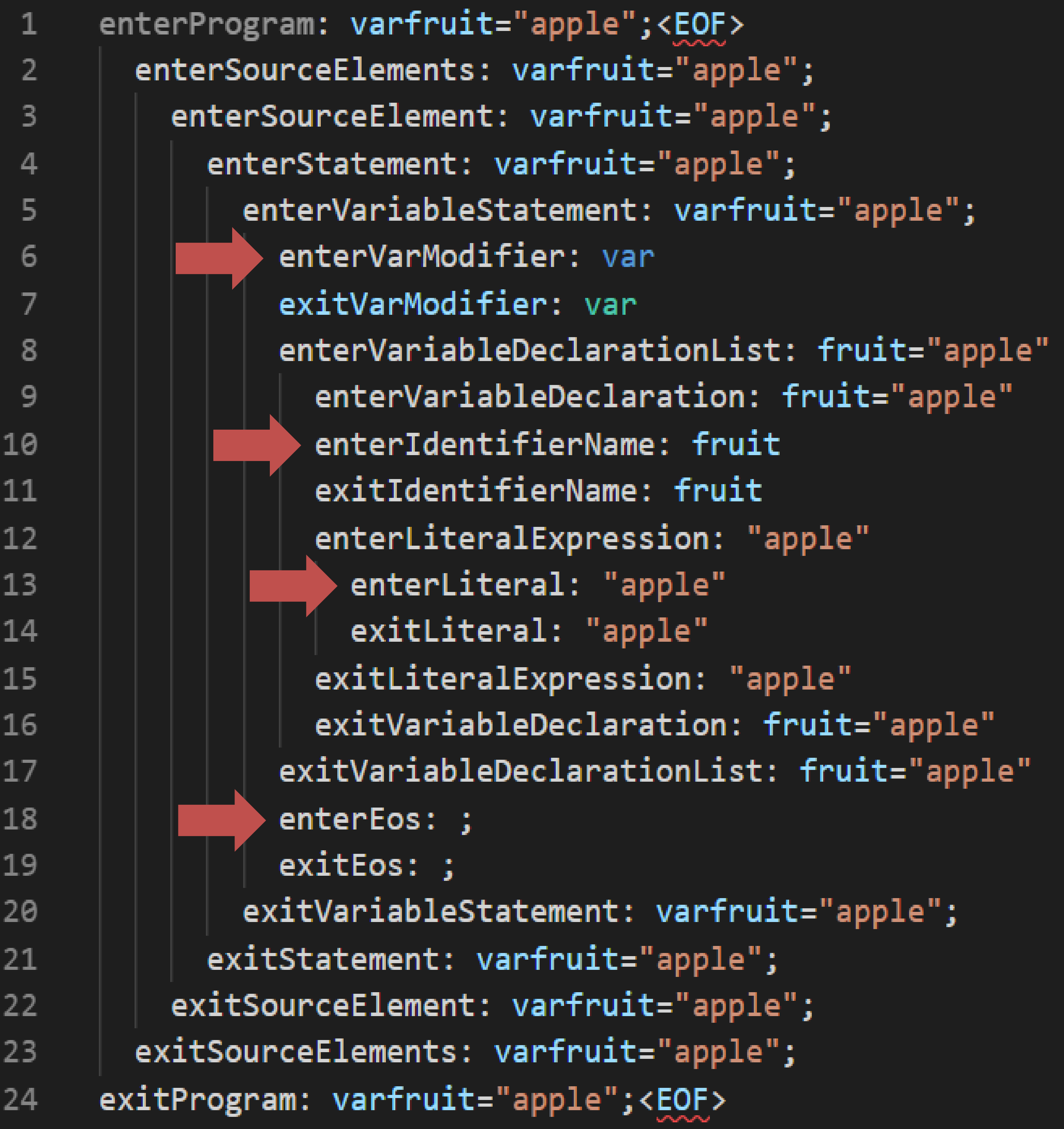


```
PS C:\Users\John\Desktop\fwip> node .\fwip.js -a .\examples\test.js
Analyzing file .\examples\test.js

Finished analyzing 1 file(s) with 0 error(s) and 0 skipped file(s).
PS C:\Users\John\Desktop\fwip>
```

remember this example parse tree?





```
enterProgram: varfruit="apple";<EOF>
  enterSourceElements: varfruit="apple";
    enterSourceElement: varfruit="apple";
      enterStatement: varfruit="apple";
        enterVariableStatement: varfruit="apple";
          enterVarModifier: var
          exitVarModifier: var
          enterVariableDeclarationList: fruit="apple"
            enterVariableDeclaration: fruit="apple"
            enterIdentifierName: fruit
            exitIdentifierName: fruit
            enterLiteralExpression: "apple"
              enterLiteral: "apple"
              exitLiteral: "apple"
            exitLiteralExpression: "apple"
            exitVariableDeclaration: fruit="apple"
          exitVariableDeclarationList: fruit="apple"
          enterEos: ;
          exitEos: ;
        exitVariableStatement: varfruit="apple";
      exitStatement: varfruit="apple";
    exitSourceElement: varfruit="apple";
  exitSourceElements: varfruit="apple";
exitProgram: varfruit="apple";<EOF>
```

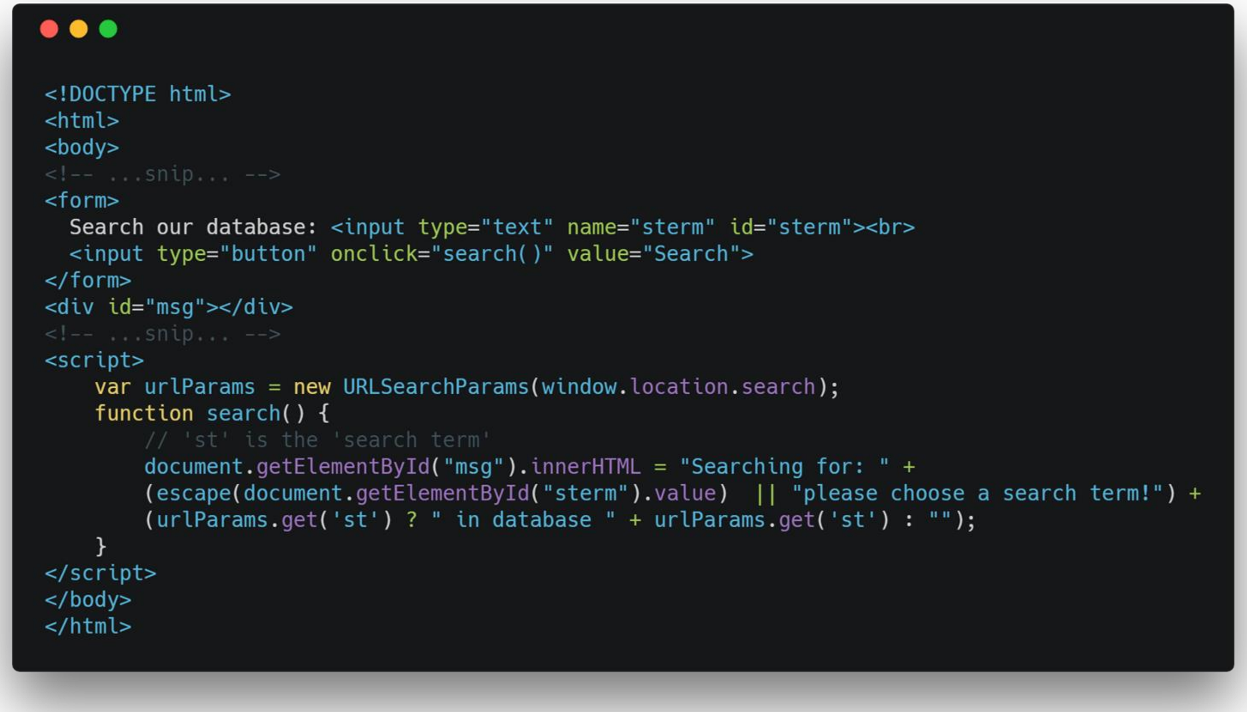
this is the result of

```
node .\fwip.js -a test.js --debug
```

where test.js is just

```
var fruit = “apple”;
```

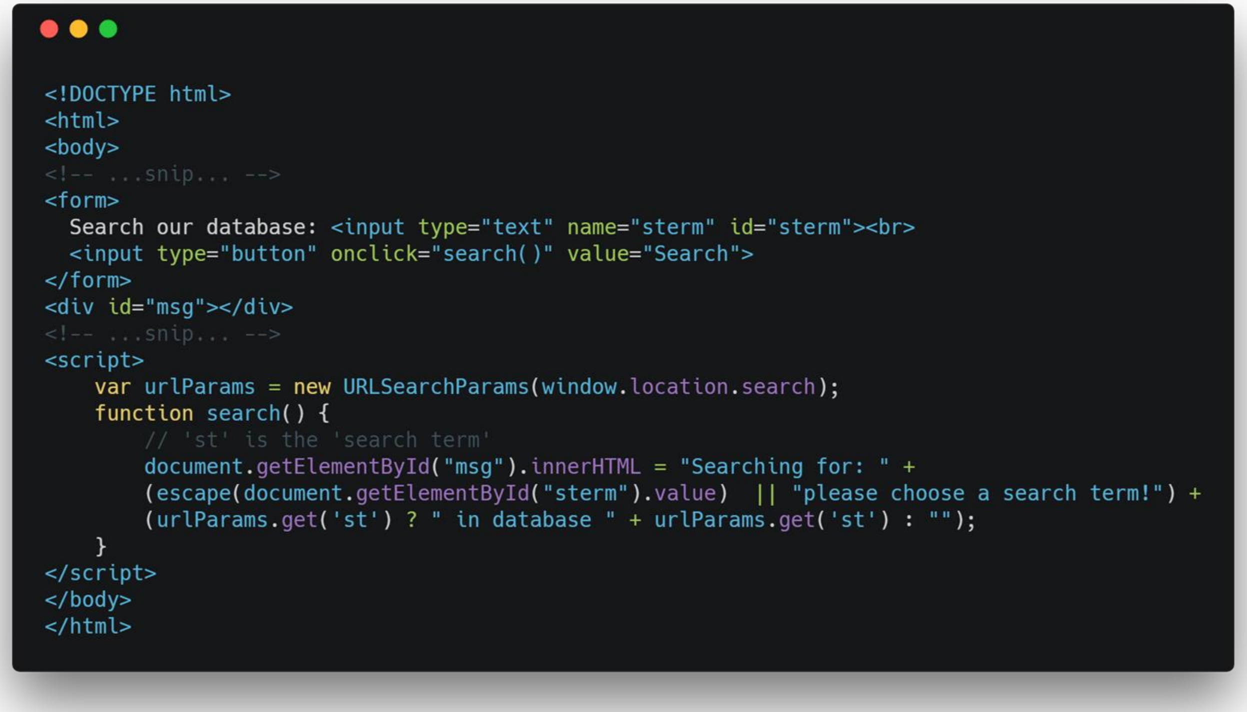
let’s revisit our example vulnerable code and build an analyzer.



```
<!DOCTYPE html>
<html>
<body>
<!-- ...snip... -->
<form>
  Search our database: <input type="text" name="sterm" id="sterm"><br>
  <input type="button" onclick="search( )" value="Search">
</form>
<div id="msg"></div>
<!-- ...snip... -->
<script>
    var urlParams = new URLSearchParams(window.location.search);
    function search() {
        // 'st' is the 'search term'
        document.getElementById("msg").innerHTML = "Searching for: " +
        (escape(document.getElementById("sterm").value)  || "please choose a search term!") +
        (urlParams.get('st') ? " in database " + urlParams.get('st') : "");
    }
</script>
</body>
</html>
```

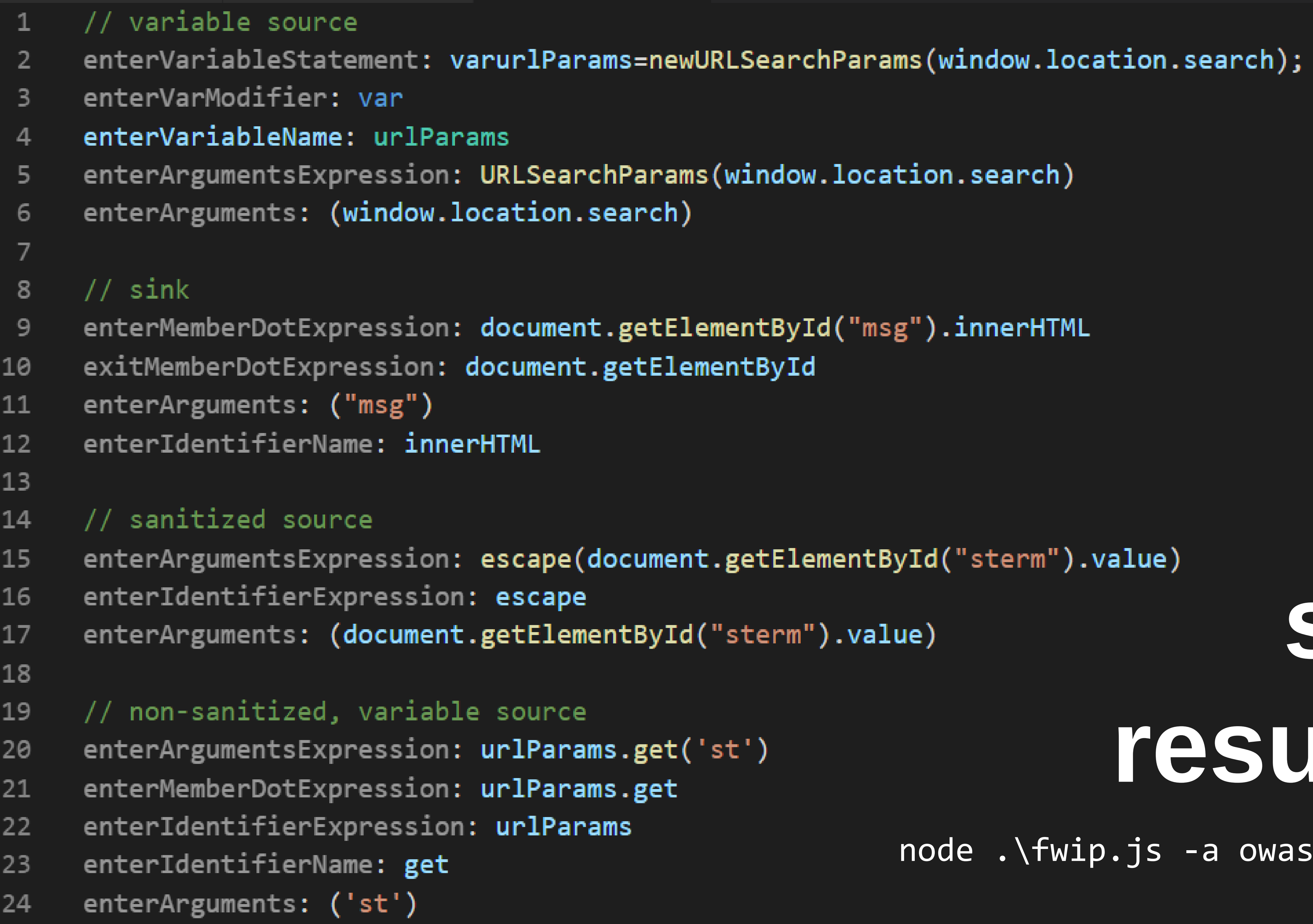
what we want to check for:

1. is a dangerous <u>sink</u> used?
2. is data from a user-controlled <u>source</u> passed to the sink?
3. iff, is a <u>sanitize</u> function used on the source data?



```html
<!DOCTYPE html>
<html>
<body>
<!-- ...snip... -->
<form>
  Search our database: <input type="text" name="sterm" id="sterm"><br>
  <input type="button" onclick="search( )" value="Search">
</form>
<div id="msg"></div>
<!-- ...snip... -->
<script>
    var urlParams = new URLSearchParams(window.location.search);
    function search() {
        // 'st' is the 'search term'
        document.getElementById("msg").innerHTML = "Searching for: " +
        (escape(document.getElementById("sterm").value)  || "please choose a search term!") +
        (urlParams.get('st') ? " in database " + urlParams.get('st') : "");
    }
</script>
</body>
</html>
```

we’ll use the *fwip --debug* switch to see what elements we should check.



```
// variable source
enterVariableStatement: varurlParams=newURLSearchParams(window.location.search);
enterVarModifier: var
enterVariableName: urlParams
enterArgumentsExpression: URLSearchParams(window.location.search)
enterArguments: (window.location.search)

// sink
enterMemberDotExpression: document.getElementById("msg").innerHTML
exitMemberDotExpression: document.getElementById
enterArguments: ("msg")
enterIdentifierName: innerHTML

// sanitized source
enterArgumentsExpression: escape(document.getElementById("sterm").value)
enterIdentifierExpression: escape
enterArguments: (document.getElementById("sterm").value)

// non-sanitized, variable source
enterArgumentsExpression: urlParams.get('st')
enterMemberDotExpression: urlParams.get
enterIdentifierExpression: urlParams
enterIdentifierName: get
enterArguments: ('st')
```

select results of

```
node .\fwip.js -a owasp.html --debug
```

there are many ways we could analyze this code. we’ll go really simple by using a stack (an array)

```
const stack = [
    // interesting stuff goes here
]
```

whenever we enter interesting conditions, we’ll push select leaf nodes to the stack.

```
const stack = [
    // interesting stuff goes here
]
```

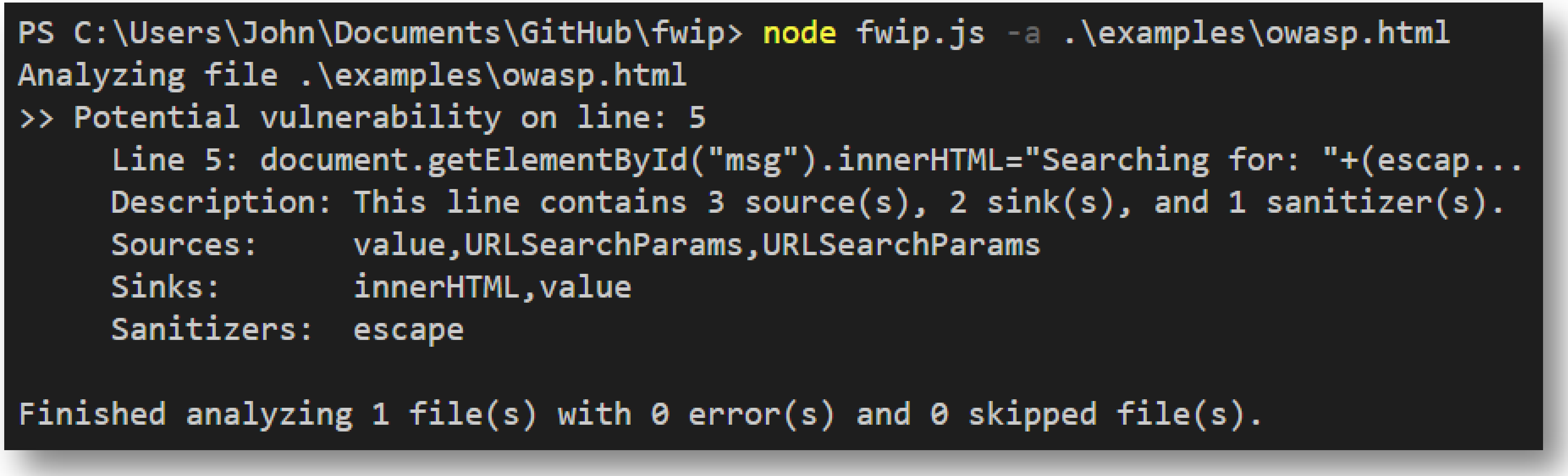
enterMemberDotExpression:
    document.getElementById(“msg”).innerHTML
…
enterIdentifierName:
    innerHTML
…
exitMemberDotExpression:
    document.getElementById(“msg”).innerHTML

```
stack.push(‘innerHTML’)
```

```
stack = [
    ‘innerHTML’,
]
```

when parsing completes we’ll call our analyzers on the program “state” we’ve built using stacks.

now onto demos and code.



```
PS C:\Users\John\Documents\GitHub\fwip> node fwip.js -a .\examples\owasp.html
Analyzing file .\examples\owasp.html
>> Potential vulnerability on line: 5
     Line 5: document.getElementById("msg").innerHTML="Searching for: "+(escap...
     Description: This line contains 3 source(s), 2 sink(s), and 1 sanitizer(s).
     Sources:     value,URLSearchParams,URLSearchParams
     Sinks:       innerHTML,value
     Sanitizers:  escape

Finished analyzing 1 file(s) with 0 error(s) and 0 skipped file(s).
```