

www.qconferences.com
www.qconbeijing.com



伦敦 | 北京 | 东京 | 纽约 | 圣保罗 | 上海 | 旧金山

London · Beijing · Tokyo · New York · Sao Paulo · Shanghai · San Francisco



QCon全球软件开发大会

International Software Development Conference

Syntaxatio

n

JavaScript: The Universal Virtual Machine

JSLint.

The Good Parts.

<http://www.JSLint.com/>

WARNING!

JSLint will hurt
your feelings.

Syntax is the least important aspect of programming language design.

Syntax is the least important aspect of programming language design.

Fashion is the least important aspect of clothing design.

Programming Languages: An Interpreter-Based Approach

Samuel N. Kamin [1990]
Lisp Clu

APL

Smalltalk

Scheme

Prolog

SASL

Minimal Syntax

Lisp

(fname arg1 arg2)

Smalltalk 80

object name: arg name2: arg2

object operator arg

[var | block]

Emotional Style

Fashionable Tolerance
of Syntaxic Ambiguity

a ♣ b ♥ c

((a ♣ b) ♥ c)
(a ♣ (b ♥ c))

Binding Power

| | |
|----|----------------------------------|
| 10 | = += -= |
| 20 | ? |
| 40 | && |
| 50 | ==== < > <= >= != |
| 60 | + - |
| 70 | * / |
| 80 | unary |
| 90 | . [(|

word

variable?

statement keyword?

operator?

special form?

Parsing

Theory of Formal Languages

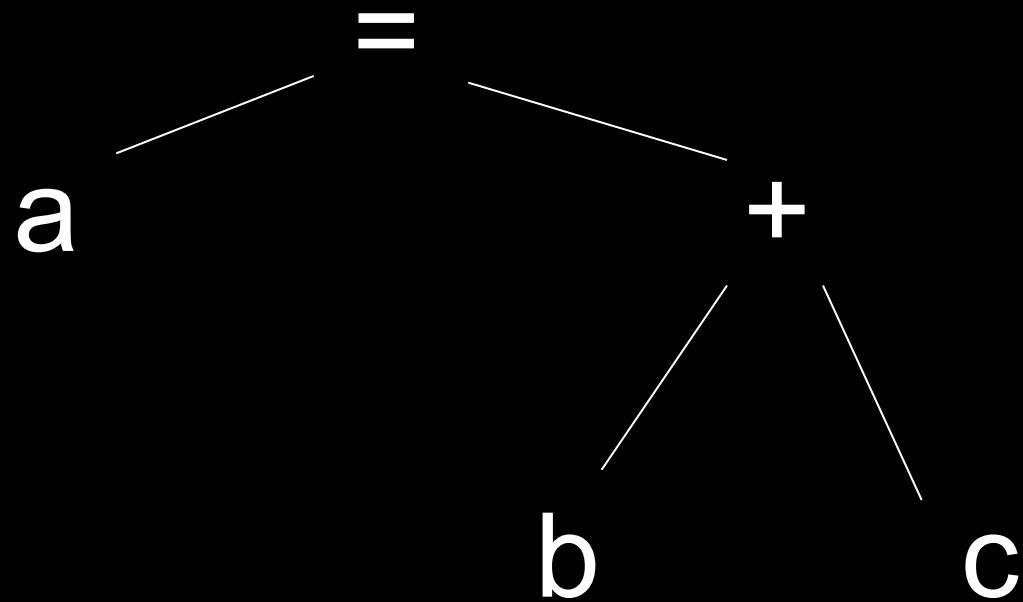
Tokens are objects

prototype ← symbol ← token

advance()

advance(id)

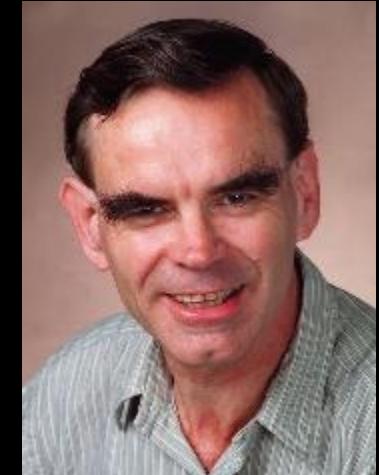
$a = b + c;$



Weave a stream of tokens into a tree

Top Down Operator Precedence

- Vaughan Pratt [POPL 1973]
- simple to understand
- trivial to implement
- easy to use
- extremely efficient
- very flexible
- beautiful



Why have you never heard of this?

- Preoccupation with BNF grammars and their various offspring, along with their related automata and theorems.
- Requires a functional programming language.
- LISP community did not want syntax.
- JavaScript is a functional language with a community that likes syntax.

What do we expect to see to
the left of the token?

left denotation led

null denotation nud

- only nud ! ~ **typeof** { prefix
- only led * . = === infix, suffix
- nud & led + - ([

```
var symbol_table = {};  
var prototype_token = {  
    nud: function () {  
        this.error("Undefined.");  
    },  
    led: function (left) {  
        this.error("Missing operator.");  
    },  
    error: function (message) {  
        ...  
    },  
    lbp: 0      // left binding power  
};
```

```
function symbol(id, bp) {
  var s = symbol_table[id];
  bp = bp || 0;
  if (s) {
    if (bp >= s.lbp) {
      s.lbp = bp;
    }
  } else {
    s = Object.create(prototype_token);
    s.id = s.value = id;
    s.lbp = bp;
    symbol_table[id] = s;
  }
  return s;
}
```

```
symbol (":") ;  
symbol (";") ;  
symbol (",") ;  
symbol (") ") ;  
symbol ("] ") ;  
symbol ("} ") ;  
symbol ("else") ;  
symbol ("(end)") ;  
symbol ("(word)") ;
```

```
symbol("+", 60).led = function (left) {  
    this.first = left;  
    this.second = expression(60);  
    this.arity = "binary";  
    return this;  
};
```

```
symbol("*", 70).led = function (left) {  
    this.first = left;  
    this.second = expression(70);  
    this.arity = "binary";  
    return this;  
};
```

```
function infix(id, bp, led) {  
    var s = symbol(id, bp);  
    s.led = led || function (left) {  
        this.first = left;  
        this.second = expression(bp);  
        this.arity = "binary";  
        return this;  
    };  
    return s;  
}
```

```
infix("+", 60);  
infix("-", 60);  
infix("*", 70);  
infix("/", 70);  
infix("===", 50);  
infix("!==", 50);  
infix("<", 50);  
infix("<=", 50);  
infix(">", 50);  
infix(">=", 50);
```

```
infix("?", 20, function (left) {  
    this.first = left;  
    this.second = expression(0);  
    advance(":");  
    this.third = expression(0);  
    this.arity = "ternary";  
    return this;  
});
```

```
function infixr(id, bp, led) {  
    var s = symbol(id, bp);  
    s.led = led || function (left) {  
        this.first = left;  
        this.second = expression(bp - 1);  
        this.arity = "binary";  
        return this;  
    };  
    return s;  
}  
  
infixr("&&", 40);  
infixr("||", 40);
```

```
function assignment(id) {
    return infixr(id, 10, function (left) {
        if (left.id !== "." &&
            left.id !== "[" &&
            left.arity !== "name") {
            left.error("Bad lvalue.");
        }
        this.first = left;
        this.second = expression(9);
        this.assignment = true;
        this.arity = "binary";
        return this;
    )) ;
}
assignment("=");
assignment("+=");
assignment("-=");
```

```
function prefix(id, nud) {  
    var s = symbol(id);  
    s.nud = nud || function () {  
        this.first = expression(80);  
        this.arity = "unary"; };  
    return s;  
}  
prefix("+");  
prefix("-");  
prefix("!");  
prefix("typeof");
```

```
prefix("(" , function () {
    var e = expression(0);
    advance(")");
    return e;
}) ;
```

Statement denotation

first null denotation

fud

```
function statement() {
    var t = token,
        e;
    if (t.fud) {
        advance();
        return t.fud();
    }
    e = expression(0);
    if (!e.assignment && e.id !== "(") {
        e.error("Bad expression statement.");
    }
    advance(";");
    return e;
}
```

```
function statements() {  
    var a = [], s;  
    while (token.id !== "}" &&  
          token.id !== "(end)") {  
        a.push(statement());  
    }  
    return a;  
}
```

```
function block() {  
    advance("{");  
    var a = statements();  
    advance("}");  
    return a;  
}  
}
```

```
function stmt(id, f) {  
    var s = symbol(id);  
    s.fud = f;  
    return s;  
}
```

```
stmt("if", function () {
    advance("(");
    this.first = expression(0);
    advance(")");
    this.second = block();
    if (token.id === "else") {
        advance("else");
        this.third = token.id === "if"
            ? statement()
            : block();
    }
    this.arity = "statement";
    return this;
});
```

```
function expression(rbp) {  
    var left,  
        t = token;  
    advance();  
    left = t.nud();  
    while (rbp < token.lbp) {  
        t = token;  
        advance();  
        left = t.led(left);  
    }  
    return left;  
}
```

Top Down Operator Precedence

- It is easy to build parsers with it.
- It is really fast because it does almost nothing.
- It is fast enough to use as an interpreter.
- Dynamic: Build DSLs with it.
- Extensible languages.
- No more reserved words.

Minimalism

- Conceptual
 - Cryptic
- Notational
 - Error resistant
 - Confusion free
- Readable

Innovate

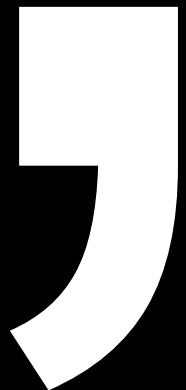
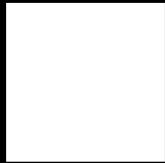
- We already have many Java-like languages.

```
CokeBottle cokeBottle = new CokeBottle();
```

- Select your features carefully.
- Beware of Sometimes Useful.
- Avoid universality.
- Manage complexity.
- Promote quality.

Innovate

- Make new mistakes.
- Let the language teach you.
- Embrace Unicode.
- Leap forward.
- Distributed programming: clouds & cores.
- State machines, constraint engines.
- Have fun.



<https://github.com/douglascrockford/TDOP>

<https://github.com/douglascrockford/JSLint>

Beautiful Code: Leading Programmers
Explain How They Think [Chapter 9]
Oram & Wilson
O'Reilly

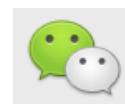
Thank you and good night.

www.infoq.com/cn

InfoQ



@InfoQ



infoqchina

软件
正在改变世界！