Elements of Programming Languages

Genesis of Some Programming Languages
JavaScript
Programming Language Popularity
StackOverflow Questions Tagged vs. Projects on Github

Corr = 0.78
Can mathematics be shown to be consistent using formal methods?
The aspiration of this ambitious work was nothing less than an attempt to derive all of mathematics from purely logical axioms, while avoiding the kinds of paradoxes and contradictions found in Frege’s earlier work on set theory.

A small part of the long proof that $1+1 = 2$ in the “Principia Mathematica”
Function Bodies

With the same goals as Alfred Whitehead and Bertrand Russell in *Principia Mathematica* (i.e., to rid mathematics of the paradoxes of the infinite and to show that mathematics is **consistent**) but also to avoid the complexities of mathematical logic - “The foundations of elementary arithmetic established by means of the recursive mode of thought, without use of apparent variables ranging over infinite domains” – Thoralf Skolem, 1923

This article can be found in “From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931 (Source Books in the History of the Sciences)”

A function is called **primitive recursive** if there is a finite sequence of functions ending with f such that each function is a successor, constant or identity function or is defined from preceding functions in the sequence by substitution or recursion.
Primitive Recursive Functions and Arithmetic
(see “A Profile of Mathematical Logic” by Howard Delong – pages 152 – 160)

“We may summarize the situation by saying that while the usual definition of a function defines it explicitly by giving an abbreviation of that expression, the recursive definition defines the function explicitly only for the first natural number, and then provides a rule whereby it can be defined for the second natural number, and then the third, and so on. The philosophical importance of a recursive function derives from its relation to what we mean by an effective finite procedure, and hence to what we mean by algorithm or decision procedure.” [DeLong, page 156]
Gödel's Incompleteness Theorems – see Delong pages, 165 - 180

Gödel showed that any system rich enough to express primitive recursive arithmetic (i.e., contains primitive recursive arithmetic as a subset of itself) either proves sentences which are false or it leaves unproved sentences which are true … in very rough outline – this is the reasoning and statement of Gödel's first incompleteness theorem. [DeLong page, 162]

Wikipedia - The first incompleteness theorem states that no consistent system of axioms whose theorems can be listed by an "effective procedure" (e.g., a computer program, but it could be any sort of algorithm) is capable of proving all truths about the relations of the natural numbers (arithmetic). For any such system, there will always be statements about the natural numbers that are true, but that are unprovable within the system. The second incompleteness theorem, an extension of the first, shows that such a system cannot demonstrate its own consistency.
Gödel's Incompleteness Theorem - Preliminaries

Algebraic Proof:

Give: \(2x - 4 = 0\)

\[
\begin{align*}
2x - 4 &= 0 & \text{Statement 1} \\
2x &= 4 & \text{Statement 2} \\
x &= 2 & \text{Statement 3}
\end{align*}
\]


Using the following table:

<table>
<thead>
<tr>
<th></th>
<th>=</th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Numbers called \(n_1\), \(n_2\), and \(n_3\) can be derived for each for statement as follows:

\[
\begin{align*}
n_1 &= 2^{**4} \times 3^{**6} \times 5^{**1} \times 7^{**5} \times 11^{**2} \times 13^{**3} = 260569237808880 \\
n_2 &= 2^{**4} \times 3^{**6} \times 5^{**2} \times 7^{**5} = 4900921200 \\
n_3 &= 2^{**6} \times 3^{**2} \times 5^{**4} = 360000
\end{align*}
\]

In a similar way, we can state that the number \(p = 2^{**n_1} \times 3^{**n_2}\) proves statement 3 (\(n_3\)) if the function Proves\((p, n_3)\) returns true.
Gödel's Incompleteness Theorem - Preliminaries

print $2^4 \times 3^6 \times 5^1 \times 7^5 \times 11^2 \times 13^3$

print $2^4 \times 3^6 \times 5^2 \times 7^5$

print $2^6 \times 3^2 \times 5^4$

def prime_factors(n):
    i = 2
    factors = []
    while i * i <= n:
        if n % i:
            i += 1
        else:
            n //= i
            factors.append(i)
    if n > 1:
        factors.append(n)
    return factors

print prime_factors($2^4 \times 3^6 \times 5^1 \times 7^5 \times 11^2 \times 13^3$)
print prime_factors($2^4 \times 3^6 \times 5^2 \times 7^5$)
print prime_factors($2^6 \times 3^2 \times 5^4$)

[2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 5, 7, 7, 7, 7, 11, 11, 13, 13, 13L]
[2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 5, 7, 7, 7, 7L]
[2, 2, 2, 2, 2, 2, 3, 3, 5, 5, 5, 5]
Gödel's Incompleteness Theorem - Preliminaries

In a similar way, using the table on the left, and Gödel Numbers for Proves and Subst, a number R can be assigned to the statement:

\[- \exists x \text{ Proves}(x, \text{Subst}(y, 17, y))\]

This statement says that there does not exist a number x such that Proves(x, Subst(y, 17, y)) returns true, i.e., Subst(y, 17, y) is not provable.

In JavaScript, Subst would be:

```javascript
function(s, x) {
    s = s.replace('17', x);
    return s
}
```
Gödel's Incompleteness Theorem

\[ R = - \exists x \text{ Proves}(x, \text{Subst}(y, 17, y)) \]

\[ G = - \exists x \text{ Proves}(x, \text{Subst}(R, 17, R)) \]

\[ G = \text{Subst}(R, 17, R) \]

\[ \therefore - \exists x \text{ Proves}(x, G) \]

i.e., If Gödel's statement is true, then it is a example of something that is true for which there is no proof. If Gödel's statement is false, then it has a proof and that proof proves the false Gödel statement true.
Good Books to Have for a Happy Life 😊

From Frege to Gödel:

From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931

Gödel’s Proof

Kurt Gödel

On Formally Undecidable Propositions of Principia Mathematica and Related Systems

Lambda-Calculus and Combinators

An Introduction

A Profile of Mathematical Logic

My Favorite

LISP 1.5 Programmer’s Manual

The Computation Center

and Research Laboratory of Electronics

Massachusetts Institute of Technology

Gödel, Escher, Bach:

An Eternal Golden Braid

DOUGLAS R. HOFSTÄTDER

Pullitzer Prize-Winner

20th-Anniversary Edition

With a new preface by the author

Dr. Philip Cannata
### Lambda Calculus

<table>
<thead>
<tr>
<th>Lambda Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x.x$</td>
</tr>
<tr>
<td>$\lambda s.(s \ s)$</td>
</tr>
<tr>
<td>$\lambda\text{func.arg.}(\text{func arg})$</td>
</tr>
</tbody>
</table>

```python
def identity = $\lambda x.x$
def self_apply = $\lambda s.(s \ s)$
def apply = $\lambda\text{func.arg.}(\text{func arg})$
def select_first = $\lambda\text{first.second.first}$
def select_second = $\lambda\text{first.second.second}$
def cond = $\lambda e1.e2.c.((c \ e1) \ e2)$
def true = select_first
def false = select_second
def not = $\lambda x.(((\text{cond false}) \ true) \ x)$
Or def not = $\lambda x.(((x \ false) \ true)$
def and = $\lambda x.y.(((\text{cond y}) \ false) \ x)$
Or def and = $\lambda x.y.((x \ y) \ false)$
def or = $\lambda x.y.(((\text{cond true}) \ y) \ x)$
Or def or = $\lambda x.y.((x \ true) \ y)$
```
## Lambda Calculus

<table>
<thead>
<tr>
<th>Lambda Calculus</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x.x )</td>
<td><code>function(x){ return x }</code> – <strong>Test:</strong> <code>function(x){ return x }</code>(3)</td>
</tr>
</tbody>
</table>
| \( \lambda s.(s\ s) \) | `function(func){return func(func)}`  
**Test:** `function(func){return func(func)}`  
`function(x){return x}` |
| \( \lambda f.\arg.(f.\arg) \) | `function(func, arg){return func(arg)}`  
**Test:** `function(func, arg){return func(arg)}`  
`function(x){return x}, 'a'` |
| `def identity = \lambda x.x`  
`def self_apply = \lambda s.(s\ s)`  
`def apply = \lambda f.\arg.(f.\arg)` | `var identity = function(x){ return x }` |
| `def select_first = \lambda first.second.first`  
`def select_second = \lambda first.second.second` | |
| `def cond= \lambda e1.e2.c.((c\ e1) e2)` | |
| `def true=select_first`  
`def false=select_second`  
`def not= \lambda x.(((cond\ false) true) x)`  
Or `def not= \lambda x.((x\ false) true)` | |
| `def and= \lambda x.y.(((cond\ y) false) x)`  
Or `def and= \lambda x.y.((x\ y) false)` | |
| `def or= \lambda x.y.(((cond\ true) y) x)`  
Or `def or= \lambda x.y.((x\ true) y)` | |
## Lambda Calculus

<table>
<thead>
<tr>
<th>Lambda Calculus</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x.x )</td>
<td>lambda x: x  - <strong>Test:</strong> (lambda x: x)(3)</td>
</tr>
<tr>
<td>( \lambda s. (s s) )</td>
<td>lambda func: func(func)</td>
</tr>
<tr>
<td></td>
<td><strong>Test:</strong> (lambda func: func(func))(lambda x: x)</td>
</tr>
<tr>
<td>( \lambda \text{func.arg.}(\text{func arg}) )</td>
<td>lambda func, arg: func(arg)</td>
</tr>
<tr>
<td></td>
<td><strong>Test:</strong> (lambda func, arg: func(arg))(lambda x: x, 4)</td>
</tr>
<tr>
<td>def identity = ( \lambda x.x )</td>
<td>identity = lambda x: x</td>
</tr>
<tr>
<td>def self_apply = ( \lambda s. (s s) )</td>
<td></td>
</tr>
<tr>
<td>def apply = ( \lambda \text{func.arg.}(\text{func arg}) )</td>
<td></td>
</tr>
<tr>
<td>def select_first = ( \lambda \text{first.second.first} )</td>
<td></td>
</tr>
<tr>
<td>def select_second = ( \lambda \text{first.second.second} )</td>
<td></td>
</tr>
<tr>
<td>def cond = ( \lambda e1.e2.c.((c e1) e2) )</td>
<td></td>
</tr>
<tr>
<td>def true = select_first</td>
<td></td>
</tr>
<tr>
<td>def false = select_second</td>
<td></td>
</tr>
<tr>
<td>def not = ( \lambda x.(((\text{cond false}) \text{true}) x) )</td>
<td>Or def not = ( \lambda x.(((x \text{false}) \text{true}) )</td>
</tr>
<tr>
<td>Or def and = ( \lambda x.y.(((\text{cond y}) \text{false}) x) )</td>
<td>Or def and = ( \lambda x.y.(((x y) \text{false}) )</td>
</tr>
<tr>
<td>Or def or = ( \lambda x.y.(((\text{cond true}) y) x) )</td>
<td>Or def or = ( \lambda x.y.(((x true) y) )</td>
</tr>
</tbody>
</table>
Lambda Calculus Arithmetic

def true = select_first
def false = select_second

def zero = λx.x
def succ = λn.λs.((s false) n)
def pred = λn.(((iszero n) zero) (n select_second))
def iszero = λn.((iszero n) zero) (n select_second)

def identity = λx.x
def self_apply = λs.(s s)
def apply = λfunc.λarg.(func arg)
def select_first = λfirst.λsecond.first
def select_second =λfirst.λsecond.second
def cond= λe1.λe2.λc.((c e1) e2)

For more but different details see Section 22.3 of the textbook.
Lambda Calculus Arithmetic

### ADDITION

```python
def addf = λf.λx.λy.
    if iszero y
    then x
    else f f (succ x) (pred y)
def add = λx.λy.
    if iszero y
    then x
    else addf addf (succ x) (pred y)

add one two
```

```python
(((λx.λy.
    if iszero y
    then x
    else addf addf (succ x) (pred y)) one) two)
```

```python
if iszero two
then one
else addf addf (succ one) (pred two)
```

```python
addf addf (succ one) (pred two)
```

```python
(((λf.λx.λy
    if iszero y
    then x
    else f f (succ x) (pred y)) addf) (succ one)) (pred two))
```

```python
if iszero (pred two)
then (succ one)
else addf addf (succ (succ one)) (pred (pred two))
```

```python
addf addf (succ (succ one)) (pred (pred two))
```

```python
(((λf.λx.λy
    if iszero y
    then x
    else f f (succ x) (pred y)) addf) (succ (succ one)) (pred (pred two)))
```

```python
if iszero (pred (pred two))
then (succ (succ one))
else addf addf (succ (succ (succ one))) (pred (pred (pred two)))
```

```python
(succ (succ one))
```

```python
three
```

### Multiplication

```python
def multf = λf.λx.λy.
    if iszero y
    then zero
    else add x (f x (pred y))
def recursive λf.(λs.(f (s s)) λs.(f (s s)))
def mult = recursive multf = λx.λy
    if iszero y
    then zero
    else add x (((λs.(multf (s s)) λs.(multf (s s))) x (pred y))
```

Church-Turing thesis: no formal language is more powerful than the lambda calculus or the Turing machine which are both equivalent in expressive power.
John McCarthy’s Takeaway

-- Primitive Recursive Functions on Lists are more interesting than PRFs on Numbers

\[
\text{prlen } x = y \ (b \ (\text{cond} \ ((==) \ []\) \ (k \ 0)) \ (b \ (s \ (b \ (+) \ (k \ 1))) \ (c \ b \ \text{cdr}))) \ x
\]
\[
\text{prsum } x = y \ (b \ (\text{cond} \ ((==) \ []\) \ (k \ 0)) \ (b \ (s \ (b \ (+) \ \text{car}))) \ (c \ b \ \text{cdr}))) \ x
\]
\[
\text{prprod } x = y \ (b \ (\text{cond} \ ((==) \ []) \ (k \ 1)) \ (b \ (s \ (b \ (*) \ \text{car}))) \ (c \ b \ \text{cdr}))) \ x
\]
\[
\text{prmap } f \ x = y \ (b \ (\text{cond} \ ((==) \ []) \ (k \ []))) \ (b \ (s \ (b \ (:) \ (f))) \ (c \ b \ \text{cdr}))) \ x
-- \text{prmap} (\lambda \ x \rightarrow (\text{car} \ x) + 2) \ [1,2,3] \text{ or}
-- \text{prmap} (\lambda \ x \rightarrow \text{pradd} \ (\text{car} \ x) \ 2) \ [1,2,3]
\]
\[
\text{prfoo } x = y \ (b \ (\text{cond} \ ((==) \ []\) \ (k \ []))) \ (b \ (s \ (b \ (:) \ (\text{cdr}))) \ (c \ b \ \text{cdr}))) \ x
\]

-- A programming language should have first-class functions as \( (b \ p_1 \ p_2 \ldots \ p_n) \), substitution, lists with \text{car}, \text{cdr} and \text{cons} operations and recursion.

\[
\text{car} \ (f:r) = f
\]
\[
\text{cdr} \ (f:r) = r
\]
\[
\text{cons} \text{ is : op}
\]

John’s 1960 paper: “Recursive Functions of Symbolic Expressions and Their Computation by Machine” – see class calendar.
Simple Lisp

LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.

I WONDER IF THE CYCLES WILL CONTINUE FOREVER.

A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.

THESE ARE YOUR FATHER’S PARENTHESES

ELEGANT WEAPONS

FOR A MORE... CIVILIZED AGE.

Alonzo Church

John McCarthy
Simple Lisp

Welcome to DrRacket, version 5.3.6 [3m].
Language: racket; memory limit: 128 MB.

> '(a b c)
'(a b c)

> (car '(a b c))
'a

> (cdr '(a b c))
'(b c)

> (cons 'a '(b c))
'(a b c)

> (let ((n (+ 1 2))) (* n 3))
9

> (letrec (((factorial (lambda (N) (if (= N 0) 1 (* N (factorial (- N 1))))))) (factorial 100))
93326215443944152681699238856266700490715968264381621468592963895217
59999322991560894146397615651828625369792082722375825118521091686400
0000000000000000000000
Last night I drifted off while reading a Lisp book.

Suddenly, I was bathed in a suffusion of blue.

At once, just like they said, I felt a great enlightenment. I saw the naked structure of Lisp code unfold before me.

My god, it's full of car's.

The patterns and metapatterns danced.
Syntax faded, and I swam in the purity of quantified conception. Of ideas manifest.

Truly, this was the language from which the gods wrought the universe.

No, it's not.

It's not?

I mean, ostensibly, yes. Honestly, we hacked most of it together with Perl.