Elements of Programming Languages

Genesis of Some Programming Languages
Can mathematics be shown to be consistent using formal methods?
<table>
<thead>
<tr>
<th>Notables in LinkedList History</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Aristotle</strong></td>
</tr>
<tr>
<td><strong>Euclid</strong></td>
</tr>
<tr>
<td><strong>Gottlob Frege</strong></td>
</tr>
<tr>
<td><strong>Giuseppe Peano</strong></td>
</tr>
<tr>
<td><strong>Georg Cantor</strong></td>
</tr>
<tr>
<td><strong>David Hilbert, Alfred Whitehead, and Bertrand Russell</strong></td>
</tr>
<tr>
<td><strong>Thoralf Skolem</strong></td>
</tr>
</tbody>
</table>
### Notables in LinkedList History

<table>
<thead>
<tr>
<th>Notable</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kurt Gödel</td>
<td><em>First Incompleteness Theorem</em></td>
</tr>
<tr>
<td></td>
<td>In any formal system capable of expressing primitive recursive functions/arithmetic the statement “I am a statement for which there is no proof” can be generated.</td>
</tr>
<tr>
<td></td>
<td><em>Second Incompleteness Theorem</em></td>
</tr>
<tr>
<td></td>
<td>No formal system capable of expressing primitive recursive functions/arithmetic can prove its own consistency.</td>
</tr>
<tr>
<td>Kurt Gödel</td>
<td><em>Certainty</em></td>
</tr>
<tr>
<td></td>
<td>Restored the notion of <em>Certainty</em> but in a manner very different from that envisioned by Hilbert, Whitehead and Russell.</td>
</tr>
<tr>
<td>Alonso Church</td>
<td><em>Recursive Functions</em></td>
</tr>
<tr>
<td></td>
<td>Functions which are defined for every input. <em>Infinity</em> returns.</td>
</tr>
<tr>
<td></td>
<td><em>Lambda Calculus</em></td>
</tr>
<tr>
<td></td>
<td>A language for defining functions and function application inspired by Gödel's recursive functions.</td>
</tr>
<tr>
<td></td>
<td><em>Undecidability of equivalence</em></td>
</tr>
<tr>
<td></td>
<td>There is no algorithm that takes as input two lambda expressions and determines if they are equivalent.</td>
</tr>
<tr>
<td>Alan Turing</td>
<td><em>Turing Machine and Halting Problem</em></td>
</tr>
<tr>
<td></td>
<td>Created a theoretical model for a machine, (Turing machine), that could carry out calculations from inputs. There is in general no way to tell if it will halt (<em>undecidability of halting</em>.)</td>
</tr>
<tr>
<td>Church-Turing <em>Thesis</em></td>
<td>Recursive Functions = Effectively Computable = Computational Completeness = calculable on a Turing machine = Turing-Complete (e.g., Lambda Calculus).</td>
</tr>
<tr>
<td></td>
<td><em>For more information see <a href="http://www.cs.utexas.edu/~ear/cs341/automatabook/">http://www.cs.utexas.edu/~ear/cs341/automatabook/</a></em></td>
</tr>
<tr>
<td>Haskell Currie</td>
<td><em>Combinator Logic</em></td>
</tr>
<tr>
<td></td>
<td>Developed a Turing-Complete language based solely upon function application of combinators.</td>
</tr>
</tbody>
</table>
## Notables in LinkedList History

<table>
<thead>
<tr>
<th>Name</th>
<th>Contribution</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>John McCarthy</td>
<td>The lisp programming language.</td>
<td>1960 paper: “Recursive Functions of Symbolic Expressions and Their Computation by Machine”, see class calendar for a copy.</td>
</tr>
</tbody>
</table>

McCarthy took concepts from Gödel's incompleteness proof (substitution), lambda calculus (function definition and function application) and combinator logic (car, cdr, and cons as primitive operations on **linked-lists**)

```lisp
(let ((l (cons 'a (cons 'b '())))) (let ((first (lambda (x) (car x)))) (first l)))
```
Cantor Set Theory

\[ A \cap B \cap C \]

\[ A \cap C \]

\[ A \cap B \cap C \]

\[ B \cap C \]

\[ A \]

\[ B \]

\[ C \]

\[ N = \text{the set of natural numbers} \]
\[ Q = \text{the set of rational numbers} \]
\[ R = \text{the set of real numbers} \]
\[ P = \text{the set of prime numbers} \]
\[ Z = \text{the set of integers} \]
\[ E = \text{the set of even integers} \]
\[ O = \text{the set of odd integers} \]
Cantor Diagonalization

<table>
<thead>
<tr>
<th>N</th>
<th>( \leftrightarrow )</th>
<th>( \text{reals in } (0,1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \leftrightarrow )</td>
<td>.835987…</td>
</tr>
<tr>
<td>2</td>
<td>( \leftrightarrow )</td>
<td>.250000…</td>
</tr>
<tr>
<td>3</td>
<td>( \leftrightarrow )</td>
<td>.559423…</td>
</tr>
<tr>
<td>4</td>
<td>( \leftrightarrow )</td>
<td>.500000…</td>
</tr>
<tr>
<td>5</td>
<td>( \leftrightarrow )</td>
<td>.728532…</td>
</tr>
<tr>
<td>6</td>
<td>( \leftrightarrow )</td>
<td>.845312…</td>
</tr>
<tr>
<td>…</td>
<td>( \leftrightarrow )</td>
<td>…</td>
</tr>
<tr>
<td>( n )</td>
<td>( \leftrightarrow )</td>
<td>( .r_1r_2r_3r_4r_5…r_n… )</td>
</tr>
<tr>
<td>…</td>
<td>( \leftrightarrow )</td>
<td>…</td>
</tr>
</tbody>
</table>

Create a new number from the diagonal by adding 1 and changing 10 to 0.
The above example would give .960143…
Now try to find a place for this number in the table above, it can’t be the first line because 8 \( \neq \) 9, it can’t be the second line because 5 \( \neq \) 6, etc. to infinity. So this line isn’t in the table above and therefore was not counted. \( \Rightarrow \) The real numbers are not countable.
Suppose there is a town with just one barber, who is male. In this town, every man keeps himself clean-shaven, and he does so by doing exactly one of two things: shaving himself; or going to the barber.

Another way to state this is that "The barber is a man in town who shaves all those, and only those, men in town who do not shave themselves."

From this, asking the question "Who shaves the barber?" results in a paradox because according to the statement above, he can either shave himself, or go to the barber (which happens to be himself). However, neither of these possibilities is valid: they both result in the barber shaving himself, but he cannot do this because he only shaves those men "who do not shave themselves".
Alright... Let’s go over the stages of the journey so far: 

1. MATHEMATICS MUST BE BASED ON LOGIC!
2. FREGÉ CREATES THE RIGHT LOGIC (BASED ON SETS)
3. I FIND PARADOX, i.e. “LOGIC IS FAULTY!”
4. WHITEHEAD & I MUST FIX IT (“PRINCIPIA”)

“2” made the quest possible... “3”, which marks my own entry, is the major crisis...

...And “4”, the struggle to overcome it!

So, what Whitehead and I were really doing, in building a paradox-free Logic that could support Mathematics, was...

...Fixing the hole I had exposed in Frege's ideas!
Principia Mathematica

*311·511. \( \vdash \text{Infin ax. } \xi \in C^\langle \Theta \rangle. Y \in C^\langle h \rangle. \exists \langle X \rangle. X \in \xi. Y +_g X \sim \epsilon \xi \)

[\( *311·51. \text{Transp} \)]

*311·52. \( \vdash \text{Infin ax. } \xi, \eta \in C^\langle \Theta \rangle. \exists \xi \Theta (\xi +_p \eta) \)

Dem.

\( \vdash *311·511. \exists \vdash : \text{Hp. } \exists : Y \in C^\langle h \rangle. \exists \langle X \rangle. X \in \xi. X +_g Y \sim \epsilon \xi : \)

[\( *311·11 \)]

\( \exists : \langle X, Y \rangle. X +_g Y \epsilon (\xi +_p \eta) = \xi : \)

[\( *310·11. *311·27 \)]

\( \exists : \xi \Theta (\xi +_p \eta) = \exists : \text{Prop} \)

*311·53. \( \vdash \text{Infin ax. } \xi, \eta \in C^\langle \Theta_n \rangle. \exists \xi \Theta_n (\xi +_p \eta) \)

[\( *311·52·33 \)]

*311·56. \( \vdash \text{Infin ax. } \xi \in C^\langle \Theta_g \rangle. \exists : \xi = \xi +_p \eta. \equiv : \eta = \iota \sigma_q \)

[\( *311·1·43·52·53 \)]

*311·57. \( \vdash \text{Infin ax. } \exists : \xi = \xi +_p \eta. \equiv : \xi = \Lambda . v . \xi \in C^\langle \Theta_g \rangle. \eta = \iota \sigma_q \)

[\( *311·56·1 \)]

*311·58. \( \vdash \text{Infin ax. } . \mu \in C^\langle \Theta \rangle. \exists : \mu = H^\langle \mu \rangle \)

[\( *304·3. *270·31 \)]

*311·6. \( \vdash \text{Infin ax. } . \mu \Theta_v. X, Y \epsilon v - \mu. XHY. M \epsilon \mu. \exists : M +_g (Y - x X) = v \)

Dem.

\( \vdash *310·11. \exists \vdash : \text{Hp. } \exists : MHX. \)

[\( *308·42·72 \)]

\( \exists : [M +_g (Y - x X)] HY \)

(1)

\( \vdash (1) . *311·58. \exists \vdash : \text{Prop} \)
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.
“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: \( 2x - 4 = 0 \)

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


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 row for statement as follows:

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

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

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

[2, 2, 2, 2, 3, 3, 3, 3, 5, 5, 7, 7, 7, 7, 11, 11, 13, 13, 13L]
[2, 2, 2, 2, 3, 3, 3, 3, 5, 5, 7, 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 $\text{Proves}(x, \text{Subst}(y, 17, y))$ returns true, i.e., $\text{Subst}(y, 17, y)$ 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:

- Gödel's Proof
- A Profile of Mathematical Logic
- Lambda-Calculus and Combinators: An Introduction

My Favorite

- From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931
- Gödel, Escher, Bach: An Eternal Golden Braid
- LISP 1.5 Programmer's Manual

Edited by Jean van Heijenoort
<table>
<thead>
<tr>
<th></th>
<th>Lambda Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\lambda x.x)</td>
</tr>
<tr>
<td>2</td>
<td>(\lambda s.(s\ s))</td>
</tr>
<tr>
<td>3</td>
<td>(\lambda\text{func.arg.}(\text{func arg}))</td>
</tr>
</tbody>
</table>
| 4 | def \(\text{id} = \lambda x.x\)  
   |   def \(\text{self}\_\text{apply} = \lambda s.(s\ s)\)  
   |   def \(\text{apply} = \lambda\text{func.arg.}(\text{func arg})\) |
| 5 | def \(\text{select\_first} = \lambda\text{first.second.first}\)  
   |   def \(\text{select\_second} = \lambda\text{first.second.second}\) |
| 6 | def \(\text{cond} = \lambda e1.e2.c.((c\ e1)\ e2)\) |
| 7 | def \(\text{true} = \text{select\_first}\)  
   |   def \(\text{false} = \text{select\_second}\)  
   |   def \(\text{not} = \lambda x.(((\text{cond}\ false)\ true)\ x)\)  
   |   Or def \(\text{not} = \lambda x.(((x\ false)\ true)\ x)\) |
| 8 | def \(\text{and} = \lambda y.(((\text{cond}\ y)\ false)\ x)\)  
   |   Or def \(\text{and} = \lambda y.(((x\ y)\ false)\ x)\) |
| 9 | def \(\text{or} = \lambda y.(((\text{cond}\ true)\ y)\ x)\)  
<p>|   Or def (\text{or} = \lambda y.(((x\ true)\ y)\ x)) |</p>
<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>
<tr>
<td>$\lambda s.(s\ s)$</td>
<td><code>function(func){return func(func)}</code></td>
</tr>
<tr>
<td></td>
<td><strong>Test:</strong> <code>function(func){return func(func)}</code></td>
</tr>
<tr>
<td></td>
<td>(<code>function(x){return x};</code>)</td>
</tr>
<tr>
<td>$\lambda func.arg.(func arg)$</td>
<td><code>function(func, arg){return func(arg)}</code></td>
</tr>
<tr>
<td></td>
<td><strong>Test:</strong> <code>function(func, arg){return func(arg)}</code></td>
</tr>
<tr>
<td></td>
<td>(<code>function(x){return x}; 'a'</code>)</td>
</tr>
<tr>
<td>def identity = $\lambda x.x$</td>
<td>var identity = <code>function(x){ return x }</code></td>
</tr>
<tr>
<td>def self_apply = $\lambda s.(s\ s)$</td>
<td></td>
</tr>
<tr>
<td>def apply = $\lambda func.arg.(func arg)$</td>
<td></td>
</tr>
<tr>
<td>def select_first = $\lambda first.second.first$</td>
<td></td>
</tr>
<tr>
<td>def select_second = $\lambda first.second.second$</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= $\lambda x.(((cond false) true) x)$</td>
<td></td>
</tr>
<tr>
<td>Or def not= $\lambda x.((x false) true)$</td>
<td></td>
</tr>
<tr>
<td>def and= $\lambda x.y.(((cond y) false) x)$</td>
<td></td>
</tr>
<tr>
<td>Or def and= $\lambda x.y.((x y) false)$</td>
<td></td>
</tr>
<tr>
<td>def or= $\lambda x.y.(((cond true) y) x)$</td>
<td></td>
</tr>
<tr>
<td>Or def or= $\lambda x.y.((x true) y)$</td>
<td></td>
</tr>
<tr>
<td>Lambda Calculus</td>
<td>Python</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
</tr>
<tr>
<td>( \lambda x.x )</td>
<td><code>lambda x: x</code> - <strong>Test:</strong> (lambda x: x)(3)</td>
</tr>
</tbody>
</table>
| \( \lambda s.(s s) \) | `lambda func: func(func)`
**Test:** (lambda func: func(func))(lambda x: x) |
| \( \lambda \text{func} . \text{arg} . (\text{func} \ \text{arg}) \) | `lambda func, arg: func(arg)`
**Test:** (lambda func, arg: func(arg))(lambda x: x, 4) |
| `def identity = \lambda x.x`  
`def self_apply = \lambda s.(s s)`  
`def apply = \lambda \text{func} . \text{arg} . (\text{func} \ \text{arg})` | `identity = lambda x: x` |
| `def select_first = \lambda \text{first} . \text{second} . \text{first}`  
`def select_second = \lambda \text{first} . \text{second} . \text{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 Arithmetic**

```python
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.((n select_first)

one = (succ zero)
    (λn.λs.((s false) n) zero)
    λs.((s false) zero)

two = (succ one)
    (λn.λs.((s false) n) λs.((s false) zero))
    λs.((s false) λs.((s false) zero))

three = (succ two)
    (λn.λs.((s false) n) λs.((s false) λs.((s false) zero)))
    λs.((s false) λs.((s false) λs.((s false) zero)))

(iszero zero)
(λn.(n select_first) λx.x)
(λx.x select_first)
select_first
```

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

**Multiplication**

```python
def multif = 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 multif = 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.
A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.
John McCarty’s Takeaway

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

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

-- A programming language should have first-class functions as (b p1 p2 . . . Pn), substitution, lists with car, cdr and cons operations and recursion.

\[
\begin{align*}
\text{car} (f:r) & \quad = \quad f \\
\text{cdr} (f:r) & \quad = \quad r \\
\text{cons} & \quad \text{is} \quad \text{: op}
\end{align*}
\]

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

Dr. Philip Cannata

David Hilbert, Jules Richard, G. G. Berry, Georg Cantor, Bertrand Russell, Kurt Gödel, Alan Turing

Alonzo Church

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.

John McCarthy

THESE ARE YOUR FATHER’S PARENTHESIS.

ELEGANT WEAPONS

FOR A MORE... CIVILIZED AGE.
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.

Huh?

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.

TRUELY, 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.