Programming Languages

Genesis of Some Programming Languages
(My kind of Fiction)
This Course

Java (Object Oriented)

Jython in Java

High Level Languages

Relation

A Snapshot of Programming Language History

Dr. Philip Cannata
<table>
<thead>
<tr>
<th>Concept</th>
<th>Notes</th>
<th>Concept</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>HLL1 &amp; HLL2</td>
<td>Functions</td>
<td>HLL4</td>
</tr>
<tr>
<td>Terminal</td>
<td>HLL1 &amp; HLL2</td>
<td>Parameters and Arguments</td>
<td>HLL4</td>
</tr>
<tr>
<td>Non-terminal</td>
<td>HLL1 &amp; HLL2</td>
<td>Parameter Passing</td>
<td>HLL4</td>
</tr>
<tr>
<td>Lexical Analysis</td>
<td>HLL1 &amp; HLL2</td>
<td>Activation Records</td>
<td>HLL4</td>
</tr>
<tr>
<td>Regular Expression</td>
<td>HLL1 &amp; HLL2</td>
<td>Runtime Stack</td>
<td>HLL4</td>
</tr>
<tr>
<td>Syntactic Analysis</td>
<td>HLL1 &amp; HLL2</td>
<td>Side Effects</td>
<td>HLL4</td>
</tr>
<tr>
<td>Regular Grammar</td>
<td>HLL1 &amp; HLL2</td>
<td>First-Class Functions</td>
<td>HLL4</td>
</tr>
<tr>
<td>Context-free Grammar</td>
<td>HLL1 &amp; HLL2</td>
<td>Recursion</td>
<td>HLL4</td>
</tr>
<tr>
<td>BNF</td>
<td>HLL1 &amp; HLL2</td>
<td>Closure</td>
<td>HLL4</td>
</tr>
<tr>
<td>EBNF</td>
<td>HLL1 &amp; HLL2</td>
<td>Tail Recursion</td>
<td>HLL4</td>
</tr>
<tr>
<td>Parse Tree</td>
<td>HLL1 &amp; HLL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Associativity</td>
<td>HLL1 &amp; HLL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precedence</td>
<td>HLL1 &amp; HLL2</td>
<td>Haskell</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Ambiguous Grammar</td>
<td>HLL1 &amp; HLL2</td>
<td>Haskell Expression</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Abstract Syntax</td>
<td>HLL1 &amp; HLL2</td>
<td>Haskell Functions</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Abstract Syntax Tree</td>
<td>HLL1 &amp; HLL2</td>
<td>Haskell Tuples</td>
<td>Haskell1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Haskell Lists</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Interpretation</td>
<td></td>
<td>Haskell Pattern Matching</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Substitution</td>
<td>PLAI 2</td>
<td>Chapter 3</td>
<td></td>
</tr>
<tr>
<td>Symbol Table and Dictionaries</td>
<td>HLL3 &amp; PLAI 3</td>
<td>Lazy Evaluation</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Scope</td>
<td>HLL3 &amp; PLAI 3</td>
<td>Strong Type System</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Nested and Dynamic Scope</td>
<td>HLL3 &amp; PLAI 3</td>
<td>Propositional Logic</td>
<td>Haskell2</td>
</tr>
<tr>
<td>Local and Non-Local Variables</td>
<td>HLL3 &amp; PLAI 3</td>
<td>Haskell and SQL</td>
<td>Haskell2</td>
</tr>
<tr>
<td>Static and Dynamic Scoping</td>
<td>HLL3 &amp; PLAI 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolving References</td>
<td>HLL3 &amp; PLAI 3</td>
<td>Syllogisms</td>
<td>Syllogism and Midter Review</td>
</tr>
<tr>
<td>Overloading</td>
<td>HLL3 &amp; PLAI 3</td>
<td>Proof by Contradiction</td>
<td>Syllogism and Midter Review</td>
</tr>
<tr>
<td>Variables</td>
<td>HLL3 &amp; PLAI 3</td>
<td>Godf's Incompleteness Theorem</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Static and Dynamic Binding</td>
<td>HLL3 &amp; PLAI 3</td>
<td>Deductive Logic</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Types</td>
<td></td>
<td>Horn Clause</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Type System</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Prolog</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Static and Dynamic Typing</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Prolog Facts and Rules</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Type System</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Prolog Lists</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Implicit Type Conversion</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Proof by Contradiction</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Type Judgments</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Resolutions</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Type Judgment Tree</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Backtracking</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Type Soundness</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Constraint Satisfaction Programming</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Type Safety</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Prolog and SQL</td>
<td>Prolog 1 &amp; 2</td>
</tr>
<tr>
<td>Type Inferencing</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relations</td>
<td>PLAI 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primitive Recursive Functions</td>
<td>Genesis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lambda Calculus</td>
<td>PLAI 1</td>
<td>Chapter 22</td>
<td></td>
</tr>
<tr>
<td>Combinator Logic</td>
<td>Genesis</td>
<td>Compressions</td>
<td>Prolog 1, Page 9</td>
</tr>
<tr>
<td>Lisp</td>
<td>PLAI 1</td>
<td>Kolmogorov Complexity</td>
<td></td>
</tr>
</tbody>
</table>

Dr. Philip Cannata
Can mathematics be shown to be consistent using formal methods?
Relations and Functions

Relations:
A Relation is a subset of the cross-product of a set of domains.

Functions:
An n-ary relation R is a function if the first n-1 elements of R are the function’s arguments and the last element is the function’s results and whenever R is given the same set of arguments, it always returns the same results. [Notice, this is an unnamed function!].
In the Beginning

(p b)
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.

In a small town there is only one barber. This man is defined to be the one who shaves all the men who do not shave themselves. The question is then asked, 'Who shaves the barber?'

If the barber doesn't shave himself, then -- by definition -- he does.

And, if the barber does shave himself, then -- by definition -- he does not.

or

Consider the statement 'This statement is false.'

If the statement is false, then it is true; and if the statement is true, then it is false.
Thoralf Skolem “The foundations of elementary arithmetic by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domains” (1919, 1923).

He assumed that the following notions are already understood:

- natural number (0 is included in the book),
- successor of x (i.e., x’),
- substitution of equals (if x = y and y = z, then x = z),
- and the recursive mode of thought (see next page).
## The addition function

<table>
<thead>
<tr>
<th>Book notation</th>
<th>Relation notation</th>
<th>Arithmetic notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x, 0) = x )</td>
<td>((x \ 0 \ x))</td>
<td>(x + 0 = x)</td>
</tr>
<tr>
<td>( f(x, y') = (f(x, y))' )</td>
<td>((x \ y' \ (x \ y \ b)'))</td>
<td>(x + y' = (x + y)')</td>
</tr>
</tbody>
</table>

**Example 2 + 3** (The similar example of 7 + 5 on pages 153-154 is actually a bit flawed as a comparison with the following will show):

\[
\begin{align*}
2 + 0 &= 2 \\
2 + 1 &= 3 \\
2 + 2 &= 4 \\
2 + 3 &= 5
\end{align*}
\]
“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]
Primitive Recursive Functions and Arithmetic
(see “A Profile of Mathematical Logic” by Howard DeLong – pages 152 – 160)

A Sequence of Functions from definitions on DeLong, page 157:

<table>
<thead>
<tr>
<th>Book notation</th>
<th>Relation notation</th>
<th>Arithmetic notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f_1(x) = x')</td>
<td>((x (+ x 1)))</td>
<td>(f_1) is the successor function</td>
</tr>
<tr>
<td>(f_2(x) = x)</td>
<td>((x x))</td>
<td>(f_2) is the identity function with (i = 1)</td>
</tr>
<tr>
<td>(f_3(y, z, x) = z)</td>
<td>((y z x z))</td>
<td>(f_3) is the identity function with (i = 2)</td>
</tr>
<tr>
<td>(f_4(y, z, x) = f_1(f_3(y,z,x)))</td>
<td>((y z x ((x (+ x 1)) (y z x z))))</td>
<td>(f_4) is defined by substitution for (f_1) and (f_3)</td>
</tr>
</tbody>
</table>

This is how you would do this in lisp

```
(let ((f1 (lambda (x) (+ x 1)))
      (f2 (lambda (x) x))
      (f3 (lambda (y z x) z)))
  (let ((f4 (lambda (y z x) (f1 (f3 y z x))))))
  (f4 2 4 6))
```

\(f_5(0, x) = f_2(x)\) \hspace{2cm} \(0 x (x x)\) \hspace{2cm} \(f_5\) is defined by recursion and \(f_2\) and \(f_4\)

\(f_5(y', x) = f_4(y, f_5(y,x), x)\) \hspace{2cm} \text{(not doable yet)}

\(f_5\) is primitive recursive addition

```
(let ((f1 (lambda (x) (+ x 1)))
      (f2 (lambda (x) x))
      (f3 (lambda (y z x) z)))
  (let ((f4 (lambda (y z x) (f1 (f3 y z x))))))
  (letrec ((f5 (lambda (a b) (if (= a 0) (f2 b) (f4 (- a 1) (f5 (- a 1) b) b))))))
  (f5 2 3))
```
Primitive Recursive Relations on DeLong, pages 158-159:

Example of eq primitive recursive relation:

(letrec ((pd (lambda (x) (if (= x 0) 0 (- x 1)))))
  (dm (lambda (x y) (if (= y 0) x (pd (dm x (pd y)))))))
  (abs (lambda (x y) (+ (dm x y)(dm y x))))
  (sg (lambda (x) (if (= x 0) 0 1)))
  (eq (lambda (x y) (sg (abs x y)))) (eq 1 1))
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 Numbering

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>17</th>
<th>19</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>'0'</td>
<td>''</td>
<td>'-'</td>
<td>'=&gt;'</td>
<td>'V'</td>
<td>'('</td>
<td>')'</td>
<td>'x'</td>
<td>'y'</td>
<td>'z'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>29</th>
<th>31</th>
<th>37</th>
<th>41</th>
<th>43</th>
<th>47</th>
<th>53</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>'='</td>
<td>'+'</td>
<td>'.'</td>
<td>'x_1'</td>
<td>'y_1'</td>
<td>'z_1'</td>
<td>'z_2'</td>
<td>...</td>
</tr>
</tbody>
</table>

1 = (0)’ = 2^{11} \times 3^1 \times 5^{13} \times 7^3

The following proof would be a sequence of symbols which would correspond to a single Gödel number (see DeLong page 167 for another example)

(0”’ 0 0”’)

(0”’ 0’ (0”’ 0 x ’))

=> (0”’ 0’ (0’’)) => (0”’ 0’ 0”’)}
Gödel's Incompleteness Theorem

If “proof” is a proof of “statement” then P is True.

If you have a statement g with variable x and if, when you substitute g for x, you produce “statement” then Q is True.

\[
\text{not P(proof, statement) } \&\& \text{ Q(x, statement) } = \text{ g}
\]

Not P(proof, statement) \&\& Q(g, statement) = s

Let g be the Gödel number for this statement.

Let s be the Gödel number for this statement but by the definition of Q that means "statement" is "s".

\[
\text{not P(proof, s) } \&\& \text{ Q(g, s) } - \text{ I am a statement that is not provable.}
\]

\[\rightarrow \text{There are Predicate Logic Statements that are True that can’t be proved True (Incompleteness) and/or there are Predicate Logic Statements that can be proved True that are actually False (\rightarrow Inconsistent Axioms or Unsound inference rules).}\]

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.
Gödel's Incompleteness Theorem

I am a statement that is not provable.

There are Predicate Logic Statements that are True that can’t be proved True (Incompleteness) and/or there are Predicate Logic Statements that can be proved True that are actually False (Inconsistent Axioms or Unsound inference rules).

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.

Logic/Math/CS

Unsound

S

F

T

Opposite is Excluded Middle

~p or p

Physics

Superposition

L

P

W

Theology

Consustantial

G

F

S

H

Plotinus

The ONE

Is nothing else but The ONE, it can’t even be finite.

Philosophy

The Forms (e.g. Justice)

Self

Trace of

Other

Finite

Plato

Dr. Philip Cannata
Good Books to Have for a Happy Life 😊

From Frege to Gödel:

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

Frege, Peano, Dedekind, Russell, Carnap, Tarski, Hilbert, von Neumann, Ackermann, von Mises, Gödel

Edited by Jean van Heijenoort

Lambda-Calculus and Combinators
An Introduction

Lambda-Calculus and Combinators
An Introduction

A Profile of Mathematical Logic
Howard DeLong

Kurt Gödel
On Formally Undecidable Propositions of Principia Mathematica and Related Systems

Gödel’s Proof

LISP 1.5 Programmer’s Manual

My Favorite
A lambda expression is a particular way to define a function:

\[
\text{LambdaExpression} \rightarrow \text{variable} \mid (M \ N) \mid (\lambda \text{variable} . \ M)
\]

\[
M \rightarrow \text{LambdaExpression}
\]

\[
N \rightarrow \text{LambdaExpression}
\]

E.g., \((\lambda x . (sq x))\) represents applying the \textit{square} function to \(x\).
A little Bit of Lambda Calculus – Properties of Lambda Expressions

In \((\lambda x . M)\), \(x\) is **bound**. Other variables in \(M\) are **free**. A substitution of \(N\) for all occurrences of a variable \(x\) in \(M\) is written \(M[x ← N]\). Examples:

\[
\begin{align*}
x[x ← y] &= y \\
(xx)[x ← y] &= (yy) \\
(zw)[x ← y] &= (zw) \\
(zx)[x ← y] &= (zy) \\
(\lambda x · (zx))[x ← y] &= (\lambda u · (zu))[x ← y] = (\lambda u · (zu)) \\
(\lambda x · (zx))[y ← x] &= (\lambda u · (zu))[y ← x] = (\lambda u · (zu))
\end{align*}
\]

• An **alpha-conversion** allows bound variable names to be changed. For example, alpha-conversion of \(\lambda x . x\) might yield \(\lambda y . y\).

• A **beta reduction** ((\(\lambda x . M\))\(N\)) of the lambda expression (\(\lambda x . M\)) is a substitution of all bound occurrences of \(x\) in \(M\) by \(N\). E.g.,

\[
(\(\lambda x . x^2\)) 5 = 5^2
\]
### 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>$\lambda \text{func}. \lambda \text{arg}. (\text{func}\ \text{arg})$</td>
</tr>
</tbody>
</table>

```python
def identity = $\lambda x. x$
def self_apply = $\lambda s. (s\ s)$
def apply = $\lambda \text{func}. \lambda \text{arg}. (\text{func}\ \text{arg})$
```

def select_first = $\lambda \text{first}. \lambda \text{second}. \text{first}$
def select_second = $\lambda \text{first}. \lambda \text{second}. \text{second}$

def cond = $\lambda e_1. \lambda e_2. \lambda c. ((c\ e_1)\ e_2)$

def true = select_first
def false = select_second
def not = $\lambda x. (((\text{cond}\ \text{false})\ \text{true})\ x)$
Or def not = $\lambda x. (((x\ \text{false})\ \text{true})$}

def and = $\lambda x. \lambda y. (((\text{cond}\ y)\ \text{false})\ x)$
Or def and = $\lambda x. \lambda y. (((x\ y)\ \text{false})$}

def or = $\lambda x. \lambda y. (((\text{cond}\ \text{true})\ y)\ x)$
Or def or = $\lambda x. \lambda y. (((x\ \text{true})\ y)$}
Lambda Calculus Substitution

In lambda calculus, if cond is defined as \( \text{def cond} = \lambda e1.\lambda e2.\lambda c.((c e1) e2) \),
\( \text{def and} = \lambda x.\lambda y.(((\text{cond } y) \text{ false}) x) \)
is equivalent to
\( \text{def and} = \lambda x.\lambda y.((x y) \text{ false}) \) because:

\[
(((\text{cond } y) \text{ false}) x) \\
(((\lambda e1.\lambda e2.\lambda c.((c e1) e2) y) \text{ false}) x) \\
((\lambda e2.\lambda c.((c y) e2) \text{ false}) x) \\
(\lambda c.((c y) \text{ false}) x) \\
((x y) \text{ false})
\]

In lambda calculus, if cond is defined as \( \text{def cond} = \lambda e1.\lambda e2.\lambda c.((c e1) e2) \),
\( \text{def or} = \lambda x.\lambda y.(((\text{cond } \text{ true}) y) x) \)
is equivalent to
\( \text{def or} = \lambda x.\lambda y.((x \text{ true}) y) \) because:

\[
(((\text{cond } \text{ true}) y) x) \\
(((\lambda e1.\lambda e2.\lambda c.((c e1) e2) \text{ true}) y) x) \\
((\lambda e2.\lambda c.((c \text{ true}) e2) y) x) \\
(\lambda c.((c \text{ true}) y) x) \\
((x \text{ true}) y)
\]
Lambda Calculus as Function Relations

Remember I said a function is a relation written as follows (param1 param2 ... body)? If we restrict ourselves to functions that only take one argument, this would be (param body). Also, function application could be written as (param body)(arg). Using this, we can re-write the following lambda expressions and applications as follows:

\((\lambda x.x \lambda x.x)\) i.e., apply \(\lambda x.x\) to itself
\((x\ x)\ (x\ x)\)
\((x\ x)\ \leftarrow\ \text{a function is returned}\)

\((\lambda s.(s\ s) \lambda x.x)\) i.e., apply \(\lambda s.(s\ s)\) to \(\lambda x.x\)
\((s\ (s\ s))\ (x\ x)\)
\(((x\ x)\ (x\ x))\)
\((x\ x)\ \leftarrow\ \text{a function is returned}\)

\((\lambda s.(s\ s) \lambda s.(s\ s))\) i.e., apply \(\lambda s.(s\ s)\) to itself
\((\lambda s\ (s\ s)\ lambda\ (s\ s))\)
\((\lambda s\ (s\ s)\ lambda\ (s\ s))\ \leftarrow\ \text{a function application is returned}\)

\(((\lambda\ func.(\lambda\ arg.(\lambda\ x.x)\ lambda\ (s\ s))\ i.e.,\ apply\ the\ "function\ application\ function"\ to\ \lambda x.x\ and\ \lambda s.(s\ s))\)
\((\lambda\ func\ (\lambda\ arg)\ ((x\ x)\ (s\ (s\ s))))\)
\((\lambda\ arg\ ((x\ x)\ arg))\ (s\ (s\ s))\)
\(((x\ x)\ (s\ (s\ s)))\)
\((s\ (s\ s))\ \leftarrow\ \text{a function is returned}\)

So, in reality, the "function application function" which looks like it takes 2 arguments really is a function that consumes one argument and returns a function which consumes the second argument.
Lambda Calculus Arithmetic

```python
def true = select_first
def false = select_second

def zero = \lambda x. x
def succ = \lambda n. \lambda s. ((s false) n)
def pred = \lambda n. (((iszero n) zero) (n select_second))
def iszero = \lambda n. ((n select_first)

one = (succ zero)
  (\lambda n. \lambda s. ((s false) n) zero)
  \lambda s. ((s false) zero)

two = (succ one)
  (\lambda n. \lambda s. ((s false) n) \lambda s. ((s false) zero))
  \lambda s. ((s false) \lambda s. ((s false) zero))

three = (succ two)
  (\lambda n. \lambda s. ((s false) n) \lambda s. ((s false) \lambda s. ((s false) zero)))
  \lambda s. ((s false) \lambda s. ((s false) \lambda s. ((s false) zero)))

(iszero zero)
  (\lambda n. (n select_first) \lambda x. x)
  (\lambda x. x select_first)
select_first

(iszero one)
  (\lambda n. (n select_first) \lambda s. ((s false) zero) )
  (\lambda s. ((s false) zero) select_first)
  ((select_first false) zero)
```

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))
```

**Multiplication**

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

\[
\begin{align*}
\text{s f g x} &= f x (g x) \\
\text{k x y} &= x \\
\text{b f g x} &= f (g x) \\
\text{c f g x} &= f x g \\
\text{y f} &= f (y f) \\
\text{cond p f g x} &= \text{if } p x \text{ then } f x \text{ else } g x -- \text{ Some Primitive Recursive Functions on Natural Numbers}
\end{align*}
\]
John McCarthy’s Takeaway

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

\[
\text{prlen x} = y \ (b \ (\text{cond} \ ((==) \ []) \ (k \ 0)) \ (b \ (s \ (b \ (+) \ (k \ 1))) \ (c \ b \ \text{cdr}))) \ x
\]

\[
\text{prsum x} = y \ (b \ (\text{cond} \ ((==) \ []) \ (k \ 0)) \ (b \ (s \ (b \ (+) \ \text{car}))) \ (c \ b \ \text{cdr}))) \ x
\]

\[
\text{prprod x} = y \ (b \ (\text{cond} \ ((==) \ []) \ (k \ 1)) \ (b \ (s \ (b \ (*) \ \text{car}))) \ (c \ b \ \text{cdr}))) \ x
\]

\[
\text{prmap f x} = y \ (b \ (\text{cond} \ ((==) \ []) \ (k \ [])) \ (b \ (s \ (b \ (:f) \ ) \ (c \ b \ \text{cdr}))) \ x \quad \quad \text{-- prmap} \ (\lambda \ x \to (\text{car} \ x) + 2) \ [1,2,3] \ \text{or}
\]

\[
\text{prfoo x} = y \ (b \ (\text{cond} \ ((==) \ []) \ (k \ [])) \ (b \ (s \ (b \ (:c) \ \text{cdr}))) \ (c \ b \ \text{cdr}))) \ x
\]

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

\[
\text{car (f:r)} = f
\]

\[
\text{cdr (f:r)} = r
\]

\[
\text{cons is : op}
\]

---

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-DISCLOYERING THE LISP ARTS.

THESE ARE YOUR FATHER’S PARENTHESES

ELEGANT WEAPONS

FOR A MORE... CIVILIZED AGE.

Alonzo Church

John McCarthy
This is a very interesting book by Gregory Chaitin! It has to do with “Algorithmic Information Theory” (Information Compression and Randomness) (also known as “Minimum Description Length”) which I think is a very interesting topic. There is a small section on lisp that I’d like you to read (i.e., pages 38 – 44 of the pdf version). DrScheme code that goes along with the reading starts on the next slide. And, if you like, you can read the entire book to feed your intellectual curiosity :-) .
Simple Lisp in Scheme

**Code for Chaitin page 40**

(if true (+ 1 2) (+ 3 4))

⇒ 3

(if false (+ 1 2) (+ 3 4))

⇒ 7

**Code for Chaitin page 41**

Instead of (` (a b c)) ⇒ (a b c)

'( a b c )

⇒ (list 'a 'b 'c)

(if (= 23 32) true false)

⇒ False

(if (= (list 1 2 3) (list 1 2 3)) true false)

⇒ . . =: expects type <number> as 1st argument, given: (list 1 2 3); other arguments were: (list 1 2 3)

Instead of (if (atom ...

(if (list? (list 1 2 3)) true false)

⇒ true

(if (list? 21) true false)

⇒ false

(if (list? 'a) true false)

⇒ false
Simple Lisp in Scheme

**Code for Chaitin page 41 continued**

Instead of \((\text{let } n (+ 1 2) (* n 3))\)

\[
(\text{let } ((n (+ 1 2))) (* n 3)) \\
\rightarrow 9
\]

Instead of \((\text{let } (f n) (* n n) (f 10))\) – see Scheme’s definition of “let” in the Scheme Tutorial at


\[
(\text{let } ((f \text{ (lambda (n) (* n n)))) (f 10)) \\
\rightarrow 100
\]

**Code for Chaitin page 42**

Instead of \((\text{car } ‘(a b c))\)

\[
(\text{car } ‘(a b c)) \\
\rightarrow ‘a
\]

Instead of \((\text{cdr } ‘(a b c))\)

\[
(\text{cdr } ‘(a b c)) \\
\rightarrow (\text{list } ‘b ‘c)
\]

Instead of \((\text{cons } ‘a (‘(b c)))\)

\[
(\text{cons } ‘a (‘(b c))) \\
\rightarrow (\text{list } ‘a ‘b ‘d)
\]
Simple Lisp in Scheme

Code for Chaitin page 43

Instead of (let (factorial N) (if (= N 0) 1 (* N (factorial (- N 1)))) (factorial 5)) – see Scheme’s definition of “letrec” in the Scheme Tutorial at http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-8.html#node_idx_288

(letrec ((factorial (lambda (N) (if (= N 0) 1 (* N (factorial (- N 1))))))) (factorial 5))
→ 120

(letrec ((factorial (lambda (N) (if (= N 0) 1 (* N (factorial (- N 1))))))) (factorial 100))
→ 9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000

More interesting code:

(letrec ((first (lambda (List) (if (null? List) (list) (car List)))) (first (list 1 2 3)))
(letrec ((rest (lambda (List) (if (null? List) (list) (cdr List)))) (rest (list 1 2 3)))
(letrec ((sum-list (lambda (List) (if (null? List) 0 (+ (car List) (sum-list (cdr List)))))) (sum-list (list 1 2 3)))
(letrec ((nth (lambda (N List) (if (not (= N 0))(nth (- N 1) (cdr List))(car List)) ) (nth 2 (list 1 2 3)))
(letrec ((head (lambda (N List) (if (= N 0) (list) (cons (car List) (head (- N 1) (cdr List)))))) (head 3 (list 1 2 3 4 5)))

Dr. Philip Cannata
Simple Lisp in Scheme

(letrec ( (first (lambda (List) (if (null? List) (list) (car List)))))
    (sum-list (lambda (List) (if (null? List) 0 (+ (car List) (sum-list (cdr List)))))))
    (nth (lambda (N List) (if (not (= N 0)) (nth (- N 1) (cdr List))(car List))))
    (head (lambda (N List) (if (= N 0) (list) (cons (car List) (head (- N 1) (cdr List)))))))

(nth 1 (list 1 2 3)))
→ 2

(letrec ( (List (list 1 2 3 4 5 6))
    (first (lambda (List) (if (null? List) (list) (car List))))
    (sum-list (lambda (List) (if (null? List) 0 (+ (car List) (sum-list (cdr List)))))))
    (nth (lambda (N List) (if (not (= N 0)) (nth (- N 1) (cdr List))(car List))))
    (head (lambda (N List) (if (= N 0) (list) (cons (car List) (head (- N 1) (cdr List)))))))

(head (nth 1 List) List)
→ (list 1 2)

**Code for Chaitin page 43 - 44**

(letrec ( (map (lambda (Function List) (if (null? List) List (cons (Function (car List)) (map Function (cdr List)))))))
    (factorial (lambda (N) (if (= N 0) 1 (* N (factorial (- N 1)))))))

(map factorial (list 4 1 2 3 5)))
→ (list 24 1 2 6 120)

**Define statement:**

(define nth (lambda (N List) (if (not (= N 0)) (nth (- N 1) (cdr List))(car List))))

(nth 2 (list 1 2 3 4 5))
→ 3
A little Bit of Lambda Calculus – Y Combinator in Scheme

Are these really the same?

(letrec ((factorial (lambda (N) (if (= N 0) 1 (* N (factorial (- N 1))))))) (factorial 50))

→ 30414093201713378043612608166064768844377641568960512000000000000000

-----------------------------------------------------------------------------------------------------------------------------------------

(lambda (X) 
  (lambda (procedure) 
    (X (lambda (arg) ((procedure procedure) arg)))))

(lambda (procedure) 
  (X (lambda (arg) ((procedure procedure) arg)))))

(lambda (func-arg) 
  (lambda (n) 
    (if (zero? n) 
      1 
      (* n (func-arg (- n 1))))) ) )

Y Combinator (red) which is applied to a function (blue)

( ( ( lambda (X) 
      ( (lambda (procedure) 
        (X (lambda (arg) ((procedure procedure) arg))))))) 
  ( lambda (procedure) 
    (X (lambda (arg) ((procedure procedure) arg))))) ) )

For more details see Section 22.4 of the textbook.

(define make-recursive-procedure
  (lambda (p) 
    ((lambda (f) 
      (f f ))
     (lambda (f) 
       (p (f f))))))
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.

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.

I MEAN, OSTEBSLBY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.

IT'S NOT?
Scheme for the Textbook

PLT Scheme is a Racket

Sure, it has parentheses, uses the keyword `lambda`, provides lexical scope, and emphasizes macros — but don't be fooled. PLT Scheme is no minimalist embodiment of 1930s math or 1970s technology. PLT Scheme is a cover for a gang of academic hackers who want to fuse cutting-edge programming-language research with everyday programming. They draw you in with the promise of a simple and polite little Scheme, but soon you'll find yourself using modules, contracts, keyword arguments, classes, static types, and even curly braces.

Racket is a Scheme

Racket is still a dialect of Lisp and a descendant of Scheme. The tools developed by PLT will continue to support R5RS, R6RS, the old `mzscheme` environment, Typed Scheme, and more. At the same time, instead of having to say "PLT's main variant of Scheme," programmers can now simply say "Racket" to refer to the specific descendant of Scheme that powers PLT's languages and libraries.

Anticipated Questions

Why change the name?

The `Scheme` part of the name `PLT Scheme` is misleading, and it is often an obstacle to explaining and promoting PLT research and tools.

For example, when you type "scheme" into Google, the first hit is a Wikipedia entry written from an R5RS perspective. That's appropriate for a Wikipedia page on Scheme, but it's not a good introduction to PLT Scheme. As long as we call our language `Scheme`, we struggle to explain our language, and we are usually forced to start the explanation with a disclaimer.

http://racket-lang.org/new-name.html
Scheme for the Textbook

Racket is a programming language.

Start Quickly

```
#lang racket
;; Finds Racket sources in all subdirs
(for ([path (in-directory)])
  (when (regexp-match? #ix"[:\r\n]rct$" path)
    (printf "source file: \"\a\n\" path))
```

Grow your Program

Racket’s interactive mode encourages experimentation, and quick scripts easily compose into larger systems. Small scripts and large systems both benefit from native-code JIT compilation. When a system gets too big to keep in your head, you can add static types.

Grow your Language

Extend Racket wherever you need to. Mold it to better suit your tasks without sacrificing interoperability with existing libraries and without having to modify the toolchain. When less is more, you can remove parts of a language or start over and build a new one.

Grow your Skills

Whether you’re just starting out, want to know more about programming language applications or models, looking to expand your horizons, or ready to dive into research, Racket can help you become a better programmer and system builder.

http://racket-lang.org/
Modelling Languages

Read Text pages 3 – 14

• Syntax important?
• Modeling Meaning (Semantics)?
• Modeling Syntax?
• Concrete Syntax
• Abstract Syntax
• read (tokenizer), parse, calc
• BNF – Terminals and Nonterminals
• Gödel's Theorem?

\[
\begin{align*}
\langle AE \rangle & ::= \langle \text{num} \rangle \\
& \quad | \{ + \langle AE \rangle \langle AE \rangle \} \\
& \quad | \{ - \langle AE \rangle \langle AE \rangle \}
\end{align*}
\]
### Scheme for Textbook Chapters 1 & 2

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Chapter Reading</th>
<th>PPT Notes</th>
<th>PDF Notes</th>
<th>Homework</th>
<th>Tests</th>
<th>Files for Assignments and Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wednesday, January 18, 2012</td>
<td>Introduction and Tools</td>
<td>Purple are Project</td>
<td>Intro &amp; Tools Intro &amp; Tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, January 23, 2012</td>
<td>Relations, Primitive Recursive Functions, Lambda Calculus, Combinator Logic and Lisp</td>
<td>PLAI Book: Chapters 22, 1, and 2</td>
<td>PLAI 1</td>
<td>PLAI 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Getting started downloads</td>
<td>Textbook</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Installing Racket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scheme for Chapter 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scheme for Chapter 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scheme Tutorial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, January 25, 2012</td>
<td>Continue from last class</td>
<td>PLAI Book: Chapter 3</td>
<td>PLAI 2</td>
<td>PLAI 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, January 30, 2012</td>
<td>Substitution</td>
<td>Scheme for Chapter 3 first subst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scheme for Chapter 3 second subst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scheme for Chapter 3 Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, February 01, 2012</td>
<td>Chomsky, Lexical Analysis and Parsing, H..</td>
<td>OO Hrn Doc.</td>
<td>HLI 2</td>
<td>HLI 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hrn Grammar and Abstract Syntax</td>
<td>OO Hrn 3 + Full First Class Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Formatting with javacc --&gt;</td>
<td>Original Hrn.zip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Original Hrn Doc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Syntax, Grammar, Parsing and Interpretation</td>
<td>HLI 3</td>
<td>HLI 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>javacc tutorial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>javacc FAQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JavaWorld javacc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>javacc demo 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>javacc demo 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unit tutorial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sed once liners</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>javacc demo 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, February 06, 2012</td>
<td>Continue from last class formatting using javacc (see information on the below) and then more lambda calculus from PLAI notes.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>javacc Formatter discussed in class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Picture of mapping of test to output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input Formatter.g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, February 08, 2012</td>
<td>Names, Types and Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Deferred Substitution, Functions, and Closures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheme for Chapter 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheme for Chapters 5 &amp; 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dr. Philip Cannata
Scheme for Textbook Chapter 1

```scheme
#lang plai

(define-type AE
  [num (n number?)]
  [add (lhs AE?) (rhs AE?)]
  [sub (lhs AE?) (rhs AE?)])

(define (parse sexp)
  (cond
   [(number? sexp) (num sexp)]
   [(list? sexp)
    (case (first sexp)
      [(+] (add (parse (second sexp))
              (parse (third sexp)))]
      [(-) (sub (parse (second sexp))
              (parse (third sexp)))]))])

Welcome to DrRacket, version 5.0.2 [3m].
Language: plai; memory limit: 256 MB.
> (parse (read))
3
(num 3)
> (parse (read))
(+ 3 4)
(add (num 3) (num 4))
> (parse (read))
(+ 3 4)
(add (num 3) (num 4))
> (parse (read))
[+ 3 4]
(add (num 3) (num 4))
> (parse '([+ 3 4])
(add (num 3) (num 4))
>
Scheme for Textbook Chapter 2

```
#lang plai

(define-type AE
  [num (n number?)]
  [add (lhs AE?) (rhs AE?)]
  [sub (lhs AE?) (rhs AE?)])

(define (parse sexp)
  (cond
   [(number? sexp) (num sexp)]
   [(list? sexp)
     (case (first sexp)
       [(+] (add (parse (second sexp))
                 (parse (third sexp)))
       [(-) (sub (parse (second sexp))
                 (parse (third sexp)))]))))

(define (calc an-ae)
  (type-case AE an-ae
    [num (n) n]
    [add (l r) (+ (calc l) (calc r))]
    [sub (l r) (- (calc l) (calc r))])))
```

Welcome to DrRacket, version 5.0.2 [3m].
Language: plai; memory limit: 256 MB.
> (calc (parse '3))
3
> (calc (parse '(+ 3 4)))
7
> |