Programming Languages

Genesis of Some Programming Languages
(My kind of Fiction)
<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 Chapters 5 &amp; 6</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 Chapters 7 &amp; 8</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>Interpretation</td>
<td>Chapters 1 &amp; 2</td>
<td>Haskell Pattern Matching</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Substitution</td>
<td>PLAI 2</td>
<td>Lazy Comprehension</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Symbol Table and Dictionaries</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Lambda Expression</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Scope</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Lazy Evaluation</td>
<td>Haskell1 Chapters 7 &amp; 8</td>
</tr>
<tr>
<td>Nested and Dynamic Scope</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Strong Type System</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Local and Non-Local Variables</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Propositional Logic</td>
<td>Haskell2</td>
</tr>
<tr>
<td>Static and Dynamic Scoping</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Haskell and SQL</td>
<td>Haskell2</td>
</tr>
<tr>
<td>Resolving References</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Syllogisms</td>
<td>Syllogism and Midder Review</td>
</tr>
<tr>
<td>Overloading</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Proof by Contradiction</td>
<td>Syllogism and Midder Review</td>
</tr>
<tr>
<td>Variables</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Gödel's Incompleteness Theorem</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Static and Dynamic Binding</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Deductive Logic</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Types</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Horn Clause</td>
<td>Prolog 1&amp;2 Chapters 32 - 34</td>
</tr>
<tr>
<td>Type System</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Prolog Facts and Rules</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Static and Dynamic Typing</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>PLAI Types &amp; HLL Types</td>
<td>Proof by Contradiction</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Type System</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Unification</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Implicit Type Conversion</td>
<td>PLAI Types &amp; HLL Types</td>
<td>Resolutions</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Type Judgments</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Backtracking</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Type Judgment Tree</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Constraint Satisfaction Programming</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Type Soundness</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Prolog and SQL</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Type Safety</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type Inferencing</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Semantic Web Programming</td>
<td></td>
</tr>
<tr>
<td>Relations</td>
<td>PLAI 1</td>
<td>Semantic Web &amp; Semantic DBs and OOPs</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Primitive Recursive Functions</td>
<td>Genesis</td>
<td>Compressions</td>
<td>Prolog 1, Page 9</td>
</tr>
<tr>
<td>Lambda Calculus</td>
<td>PLAI 1 Chapter 22</td>
<td>Kolmogorov Complexity</td>
<td></td>
</tr>
<tr>
<td>Combinator Logic</td>
<td>Genesis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lisp</td>
<td>PLAI 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

$$
P(0) = 0 \quad P(S(x)) = x
$$
$$
x - 0 = x \quad x - S(y) = P(x - y)
$$
$$
|x - y| = (x - y) \lor (y - x).
$$
Lambda Calculus Arithmetic

```lambda
def true = select_first
def false = select_second

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

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

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

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

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

(iszero one)
(\n.(n select_first) \s.((s false) zero))
(\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**

```
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**

```
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.
Function Bodies using Combinators

A function is called \textit{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.

\[
\begin{align*}
s f g x &= f x (g x) \\
k x y &= x \\
b f g x &= f (g x) \\
c f g x &= f x g \\
y f &= f (y f) \\
\text{cond } p f g x &= \text{if } p x \text{ then } f x \text{ else } g x \quad \text{-- Some Primitive Recursive Functions on Natural Numbers} \\
\text{pradd } x z &= y (b (\text{cond } ((==) 0) (k z)) (b (s (b (+) (k l)) ) (c b \text{ pred}) ) \text{ x} \\
\text{prmul } x z &= y (b (\text{cond } ((==) 0) (k 0)) (b (s (b (+) (k z)) ) (c b \text{ pred}) ) \text{ x} \\
\text{prexp } x z &= y (b (\text{cond } ((==) 0) (k l)) (b (s (b (*) (k x)) ) (c b \text{ pred}) ) \text{ z} \\
\text{prfac } x &= y (b (\text{cond } ((==) 0) (k l)) (b (s (*) ) (c b \text{ pred}) ) \text{ x} \\
\text{pradd1 } x z &= y (b (\text{cond } ((==) 0) (k z)) (b (s (b (+) (k l)) ) (c b \text{ pred}) ) \text{ x} \\
\text{prmul1 } x z &= y (b (\text{cond } ((==) 0) (k 0)) (b (s (b (pradd1) (k z)) ) (c b \text{ pred}) ) \text{ x} \\
\text{prexpl1 } x z &= y (b (\text{cond } ((==) 0) (k l)) (b (s (b (prmul1) (k x)) ) (c b \text{ pred}) ) \text{ z} \\
\text{prfac1 } x &= y (b (\text{cond } ((==) 0) (k l)) (b (s (prmul1) ) (c b \text{ pred}) ) \text{ x}
\end{align*}
\]

No halting problem here but not Turing complete either

Implies recursion or bounded loops, if-then-else constructs and run-time stack.
A Strange Proposal

\[
s f g x = f x (g x) \\
k x y = x \\
b f g x = f (g x) \\
c f g x = f x g \\
y f = f (y f) \\
cond p f g x = \text{if } p x \text{ then } f x \text{ else } g x
\]

-- Some Primitive Recursive Functions on Natural Numbers

\[
pradd x z = y (b (\text{cond } (==) 0) (k z)) (b (s (b (+) (k l))) (c b \text{ pred})) x \\
prmul x z = y (b (\text{cond } (==) 0) (k 0)) (b (s (b (+) (k z))) (c b \text{ pred})) x \\
prexp x z = y (b (\text{cond } (==) 0) (k l)) (b (s (b (*) (k x))) (c b \text{ pred})) z \\
prfac x = y (b (\text{cond } (==) 0) (k l)) (b (s (*)) (c b \text{ pred})) x
\]

To find Primitive Recursive Functions, try something like the following:

\[
\text{prf } x z = y (b (\text{cond } (==) 0) (k A)) (b (s AAAAAA . . . AAAAAAAAAA) (c b \text{ pred})) A
\]

Where each A is a possibly different element of \{x z 0 1 + * s k b c () "\} but at least having parentheses match.

Gödel may be lurking!

Dr. Philip Cannata
John McCarthy’s Takeaway

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

prlen x = y (b (cond ((==) [])) (k 0)) (b (s (b (+) (k 1))) (c b cdr))) x
prsum x = y (b (cond ((==) [])) (k 0)) (b (s (b (+) (car))) (c b cdr))) x
prprod x = y (b (cond ((==) []) (k 1)) (b (s (b (*) (car))) (c b cdr))) x
prmap f x = y (b (cond ((==) [])) (k [])) (b (s (b (: (f) ) (c b cdr)))) x
prfoo x = y (b (cond ((==) [])) (k [])) (b (s (b (: (cdr)) (c b cdr)))) x

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

(car (f:r)) = f
cdr (f:r) = r
: -- cons
Dr. Philip Cannata

Other Takeaways?

Are these just meaningless symbols?

\[ s \ f \ g \ x = f \ x \ (g \ x) \]
\[ k \ x \ y = x \]
\[ b \ f \ g \ x = f \ (g \ x) \]
\[ c \ f \ g \ x = f \ x \ (g \ x) \]
\[ y \ f = f \ (y \ f) \]
\[ \text{cond } p \ f \ g \ x = \text{if } p \ x \ \text{then } f \ x \ \text{else } g \ x \ -- \text{Some Primitive Recursive Functions on Natural Numbers} \]
\[ \text{pradd } x \ z = y \ (b \ (\text{cond } ((==) \ 0) \ (k \ z)) \ (b \ (s \ (b \ (+) \ (k \ 1))) \ (c \ b \ \text{pred}))) \ x \]
\[ \text{prmul } x \ z = y \ (b \ (\text{cond } ((==) \ 0) \ (k \ 0)) \ (b \ (s \ (b \ (+) \ (k \ z))) \ (c \ b \ \text{pred}))) \ x \]
\[ \text{prexp } x \ z = y \ (b \ (\text{cond } ((==) \ 0) \ (k \ 1)) \ (b \ (s \ (b \ (*) \ (k \ x))) \ (c \ b \ \text{pred}))) \ z \]
\[ \text{prfac } x = y \ (b \ (\text{cond } ((==) \ 0) \ (k \ 1)) \ (b \ (s \ (*) \ (c \ b \ \text{pred}))) \ x \]
Propositions:
Statements that can be either True or False

Truth: Are there well formed propositional formulas (i.e., Statements) that return True when their input is True

truth1 :: (Bool -> Bool) -> Bool
truth1 wff = (wff True)

truth2 :: (Bool -> Bool -> Bool) -> Bool
truth2 wff = (wff True True)

(( p -> not p)
( p q -> (p && q) || (not p ==> q))
( p q -> not p ==> q)
( p q -> (not p && q) && (not p ==> q))

1. prfac x = y (b (cond ((==) 0) (k 1)) (b (s (*) ) (c b pred))) x
2. not P(proof, s) && Q(g, s) - I am a statement that is not provable.

If Gödel's statement is true, then it is an example of something that is true for which there is no proof (Incomplete).
If Gödel's statement is false, then it has a proof and that proof proves the false Gödel statement true (Unsound).

3. If it was never possible for it not to be True that something was going to exist, and it will never be possible for it not to be True that something existed in the past then it is impossible for Truth ever to have had a beginning or ever to have an end. That is, it was never possible that Truth cannot be conceived not to exist.

4. If R is something that can be conceived not to exist and T is something that cannot be conceived not to exist and T is greater than R and God is that, than which nothing greater can be conceived, then God exists and is Truth.
Good Books to Have for a Happy Life 😊

From Frege to Gödel:
- From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931
  - Edited by Jean van Heijenoort

Lambda-Calculus and Combinators
An Introduction

A Profile of Mathematical Logic

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

Gödel, Escher, Bach: An Eternal Golden Braid

My Favorite

Dr. Philip Cannata