Programming Languages
High Level Languages

Relation

This Course

Jython in Java

ACL2 (Propositional Induction)

Algorithmic Information Theory (Information Compression and Randomness) - Kolmogorov Complexity

Orc (Parallel Computing)

GpH (Parallel Computing)

RDF (Horn Clause Deduction, Semantic Web)
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!].
A little Bit of Lambda Calculus – Lambda Expressions

The function *Square* has $\mathbb{R}$ (the reals) as domain and range.

\[ \text{Square} : \mathbb{R} \to \mathbb{R} \]

\[ \text{Square}(n) = n^2 \]

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

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

\[ M \to \text{LambdaExpression} \]
\[ N \to \text{LambdaExpression} \]

E.g., \( \lambda x . x^2 \) represents the *Square* function.
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 \(M[x \leftarrow N]\). Examples:

\[
\begin{align*}
x[x \leftarrow y] &= y \\
(x x)[x \leftarrow y] &= (y y) \\
(z w)[x \leftarrow y] &= (z w) \\
(z x)[x \leftarrow y] &= (z y) \\
(\lambda x \cdot (z x))[x \leftarrow y] &= (\lambda u \cdot (z u))[x \leftarrow y] = (\lambda u \cdot (z u)) \\
(\lambda x \cdot (z x))[y \leftarrow x] &= (\lambda u \cdot (z u))[y \leftarrow x] = (\lambda u \cdot (z u))
\end{align*}
\]

- 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 and Lambda Calculus in Python (sort of)

<table>
<thead>
<tr>
<th>Lambda Calculus</th>
<th>Python Version</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x.x )</td>
<td>lambda x: x</td>
<td>(lambda x: x)(4)</td>
</tr>
<tr>
<td>( \lambda s.(s\ s) )</td>
<td>lambda s: (s)(s)</td>
<td>s=(lambda s: (s)(s))</td>
</tr>
<tr>
<td>( \lambda \text{func.}\lambda \text{arg.}(\text{func arg}) )</td>
<td>lambda func, arg: (func)arg</td>
<td>(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>
<td>(lnot)(true)</td>
</tr>
<tr>
<td>def self_apply = ( \lambda s.(s\ s) )</td>
<td>self_apply = lambda s: (s)(s)</td>
<td></td>
</tr>
<tr>
<td>def apply = ( \lambda \text{func.}\lambda \text{arg.}(\text{func arg}) )</td>
<td>apply = lambda func, arg: (func)arg</td>
<td>(select_first)(3, 4)</td>
</tr>
<tr>
<td>def select_first = ( \lambda \text{first.}\lambda \text{second.first} )</td>
<td>select_first=lambda f, s: f</td>
<td></td>
</tr>
<tr>
<td>def select_second = ( \lambda \text{first.}\lambda \text{second.second} )</td>
<td>select_second=lambda f, s: s</td>
<td></td>
</tr>
<tr>
<td>def cond= ( \lambda e1.\lambda e2.\lambda c.((c\ e1)\ e2) )</td>
<td>cond=lambda c, e1, e2: (c)(e1, e2)</td>
<td></td>
</tr>
<tr>
<td>def true=select_first</td>
<td>true=select_first</td>
<td></td>
</tr>
<tr>
<td>def false=select_second</td>
<td>false=select_second</td>
<td></td>
</tr>
<tr>
<td>def not= ( \lambda x.(((\text{cond false})\ true)\ x) )</td>
<td>Inot=lambda x: (x)(false, true)</td>
<td></td>
</tr>
<tr>
<td>Or def not= ( \lambda x.((x\ false)\ true) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>def and= ( \lambda x.\lambda y.(((\text{cond y})\ false)\ x) )</td>
<td>land=lambda x, y: (cond)(x, y, false)</td>
<td></td>
</tr>
<tr>
<td>Or def and= ( \lambda x.\lambda y.((x\ y)\ false) )</td>
<td>(land)(true, true)(&quot;true&quot;, &quot;false&quot;)</td>
<td></td>
</tr>
<tr>
<td>def or= ( \lambda x.\lambda y.(((\text{cond true})\ y)\ x) )</td>
<td>lor=lambda x, y: (cond)(x, true, y)</td>
<td></td>
</tr>
<tr>
<td>Or def or= ( \lambda x.\lambda y.((x\ true)\ y) )</td>
<td>(land)(true, true)(&quot;true&quot;, &quot;false&quot;) (land)(true, false)(&quot;true&quot;, &quot;false&quot;)</td>
<td></td>
</tr>
</tbody>
</table>

Dr. Philip Cannata
Lambda Calculus Normal Order Substitution

Normal Order (call-by-name Algol 60, lazy evaluation) v. Applicative Order (call-by-value, eager evaluation)

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:

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

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:

\[ (((\text{cond} \text{true}) y) x) \]
\[ (((\lambda e1.\lambda e2.\lambda c.((c e1) e2) \text{true}) y) x) \]
\[ ((\lambda e2.\lambda c.((c \text{true}) e2) y) x) \]
\[ ((x \text{true}) y) \]
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
\((x \ x) \ (x \ x)\)
\((x \ x) \leftrightarrow \text{a function is returned}\)

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

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

etc.

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

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.
A little Bit of Lambda Calculus – 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.(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)  (iszero one)
(λx.x select_first)           (λn.(n select_first) λs.((s false)
select_first                zero) )
                         (λs.((s false) zero) select_first)
                         ((select_first false) zero)
A little Bit of Lambda Calculus – Lambda Calculus Arithmetic

**ADDITION**

```lambda
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**

```lambda
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))) 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 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))

→ 3041409320171337804361260816606476884437764156896051200000000000000000000

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

(( ( 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))))) ) 50)

→ 3041409320171337804361260816606476884437764156896051200000000000000000000

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

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 )))))

Dr. Philip Cannata
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
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))
\[ \rightarrow 3 \]

(if false (+ 1 2) (+ 3 4))
\[ \rightarrow 7 \]

**Code for Chaitin page 41**

Instead of (` (a b c)) \( \rightarrow \) (a b c)
\[
'(a b c)
\rightarrow \text{(list 'a 'b 'c)}
\]

(if (= 23 32) true false)
\[ \rightarrow \text{False} \]

(if (= (list 1 2 3) (list 1 2 3)) true false)
\[ \rightarrow \ldots \text{:= expects type <number> as 1st argument, given: (list 1 2 3); other arguments were: (list 1 2 3)} \]

Instead of (if (atom ...)
\[
\text{(if (list? (list 1 2 3)) true false)}
\rightarrow \text{true}
\text{(if (list? 21) true false)}
\rightarrow \text{false}
\text{(if (list? 'a) true false)}
\rightarrow \text{false} \]
Simple Lisp in Scheme

**Code for Chaitin page 41 continued**

Instead of `(let n (+ 1 2) (* n 3))`

```scheme
(let ((n (+ 1 2))) (* n 3))
→ 9
```

Instead of `(let (f n) (* n n) (f 10))` – see Scheme’s definition of “let” in the Scheme Tutorial at [http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-7.html#node_idx_274](http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-7.html#node_idx_274)

```scheme
(let ((f (lambda (n) (* n n)))) (f 10))
→ 100
```

**Code for Chaitin page 42**

Instead of `(car (‘ (a b c )))`

```scheme
(car '(a b c))
→ ’a
```

Instead of `(cdr (‘ (a b c )))`

```scheme
(cdr '(a b c))
→ (list 'b 'c)
```

Instead of `(cons (‘ a) (‘ (b c )))`

```scheme
(cons 'a '(b d))
→ (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](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))
→ 9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000000000

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

**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)))
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))
\rightarrow 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)
\rightarrow (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))
\rightarrow (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))
\rightarrow 3

Dr. Philip Cannata
LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.

HUH?

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.
Scheme for the Textbook

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

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?

\[
<\text{AE}> ::= <\text{num}>
| \{ + <\text{AE}> <\text{AE}> \}
| \{ - <\text{AE}> <\text{AE}> \}
\]
# 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>
</tr>
</thead>
<tbody>
<tr>
<td>Wednesday, August 24, 2011</td>
<td>Introduction and Tools</td>
<td>Purple are Project suggestions.</td>
<td>Intro &amp; Tools</td>
<td>Intro &amp; Tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Syntax, Grammar, Parsing and Interpretation</td>
<td></td>
<td></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>
</tr>
<tr>
<td></td>
<td>javacc FAQ</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>
</tr>
<tr>
<td></td>
<td>javacc demo1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>javacc demo2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unix tutorial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sed one liners</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>javacc demo3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, August 29, 2011</td>
<td>Modeling Languages and Interpreting Arithmetic</td>
<td>PLAI Book: Chapter 22, 1, and 2</td>
<td>PLAI 1</td>
<td>PLAI 1</td>
<td>HW1</td>
<td>Quiz 1</td>
</tr>
<tr>
<td></td>
<td>Getting started downloads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Textbook</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Installing Racket</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheme for Chapter 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheme for Chapter 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheme Tutorial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, August 31, 2011</td>
<td>Chomsky, Lexical Analysis and Parsing, H--</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hmm Grammar and Abstract Syntax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, September 05, 2011</td>
<td>Substitution</td>
<td></td>
<td>PLAI Book: Chapter 3</td>
<td>PLAI 2</td>
<td>PLAI 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheme for Chapter 3 first subst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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>Scheme for Chapter 3 Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tuesday, September 06, 2011</td>
<td>Notes used for Dr. Cook's Class</td>
<td></td>
<td>DCC</td>
<td>DCC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, September 07, 2011</td>
<td>Formatting with javacc --&gt;</td>
<td></td>
<td></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>
</tr>
<tr>
<td></td>
<td>javacc FAQ</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>
</tr>
</tbody>
</table>

Dr. Philip Cannata
#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

```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)))])])

(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
> (calc (parse '+ 3 4))
7
> |
```

Dr. Philip Cannata