Programming Languages
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)

High Level Languages

Relation

A Snapshot of Programming Language History
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!].
The function \textit{Square} has \( \mathbb{R} \) (the reals) as domain and range.

\textit{Square} : \( \mathbb{R} \to \mathbb{R} \)

\textit{Square}(n) = n^2

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

\textit{LambdaExpression} \to \textit{variable} \mid ( \textit{M N} ) \mid ( \lambda \textit{variable} . \textit{M} )

\textit{M} \to \textit{LambdaExpression}

\textit{N} \to \textit{LambdaExpression}

E.g., ( \( \lambda x . x^2 \) ) represents the \textit{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:

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

- 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 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) (lambda x: x)(lambda x: x+2) f=(lambda x: x)(lambda x: x+2) f(4)</td>
</tr>
<tr>
<td>( \lambda s. (s \ s) )</td>
<td>lambda s: (s)(s)</td>
<td>s=(lambda s: (s)(s)) (s)(lambda x: x) (s)(s)</td>
</tr>
<tr>
<td>( \lambda \text{func} \lambda \text{arg}. (\text{func} \ \text{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 ) def self_apply = ( \lambda s. (s \ s) ) def apply = ( \lambda \text{func} \lambda \text{arg}. (\text{func} \ \text{arg}) )</td>
<td>Identity = lambda x: x self_apply = lambda s: (s)(s) apply = lambda func, arg: (func)(arg)</td>
<td></td>
</tr>
<tr>
<td>def select_first = ( \lambda \text{first} \lambda \text{second}. \text{first} ) def select_second = ( \lambda \text{first} \lambda \text{second}. \text{second} )</td>
<td>select_first=lambda f, s: f select_second=lambda f, s: s</td>
<td>(select_first)(3, 4)</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 def false=select_second def not= ( \lambda x. (((\text{cond} \ false) \ true) \ x) ) Or def not= ( \lambda x. ((x \ false) \ true) )</td>
<td>true=select_first false=select_second not=lambda x: (x)(false, true)</td>
<td>(lnot)(true) (lnot)(true)(&quot;true&quot;, &quot;false&quot;)</td>
</tr>
<tr>
<td>def and= ( \lambda x. \lambda y. (((\text{cond} \ y) \ false) \ x) ) Or def and= ( \lambda x. \lambda y. ((x \ false) \ y) )</td>
<td>land=lambda x, y: (cond)(x, y, false)</td>
<td>((land)(true, true))(&quot;true&quot;, &quot;false&quot;) ((land)(true, false))(&quot;true&quot;, &quot;false&quot;)</td>
</tr>
<tr>
<td>def or= ( \lambda x. \lambda y. (((\text{cond} \ true) \ y) \ x) ) Or def or= ( \lambda x. \lambda y. ((x \ true) \ y) )</td>
<td>lor=lambda x, y: (cond)(x, true, y)</td>
<td>((land)(true, true))(&quot;true&quot;, &quot;false&quot;) ((land)(true, false))(&quot;true&quot;, &quot;false&quot;)</td>
</tr>
</tbody>
</table>
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 def \( \text{cond} = \lambda e_1.\lambda e_2.\lambda c.((c \ e_1) \ e_2) \),
\[
\text{def and} = \lambda x.\lambda y.(((\text{cond} \ y) \ \text{false}) \ x)
\]
is equivalent to
\[
\text{def and} = \lambda x.\lambda y.((x \ y) \ \text{false}) \text{ because:}
\]
\[
(((\text{cond} \ y) \ \text{false}) \ x)
\]
\[
(((\lambda e_1.\lambda e_2.\lambda c.((c \ e_1) \ e_2) \ y) \ \text{false}) \ x)
\]
\[
(\lambda c.((c \ y) \ \text{false}) \ x)
\]
\[
((x \ y) \ \text{false})
\]

In lambda calculus, if cond is defined as def \( \text{cond} = \lambda e_1.\lambda e_2.\lambda c.((c \ e_1) \ e_2) \),
\[
\text{def or} = \lambda x.\lambda y.(((\text{cond} \ \text{true}) \ y) \ x)
\]
is equivalent to
\[
\text{def or} = \lambda x.\lambda y.((x \ \text{true}) \ y) \text{ because:}
\]
\[
(((\text{cond} \ \text{true}) \ y) \ x)
\]
\[
(((\lambda e_1.\lambda e_2.\lambda c.((c \ e_1) \ e_2) \ y) \ \text{true}) \ x)
\]
\[
((\lambda e_2.\lambda c.((c \ \text{true}) \ e_2) \ y) \ x)
\]
\[
((x \ \text{true}) \ y)
\]
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 \(\lambda x . x\)
\((s \; (s \; s)) \; (x \; x)\)
\(((x \; x) \; (x \; x))\)
\((x \; x) \leftarrow \text{a function is returned}\)

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

\(\ldots\)

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

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.
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)
(λx.x select_first)
select_first

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

### ADDITION

```plaintext
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)
  (((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))
  (((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

```plaintext
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.
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))

→ 30414093201713378043612608166064768844377641568960512000000000000

( ( ( 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)

→ 30414093201713378043612608166064768844377641568960512000000000000

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 )))))))
Definition 8 (Substitution, take 4) To substitute identifier i in e with expression v, replace all non-binding identifiers in e having the name i with the expression v, except within nested scopes of i.

Finally, we have a version of substitution that works. A different, more succinct way of phrasing this definition is

Definition 9 (Substitution, take 5) To substitute identifier i in e with expression v, replace all free instances of i in e with v.
Develop Substitution for the Following Expressions

Start with schema from Chapter 2

Get the following expressions to work:

(subst 'x (num 1) (num 2))

(subst 'x (num 1) (id 'x))

(subst 'x (num 1) (id 'y))

(subst 'x (num 1) (add (id 'x) (id 'x)))

(subst 'x (num 1) (with 'y (num 2) (id 'x)))

(subst 'x (num 1) (with 'y (num 2) (add (id 'x) (id 'y))))

(subst 'x (num 1) (with 'y (id 'x) (add (id 'x) (id 'y))))

(subst 'x (num 1) (with 'x (id 'x) (id 'x)))

(calc (subst 'x (num 1) (with 'y (add (num 2) (id 'x)) (add (id 'y)(id 'x)))))

(calc (subst 'x (num 1) (with 'y (add (num 2) (id 'x)) (add (id 'y)(id 'x))))
(define-type AE
    [num (n number?)]
    [add (lhs AE?) (rhs AE?)]
    [sub (lhs AE?) (rhs AE?)]
    [id (i symbol?)]
    [with (i symbol?) (v AE?) (e AE?)])
(define (subst i v e)
    (type-case AE e
        (num (n) e]
        [add (l r) (add (subst i v l) (subst i v r))]
        [sub (l r) (sub (subst i v l) (subst i v r))]
        [id (i?) (if (symbol=? i i?) v e)]
        [with (i? v? e?) (if (symbol=? i i?) (with i? (subst i v v?) e?) (with i? (subst i v v?) (subst i v e?)))]
    ))
(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))]
        [id (i) (error 'calc "free id")]
        [with (i v e) (calc (subst i v e))]))
Programming Language Concepts

Chapters 1 and 2

Concepts:

• Concrete Syntax
• Abstract Syntax
• Token (Terminal)
• Non-Terminal
• BNF
• s-expression
• Parse
• Interpret (calc)

Chapter 3

Concepts:

• Identifier
• Substitution
• Binding Instance
• Bound Identifier
• Free Instance (Free Identifier)
• Scope
• Eager Evaluation
• Lazy Evaluation
Programming Language Concepts
Eager and Lazy Evaluation

We began this material motivating the introduction of with: as a means for eliminating redundancy. Let’s revisit this sequence of substitutions (skipping a few intermediate steps):

{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}} 
= {with {x 10} {with {y {- x 3}} {+ y y}}} 
= {with {y {- 10 3}} {+ y y}} 
= {with {y 7} {+ y y}} 
= {+ 7 7} 
= 14

Couldn’t we have also written it this way?

{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}} 
= {with {y {- {+ 5 5} 3}} {+ y y}} 
= {+ {- {+ 5 5} 3} {- {+ 5 5} 3}} 
= {+ {- 10 3} {- {+ 5 5} 3}} 
= {+ 7 {- 10 3}} 
= {+ 7 7} 
= 14

In the top example, we invoke calc before substitution (because the result of calc is what we supply as an argument to subst). This model of substitution is called **eager**: we “eagerly” reduce the named expression to a value before substituting it. This is in contrast to the second example of reductions above, which we call **lazy**, wherein we reduce the named expression to a value only when we need to (such as at the application of an arithmetic primitive).