Programming Languages

• Syllogisms and Proof by Contradiction

• Midterm Review
Dr. Philip Cannata

Notions of Truth

Propositions:
Statements that can be either True or False

Truth: $\top$
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) )

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.

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.
Proof by Contradiction

1. A is an Animal.
2. A Barks.
3. A is a Dog :- A is an Animal, A Barks.
4. -(A is a Dog)

Can this reasoning be automated?
## Syllogisms

Note: this is Prolog notation. In “standard” notation, this would be $P \rightarrow Q$.

P implies Q (i.e., Q is True if P is True or If P is True then Q is True)

### Mixed Hypothetical Syllogisms

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>$Q \Leftarrow P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Valid

### Conjunctive Syllogisms

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>$(R &amp; S)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Valid

### Disjunctive Syllogisms

<table>
<thead>
<tr>
<th>T</th>
<th>U</th>
<th>$(T &amp; U)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Invalid

### Syllogism – e.g., Modus Ponens – 1). P is True; 2). P $\rightarrow$ Q is True, therefore Q is True.

Green – assume 2 things; the implied Result is in Red; if the result is just True, then the syllogism is Valid, if the results are True and False, then the syllogism is Invalid.
Proof by Contradiction

1. Let $P = \text{It's raining, I'm outside}$ (comma means “&&”)
2. $P_1$. ($P_1$ is True, i.e., it's raining)
3. $P_2$. ($P_2$ is True, i.e., I'm outside)
4. $Q :- P = \text{I'm wet :- It's raining, I'm outside}$. (if it's raining and I'm outside then I'm wet)
5. $-Q$ (To answer the Query “Am I wet” against the Database, assume I'm not wet)
6. $-(\text{It's raining, I'm outside})$ (From 4 and 5 and Pattern 1)
7. $-\text{I'm outside}$ (From 2 and 6 and Pattern 2)
8. Contradiction – Therefore I’m wet (From 3 and 7 and Pattern 3)

Pattern 1
(Modus Tollens):
$Q :- (P_1, P_2)$
$-Q$ \[ \Rightarrow -(P_1, P_2) \]

Pattern 2
(Affirming a Conjunct):
$P_1$.
$-(P_1, P_2)$ \[ \Rightarrow -P_2 \]

Pattern 3:
$P_2$.
$-P_2$ \[ \Rightarrow \text{Contradiction} \]
Proof by Contradiction

1. A is an Animal.
2. A Barks.
3. A is a Dog :- A is an Animal, A Barks.

4. -(A is a Dog)
5. 3 and 4 and Pattern 1 ⇒ -(A is an Animal, A Barks)
6. 1 and 5 and Pattern 2 ⇒ -A Barks
7. 2 and 6 and Pattern 3 ⇒ Contradiction
8. Therefore A is a Dog

Database

Query (If you want to know if A is a Dog based upon the Facts and Rules in the Database try to see if A is not a Dog.)
What we cannot speak about we must pass over in silence.

- The world consists of independent atomic facts—existing states of affairs—out of which larger facts are built.
- Language consists of atomic, and then larger-scale, propositions that correspond to these facts by sharing the same "logical form".
- Thought, expressed in language, "pictures" these facts.

Tractatus
# Previous Midterm Exam

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Materials</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wednesday, September 14, 2011</td>
<td>Continue from last class - formatting using javac (see information on the below) and then move lambda calculus from PLAI notes.</td>
<td>Homework 3 (Also see below) HW3 Solutions quizzes FormatDTraceProfile.g</td>
</tr>
<tr>
<td>Monday, September 19, 2011</td>
<td>Names, Types and Functions</td>
<td>quizzes</td>
</tr>
<tr>
<td>Monday, September 26, 2011</td>
<td>Arity</td>
<td>Antr Video Tutorials</td>
</tr>
<tr>
<td>Wednesday, September 28, 2011</td>
<td>Syllogisms, Proof by Contradiction and Midterm Review</td>
<td>Extra Credit</td>
</tr>
<tr>
<td>Monday, October 01, 2011</td>
<td>In Class Mid-Term</td>
<td>Previous Midterm</td>
</tr>
<tr>
<td>Monday, October 10, 2011</td>
<td>Semantic Web Programming</td>
<td>Oracle Semantic Tutorial</td>
</tr>
<tr>
<td></td>
<td>RDF and RDFs</td>
<td>JDBCS Example for connecting to rising sun with two statements from</td>
</tr>
</tbody>
</table>

---

Dr. Philip Cannata
**Study the Homework and Quizzes**

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Textbooks</th>
<th>Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monday, August 29, 2011</td>
<td>Modeling Languages and Interpreting Arithmetic</td>
<td>PLAI Book: Chapter 22, 1, and 2, PLAI 1, PLAI 1, Homework 1, HW1 Solutions, Quiz 2.1</td>
<td></td>
</tr>
<tr>
<td>Wednesday, August 31, 2011</td>
<td>Continue from last class</td>
<td>Homework 2, HW2 Solution, Quiz 2, HW1كرم, Practice</td>
<td></td>
</tr>
<tr>
<td>Monday, September 5, 2011</td>
<td>Holiday, No Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thursday, September 8, 2011</td>
<td>Notes used for Dr. Cook’s Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, September 7, 2011</td>
<td>Substitution</td>
<td>PLAI Book: Chapter 3, PLAI 2, PLAI 2, Quiz 3</td>
<td></td>
</tr>
<tr>
<td>Wednesday, September 14, 2011</td>
<td>Continue from last class, formatting using java (see information on the board) and then move forward to PLAI notes</td>
<td>Homework 2, Quiz 4, HW3 Solution, Example [TextFile1]</td>
<td></td>
</tr>
<tr>
<td>Monday, September 19, 2011</td>
<td>Names, Types, and Functions</td>
<td>PLAI Book: Chapters 4, 5, and 6, PLAI 2 &amp; PLAI 2, Quiz 5</td>
<td></td>
</tr>
<tr>
<td>Wednesday, September 21, 2011</td>
<td>Continue from last class, run type tests, PLAI function application, data type substitution, and closures</td>
<td>Homework 4, Quiz 7A, Quiz 7B, Quiz 7</td>
<td></td>
</tr>
<tr>
<td>Monday, September 26, 2011</td>
<td>Java</td>
<td>Java Video Tutorials, Modifying jython, Modifying Python, Last Exception, HW to add “.cat” to import, HW1 Solutions - Python 1, Partial, Python 2, Java, Call Java, Lambda Live</td>
<td></td>
</tr>
<tr>
<td>Wednesday, September 28, 2011</td>
<td>Syllogisms, Proof by Contradiction and Midterm Review</td>
<td>Partial and Midterm Review, Partial and Midterm Review</td>
<td></td>
</tr>
</tbody>
</table>

Dr. Philip Cannata
You should now know the concepts highlighted in green.

---

<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>6 Lexical Analysis</td>
<td>HLL1 &amp; HLL2</td>
<td>Activation Records</td>
<td>HLL4*</td>
</tr>
<tr>
<td>7 Regular Expression</td>
<td>HLL1 &amp; HLL2</td>
<td>Runtime Stack</td>
<td>HLL4*</td>
</tr>
<tr>
<td>8 Syntactic Analysis</td>
<td>HLL1 &amp; HLL2</td>
<td>Side Effects</td>
<td>HLL4*</td>
</tr>
<tr>
<td>9 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>11 BNF</td>
<td>HLL1 &amp; HLL2</td>
<td>Closure</td>
<td>HLL4*</td>
</tr>
<tr>
<td>12 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>Haskell</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Precedence</td>
<td>HLL1 &amp; HLL2</td>
<td>Haskell Expression</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Ambiguous Grammar</td>
<td>HLL1 &amp; HLL2</td>
<td>Haskell Functions</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Abstract Syntax</td>
<td>HLL1 &amp; HLL2</td>
<td>Haskell Tuples</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Abstract Syntax Tree</td>
<td>HLL1 &amp; HLL2</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>Chapter 3</td>
</tr>
<tr>
<td>Symbol Table and Dictionaries</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>List Comprehension</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Scope</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Lambda Expression</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Nested and Dynamic Scope</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Lazy Evaluation</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Static and Dynamic Scoping</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Strong Type System</td>
<td>Haskell1</td>
</tr>
<tr>
<td>Resolving References</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Propositional Logic</td>
<td>Haskell2</td>
</tr>
<tr>
<td>Overloading</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Haskell and SQL</td>
<td>Haskell2</td>
</tr>
<tr>
<td>Variables</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Syllogisms</td>
<td>Haskell2</td>
</tr>
<tr>
<td>Static and Dynamic Binding</td>
<td>HLL 3 &amp; PLAI 3</td>
<td>Proof by Contradiction</td>
<td>Haskell2</td>
</tr>
<tr>
<td>Type</td>
<td>PLAI Types &amp; HLL Types</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Prolog 1&amp;2</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>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>Unification</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Type Judgment Tree</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Resolutions</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Type Soundness</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Backtracking</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Type Safety</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Constraint Satisfaction</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Explicit Polymorphism</td>
<td>Chapters 24 - 29 &amp; 31</td>
<td>Prolog and SQL</td>
<td>Prolog 1&amp;2</td>
</tr>
<tr>
<td>Lisp</td>
<td>PLAI 1</td>
<td>Semantic Web Programming</td>
<td>Semantic Web &amp; Semantic</td>
</tr>
<tr>
<td>Lambda Calculus</td>
<td>PLAI 1</td>
<td>Chapter 22</td>
<td>DBs and OO PLs</td>
</tr>
<tr>
<td>Relations</td>
<td>PLAI 1</td>
<td>Compressions</td>
<td>Prolog 1,</td>
</tr>
<tr>
<td>Composition of Relations</td>
<td>Haskell2</td>
<td>Kolmogorov Complexity</td>
<td>Page 9</td>
</tr>
</tbody>
</table>
We’ll be starting with javacc ➔ moving to ANTLR later

Instance of a Programming Language:

```java
int main ()
{
    return 0 ;
}
```

Program (abstract syntax):
Function = main; Return type = int
params =
Block:
Return:
    Variable: return#main, LOCAL addr=0
    IntValue: 0

Abstract Syntax
Javacc Parsers

Javacc uses these to build a Lexical Analyzer (Token Manager) and a Syntactic Analyzer (Parser)

Grammar Production Rules and Actions

void ae():
{ Token n; }
{ n = INTEGER { System.out.print("(num " + n +")"); } |
  list(); }
}

void list():
{}
{ |
  LOOKAHEAD(2) <LCURLY> <PLUS> { System.out.print("(add ");} (ae())* |
  <LCURLY> <MINUS> { System.out.print("(sub ");} (ae())* |
  <RCURLY> { System.out.print("\")}; }
}

Tokens, Terminals

PARSER_BEGIN(Parser)
import java.io.*;
import java.util.*;

public class Parser {
  public static void main(String args[]) throws ParseException {
    Parser parser = new Parser (System.in);
    parser.ae();
  }
}
PARSER_END(Parser)

SKIP:
{ " " |
  "\t" |
  "\n" |
  "\r" |
  "<//" (~["\n","\r"])* ("\n"|"\r") >
}

TOKEN:
{ < LCURLY: "{" > |
  < RCURLY: "}" > |
  < MINUS: "-" > |
  < PLUS: "+" > |
  INTEGER: ("0"-"9")+ |
  ERROR: ~[] >
}

Dr. Philip Cannata
Example Javacc Parser Input and Output

Input
Relation X Y Z
Relation a
Compose Y with X
Compose a with Z

Output
$ cat test.r | java Rel
    Saw IDENTIFIER X
    Saw IDENTIFIER Y
    Saw IDENTIFIER Z
Saw a RELATION
    Saw IDENTIFIER a
Saw a RELATION
Saw Composiiton of Y with X
Saw Composiiton of a with Z
import java.io.*;
import java.util.*;

public class Rel {
    public static void main(String args[]) throws ParseException {
        Rel parser = new Rel(System.in);
        parser.program();
    }
}

void program() {
    {}
    ( rels() )+ ( comps() )* 
    }

void rels() {
    String id; 
    {
        <RELATION> ( id = identifier() { System.out.println("   Saw IDENTIFIER "+id); } )+
        { System.out.println("Saw a RELATION"); }
    }

String identifier() {
    Token t; 
    t = <IDENTIFIER> { return new String(t.image); }
    }

void comps() {
    String id1, id2; 
    {
        <COMPOSE> id1 = identifier() <WITH> id2 = identifier()
        { System.out.println("Saw Composition of "+id1 +" with "+id2); }
    }

Example Javacc Parser

PARSER_BEGIN(Rel)

import java.io.*;
import java.util.*;

public class Rel {
public static void main(String args[]) throws ParseException {
    Rel parser = new Rel(System.in);
parser.program();
}
PARSER_END(Rel)

Skip:
{
    " "
    "\t"
    "\n"
    "\r"
    <"/" (~["\n","\r"])* ("\n"|"\r")>
}

Token:
{
    < RELATION: "Relation" >
    < COMPOSE: "Compose" >
    < WITH: "with" >
}

Token: /* Literals */
{
    < IDENTIFIER: ["A"-"Z", "a"-"z"]
        ( ["A"-"Z", "a"-"z", "0"-"9", ","] )*
    >
}

Token:
{
    <ERROR: ~[] >
}
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!].
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)

For more but different details see Section 22.3 of the textbook.
A little Bit of Lambda Calculus – 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
```
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) \[ (a b c) \]
\[ (list 'a 'b 'c) \]

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

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

Instead of (if (atom ...)
\[ \rightarrow \text{true} \]
\[ \rightarrow \text{false} \]
\[ \rightarrow \text{false} \]
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

```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))
→ 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414
6397615651828625369792082722375825118521091686400000000000000000000000000000000

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

**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))
→ 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
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)))(cal (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
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):

\[
\{\text{with } \{x \{+ 5 5\}\} \{\text{with } \{y \{- x 3\}\} \{+ y y\}\}\} \\
= \{\text{with } \{x 10\} \{\text{with } \{y \{- x 3\}\} \{+ y y\}\}\} \\
= \{\text{with } \{y \{- 10 3\}\} \{+ y y\}\} \\
= \{\text{with } \{y 7\} \{+ y y\}\} \\
= \{+ 7 7\} \\
= 14
\]

Couldn’t we have also written it this way?

\[
\{\text{with } \{x \{+ 5 5\}\} \{\text{with } \{y \{- x 3\}\} \{+ y y\}\}\} = \{\text{with } \{y \{- \{+ 5 5\} 3\}\} \{+ y y\}\} \\
= \{+ \{- \{+ 5 5\} 3\} \{- \{+ 5 5\} 3\}\} \\
= \{+ \{- 10 3\} \{- \{+ 5 5\} 3\}\} \\
= \{+ \{- 10 3\} \{- 10 3\}\} \\
= \{+ 7 \{- 10 3\}\} \\
= \{+ 7 7\} \\
= 14
\]

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).
Regular Grammar

• Simplest; least powerful
• Equivalent to:
  – Regular expression (think of perl)
  – Finite-state automaton
• Right regular grammar:
  \[ \omega \in \text{Terminal}^*, \]
  \[ A \text{ and } B \in \text{Nonterminal} \]
  \[ A \to \omega B \]
  \[ A \to \omega \]
• Example:
  \[ \text{Integer} \to 0 \text{ Integer} \mid 1 \text{ Integer} \mid \ldots \mid 9 \text{ Integer} \mid 0 \mid 1 \mid \ldots \mid 9 \]
Context-Free Grammar

Production:

$$\alpha \rightarrow \beta$$

$$\alpha \in \text{Nonterminal}$$

$$\beta \in (\text{Nonterminal} \cup \text{Terminal})^*$$

ie, lefthand side is a single nonterminal, and righthand side is a string of nonterminals and/or terminals (possibly empty).
Context-Sensitive Grammar

Production:

\[ \alpha \rightarrow \beta \quad |\alpha| \leq |\beta| \]

\[ \alpha, \beta \in (\text{Nonterminal } \cup \text{ Terminal})^* \]

ie, lefthand side can be composed of strings of terminals and nonterminals
Grammars

Grammars: Metalanguages used to define the concrete syntax of a language.

Backus Normal Form – Backus Naur Form (BNF)

• Stylized version of a context-free grammar (cf. Chomsky hierarchy)
• First used to define syntax of Algol 60
• Now used to define syntax of most major languages

Production:
\[ \alpha \rightarrow \beta \]
\[ \alpha \in \text{Nonterminal} \]
\[ \beta \in (\text{Nonterminal} \cup \text{Terminal})^* \]

ie, lefthand side is a single nonterminal, and \( \beta \) is a string of nonterminals and/or terminals (possibly empty).

• Example
  \[ \text{Integer} \rightarrow \text{Digit} | \text{Integer Digit} \]
  \[ \text{Digit} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]
Extended BNF (EBNF)

Additional metacharacters

- `{ }` a series of zero or more
- `( )` must pick one from a list
- `[ ]` pick none or one from a list

Example

Expression -> Term `{ ( + | - ) Term }`
IfStatement -> if ( Expression ) Statement [ else Statement ]

EBNF is no more powerful than BNF, but its production rules are often simpler and clearer.

**Javacc EBNF**

- `( ... )*` a series of zero or more
- `( ... )+` a series of one or more
- `[ ... ]` optional
Instance of a Programming Language:

```c
int main ()
{
    return 0 ;
}
```

Program (abstract syntax):
Function = main; Return type = int
params =
Block:
Return:
    Variable: return#main, LOCAL addr=0
    IntValue: 0

Abstract Syntax
**Parse Trees**

\[
\text{Integer} \rightarrow \text{Digit} \mid \text{Integer Digit} \\
\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

Parse Tree for 352 as an Integer

```
  Integer
    /   \   \\
  Integer  Digit
    /   \   \\
 Digit   5
    \   \\
  3
```
Different values are usually used to solve different instances of a problem. Would it be efficient to build the particular values into the program?

<table>
<thead>
<tr>
<th>High Level Languages</th>
<th>Relation-based Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Imperative or Procedural Languages)</td>
<td>(Functional Languages)</td>
</tr>
<tr>
<td>• Traditionally the concept of a <strong>variable</strong> with a changeable association between a <strong>name</strong> and a <strong>value</strong> is used.</td>
<td>• Traditionally the concept of a <strong>function application</strong> is used.</td>
</tr>
<tr>
<td>• Assignments are made to the variable usually multiple times.</td>
<td>• In a function application, a function receives argument values and returns values.</td>
</tr>
<tr>
<td>• In other words, the same name may be associated with different values.</td>
<td>• In other words, a <strong>name</strong> is only associated with one value.</td>
</tr>
</tbody>
</table>
### Scope Concepts

The **lifetime** of a variable is the time interval during which the variable has been allocated a block of memory.

#### Symbol Table

<table>
<thead>
<tr>
<th>Scope Level</th>
<th>Scope Details</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Outer scope</td>
<td>&lt;i, 1&gt;&lt;h, 1&gt;&lt;i, 1&gt;&lt;B, 2&gt;&lt;A, 8&gt;&lt;main, 14&gt;&lt;sort, 21&gt;</td>
<td>1 - 33</td>
</tr>
<tr>
<td>2. Function B nested scope</td>
<td>&lt;w, 2&gt;&lt;j, 3&gt;&lt;k, 3&gt;</td>
<td>2 - 7</td>
</tr>
<tr>
<td>3. Function A nested scope</td>
<td>&lt;x, 8&gt;&lt;y, 8&gt;&lt;i, 9&gt;&lt;j, 9&gt;</td>
<td>8 - 13</td>
</tr>
<tr>
<td>4. Function main nested scope</td>
<td>&lt;a, 15&gt;&lt;b, 15&gt;</td>
<td>15 - 20</td>
</tr>
<tr>
<td>5. Function sort nested scope</td>
<td>&lt;a, 21&gt;&lt;size, 21&gt;&lt;i, 22&gt;&lt;j, 22&gt;</td>
<td>21 - 33</td>
</tr>
</tbody>
</table>

#### Nested scope

#### Dictionaries

For **static scoping**, the referencing environment (the set of statements which can validly reference a name) for a name is its defining scope and all nested subscopes.

Reference to i (4) resolves to <i, 1> in Outer scope

In **dynamic scoping**, a name is bound to its most recent declaration based on the program’s call history.

Since main (17) calls \( \rightarrow \) A (10) sets I, A(11) calls \( \rightarrow \) B, the reference to i (4) resolves to <i, 9> in A. What about i at (5)?
int h, i;
void B(int w) {
    int j, k;
    i = 2*w;
    w = w+1;
}
void A(int x, int y) {
    bool i, j;
    B(h);
}
int main() {
    int a, b;
    h = 5; a = 3; b = 2;
    A(a, b);
}
A Taxonomy of Functions

6.1 A Taxonomy of Functions

- **first-order** Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in F1WAE are of this nature, which explains the 1 in the name of the language.
- **higher-order** Functions can be defined anywhere in a program and return other functions as values.
- **first-class** Functions are values with all the rights of other values. In particular, they can be supplied as the value of arguments to functions, returned by functions as answers, and stored in data structures. [They can also be defined anywhere in a program.]
Binding

• A *binding* is an association between an entity (such as a variable) and a property (such as its value).

• A binding is *static* if the association occurs before run-time.

• A binding is *dynamic* if the association occurs at run-time.

• Name bindings play a fundamental role.

• The lifetime of a variable name refers to the time interval during which memory is allocated.
Dr. Philip Cannata

Something (e.g., Variable)

<table>
<thead>
<tr>
<th>Static Bind.</th>
<th>Dynamic Bind.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>Lifetime</td>
<td></td>
</tr>
</tbody>
</table>

Static v. Dynamic Scoping

Static v. Dynamic Typing

Static v. Dynamic Binding ➔
(with (x 1) (+ x x))
(\x. (+ x x) 1)

(parse '(with (x 1) (+ x x)))
(app (fun 'x (add (id 'x) (id 'x))) (num 1))

(interp (parse '(with (x 1) (+ x x))) (mtSub))
(numV 2)

(let ((x 1)) (+ x x))
2

((lambda (x) (+ x x)) 1)
2
(with (x 2) (with (y 1) (+ x y)))
((\x.\y.(+ x y) 2) 1)

(parse '(with (x 2) (with (y 1) (+ x y))))
(app (fun 'x (app (fun 'y (add (id 'x) (id 'y))) (num 1)))
(num 2))

(interp (parse '(with (x 2) (with (y 1) (+ x y)))) (mtSub))
(numV 3)

(let ((x 2)) (let ((y 1)) (+ x y)))
3

(((lambda (x) (lambda (y) (+ x y))) 2) 1)
3
PLAI Chapters 4, 5 and 6

(with (x 5) (with (f (fun (y) (+ x y))) (with (x 3) (f 4))))
(((λx.λf.λx.(f 4) 5) (λy.(+ x y)) 3)

(λy.(+ x y) 4)
-----
X = 3
-----
f = (λy.(+ x y))
   { x = 5 }
-----
X = 5

(parse '(with (x 5) (with (f (fun (y) (+ x y))) (with (x 3) (f 4)))))
(app (fun 'x (app (fun 'f (app (fun 'x (app (id 'f) (num 4))) (num 3))) (fun 'y (add (id 'x) (id 'y))))) (num 5))

Static Scoping:
(interp (parse '(with (x 5) (with (f (fun (y) (+ x y))) (with (x 3) (f 4))))) (mtSub))
(numV 9)

Dynamic Scoping:
(interp (parse '(with (x 5) (with (f (fun (y) (+ x y))) (with (x 3) (f 4))))) (mtSub))
(numV 7)

(let ((x 5)) (let ((f (lambda (y) (+ x y)))) (let ((x 3)) (f 4)))) 9

(let ((Z 5)) (let ((f (lambda (y) (+ x y)))) (let ((x 3)) (f 4))))
. . . reference to an identifier before its definition: x