Programming Languages
Dr. Philip Cannata
Relations and Functions

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

The function \( \text{Square} \) 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:

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

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

E.g., \((\ \lambda\ x\.\ x^2)\) represents the \(\text{Square}\) function.
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 \(M[x \leftarrow N]\). 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., \((\lambda x . x^2) 5) = 5^2\)
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

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 )))))))

→ 30414093201713378043612608166064768844377641568960512000000000000

Dr. Philip Cannata
A little Bit of Lambda Calculus – Lambda Calculus Arithmetic

zero  (lambda (f ) (lambda (x) x))
one  (lambda (f ) (lambda (x) (f x))))
two  (lambda (f ) (lambda (x) (f (f x))))
   i.e., in Scheme - ((lambda (f ) ((lambda (x) (f (f x))) 4)) (lambda (z) (+ z z)))
three (lambda (f ) (lambda (x) (f (f (f x)))))
   i.e., in Scheme - ((lambda (f ) ((lambda (x) (f (f (f x)))) 4)) (lambda (z) (+ z z)))
succ
   (lambda (n)
       (lambda (f )
           (lambda (x)
               (f ((n f ) x))))))
   i.e., in Scheme -
((lambda (n) (n ((lambda (f ) ((lambda (x) (f (f x))) 4)) (lambda (z) (+ z z)))))
   or ((lambda (n) ((lambda (z) (+ z z)) n)) ((lambda (f ) ((lambda (x) (f (f x))) 4)) (lambda (z) (+ z z))))
   or (define succ(lambda (n) ((lambda (z) (+ z z)) n)))
       (succ ((lambda (f ) ((lambda (x) (f (f x))) 4)) (lambda (z) (+ z z)) )
sum
   (lambda (m)
       (lambda (n)
           ((n succ) m))))
prod
   (lambda (m)
       (lambda (n)
           ((n (sum m)) zero)))
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 :-).
<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>Text</th>
<th>Plan for Assignment and Examinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wednesday, January 19, 2011</td>
<td>Introduction and Tools</td>
<td>System Overview, Design, and Analysis</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Monday, January 24, 2011</td>
<td>Building Languages and Interpreting Assembly</td>
<td>PL/AL Book, Chapters 2, 3, and 4</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Wednesday, January 26, 2011</td>
<td>Binaural Model Analysis and Testing AC</td>
<td>PL/AL Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Monday, January 31, 2011</td>
<td>Substitution</td>
<td>PL/AL Book, Chapter 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Wednesday, February 2, 2011</td>
<td>Audio and Art</td>
<td>Audio-Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Monday, February 7, 2011</td>
<td>Audio and Art</td>
<td>Audio-Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Wednesday, February 9, 2011</td>
<td>Audio and Art</td>
<td>Audio-Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Monday, February 14, 2011</td>
<td>Sound and Music</td>
<td>Audio-Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Wednesday, February 16, 2011</td>
<td>Functions and Programming</td>
<td>Audio-Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Monday, February 21, 2011</td>
<td>Project Disposition</td>
<td>Project Requirements</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Wednesday, March 9, 2011</td>
<td>Music</td>
<td>Audio-Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Monday, March 14, 2011</td>
<td>Music</td>
<td>Audio-Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
<tr>
<td>Wednesday, March 16, 2011</td>
<td>Music</td>
<td>Audio-Book, Chapters 1, 2, and 3</td>
<td>PPT 1</td>
<td>PDF 1</td>
<td>Homework 1</td>
<td>Text 1</td>
<td>Plan for Assignment and Examinations</td>
</tr>
</tbody>
</table>
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\(\text{(let n (+ 1 2) (* n 3))}\)

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

Instead of\(\text{(let (f n) (* n n) (f 10))}\) – 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

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

**Code for Chaitin page 42**

Instead of\(\text{(car ' (a b c ))}\)

\[
\begin{align*}
\text{(car '(a b c))} \\
\rightarrow 'a
\end{align*}
\]

Instead of\(\text{(cdr ' (a b c ))}\)

\[
\begin{align*}
\text{(cdr '(a b c))} \\
\rightarrow \text{(list 'b 'c)}
\end{align*}
\]

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

\[
\begin{align*}
\text{(cons 'a '(b d))} \\
\rightarrow \text{(list 'a 'b 'd)}
\end{align*}
\]
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))
→ 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414
6397615651828625369792082722375825118521091686400000000000000000000000000

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

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
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.

TRULY.

NO, IT'S NOT.

IT'S NOT?

I MEAN, OSTEENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.
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 `schemebound` 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

Dr. Philip Cannata
<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>Text</th>
<th>Plans for Assessment and Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wednesday, January 19, 2011</td>
<td>Introduction and Tools</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, January 24, 2011</td>
<td>Machine Language and Interpreting Assembly</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, January 26, 2011</td>
<td>Machine-Related Analysis and Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, January 31, 2011</td>
<td>Encryption and Abstract Syntax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, February 02, 2011</td>
<td>Arithmetic and Anti-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, February 07, 2011</td>
<td>Arithmetic and Anti-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, February 09, 2011</td>
<td>Arithmetic and Anti-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, February 14, 2011</td>
<td>Functions and Subroutines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, February 16, 2011</td>
<td>Functions and Subroutines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, February 21, 2011</td>
<td>Project Disposition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, March 09, 2011</td>
<td>Subroutines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monday, March 14, 2011</td>
<td>Subroutines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wednesday, March 23, 2011</td>
<td>Subroutines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dr. Philip Cannata
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
Notions of Truth

Propositions:
Statements that can be either True or False

Truth: \( \models \)
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)

\( ( \lambda p \rightarrow \neg p ) \)
\( ( \lambda p \ q \rightarrow (p \&\& q) \| (\neg p \Longrightarrow q) ) \)
\( ( \lambda p \ q \rightarrow (\neg p \&\& q) \&\& (\neg p \Longrightarrow q) ) \)

<table>
<thead>
<tr>
<th>Clausal Form</th>
<th>Conventional Syntax</th>
<th>PROLOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>( { p } )</td>
<td>( p )</td>
<td>( p. )</td>
</tr>
<tr>
<td>( { q } )</td>
<td>( q )</td>
<td>( q. )</td>
</tr>
<tr>
<td>( { r } )</td>
<td>( r )</td>
<td>( r. )</td>
</tr>
<tr>
<td>( { s, \neg p, \neg q } )</td>
<td>( p &amp; q \rightarrow s )</td>
<td>( s : p, q. )</td>
</tr>
<tr>
<td>( { t, \neg s, \neg r } )</td>
<td>( s &amp; r \rightarrow t )</td>
<td>( t : s, r. )</td>
</tr>
<tr>
<td>( { \neg t } )</td>
<td>( \neg t \rightarrow \bot )</td>
<td>( \neg t. )</td>
</tr>
</tbody>
</table>

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.
How Robots Reason

How does a robot scientist “reason?” It uses the same options people use. One is deductive inference, which is the foundation for mathematics and computer science. Deductive reasoning is “sound.” That is, if you start with truth you can infer only new truths. Unfortunately, in the absence of a consummate “theory of everything,” deduction is insufficient for science, because it can work out only the consequences of what is already known.

A second option, abductive reasoning, is not sound, as is obvious from the swan example below; many things are white but are not swans. Yet abduction does provide a way of generating hypotheses that may be true. The great insight of science is that the way to decide truth is not by pure deduction from assumptions but rather by experimenting on the physical world. If Adam hypothesizes that Daisy is a swan, then the way to decide on the truth of this proposition is for Adam to experimentally catch Daisy and test whether she is a swan, a duck or something else.

Induction, like abduction, provides a way to infer new hypotheses. If every swan we see is white, it is natural to infer, as Aristotle actually did, that all swans are white. But induction is not sound, and Aristotle's induction was disproved by the discovery of black swans in Australia. We constantly use induction in our daily lives. It reassures us that the sun will rise tomorrow and that our breakfast won't poison us. Induction's role in science is controversial, however, because the main justification for induction is that it generally works, which is itself an induction.

Like humans, robots can use various methods of reasoning. The methods may or may not be sound, but they provide ways to form hypotheses and suggest experiments that can be performed to test those hypotheses.
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 and you produce “statement” then Q is True.

not P(proof, statement) && Q(x, statement) = g

Let g be the Gödel number for this statement,

not P(proof, statement) && Q(g, statement) = s

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

not P(proof, s) && Q(g, s) - 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.
Prolog Example

parent(hank, ben).
parent(hank, denise).
parent(irene, ben).
parent(irene, denise).
parent(alice, carl).
parent(ben, carl).
parent(ben, carl).
parent(denise, frank).
parent(denise, gary).
parent(earl, frank).
parent(earl, gary).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

Besides being a Prolog Program this is also an example of a Compression.
Running Prolog

For Linux it should be - /usr/opt/gprolog-1.3.0/bin/gprolog
You need to set your PATH environment variable as follows:
export PATH=“/usr/opt/gprolog-1.3.0/bin”:SPATH

To download prolog for windows go to: http://www.gprolog.org/#download

Don’t forget the dot

Turn on trace.

Turn off trace.
Prolog Example

parent(hank, ben).
parent(hank, denise).
parent(irene, ben).
parent(irene, denise).
parent(alice, carl).
parent(ben, carl).
parent(denise, frank).
parent(denise, gary).
parent(earl, frank).
parent(earl, gary).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

?- parent(hank, ben).
no
?- parent(hank, denise).
yes
X = hank ? ;
X = irene ? ;
(16 ms) yes
?- parent(irene, ben).
no
?- parent(irene, denise).
yes
X = hank ? ;
X = irene ? ;
no
?- parent(alice, carl).
no
?- parent(ben, carl).
yes
X = hank ? ;
X = irene ? ;
no
?- parent(denise, frank).
yes
X = hank ? ;
X = irene ? ;
no
?- parent(denise, gary).
yes
X = hank ? ;
X = irene ? ;
no
?- parent(earl, frank).
yes
X = hank ? ;
X = irene ? ;
no
?- parent(earl, gary).
yes
X = hank ? ;
X = irene ? ;
no
?- grandparent(X, carl).
no
?- grandparent(hank, X).
yes
X = carl ? ;
X = frank ? ;
X = gary
yes
?- grandparent(hank, ben).
no
?- grandparent(irene, denise).
yes
X = hank ? ;
X = irene ? ;
no
?- grandparent(alice, carl).
no
?- grandparent(ben, carl).
no
?- grandparent(denise, frank).
yes
X = hank ? ;
X = irene ? ;
no
?- grandparent(denise, gary).
yes
X = hank ? ;
X = irene ? ;
no
?- grandparent(earl, frank).
yes
X = hank ? ;
X = irene ? ;
no
?- grandparent(earl, gary).
yes
X = hank ? ;
X = irene ? ;
no