ב״ה

Jerusalem College of Technology בית ספר גבוה לטכנולוגיה ירושלים

המחלקה למדעי המחשב

תכנות פונקציונלי ולוגי

Part 1: Reasoning about Programs using Specifications and Induction

Part 2: Functional Programming in LISP

Part 3: Logic Programming in PROLOG

דפי הקורס מיועדים לעזור לתלמידים שמשתתפים בהרצאות להבין את החומר, ואינם מיועדים ללימוד עצמי.

ר.ב. יחזקאל R.B. Yehezkael

שבט תשס״ה - 1/2005

Part 1

Reasoning about Programs using Specifications and Induction

Inductive methods of reasoning about programs are explained. The methods presented are not novel; our aim being to facilitate their understanding and use by computer scientists and engineeers.

What does it mean for a program to be correct? What is correctness?

Partial correctness: Halting is not guaranteed for all values of the inputs, but whenever the program halts normally, the results are correct according to the specifications.

Total Correctness: The program always halts and gives values which are correct according to the specifications.

Specifications: What the program should do and not how it does it, that is, what are the inputs and the outputs. While it is desirable to add to the specifications what happens when there are errors in the inputs, we shall not require this. We take the view that failure because of illegal inputs is a case of misuse of the program and not incorrectness. (Analogy: A light bulb which burns out immediately because it is connected to a very high voltage supply is not considered defective.) In writing specifications, one should give a clear and direct description of the final result only, and not describe intermediate results. The description should avoid the use of imperatives.

Computational induction: This is a technique for reasoning about programs. Specifically this technique can be used to prove or to justify that a program is partially correct. Proof of halting or total correctness will not be discussed here. For now, run the program on test data to gain confidence that it will halt.

Now for some examples.

Example 1

```
FUNCTION even_test(IN n:integer) RETURN boolean;
-- SPECIFICATION
-- the function even_test tests if n is even

FUNCTION odd_test(IN n:integer) RETURN boolean;
-- SPECIFICATION
-- the function odd_test tests if n is odd
```

```
--THE BODIES OF THE FUNCTIONS

FUNCTION even_test(IN n:integer) RETURN boolean IS
BEGIN
        IF n = 0
        THEN RETURN true;
        ELSIF n>0
        THEN RETURN odd_test(n-1);
        ELSE RETURN odd_test(n+1);
        END IF;
END even_test;

FUNCTION odd_test(IN n:integer) RETURN boolean IS
BEGIN
        IF n = 0
        THEN RETURN false;
        ELSIF n>0
        THEN RETURN even_test(n-1);
        ELSE RETURN even_test(n+1);
        END IF;
END odd_test;
```

How can we prove or justify that even_test successfully tests that n is even and that odd_test indeed tests that n is odd. If we try to read the program the way it is executed we get into a mess as even_test calls odd_test and odd_test calls even_test and we can go around endlessly trying to understand the program. The inductive way of reading the program in order to understand it is to read each function from beginning to end and try to justify each line in the program. When we reach a function or procedure call we do not jump anywhere in our reading of the program but instead simply assume that the call works to specification. If we do this and every line can be justified then theory shows that the program is partially correct.

Actually, computational induction is based on a form of simple induction on the length of the computation. (Induction on the number of function/procedure calls executed, may also be used.)

Let us now present two parallel proofs that even_test is partially correct using both these techniques. We use two columns where there are differences between the proofs and write the common parts once only across the width of the page.

if n=0 then even_test(n) = true and so even_test correctly tests if n is even.

if n>0 then even_test(n) = odd_test(n-1).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call odd_test(n-1) works correctly.	Assuming the computation halts normally, the length of computation of odd_test(n-1) is shorter than the length of computation of even_test(n). Therefore by simple induction we may assume that odd_test(n-1) works correctly.

So, odd_test(n-1) will test if n-1 is odd. But n-1 is odd means that n is even. So in this case even_test(n) tests correctly if n is even.

if n<0 then even_test(n) = odd_test(n+1).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call odd_test(n+1) works correctly.	Assuming the computation halts normally, the length of computation of odd_test(n+1) is shorter than the length of computation of even_test(n). Therefore by simple induction we may assume that odd_test(n+1) works correctly.

So, odd_test(n+1) will test if n+1 is odd. But n+1 is odd means that n is even. So in this case as well, even_test(n) tests correctly that n is even.

Similarly we can check the definition of odd_test in the same way.

So assuming halting, correct results will be obtained. That is, even_test and odd_test are partially correct.

Regarding halting, it has been shown by Turing and Godel that this can not be determined by a program or algorithm. There are however, systematic but not fully general methods, for proving halting. In this specific case, we see that the absolute value of n is being reduced (or simplified) on each function call so we can see that halting will occur as zero must be reached. This idea of "simplification" can be made mathematically precise - see Zohar Manna's book "Mathematical Theory of Computation" in particular the material on well founded orderings. We shall not discuss these concepts here.

Class discussion: How can a proof by computational induction be translated into a proof by induction on the length of computation? Does the existence of such a translation justify computational induction?

Exercise: Check the definition of odd_test by using computational induction and simple induction on the length of the computation.

Class Discussion: What would happen is we used computational induction to check the following definition of even_test. (Here, execution is non terminating when n≠0.)

```
FUNCTION even_test(IN n:integer) RETURN boolean IS
BEGIN
        IF n = 0
        THEN RETURN true;
        ELSIF n>0
        THEN RETURN odd_test(n+1);    -- error, should be n-1
        ELSE RETURN odd_test(n-1);    -- error, should be n+1
        END IF;
END even_test;
```

Example 2

```
size: CONSTANT integer := 100;

TYPE vector IS ARRAY (1..size) OF integer;

PROCEDURE swap (IN OUT v:vector; IN low,high: integer) IS
-- SPECIFICATION
-- exchange the values of the v(low) and v(high).

        .....
BEGIN
        .....
END swap;

PROCEDURE reverse(IN OUT v:vector; IN low,high: integer) IS
-- SPECIFICATION
-- When  low<high,
-- reverse the order of the elements of the vector v between "low" and "high".
-- When high≤low, no change is made to v.

BEGIN
        IF high > low
        THEN swap(v,low,high);
                reverse(v,low+1,high-1);
        ENDIF;
END reverse;
```

Again let us use computational induction and simple induction on the length of the computation to justify that the procedure reverse works.

Originally the elements are in the order :-
.....,v(low),v(low+1),.....,v(high-1),v(high),.....

If high ≤ low then the procedure does nothing which is in agreement with the specification.

If high > low then we execute swap(v,low,high).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call swap(v,low,high) works correctly.	Assuming the computation halts normally, the length of computation of swap(v,low,high) is shorter than the length of computation of reverse(v,low,high). Therefore by simple induction we may assume that swap(v,low,high) works correctly.

This means that the elements are in the order :-
.....,v(high),v(low+1),.....,v(high-1),v(low),.....

Then reverse(v,low+1,high-1) is called.

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call reverse(v,low+1,high-1) works correctly.	Assuming the computation halts normally, the length of computation of reverse(v,low+1,high-1) is shorter than the length of computation of reverse(v,low,high). Therefore by simple induction we may assume that reverse(v,low+1,high-1) works correctly.

So this means that the elements of v between low+1 and high-1 will be reversed. This means that the elements will now be in the order:-

.....,v(high),v(high-1),.....,v(low+1),v(low),.....

Which means that all the elements of the vector v between low and high have been reversed.

So assuming halting, correct results will be obtained. That is, the procedure reverse is partially correct.

Class Discussion

Can computational induction be used to check correctness of a single execution, assuming halting?
For example, to check the execution of reverse (c, 1, 5)
where c = (1, 2, 3, 4, 7) we have to perform swap (c, 1, 5)
so c is now (7, 2, 3, 4, 1) assuming swap (c, 1, 5) works.
Now we execute reverse (c, 2, 4)
and so c is now (7, 4, 3, 2, 1) assuming reverse (c, 2, 4) works.
So we see that in this single execution, assuming halting, that c is reversed.
The above is not a valid argument and c may not actually get this value. Why?

Example 3

In this example the programmer has made an error as indicated.

```
PROCEDURE reverseb(INOUT v:vector; IN low,high: integer) IS
-- SPECIFICATION AS BEFORE.

BEGIN
        IF high > low
        THEN swap(v,low,high);
                reverseb(v,low+2,high-2);   -- programmer error here
        ENDIF;
END reverseb;
```

Again let us use computational induction and simple induction on the length of the computation to find out that there is an error.

Originally the elements are in the order :-
.....,v(low),v(low+1),v(low+2),..... ,v(high-2),v(high-1),v(high),.....

If high $\leq$ low then the procedure does nothing which is in agreement with the specification.

If high > low then we execute swap(v,low,high).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call swap(v,low,high) works correctly.	Assuming the computation halts normally, the length of computation of swap(v,low,high) is shorter than the length of computation of reverseb(v,low,high). Therefore by simple induction we may assume that swap(v,low,high) works correctly.

This means that the elements are in the order :-
.....,v(high),v(low+1),v(low+2),..... ,v(high-2),v(high-1),v(low),.....

Then reverseb(v,low+2,high-2) is called.

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call reverseb(v,low+2,high-2) works correctly.	Assuming the computation halts normally, the length of computation of reverseb(v,low+2,high-2) is shorter than the length of computation of reverseb(v,low,high). Therefore by simple induction we may assume that reverseb(v,low+2,high-2) works correctly.

So this means that the elements of v between low+2 and high-2 will be reversed. This means that the elements will now be in the order:-

.....,v(high),v(low+1),v(high-2),..... ,v(low+2),v(high-1),v(low),.....

Which means there is an error.

Class Discussion

Can computational induction be used to check that a single execution is incorrect, assuming halting?
For example, to check the execution of reverseb (c, 1, 5)
where c = (1, 2, 3, 4, 7) we have to perform swap (c, 1, 5)
so c is now (7, 2, 3, 4, 1) assuming swap (c, 1, 5) works.
Now we execute reverseb (c, 3, 3)
and so c is now (7, 2, 3, 4, 1) assuming reverseb (c, 3, 3) works.
So we see that in this single execution, assuming halting, that there is an error and c is not reversed.
The above is not a valid argument and c may not actually get this value. Why?
Can we conclude there is an error in the function?

A subtle point

Let us assume that the program halts and that we use the method of computational induction to check it.

If it is correct the results obtained from the program, the specification, and from the computational induction proof itself, will all be identical.

Here is what happens when it is not correct. There will be different results from the specification and program. There will be different results from the specification and the computational induction proof. There may or may not be different results from the program and the computational induction proof.

The following example shows what can happen when execution does not halt.

Example 4

With this example, execution does not always halt.

```
FUNCTION ll (IN n:integer) RETURN integer IS
BEGIN
        IF n < 0
        THEN RETURN 0;
        ELSE RETURN ll (n+1);
        END IF;
END ll;
```

Also, it can be shown by computational induction that the above definition is partially correct with respect to both specifications below.

Specification 1: ll (n) = 0 when n < 0 and
ll (n) = -5 when n ≥ 0.

Specification 2: ll (n) = 0 when n < 0 and
ll (n) = +7 when n ≥ 0.

There seems to be a contradiction here but it is not a real contradiction. The apparent contradiction occurs for those values of n where the execution does not halt normally and the method of computational induction by definition says nothing about this situation. Indeed, if these apparent contradictions occur, this effectively shows that execution does not halt.

Class Discussion: How does the above example of not halting differ from the non halting execution presented in the class discussion following example 1?

Class Discussion: If there are operations to stop a (sub) computation, you can not assume that the length of a subcomputation is shorter that the computation itself. Under what circumstances can computation induction be used even in such a case?

NOTES

1) Computational induction does not use an explicit basis as the technique only guarantees that no false results are produced.

2) It is similar to the "step over" feature in debuggers for avoiding the displaying of detailed steps when executing functions or procedures. We recommend the use of the "step over" feature when debugging programs, and the use of computational induction when reasoning about programs. (They both prevent being swamped by too much information.)

3) Computational induction can be used for procedures and functions whether recursive or not, the use being identical in all cases.

Programming Large Systems

We have demonstrated how specifications and computational induction can be used to prove correctness of programs. For expository reasons, the examples we have given have been simple ones. Without automatic methods, large programs can not be proved correct because quite simply the proof is usually longer than the program itself. So if one has doubts about the correctness of the program, one will have even greater doubts about the validity of the proof! However the technique of assuming that internal function and procedure calls work when reading programs is a very powerful technique for understanding and checking large programs and systems.

For example, suppose we have a very large system in which P calls Q1,Q2,Q3. To check P using these methods, read the specifications of P,Q1,Q2,Q3 and assume

that calls to them will work. Then check that the statements of P make sense with respect to these assumptions. No other specifications and no other statements need to be looked at when checking or writing P.

This ability of looking at only one item (i.e. a function or procedure) in detail and items directly connected to it in summary (i.e. specifications) makes it easier to develop programs and systems within teams. A team member needs to know the specifications of the procedures and functions that he calls. He does not need to know the detailed statements in the procedures and functions written by others.

Finally, this is a fundamental technique of thinking and enables one to decide what needs to be known in summary, what needs to be known in detail, and what may be ignored. It enables the application of mental effort where it is needed and eases problem solving.

Class Discussion: A technique used for checking plans is to be optimistic about the outcome of all sublans tasks etc. How is this technique similar to Computational Induction?

Inductive Assertions or Loop Invariants

The principle of computational induction is an excellent aid for reasoning about procedure and function calls. Inductive assertions or loop invariants enable us to reason about loops. Again this technique assures partial correctness only - termination is not guaranteed. We illustrate this technique with an example

```
-- SPECIFICATION:
-- sq and n are non negative integers. sq = n^2 after executing the statements below.

i:=0; sq := 0;
LOOP

  -- Inductive Assertion or Loop Invariant:-  we claim that  sq = i^2
  WHEN i = n EXIT;
  sq := sq + 2*i +1;   -- the reason for this form will become clear below
  i := i + 1;

END LOOP;
```

Proof:

We have to guess some property that remains invariant when executing the loop. The property $sq = i^2$ is similar to the specification but uses the loop index i instead of n its final value. This is a good guess. We write this property before the exit conditions of the loop. i.e the WHEN statements in the loop.

Now we check if that this property is true when we first reach the assertion. Clearly this is so because sq = i = 0 and so $sq = i^2$ on loop entry.

Assume this property holds at the head of the loop and then prove that it will continue to hold after executing one more iteration of the loop. Let us say that the value of i and sq on loop entry are i = k and sq = $i^2 = k^2$. Now if we execute the loop once more the value of sq and i will now be :-

sq = k^2+ 2k + 1 =$(k+1)^2$
i=k+1

After executing the statements of the loop once more we have sq = i^2 just as before. So no matter how many times we go around the loop, sq = i^2 remains correct. We can only leave the loop when i = n and so when we exit the loop sq = n^2.

This demonstrates partial correctness of the statements above.

NOTE - to reason about the loop we had to introduce symbols for the values of i and sq at the head of the loop. This is common when reasoning about loops.

Handling Loops by Transforming them into Procedures

Another way of handling the proof of correctness of a loop, is to convert it to procedure(s), and in fact this can always be done systematically. Then use computational induction to prove the partial correctness of the procedures. For example, the previous loop and initialization code can be written as two procedures as follows.

```
PROCEDURE square(IN n:integer; OUT result:integer) IS
-- SPECIFICATION
-- result = n^2.

PROCEDURE looping(IN n,i,sq:integer; resullt:OUT integer) IS
-- SPECIFICATION
-- result = sq+n^2-i^2

--THE BODIES OF THE FUNCTIONS

PROCEDURE square(IN n:integer; OUT result:integer) IS

BEGIN -- square
  looping(n,0,0,result);
  -- this effectively initialize i and sq to zero for procedure looping.
END square;

PROCEDURE looping(IN n,i,sq:integer; resullt:OUT integer) IS

BEGIN -- looping
 IF i = n
 THEN result:= sq;
 ELSE looping(n, i + 1, sq + 2*i +1, result);
 ENDIF;
END looping;
```

Exercise: Prove by computational induction that the previous two procedures are partially correct.

SUMMARY

Specifications - What has to be done NOT how to do it.

Computational Induction - proof method for procedures and functions based on simple induction on the length of computation. Enables you to decide what can be ignored, what needs to be studied in summary (i.e. specifications), and what must be looked at in every detail (i.e. the procedure or function being checked).

Inductive Assertions or Loop Invariants - Proof technique for loops.

Handling Loops by Transforming them into Procedures - Proof technique for loops.

Other Inductive Techniques - Structural Induction can be used for proving halting or total correctness. It is a generalization of the principle of simple induction. We do not discuss this here and the interested reader is referred to Zohar Manna's book "Mathematical Theory of Computation".

Exercises:

1) Prove by computational induction that the following functions are partially correct

```
FUNCTION f(IN n:integer) RETURN integer IS
-- SPECIFICATION
-- f(n) = n!

  FUNCTION ff(IN m,n:integer) RETURN integer IS
  -- SPECIFICATION
  -- ff(n) = n! * m

  BEGIN -- ff
        IF n = 0
        THEN RETURN m;
        ELSE  RETURN ff(m*n,n-1);
        END IF;
  END ff;

BEGIN -- f
  RETURN ff(1,n);
END f;
```

2) Complete the following function using recursion. Use computational induction to check your procedure. DO NOT RUN the procedure.

```
size: CONSTANT integer :=100;

TYPE vector IS ARRAY (1..size) OF integer;

FUNCTION palindrome (IN a:vector; IN low,high:integer) RETURN boolean;
-- SPECIFICATION
-- the elements of the vector a(low),a(low+1), .... ,a(high-1),a(high) are not
-- affected by reversing them
```

3) The towers of Hanoi problem. You are given a pile of n disks of decreasing size where disk 1 is the largest and disk n the smallest. There are three pegs (A,B,C) on which disks may be placed. You are allowed to move one disk from one peg to another provided it goes on a larger disk. Initially all the disks are on peg A with the disk 1 at the bottom and the disk n on the top. The following program will move the disks to peg C in such a way that at no stage will you have a large disk on a small disk. Use computational induction to check this claim.

```
TYPE peg IS ('A','B','C');

PROCEDURE hanoi(IN n: integer; IN start,finish,extra:peg) IS

-- SPECIFICATION
-- Prints the moves needed for
-- moving disks 1,2,3, ... ,n  from peg "start" to peg "finish" using peg "extra"
-- as an auxiliary peg.  Never put a small disk on a large disk.

BEGIN -- hanoi
        IF n=1
        THEN put('MOVE'); put(start); put('TO'); put(finish); new_line;
        ELSE  hanoi(n-1,start,extra,finish);
                put('MOVE'); put(start); put('TO'); put(finish); new_line;
                hanoi(n-1,extra,finish,start);
        ENDIF;
END hanoi;

hanoi(4,'A','B','C');
```

Part 2

Functional Programming in LISP

CONTENTS

SUGGESTED READING

LISP by P. Winston
Common Lisp - the Language by G. L. Steele

History of the **LISP** language

First version developed about 1959 by Prof. John Mcarthy ran on the IBM7090 computer which had 32K words of main memory.

Language designed for processing symbolic data.

Typical uses include Theorem proving, Formula mamipulation, Symbolic Integration and differentiation, Game playing, Expert Systems, Fast Prototyping etc.

Early implementations of the language were interpreted and not compiled.

Language has a very primitive syntax but the language is easy to extend and is provided with many functions. Program and data have the same form and data can be evaluated (or executed).

Special purpose LISP machines have a very advanced user friendly program development environment and are provided with 30000 functions. Very good for building prototype systems quickly (fast prototyping).

Language is not strongly typed - variables can take values of any type. However, type checking takes place at run time.

Pure LISP is a functional language and requires a recursive programming style as there are no loops, no jumps and no possibility to update variables.

Data types

Lists (Trees):	(A B C)	(A (A B) (C D) E)		
	( )	NIL {Empty List}		
Atoms				
	Symbols:	A	BIG-NUMBER	NIL
	Numbers:			
		Integer:	27	
		Floating Point:	27.3	4E-6

Entering Constants and Expressions to the Interpreter

Numbers are written without special annotation. Symbols and lists are quoted.

Examples:	12.34	'XYZ	'(A B C D)

Expressions are written as lists in prefix notation: (**fn** ARG_1 ARG_2 . . . . ARG_n)

Examples:	(**+** 1 2 3 4)	(**max** 1 2 3 4)

User Interface

The user interface is similar to a desk calculator. The **LISP** system reads expressions , evaluates them , and prints their results without the leading "quote". We illustrate this first with constants and arithmetic functions and then go on to functions for manipulating lists.
Note that ";" indicates the start of a comment - till carriage return.

User enters expression	Value	System responds
123	123	123
'x	'x	x ; abbreviated quote
(quote x)	(quote x)	x ; standard quote
'(A B (C D))	'(A B (C D))	(A B (C D))
' ' X	(quote (quote X))	(quote X)
'(A B '(C D))	??	??
(A B (C D))	??	??

Arithmetic functions

Expression	Value
486	486
(**+** 123 654)	777
(* 5 10)	50
(*/* 10 5)	2
(**-** (* 5 10) (*/* 10 5))	48
(**-** (* 5 10) (*/* 10 5))	48
(**max** 1 2 3)	3
(**min** 1 2 3)	1
(**expt** 2 3)	8
(**sqrt** 4.0)	2.0
(**abs** -5)	5

(**float** 5)	5.0
(**truncate** 4.5)	4
(**round** 4.5)	5
(**truncate** 14 4)	3 ; quotient
(**rem** 14 4)	2 ; remainder

List manipulation functions

car , **first** - First element of a list
cdr , **rest** - Remaining elements of a list

IMPORTANT NOTE:
You may not assume that (first '()) = (rest '()) = (car '()) = (cdr '()) = '() even though some LISP systems work that way. As far as we are concerned, these functions are undefined in this case.

Expression	Value
(**car** '(A B C))	'A
(**first** '(A B C))	'A
(**cdr** '(A B C))	'(B C)
(**rest** '(A B C))	'(B C)
(**first** '((A B) (A B)))	??
(**rest** '((A B) (A B)))	??
(**car** '(**cdr** (A B C)))	??
(**car** (**cdr** '(A B C)))	??
(**car** (**cdr** (A B C)))	??

You can abbreviate combinations of car and cdr

(**cadr** . . . .) = (**car** (**cdr** . . . .))
(**cdadr** . . . .) = (**cdr** (**car** (**cdr** . . . .)))

Exercises

1) Are the following lists , symbols , numbers , or something else ?
   a) (jerusalem college - computer department)
   b) ((A B) (C D))
   c) 3
   d) (3)
   e) atomic-bomb
   f))(
   g) ((( )))
   h) ((A B C
   i) *

2) What values are returned by the following expressions?
   a) (**/** (**+** 3 1) (**-** 3 1))
   b) (**min** (**max** 3 4 5) (**min** 8 9 10))

3) Draw trees corresponding to the expressions in question 2.

4) How would you represent lists in the style of Pascal?

5) What are the values of the following expressions?
   a) (**first** (**rest** '((A B) (C D))))
   b) (**rest** (**first** '((A B) (C D))))
   c) (**rest** (**first** (**rest** '((A B) (C D)))))
   d) (**first** (**rest** (**first** '((A B) (C D)))))
   e) (**first** '(**first** (**first** (**first** (A B)))))
   f) '(**rest** (**rest** (**rest** (**rest** (A B)))))

6) Write combinations of **first**'s and **rest**'s for accessing X from the following lists.
   a) '(A B X D)
   b) '(A (B (X D)))
   c) '(((A (B (X) D))))

7) Rewrite ' ' (D) in the form of (**quote** . . .)

Truth values and predicates

Truth - T

False – Nil also ()

T, Nil, () evaluate to themselves no need to write 'T, 'Nil, '().

Expression	Value
T	T
Nil	Nil

atom - tests if argument is an atom.
listp - tests if argument is a list.

Note : Nil is both an atom and a list. Therefore
(**atom** nil) = (**atom** ( )) = T
(**listp** nil) = (**listp** ( )) = T.

Note : When the **LISP** system tests a value for true or false, it treats any value other than nil as true.

(**member** <atom> <list>) returns nil if <atom> is not a member of <list>.

Expression	Value
(**member** 'X '(A B))	nil
; Look what happens when <atom> is a member of <list>	
(**member** 'X '(A B X D))	'(X D)
(**member** 'X '(X D))	??
(**member** 'X (**rest** (**member** 'X '(U V X A B X D))))	??

(**eql** ARG ARG_2) - Equality test for atoms.
(**equal** ARG_1 ARG_2) - Equality test for lists (trees) or atoms.

Expression	Value
(**eql** 1 1.0)	T
(**eql** '(A B) '(A B))	unknown value returned
(**equal** '(A B) '(A B))	T

(**numberp** ARG) - true if argument is a number.
(**zerop** ARG) - true if argument equals zero.
(**lessp** ARG_1 ARG_2 . . . ARG_n) - true if $ARG_1 < ARG_2 < . . . < ARG_n$.
(**greaterp** ARG_1 ARG_2 . . . ARG_n) - true if $ARG_1 > ARG_2 > . . . > ARG_n$.
(**null** ARG) true if ARG = nil.
(**consp** ARG) - true if ARG isn't an atom (meaning, ARG is a non-empty list).
(**not** ARG) - negation
(**and** ARG_1 ARG_2 . . . ARG_n) - and (incrementally evaluated)
(**or** ARG_1 ARG_2 . . . ARG_n) - or (incrementally evaluated)

Conditional expressions

(**if** <TEST> <EXPRESSION>)

(**if** <TEST> <EXPRESSION$_1$> <EXPRESSION$_2$>)

(**cond** (<TEST$_1$> <EXPRESSION$_1$>)
(<TEST$_2$> <EXPRESSION$_2$>)
..
..
(<TEST$_n$> <EXPRESSION$_n$>))

Let and **let*** - once only assignment (local variables)
(**let** ((<VARIABLE$_1$> <EXPRESSION$_1$>). . . .(<VARIABLE$_n$> <EXPRESSION$_n$>))
<EXPRESSION>) ; value of the **let**
{also (**let*** . . .)}

example :
(**let** ((pi 3.14159) (radius 2) (area (***** pi radius radius))) ; end of variable list
area) ; value of **let**

This would cause an error. **Let** makes parallel value-to-variable assignments. Therefore pi and radius can't be used to give a value to area.
There are two ways around this.

a) **let***
Let* makes a serial value-to-variable assignment therefore variables that have already been assigned values can be used in assigning values to later variables.

b) nested **let**
(**let** ((pi 3.14159) (radius 2)) ; end of variable list
(**let** ((area (***** pi radius radius))) ; end of nested variable list
area))

Class discussion: What is the value of the following nested "let"?
(**let** ((v 5))
(+ v v (**let** ((v 7)) (* v v)) (**let** ((v (- v 5))) (* v v)) v))

Assignment and global variables
(NOT TO BE USED IN THE EXERCISES UNLESS EXPLICITLY PERMITTED)

Expression	Value
4	4
L	; error - undefined variable
(**setq** L '(A B))	'(A B)
L	?
'L	?

(**first** L)	?
(**rest** L)	?
(**first** 'L)	?
(**rest** 'L)	?

Setq does not evaluate its first argument. **Set** is like **setq** but it evaluates its first argument (i.e. (**setq** X . . .) is short for (**set** 'X . . .). This feature gives a form of indirect assignment).

(**setq** X 'A)	'A
(**set** X 5)	5
X	?
A	?

Exercises

1) What is the effect of evaluating :

a) (**setq** L1 (**setq** L2 '(A B C)))

b) (**first** (**setq** X '(A B C)))

c) (**first** '(**setq** X (A B C)))

2) What is the effect of evaluating (**set** (**first** '(A B)) 5) ?

Append , **list** , and **cons**

These functions are useful for building lists.

Expression	Value
(**setq** L '(A B))	'(A B)
(**append** L '(C D))	'(A B C D)
(**append** L L L)	??
(**append** '(A) '( ) '(B) '( ))	??
(**append** 'L L)	??
(**append** '((A) (B)) '((C) (D)))	'((A) (B) (C) (D))
(**list** 1 2 'A)	'(1 2 A)

(**list** L '(C D))	??
(**list** L L L)	??
(**list** 'L L)	??
(**list** '((A) (B)) '((C) (D)))	'(((A) (B)) ((C) (D)))
(**list** 'L)	'(L)
(**cons** 'A '(B C))	'(A B C)
(**cons** L '(C D))	??
(**cons** L L)	??

Length , **reverse** , **last** , and **subst**

Expression	Value
(**setq** L '(A B))	'(A B)
(**length** '(A B))	2 ;length of a list
(**length** '((A B) (C D)))	??
(**length** L)	??
(**length** (**cons** L L))	??
(**length** (**append** L L))	??
(**length** (**list** L L))	??
(**reverse** '(A B))	'(B A) ; reverse a list
(**reverse** '((A B) (C D)))	??
(**reverse** L)	??
L	??
(**reverse** (**cons** L L))	??
(**reverse** (**append** L L))	??
(**last** '(A B C))	'(C)

(**last** '(((A B) C D))) ??

(**last** 'X) ??

; **subst** - substitute ARG_1 in place of ARG_2 (should be an atom) in ARG_3

(**subst** 'A 'B '(A B C)) '(A A C)

(**subst** 'B 'A '(A B C)) '(B B C)

(**subst** '(**+** A 1) 'X '(* X X)) ??

Exercises (**append** , **list**, and **cons**)

1) Evaluate

(**append** '(A B C) '( ))

(**list** '(A B C) '( ))

(**cons** '(A B C) '( ))

2) For a non-empty list L, (**cons** (**first** L) (**rest** L)) = L.
Verify this for L = '(A B C) and for L = '(A).

3) Evaluate the following in order :

(**setq** authors '(dickens shakespeare))
(**cons** 'longfellow authors)
authors
(**setq** authors (**append** '(poe fleming) authors))
authors

4) Suppose that L = '(A B C).

a) Construct '(A B C D) using L.
b) Construct '(A B C (A B C)) using L.

5) Construct the list '(a (b) c) from 'a 'b 'c '( ) using only the function **cons**.

Exercises (**length** , **reverse** , **last** , and **subst**)

1) Evaluate in the order written:

(**setq** philosophers '(plato socrates aristotle))
(**length** philosophers)
(**reverse** philosophers)
(**subst** 'xeno 'aristotle philosophers)
philosophers
(**last** philosophers)

2) Evaluate (**subst** '(**+** A 1) 'X (**subst** '(**+** B 1) 'Y '(**+** (***** X Y) 1)))

The evaluation (or execution) of data.

Eval - The **LISP** interpreter evaluates **LISP** expressions

Expression	Value
(**setq** X '(**max** 1 2 3))	'(**max** 1 2 3)
X	'(**max** 1 2 3)
(**eval** X)	3 ; evaluate the value of X
'(**+** 1 2 3)	'(**+** 1 2 3)
(**eval** '(**+** 1 2 3))	6

Exercises

1) Evaluate (**eval** ' '(X))

2) Evaluate in the order written:

```
(setq  v1  'v2)
(setq v2  'v3)
(eval  (eval  'v1))
```

3) Evaluate (**eval** '(**+** 1 2 (**eval** '(***** 3 4))))

A Pascal like description of **eval**

```
function  eval (L : listoratom) : listoratom;
begin
        if L is an atom
        then if  L is a number
                then return  L
                else  return value of  L  {L is a symbol denoting a variable}
        else {L is a list}
                if  (first  L)  is a special form {LET, COND, DEFUN, IF, QUOTE, etc.}
                then process-special-form (L)
                else  { (first  L) is a function - so apply it}
                        begin
                        1)  use eval to evaluate each element of (rest  L)
                        2)  apply (first L) to the list of values obtained in 1)
                        3)  return the value obtained in 2)
                        end;
end;
```

Defining Functions

(**defun** <function name> (<parameters>) ; function header
<expression-1> . . . <expression-n>)

NOTE: With functional programming, only one expression may be written.

Value of function is the value of the last expression.
Value of (**defun** . . . .) is the function name.

Expression	Value
(**defun** f-to-c (temp) ; first definition ; SPECIFICATION ; "temp" is the temperature in Fahrenheit. ; The function returns the temperature in centigrade. (**/** (**-** temp 32) 1.8))	'f-to-c
(**f-to-c** 100)	37.77
(**setq** fever 100)	100
(**f-to-c** fever)	37.77
(**defun** f-to-c (temp) ; second definition ; SPECIFICATION ; "temp" is the temperature in Fahrenheit. ; The function returns the temperature in centigrade. (**setq** temp (**-** temp 32)) (**/** temp 1.8))	'f-to-c
(**f-to-c** fever)	37.77
fever	100
(**defun** pair-swap (pair) ; SPECIFICATION ; "pair" is a list of two elements. ; The function returns a list with the same two elements but interchanged. (**list** (**second** pair) (**first** pair)))	'pair-swap
(**pair-swap** '(1 2))	'(2 1)

Exercises

1) Suppose that <list> has at least two elements. Write a function (**swap2** <list>) whose value is like <list> except that the first two elements are reversed.
e.g. (**swap2** '(1 2 3)) = '(2 1 3).

2) Write a function for rotating a list to the left.
e.g. (**rotate-left** '(A B C)) = '(B C A).

3) Write a function for rotating a list to the right.
e.g. (**rotate-right** '(A B C)) = '(C A B).

Anonymous functions - **lambda** expressions

For example (**lambda** (x y) (**+** x y (***** x y)))

You can write a **lambda** expression anywhere that you can write a function name. Its main use is with functions like **mapcar** (see below) where it is passed as a parameter.

You can use a **lambda** function in the same way you use any function. For example
((**lambda** (x y) (**+** x y (***** x y))) 2 3) evaluates to 11.
function args.

Apply

(**apply** <**fn**> <parameter-list>) is similar to
(**eval** (**cons** <**fn**> <parameter-list>)))

For example, (**apply #'+** '(1 2 3)) would evaluate to 6.
#'+ means treat **+** as a function and not as an atom. **#'** is used with function parameters

Mapcar

Applies a function to elements of list(s) and produces a list of corresponding function values. If **F** is a function of one argument then (**mapcar #'F** '(a_1 … a_n)) is equivalent to (**list** (**F** 'a_1) … (**F** 'a_n)). If **G** is a function of two arguments then (**mapcar #'G** '(a_1 … a_n) '(b_1 … b_n)) is equivalent to (**list** (**G** 'a_1 'b_1) … (**G** 'a_n 'b_n)) etc .

Expression	Value
(**mapcar #'add1** '(1 2 3))	'(2 3 4)
(**mapcar #'+** '(1 2 3) '(4 5 6))	'(5 7 9)
(**apply #'+** (**mapcar #'F** '(1 2 3)))	value of **F**(1) + **F**(2) + **F**(3)
(**apply #'and** (**mapcar #'P** '(1 2 3)))	value of **P**(1) and **P**(2) and **P**(3)
(**mapcar #'(lambda**(x) (+ x x 1)) '(1 2 3))	'(3 5 7)

The function **count** for counting the number of atoms in a list, including atoms in sublists, can be defined with and without **mapcar**.

```
(defun  count1 (L)  ; with mapcar
; SPECIFICATION
; L is a list or atom.
; The function returns the number of atoms in L including atoms in internal lists.

        (cond  ((null  L)  0)
               ((atom L)  1)
               (t (apply  #'+  (mapcar  #'count1  L)))))

(defun  count2 (L)  ; without mapcar
; SPECIFICATION
; L is a list or atom.
; The function returns the number of atoms in L including atoms in internal lists.

        (cond  ((null  L)  0)
               ((atom L)  1)
               (t (+  (count2  (first L))  (count2  (rest  L))))))
```

Exercise

Given a list of values $(K_1 \ . \ . \ . \ K_n)$, a function F, and a test predicate P, use **mapcar** and **apply** to calculate $F(K_1)$ ***** . . . ***** $F(K_n)$
as well as $P(K_1)$ **or** . . . **or** $P(K_n)$.

Backquote , comma , and at-sign

These operators enable "selective evaluation" within lists.

	Expression	Value
	'(A B (**reverse** '(A B)))	(quote (A B (**reverse** (**quote** (A B)))))
but	`(A B **,**(**reverse** '(A B)))	'(A B (B A))
also	`(A B **,@**(**reverse** '(A B)))	'(A B B A)

Defining Macros

```
(defmacro  swap(L)
 ; SPECIFICATION
; L is a list of two elements.
; The macro returns a list with the same two elements but interchanged.

`(list  (second  ,L)  (first  ,L)) )
```

When compiling, (**swap** '(1 2)) is replaced by (**list** (**second** '(1 2)) (**first** '(1 2))) and this is what would be evaluated subsequently.

Symbolic Differentiation

Here is a **LISP** program for determining the formula of the differential coefficient of a fully bracketed expression built from the constant C, the variable X, and the operators +, -, *, /. For example, an expression such as (C + ((X - C) * (C / X))). The expression must be written as a nested list with spaces around the operators.

```
(defun d(E)
; SPECIFICATION
; E is a fully bracketed expression as above.
; The function returns an unsimplified formula for dE/dX.

(cond ( (equal E 'C)  0 )
      ( (equal E 'X)  1 )
      (t  (let ( (u (first E)) (op (second E)) (v (third E)) )
            (cond ( (member op '(+ -))  `(,(d u) ,op ,(d v))
                  )
                  ( (equal op '*)        ; please complete
                  )
                  ( (equal op '/)        ; please complete
                  )
            ))))
```

Pure **LISP**

Pure **LISP** is the subset of **LISP** which excludes features such as update of variables {**set**, **setq**, **setf**, etc.} and update of data structures {**rplaca**, **rplacd**, **nconc**, etc. }. Computaional induction is easiest to use in Pure **LISP**.

The **let** feature for defining local variables can be looked at as a "once only assignment". Its use is also permitted in **Pure LISP**. A read/write to a file would not be acceptable in **Pure LISP** as it is similar to an update of data structures.

The student is required to program all exercises in **Pure LISP** as it is assumed that he has experience in conventional programming techniques and does not need further practice of these techniques in **LISP**.

Example of Computational Induction in LISP

```
(defun  F  (E  L)
; SPECIFICATION
; E is an atom ,  L is a list
; (F  E  L) = ( ) if the atom E is not a member of the list L.
; (F  E  L) ≠ ( ) if the atom E is a member of the list L.

        (cond  ((null L)  L)
               ((equal  E  (first  L)) L)
               (T  (F  E  (rest  L)))))
```

Proof that the function works to specification

a) if L = ( ) the value of the function from the function definition is L = ( ). Satisfies specification.
b) otherwise L is a non-empty list. If (**first** L) = E then the value of the function is the non-empty list L. Satisfies specification.
c) otherwise L is a non-empty list and (**first** L) does not equal E. So from the function definition, (**F** E L) = (**F** E (**rest** L)).

Assume that our claim is true for the recursive call, that is,
(**F** E (**rest** L)) = ( ) if atom E is not a member of (**rest** L) and
(**F** E (**rest** L)) does not equal ( ) if atom E is a member of (**rest** L).

In addition we know that (**first** L) does not equal E. Therefore E is a member of L if and only if E is a member of (**rest** L). Therefore :

(**F** E L) equals ( ) if the atom E is not a member of the list L.
(**F** E L) does not equal ( ) if the atom E is a member of the list L.

So, by assuming that the internal call (**F** E (**rest** L)) returns a correct result, we proved that (**F** E L) will also return a result satisfying the specification.

Whenever the function **F** halts, it returns a value satisfying its specification.

How do we show that the function does not go into an infinite recursion? By showing that the parameter list is ,in some sense, getting "simpler" on the recursive calls. Providing we choose a meaning of "simpler" which does not allow an infinite sequence of simplifications, this guarantees that the function does not have an infinite recursion. In our case, the value of the second argument is getting shorter on each recursive call.
Since a finite list can't be shortened an infinite number of times, this guarantees that there won't be an infinite recursion.

Evaluation sequence
We can present in mathematical style the sequence of expressions needed to evaluate a function.

example 1 : (**F** 'C '(A B C))
= (**F** 'C '(B C))
= (**F** 'C '(C))
= '(C)

example 2 : (**F** 'C '(A B))
= (**F** 'C '(B))
= (**F** 'C '( ))
= '( )

Further Discussion of Computational Induction in LISP

Consider the following LISP definitions.

```
(defun square (n)
; specification
; n must be non negative integer
; the function returns n^2

        (square-aux n 0) )

(defun square-aux (n r)
;  n is a non-negative integer
;  r is any integer
;  the function returns n^2 + r

        (cond (( = n 0 )  r)
                (t  (square-aux  ( - n 1) ( + r n n -1) )
```

Class discussion: How do we prove partial correctness of the functions "square" and "square-aux" ?

Consider the following LISP definitions.

```
(defun sort (l)
; specification
; l is a list of numbers
; the function returns a list containing the
; same elements as l but in non-decreasing order

 (cond (( < (length l)  2)  l)
        (t (let ( (l1  (sort (oddones l )))       (l2 (sort (evenones l ))) )
                (merge  l1 l2 ))))))

(defun oddones (l)
; please complete specification and function
)

(defun evenones (l)
; please complete specification

        (if       (null l)
                  ()
                  (oddones (rest l)))
)

 (defun merge (l1  l2)
; please complete specification and function
)
```

Class discussion: What are the specifications of evenones, oddones, and merge. How do we prove partial correctness of the function "sort" ?

A methodology for recursive programming

1) Write the function headers and specifications and (then) the function definitions. In writing a function definition for example of a function F which receives a list L as a parameter, ask yourself how you can use the values of (F (first L)) and (F (rest L)) to compute the value of (F L), relying on computational induction as appropriate. Similarly if a function G receives a numeric parameter, ask yourself how you can use the values of (G (- N 1)), (G (- N 2)), ... (G (+ N 1)), (G (+ N 2)) ... to compute (G N). Carry on in the same way for other functions you are writing.

2) Use computational induction to check if the functions you are defining will work to their specifications. If you find an error, make appropriate changes and check again using computational induction. Repeat until you do not find errors.

3) However, infinite recursion is still possible. Now check if the arguments are getting simpler on the recursive calls. This means you won't have an infinite recursion. Now run and debug the program, checking the changes you make using computational induction.

DO NOT TRY TO UNDERSTAND THE BEHAVIOR OF THE RECURSIVE CALLS AD INFINITUM.

Note: Computational induction can be used for a set of (mutually recursive) functions as explained in part 1 of these notes.

Exercises

1) Prove by computational induction that
(**F** E L) = ( ) if E is not a member of L
= a non-empty list like L but starting from the first occurrence of E.
Do we need to check halting again?

2) Suppose that M and N are non-negative integers and P(M,N) is defined by:

```
(defun  P  (M  N)
 ; SPECIFICATION
; M,N are non negative integers.
; (P  M  N)  =  (M  +  N)!
                  M! N!

        (cond  ((or  (zerop  M)  (zerop  N))  1)
               (T (+  (P  (-  M  1)  N)  (P  M  (-  N  1))))))
```

a) show the evaluation sequence of (P 1 2).
b) why is infinite recursion impossible for non negative integer values of M and N ?
c) Prove by computational induction that (P M N) works to specification.

3) Show using computational induction that the following function is partially correct.

```
(defun sort3 (a b c)
; SPECIFICATION
; a, b, c are numbers
; the function returns a sorted list of these three numbers

(cond    ((> a b)   (sort3 b a c))
         ((> b c)   (sort3 a c b))
         (t   (list a b c))))
```

Tail recursion

A collection of functions, $F_1 \ldots F_n$, is said to be tail recursive if the(recursive) calls to $F_1 \ldots F_n$ do not occur "inside" any other function or as the "test" of a conditional. The function F(E L) that we defined earlier is tail recursive whereas P(M N) is not.

The functions **rev1** and **rev2** given below will reverse a list. (**Rev2-aux** is an auxiliary function.) **Rev1** is not tail recursive. **Rev2** and **Rev2-aux** are tail recursive.

```
(defun  rev1 (L)
; SPECIFICATION
; L is a list.
; L the function returns a list like L but with the elements reversed.
; However the elements in internal lists are not reversed.

         (cond  ((null L)  ( ))
                (T (append  (rev1  (rest L))  (list  (first  L))))))

(defun  rev2 (L)  ; L is a list
; SPECIFICATION
; L is a list.
; the function returns a list like L but with the elements reversed.
; However the elements in internal lists are not reversed.

         (rev2-aux  L  ( )))

(defun  rev2-aux  (L1  R)  ; L1 is a list , R  "accumulates" the result list
; SPECIFICATION
; L1 is a list.
; the function returns a list like L1 but with the elements reversed followed by
; the elements of R with no reversal..
; However the elements in internal lists are not reversed.

         (cond  ((null L1)  R)
                (T (rev2-aux  (rest L1)  (cons  (first  L1)  R)))))
```

Note: You will find it common in converting a recursive function into a tail recursive function that you will need an auxiliary function and an auxiliary variable to accumulate the result. The call (**rev2-aux** L ( )) effectively "initializes" the variables L1 and R.

Rev2 and **rev2-aux** are similar to the following Pascal like version.

```
read(L);
R  :=  '( );
L1  := L;
while L1 <>  ( )
do  begin
     R  := (cons (first L1)  R);
     L1 := (rest  L1);
     end;
write(R);
```

Note: The result is accumulated in R.

Tail recursion is important because the execution can be optimized so that the functions run in a bounded stack. You can get a feel for this by comparing the evaluation sequences for (**rev1** '(A B C)) and (**rev2** '(A B C)) given below.

```
(rev1 '(A B C))
= (append (rev1 '(B C)) '(A))
= (append (append (rev1 '(C)) '(B)) '(A))
= (append (append (append (rev1 '()) '(C)) '(B)) '(A))
= (append (append (append '() '(C)) '(B)) '(A))
= (append (append '(C) '(B)) '(A))
= (append '(C B) '(A))
= '(C B A)

(rev2 '(A B C))
= (rev2-aux '(A B C) '())
= (rev2-aux '(B C) '(A))
= (rev2-aux '(C) '(B A))
= (rev2-aux '() '(C B A))
= '(C B A)
```

Exercises

1) Present the execution sequence of the Pascal like version of **Rev2** and **rev2-aux** for the input '(A B C). The result should be similar to the execution sequence of (**Rev2** '(A B C)).

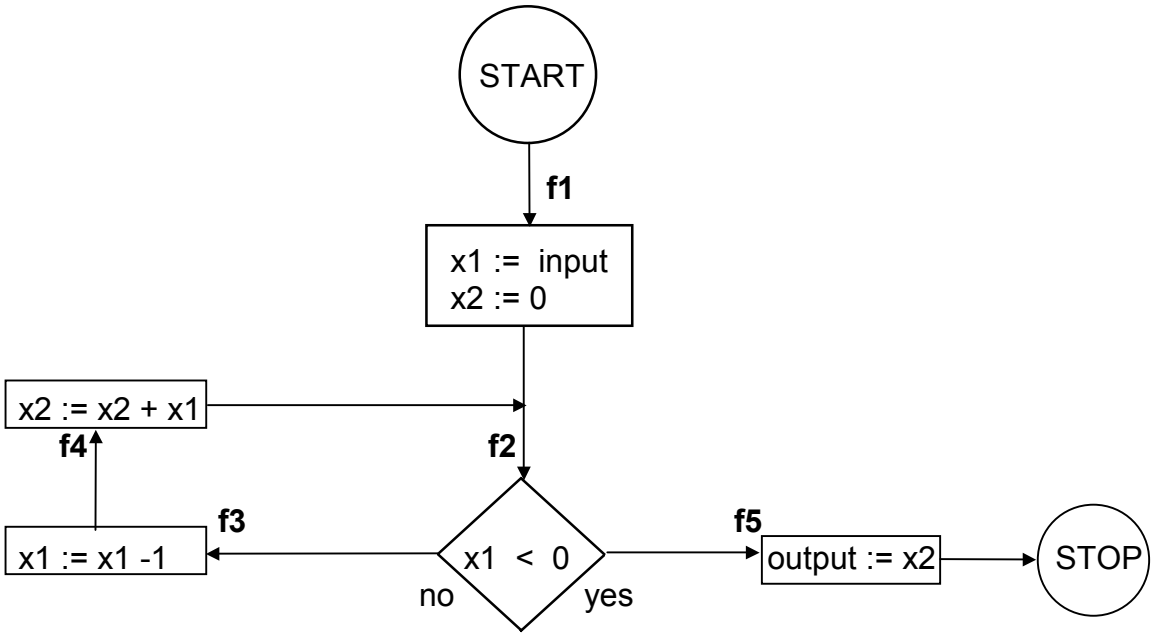
2) Write the usual recursive definition of the factorial function and then try to write a tail recursive definition.

3) Write evaluation sequences for 3! using both the definitions you wrote in answer to question (1).

4) Prove by computational induction that
(**rev2-aux** L R) = (**append** (**rev**1 L) R) and then deduce that
(**rev2** L) = (**rev1** L).

Converting flowcharts into tail recursive form

Any flowchart program and any sequential program can be converted into a set of tail recursive functions. We illustrate this with an example. Suppose the flowchart uses the variables x1 and x2.



Long form

f1 (input) = **f2** (input , 0)
f2 (x1 , x2) = **f5** (x1 , x2) if x1 < 0
f2 (x1 , x2) = **f3** (x1 , x2) if not (x1 < 0)
f3 (x1 , x2) = **f4** (x1 - 1 , x2)
f4 (x1 , x2) = **f2** (x1 , x2 + x1)
f5 (x1 , x2) = x2 {the output}

Short form

f1 (input) = **f2** (input , 0)
f2 (x1 , x2) = x2 if x1 < 0
f2 (x1 , x2) = **f2** (x1 - 1 , x2 + (x1 - 1)) if not (x1 < 0)

Here is the execution of one iteration of the loop in the flowchart when input=2.

place	input	x1	x2	output
f1	2			
f2		2	0	
f3		2	0	
f4		1	0	
f2		1	1	

Here is here is a similar computation using the functions in long form. Note the close correspondence between the two.

f1 (2) = f2 (2, 0) = f3 (2, 0) = f4 (1, 0) = f2 (1, 1)

Exercises

1) Complete the above executions until results are obtained. Present the execution of f1(2) in full using the functions in the short form above.

2) Write the following in tail recursive form in LISP.

```
read (m , n);
while n >  m do n := n - m;
write (n);
```

3) Present the computation steps of the above loop and the evaluation sequence of the recursive function(s) you wrote for m=3 and n=13.

Searching graphs and finding paths

There are many methods of searching, most of which are based on depth-first or breadth-first searches.

```
(defun  getsons (from)
; SPECIFICATION
; The function returns a list of nodes which can be reached in one step
; from the node "from".

;Please represent directed graphs in LISP and complete the function.
)

(defun  depth (from  to  graph)
; SPECIFICATION
; "graph" is a list representing a directed graph and "from", "to" are nodes.
; The function returns t if there is a path from node "from" to node "to"
; and returns () otherwise.

    (cond  ((member  to  (getsons  from  graph))  t)   ; "to" is a son of "from"
           (t (depth-sons  (getsons  from  graph)  to  graph))))
```

```
(defun depth-sons (son-list to graph)
; SPECIFICATION
; "graph" is a list representing a directed graph.
; "son-list" is a list of nodes and "to" is a node.
; The function returns t if there is a path from some node in "son-list" to node "to"
; and returns () otherwise.

    (cond ((null son-list) nil)
          ((depth (first son-list) to graph) t)
          (t (depth-sons (rest son-list) to graph))))
```

In the above, the functions return a boolean value only.
In the following function definitions, the path is returned.

```
(defun depth-path (from to graph)
; SPECIFICATION is similar to that of "depth" above except that
; if there is a path then one of the paths is returned as a list of nodes.

    (cond ((member to (getsons from graph)) (list from to))   ; "to" is a son of "from"
          (t ( let ((subpath (depth-sons-path (getsons from graph) to graph)))
                 (cond ((null subpath) nil)  ; value of let given by cond
                       (t (cons from subpath)))))))

(defun depth-sons-path (son-list to graph)
; SPECIFICATION is similar to that of "depth-sons" above except that
; if there is a path then one of the paths is returned as a list of nodes.

    (cond ((null son-list) nil)
          (t (let ((path (depth-path (first son-list) to graph)))
                 (cond ((null path) (depth-sons-path (rest son-list) to graph))
                       (t path))))))
```

Exercises:

1) Define a representation of directed graphs in LISP.

2) Define the function "getsons" using the representation you defined in the previous question.

3) Write functions breadth, breadth-sons, breadth-path, breadth-sons-path for breadth first search similar to the functions we gave for depth first search.

Structure of search algorithms using a nodelist

```
(defun search (from to graph)
; SPECIFICATION
; "graph" is a list representing a directed graph and "from", "to" are nodes.
; The function returns t if there is a path from node "from" to node "to"
; and returns () otherwise.

        (search1 (list from) to graph))

(defun search1 (from-list to graph)
; SPECIFICATION
; "graph" is a list representing a directed graph.
; "from-list" is a list of nodes and "to" is a node.
; The function returns t if there is a path from some node in "from-list" to node "to"
; and returns () otherwise.

        (cond ((null from-list) nil)
              ((eq (first from-list) to) t)
              (t (search1 "new-from-list" to graph))))
```

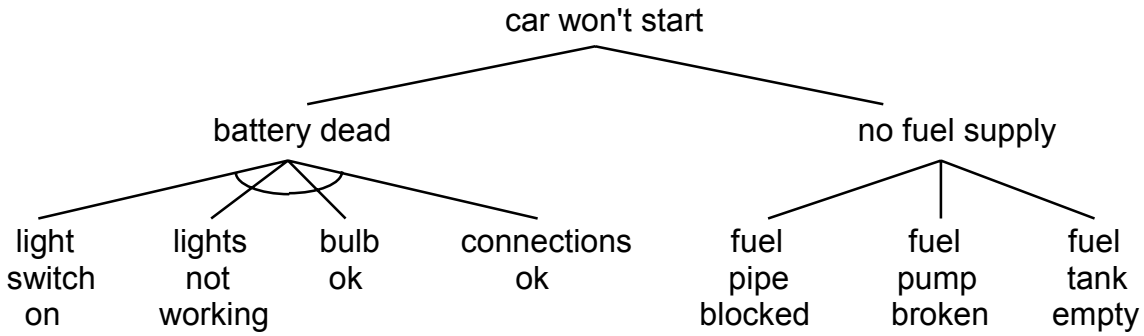
The form of "new-from-list" depends on the search. Below we list several possibilities. More are listed in the book "LISP" by P. Winston.

Search algorithm	new-from-list
depth first	(append (getsons (first from-list) graph) (rest from-list))
breadth first	(append (rest from-list) (getsons (first from-list) graph))
best first	like breadth first or depth first but (append ...) is sorted by estimated distance to the goal node "to". For greater efficiency do not use append. Instead, sort (getsons ...) only and merge with (rest ...).
hill climbing	like depth first but only sort getsons
beam search	like best first but keep limit new-from-list to a maximum of "n" nodes
branch and bound	like best first but sort/merge according to actual distance from the start node "from", plus the estimated distance to the destination node "to". This has the effect of determining the shortest path providing that estimated distance is less than or equal to the actual distance.

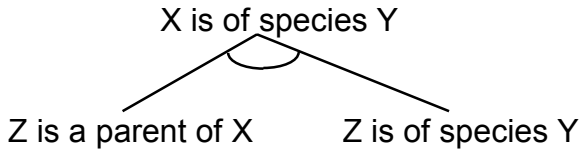
And/or trees (and graphs)

Two kinds of nodes : And node Or node

Example 1 Knowledge about a simple car repair



Example 2 If we allow variables in the graph or tree, we have a finite representation of a potentially infinite graph.



This can also be written in "rule form"

If Z is a parent of X **and**
Z is of species Y
then X is of species Y

In practice, we could use an and/or tree in two ways.

1) If we are given whether certain facts hold, we can deduce new facts higher up in the tree (see first definition of **given**)

2) If we are given nothing at all, we can use the and/or tree to ask the user questions, but only when needed, and on the basis of his answers make deductions. (see second definition of **given**)

Searching an and/or tree (or graph)

```
(defun checkandor (node  tree)
; SPECIFICATION
; "node" is a node and "tree" is a list representing an and or tree.
; The function returns the truth value of the node
; based on the truth values of the leaves.

    (cond  ((null (getsons node tree)) (given node))    ; leaf node
           ((andnode node) (checkand (getsons node tree) tree))
           ((ornode node) (checkor (getsons node tree) tree))
           (t (error 'tree-structure-error))))

(defun checkand (sonlist tree)
; SPECIFICATION
; "sonlist" is a list of nodes and "tree" is a list representing an and or tree.
; The function returns the logical and of the truth values of the nodes in "sonlist"
; based on the truth values of the leaves.

    (cond  ((null sonlist) t)
           ((checkandor (first sonlist) tree)   (checkand (rest sonlist) tree))
           (t nil)))

(defun checkor (sonlist tree)
; SPECIFICATION
; "sonlist" is a list of nodes and "tree" is a list representing an and or tree.
; The function returns the logical or of the truth values of the nodes in "sonlist"
; based on the truth values of the leaves.

    (cond  ((null sonlist) nil)
           ((checkandor (first sonlist) tree) t)
           (t (checkor (rest sonlist) tree))))

; specific definitions of andnode and ornode for the simple car repair example

(defun andnode (node)
; SPECIFICATION
; "node" is a node and the function tests if it is an and node.

    (member node '(battery-dead)))

(defun ornode (node)
; SPECIFICATION
; "node" is a node and the function tests if it is an or node.

    (member node '(car-won't-start no-fuel-supply)))
```

First definition of **given** - (basic facts known at start)

```
; specific definitions of given for the simple car repair example
```

```
(defun given (leaf)
; SPECIFICATION
; "leaf" is a leaf and the function returns the truth value associated with the leaf.

     (member leaf '(light-switch-on fuel-pipe-blocked) ))
```

Second defintion of **given** - (nothing known at start)

```
; Initialization
(setq true-ones ( ) )
(setq false- ones ( ) )

(defun given (leaf)
; SPECIFICATION
; "leaf" is a leaf and the function returns the truth value associated with the leaf
; by asking the user for the truth value.

     (cond ((member leaf true-ones) t)
           ((member leaf false-ones) nil)
           (t    (print leaf)
                 (terpri) ; t or nil answer assumed
                 (cond  ((read) (setq true-ones (cons leaf true-ones))
                        t) ; value returned
                        (t (setq false-ones (cons leaf false-ones))
                        nil))))) ; value returned
```

Exercises:

1) Present the evaluation sequence of checkandor for a small and/or tree.

2) By adding parameters to the functions checkandor, checkand, checkor, rewrite them to avoid the use of **setq** in the second definition of the function "given".

Part 3

Logic Programming in PROLOG

CONTENTS

CONTENTS (continued)

SUGGESTED READING

Clocksin, W.F. and Mellish, C.S., "Programming in Prolog", Springer Verlag, 2nd edition 1984.

Coehlo H., Cotta J. C., and Pereira L.M., "How to Solve it in Prolog", Laboratorio Nacional de Engenheria Civil, Lisbon, 2nd edition 1980

Cohen J., "Describing Prolog by its Interpretation and Compilation", CACM Vol 28 No. 12 December 1985 pp 1311 - 1325

Colmerauer A., "Prolog in 10 Figures", CACM Vol 28 No. 12 December 1985 pp 1296 - 1310

Kowalski, R. A. "Logic for Problem Solving", North Holland, 1979

Sterling L. and Shapiro E., "The Art of Prolog", MIT Press 1986

HISTORY OF PROLOG

PROLOG - PROgramming in LOGic

Originally developed by Alain Colmerauer at Marseille for solving problems Linguistic Analysis

Further developed at Edinburgh by David Warren

Based on a subset of Predicate Logic - Horn Clauses

Computation via a smart Theorem prover

Extra Logical features for efficiency

Logic Programming Foundations - R. A. Kowalski and J.A. Robinson

Parallel Prologs

- Flat Concurrent Prolog, Ehud Shapiro, Weizmann Institute
- Parlog, Keith Clark and Steve Gregory, Imperial College London
- Guarded Horn Clauses, K. Ueda, ICOT Japan

A TASTE OF PROLOG

Family Relationships

Clauses defining facts:

```
% SPECIFICATION: parent(X,Y) - X is a parent of Y.
1)  parent(adam, abel).
2)  parent(eve, abel).
3)  parent(adam, cain).
4)  parent(eve, cain).
5)  parent(cain, enoch).

% SPECIFICATION: male(X) - X is male.
6)  male(adam).

% SPECIFICATION: female(X) - X is female.
7)  female(eve).
8)  female(ada).
```

Clauses defining rules:

```
% SPECIFICATION: mother(X,Y) - X is mother of Y.
9  )  mother(X, Y) :- female(X),  parent(X, Y).

% SPECIFICATION: father(X,Y) - X is father of Y.
10)  father(X, Y) :- male(X),  parent(X, Y).

% SPECIFICATION: ancestor(X,Y) - X is an ancestor of Y.
11)  ancestor(X, Y) :- parent(X,Y).
12)  ancestor(X, Y) :- parent(X,Z), ancestor(Z, Y).
```

Queries and their answers:
(Note that ";" typed by the user after an answer, indicates a request for another answer)

```
?- parent(adam, eve) .
 no
?- parent(adam, Y) .
  Y = abel ;
  Y = cain ;
  no
?- parent(X, abel) .
  X = adam ;
  X = eve ;
  no
?- parent(adam, abel) .
  yes
?- father(X,abel) .
  X = adam
  yes
?- parent(X,Y) .
  X = adam  Y = abel ;
  X = eve  Y = abel ;
  X = adam  Y = cain ;
  X = eve  Y = cain ;
  X = cain  Y = enoch ;
  no
```

Exercises.

1) Translate the following queries into English and guess what answers a Prolog system will give to them.

```
?- ancestor(adam, enoch) .
?- ancestor(X, enoch) .
?- ancestor(adam, Y) .
```

2) Define the relations child(X,Y) and sibling(X,Y) in terms of the relations parent, mother, father.

3) Try and rewrite the family relationships example above so that the relationships mother, father, are defined by facts and the relationships parent, male, female are defined by rules. Note that you will not succeed completely, and you should indicate where there are problems.

Structures and Arithmetic in Prolog

Arithmetic Expressions are structures or trees. These structures can be decomposed into sub expressions. They are only evaluated on the right side of the "is" predicate.

```
?- X=5+3 .
  X = 5+3
  yes
?- X+Y = 5*3+6/7 .
  X = 5*3  Y = 6/7
  yes
?- 5*3+6/7 = X+Y .
  X = 5*3  Y = 6/7
  yes
?- X is 5+3, Y is X+1 .
  X = 8  Y = 9
  yes
?- X is 5+3, X is X+1 .
  no
?- 5+3 is 5+3 .
  no
```

Functors (function symbols) like operator symbols are used for defining tree like or record structures and are not used with the "is" predicate. Their purpose is **solely** to define data structures and not to compute a result.

```
?- X = tree(5, 9) .
  X = tree(5, 9)
  yes
?- tree(L,R)=tree(5*3, tree(1, 2)) .
  L = 5*3  R = tree(1, 2)
  yes
```

Symbolic Differentiation

Here is a PROLOG program for determining the formula of the differential coefficient of an expression built from the constant c, the variable x, and the operators +, -, *, /. For example, an expression such as c + ((x - c) * c / x) .

```
% SPECIFICATION: dd(E, D) means that D=dE/dx. The formula D is not simplified.
% E is an expression as above.
dd(c, 0 ).
dd(x, 1 ).
```

```
dd(U+V, DU+DV ) :- dd(U, DU), dd(V, DV).
dd(U-V, DU-DV ) :- dd(U, DU), dd(V, DV).
dd(U*V, V*DU+U*DV ) :- dd(U, DU), dd(V, DV).
dd(U/V, (V*DU-U*DV)/(V*V) ) :- dd(U, DU), dd(V, DV).
```

Exercise: Add clauses to the predicate "dd" to handle exp(x)=e^x, sin(x), cos(x).

Summary

Fact ,Rule, Query Based Programming

Constants, Variables and Predicates

Variables assigned to only once

Recursive definitions of predicates

Data Structures using Operators and Functors

Computation using Predicates

Response to Query - First Answer.
Further answers are obtained by using ";".
(Other Prolog Versions - All Answers)

"=" means unifiable, i.e. succeeds if substitutions for variables can be found to make the two sides identical. It can be used both to construct and to decompose data structures.

LOGICAL FOUNDATIONS (Mathematical Background)

Syntax of Clauses

A clause is a statement of the form :-

(A_1 **or** A_2 **or** ... A_m) **if** (B_1 **and** B_2 **and** ... B_n)

where each A_i and B_j are atomic formulae and $m,n \geq 0$.

An atomic formula (atom) is composed from a single (outermost) predicate and terms and has the form :-

predicate ($term_1$, $term_2$, ... $term_k$) (also $term_1 = term_2$)

A term is a constant, or variable or a composite term which may take two forms :-

functor ($subterm_1$, $subterm_2$, ... $subterm_k$)
or
$subterm_1$ operator $subterm_2$

The difference between functors and operators is purely syntactic. There is no difference semantically.

Horn Clauses (Prolog Clauses)

Horn Clauses (Prolog Clauses) require m=0, 1. i.e. **or** does not appear in the clause. In Prolog the **if** is written as ":-" . The **and** is written as "," .

Exercise: What are the atomic formulae and main terms in the following Prolog clauses.

a) female(eve).
b) mother(X, Y) :- female(X), parent(X, Y).
c) is-a-tree(tree(X,Y)) :- is-a-tree(X), is-a-tree(Y).

Semantics

A clause (A_1 **or** A_2 **or** ... A_m) **if** (B_1 **and** B_2 **and** ... B_n) containing the variables X_1, X_2 ..., X_k is to be interpreted as if it were universally quantified. i.e. in the general case it is understood as saying :-

" For all X_1, X_2 ... X_k if every formula B_j is true then at least one of the formulae A_i is true. "

Interpretation in the special cases when m=0 or n=0 :-

n=0, m>0 (fact)	" For all X_1, X_2 ... X_k at least one A_i is true. "
n>0, m=0 (query)	" For all X_1, X_2 ... X_k at least one B_j is false."
n=0, m=0 (empty clause)	"False or Contradiction"

Sets of Clauses

A set of clauses is to be interpreted as their conjuction. i.e. the claim is that they are all true.

Equivalence to Standard form

It may be shown that any logical formula built using the connectives **and, or, not, if, iff**, the quantifiers "**for all**", "**there exists**" may be converted to an equivalent set of clauses.

In general, it is not possible to produce an equivalent set of Prolog clauses (Horn Clauses) for every such formula.

Exercise: Convert the formula " a(X) **if** (b(X) **or** c(X)) " into a set of two Horn (Prolog) clauses.

Prolog Resolution Principle

Prolog uses a deduction rule called "Resolution" to deduce a new query (goal list) from a query (goal list) and a Prolog clause.

e.g Query: ?- a, b, c .
Clause: a :- e, f.
New query: ?- e, f, b, c .

e.g Query: ?- a, b, c .
Clause: a.
New query: ?- b, c .

Prolog attempts to answer queries by using the above principle to deduce the empty query.

No questions to answer - problem solved ! ! !

The Prolog resolution principle is a special case of a more general Resolution Principle which was devised by J. A. Robinson.

General Resolution Principle

Clauses given :-

$(A_1$ **or** A_2 **or** ... A_m **or** $R)$ **if** $(B_1$ **and** B_2 **and** ... $B_n)$
$(C_1$ **or** C_2 **or** ... $C_p)$ **if** $(R$ **and** D_1 **and** D_2 **and** ... $D_q)$

Conclusion clause :-

$(A_1$ **or** A_2 **or** ... A_m **or** C_1 **or** C_2 **or** ... $C_p)$
if
$(B_1$ **and** B_2 **and** ... B_n **and** D_1 **and** D_2 **and** ... $D_q)$

To justify this suppose all the B's and D's are true. If R is true then one of the C's is true by the second clause given. If R is false then one of the A's is true by the first clause. So in any case an A or a C is true which is what we require for the conclusion clause to hold.

Class discussion: Are the classical rules of deduction "modus ponens" and "modus tolens" included in the general resolution principle? What is the connection between the Prolog resolution principle and the general resolution principle?

Example - Family Relationships revisited

Prolog resolution to answer "?- mother (X, Y)." (Remember, we try to deduce the empty query)

First Solution of X

```
?- mother (X, Y) .                          ; initial query
?- female(X), parent(X,Y) .                 ; by clause 9)
?- female(eve), parent(eve,Y) .             ; substitution X=eve
?- parent (eve,Y) .                         ; by clause 7)
```

NOTE so far we have found that X=eve. For this value of X, Prolog Resolution will find the solutions of Y as follows.

First Solution of Y

```
?- parent(eve, Y) .
?- parent(eve, abel) .                      ; substitution Y=abel
? empty query - success                     ; by clause 2)
```

Second Solution of Y

```
?- parent(eve, Y) .                         ; solve another way
?- parent(eve, cain) .                      ; substitution Y=cain
? empty query - success                     ; by clause 4)
```

No More Solutions of Y when X=eve

Second Solution of X

```
?- female(X), parent(X,Y) .                 ; solve another way
?- female(ada), parent( ada,Y) .            ; substitution X=ada
?- parent ( ada,Y) .                        ; by clause 8)
```

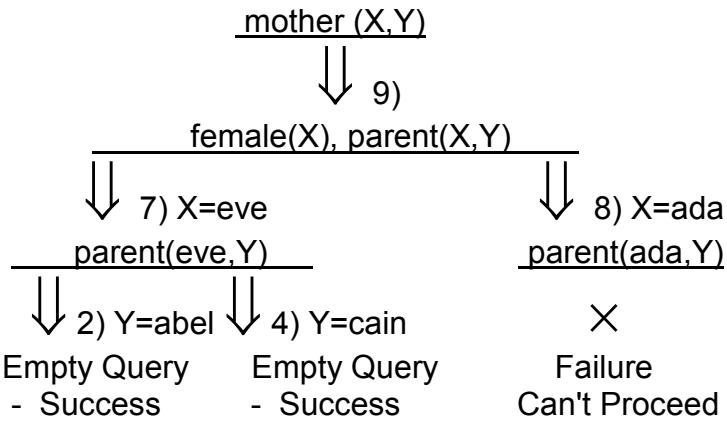
No Solutions of Y when X= ada

```
?- mother (X, Y) .                          ; not possible to solve another way
```

In this way Prolog resolution can find all solutions to the original query - namely

```
X=eve  Y=abel ;
X=eve  Y=cain ;
no
```

The above is compactly represented in a search tree. (Resolution is on the leftmost atom in a tree node.)



Instantiation and Unification

Part of the proof above involved substituting or instantiating variables with values or in the general case with terms.

A variable is said to be instantiated if it has received a substitution. A variable is uninstantiated if it has not been substituted.

For the proof to proceed in a sensible way, substitutions are made on the query and/or on a copy of a Prolog clause so that the head of this copy matches exactly one of the atomic formulae in the query (In Prolog matching is on the first formula of the query). Then resolution is used.

This process of making substitutions to make a match is called unification or matching.

Prolog will try to make the most general unification as part of its matching process.

e.g. We can unify p(X,Y) with p(f(U), g(V)) by substituting X=f(U), Y=g(V) or by making a more specific substitution X=f(U), Y=g(U), V=U. Prolog will make the first choice.

The concept of "most general unification" has a precise mathematical definition and algorithms are used for its determination.

Summary

Resolution - Proof or computational method

Unification - Most general substitutions

Search Trees

Exercises.

1) Use Prolog resolution to find an answer to the query
?- ancestor(adam, enoch) .

2) Draw a search tree for the query "mother(X, Y) ?" in depth first left to right order. The relation mother is defined by :-

mother (X, Y) :- parent (X, Y), female(X).

(NOTE the change in the order in the predicates parent and female and the effect this has on the search tree.)

3) What are the most general substitutions for unifying
a) p(f(U,g(V,a))) with p(f(X,Y))
b) p(X,a) with p(b,Y)

LANGUAGE FEATURES

Prolog Syntax

We use BNF syntax with the extensions that braces { } are used for grouping and <a>,,, means a non empty list of <a>'s separated by commas.

<query> ::= ?- <atomic formula> ,,,.
<clause> ::= <head>. | <head> :- <tail>.
<head> ::= <atomic formula>
<tail> ::= { <atomic formula> | <cut> },,,
<atomic formula> ::= <predicate> | <predicate> (<term>,,,)
<cut> ::= !
<term> ::= <constant> | <variable> | <functor>(<term>,,,)

A <constant> is either a number or an identifier starting with a small letter.

A <variable> is an identifier starting with a capital letter or with the underscore character "_" . All occurences of a variable in a clause have the same "value", but see anonymous variables below.

The variables "_" are anonymous variables. i.e. their values are not printed. (NOTE: if "_" occurs more than once in a clause each occurence is treated as a separate anonymous variable taking values independently of each other)

A <predicate> is an identifier starting with a small letter.

A <functor> is an identifier starting with a small letter.

For simplicity we omit the definition of operators.

Pure Prolog

In Pure Prolog the <cut> is not used and special predicates for I/O or for updating the set clauses are not used either.

Declarative Interpretation of Pure prolog

A clause in pure prolog is understood as previously described in the subsection "semantics". This is the declarative interpretation.

Procedural Interpretation of Prolog

A procedural Interpretation can be given to Prolog clauses whether they are pure or not. This is done by effectively specifying the order in which the search tree is built.

1) To solve the query "?-." (empty query) nothing needs to be done, it succeeds immediately.

2) To solve a query "?- Q_1, Q_2 ,... Q_N.", first solve the atom Q_1 and then solve the query "?- Q_2 ,... Q_N." . Note that each Q_i is an atom (atomic formula).

3) To solve an atom Q search the list of Prolog clauses in the order they are written for clauses whose <head> unify with Q. As a matching clause is encountered, solve the query made up from a copy of the right hand side of this clause with variable renamings and appropriate substitutions. (NOTE: if a clause has no right hand side an empty query will be produced which will succeed immediately by (2) above.) Of course, if no matching clauses can be found Q has no solution.

NOTES:

1) In the above recall that variables are substituted on at most once.

2) The purpose of renaming variables is to provide multiple copies of the variable for repeated substitutions. A process similar is used for handling variables in recursively defined procedures in conventional programming languages.

Class discussion: How could we define a predicate "pg(X, Y)" meaning X is the paternal grandfather of Y? How would the search tree for the query "?- pg(X, Y)." be built when variables are renamed?

The <cut>

The <cut> is denoted by "!" and its purpose is to reduce the size of the search tree. Its purpose is explained by examples.

Example 1 p(...) :- q(...), r(...) , ! , s(...) , t(...) .
p(...) :- u(...), v(...) .

The system will solve p(...) using the following method.

a) Try and find the **first solution only** of q(...), r(...)

b) If a solution for q(...), r(...) exists use **this solution only** and find all solutions of s(...), t(...) to solve p(...). **Do not search** for solutions using following clauses.

c) If no solution for q(...), r(...) exists, search for solutions for p(...), if any, in clauses following the above clause.

Example 2

```
% SPECIFICATION: abs(X,Y) means Y = | X |
abs(X, Y) :- X<0, Y is -X.
abs(X, X) :- X>0.
abs(0,0).
```

Now, if we need | -5 | we ask the query "?- abs(-5, Y) ." and get the result Y=5.

However, as the Prolog system attempts to find **all** solutions, it will attempt to try the second and third rules to find a solution even though we know there is only one solution and only one of the three rules will give a solution for y. We can rewrite the rules with a cut as follows

```
abs(X, Y) :- X<0, ! , Y is -X.
abs(X, X) :- X>0, ! .
abs(0,0).
```

A cut is like a point of no return. If in trying to solve a query we reach a cut, the cut immediately succeeds. Then the system will prevent searching for solutions using clauses following the current clause or finding other solutions to the atomic formulae preceding the cut.

Conditional Predicates

These can only be used on the right hand side of a Prolog clause.

When we write	*We intend the logical meaning*
P , Q	P and Q
P ; Q	P or Q
P -> Q ; R	If P (first solution) then Q else R.
if(P,Q,R)	If P (all solutions) then Q else R.

Negation as Failure - Closed World Assumption

Prolog permits the use of "not" in a query or on the right hand side of a clause. "not" means "failure to prove with the clauses available". That is we have a closed world assumption. "not" can be expressed in terms of the cut as follows:-

```
not(Atomic_formula) :- Atomic_formula, ! , fail.      % "fail" is a system predicate
not(Atomic_formula).                                  % which always fails.
```

The built-in command for "not P" is "\+ P"

\+ P If the goal P has a solution, fail, otherwise "not" succeeds.
No cuts are allowed in P.

Prolog List Structures

Prolog has a built in facility for handling lists.

```
[ ]              - the empty list
[a, b, c, d]     - a four element list
[a, [b, c, d]]   - nested list (list inside a list)
```

Prolog also supports the notation [H | T] to denote a list whose head is H and whose tail is the list T.

Thus [a,b,c] = [a | [b, c]] = [a | [b | [c]]] = [a | [b | [c | [ ]]]] .

We can mix notations
[H1, H2 | T] - first two elements H1 , H2. and remaining elements T.
[H | T1, T2] - badly formed list.

Data Structures Incorporating Variables

Prolog can manipulate data structures containing variables - that is partially specified structures.

Thus for example [a,b, X | Y] denote a list of at least three elements where a is its first, b its second, X its third is unknown and the remaining elements Y are unknown.

cons, first, rest in PROLOG

We use PROLOG lists to represent LISP list structures and write predicates for first, rest and cons in PROLOG. These predicates have extremely simple definitions. The last argument gives the result.

```
% SPECIFICATION: cons(L,R,Result)  means that Result is a list
% with head L and tail R.
cons (L ,R, Result) :- Result=[L | R] .

% SPECIFICATION: first(L,Result)  means that Result is
```

```
% the first element of the list L.
first(L, Result) :- L =[ Result | _ ] .

% SPECIFICATION: rest(L ,Result)  means that Result is
% a list like L but without the first element.
rest(L, Result) :- L =[ _ | Result] .
```

These can be written more concisely and efficiently as follows.

```
cons (L, R, [L | R]) .
first([Result | _ ], Result) .
rest([ _ | Result], Result) .
```

Note that in situations like this, the "=" predicate can be avoided altogether and this results in more efficient but less understandable programs.

Extra Logical Predicates

For reference we give a short list of these predicates.

read(X) - input a term terminated by full stop.
write(X) - output a term.
nl - new line.
op(Priority, Type, Name_as_string) - Operator Definition

A Caution: Multifile and dynamic predicate declarations must precede any other declarations for the same predicates!

```
:- dynamic Predspec, ...., Predspec.
```

For assert and retract, the predicate (or clause) concerned must be "dynamic" or undefined. The predicate must be enclosed in "()".

assert(C) - add a clause C to the existing clauses.
retract(C) - erase first clause in current set of clauses matching C.
listing - lists onto the current output stream all the clauses in the current interpreted program.

setof - Read this as "Set is the ordered set of all instances of Template such that Goal is satisfied, where that set is non-empty."

bagof - same as setof except the list returned will not be ordered, and may contain duplicates. This is faster.

```
| ?- setof(X, likes(X,Y), S).
```

might produce two alternative solutions

```
Y = yogurt      S = [david, joseph, moshe] ;
Y = cider       S = [bill, jan, joseph] ;
no
```

But,

```
| ?- setof((Y,S),setof(X,likes(X,Y),S),SS).
SS = [ (yogurt,[david, joseph, moshe]), (cider,[bill, jan, joseph])]
yes

| ?- setof(X, Y^(likes(X,Y)),S).
S = [bill, david, jan, joseph, moshe]
yes
```

Summary

Pure Prolog - No cut and extra-logical features

- Declarative Interpretation

Procedural interpretation for Prolog

<cut> to reduce search tree

not - failure to prove

Partially specified data structures

Exercises

1) p = q if r is true
s otherwise

Write this in prolog without a cut and with a cut.

2) Write a predicate for extracting the third element of a Prolog list.

3) Suppose we add the following definitions of the grandparent relation to our family relationships example.

g1(Person1, Person2) :- parent (Person1, Person3), parent (Person3, Person2).

g2(Person1, Person2) :- parent (Person3, Person2), parent (Person1, Person3).

g3(Person1, Person2) :- parent (Person1, Person3), ! , parent (Person3, Person2).

g4(Person1, Person2) :- parent (Person3, Person2), ! , parent (Person1, Person3).

a) Which of the relations will give the correct answer to the queries

"?- g_n(adam, enoch) ." n=1,2,3,4

b) What answers will be given to the queries

(i) ?- g3(adam, Person2) .
(ii) ?- g4(adam, Person2) .
(iii) ?- g3(Person1, enoch) .
(iv) ?- g4(Person1, enoch) .

4) Use the definitions of ancestor and parent given previously to draw a detailed search tree for the following queries.

a) ?- ancestor(adam, enoch). b) ?- ancestor(adam, eve).

5) Suppose we modify the definition of ancestor in the family relationships example as follows :-

```
ancestor(X, Y) :- parent(X,Y).
ancestor(X, Y) :- ancestor(X, Z), parent(Z,Y) .
```

This form causes an infinite loop - Try drawing a search tree in depth first left to right order for the query "?- ancestor(adam, eve)." .

PROGRAMMING TECHNIQUES AND EXAMPLES

Overview

As Prolog has no assignment statement, programs are inevitably recursive.

As no computation is done via functors, so to compute a function f(X1, X2, ... XN), we have to define a predicate f_p(X1, X2, ... XN, RESULT) in prolog whose meaning is RESULT = f(X1, X2, ... XN). Note the use of an extra variable for the result. (Similarly, when replacing a function by a procedure in a conventional programming language, an extra variable for the result is also needed)

In many cases we can reverse the roles of the input and output variables and run the program the other way round. i.e. Prolog **does not** have a fixed designation of input or output variables.

We now illustrate these and other ideas with examples. Several of the examples have been adapted from "How to solve it Prolog" by Coelho, Cotta and Pereira which has a wealth of examples, many of them classics.

Member

```
% SPECIFICATION: member(E,L) means E is a member of L
member(E, [E | _ ]).
member(E, [ _ | T]) :- member(E, T).
```

This will test for list mebership but what else will it do ?

```
?- member(3, [-1, -2, 3, 4] ) .
?- member(E, [-1, -2, 3, 4] ) .
?- member(E, [-1, -2, 3, 4] ]), E>0 .
?- member(5, L) .
```

If we use the cut, as below how is the above definition restricted and what do we get from the above queries.

```
member(E, [E | _ ]) :- !.
member(E, [ _ | T]) :- member(E, T).
```

Append

```
% SPECIFICATION: append (LIST1,LIST2,RESULT) means
% RESULT is LIST1 appended before LIST2.
append([ ], LIST, LIST).
append([HEAD | TAIL], LIST, [HEAD | TEMP] ) :- append(TAIL, LIST, TEMP) .
```

This will append lists but what else can it do ?

```
?- append([a,b], [c,d], L) .
?- append(L1, L2, [a, b, c]) .
?- append(L1, [c,d], [a,b,c,d]) .
```

If append is written with a cut as follows, how are things changed ?

```
append([ ], LIST, LIST) :- ! .
append([HEAD | TAIL], LIST, [HEAD | TEMP] ) :- append(TAIL, LIST, TEMP) .
```

Class discussion: What does a search tree for "?- append(A, B, [1, 2])." look like when the first definition of append (no cut) is used? What care needs to be taken regarding the variables in the tree?

To determine if L1 is a sublist of L2 the first definition of append (no cut) can be used as follows. (THIS DEFINITION OF sublist IS VERY INEFFICIENT.)

```
sublist(L1,L2) :-  append(Prefix, L1,Temp), append(Temp, Suffix, L2) .
```

Exercise: The previous definition of sublist is very inefficient. Write a definition of sublist which does not use append.

Computational Induction in PROLOG

When using computational induction in PROLOG, we assume that everything on the right hand side of a clause works and verify that the values received on the left hand side of the clause are as specified or claimed.

Class discussion: Consider the definition of the "member" predicate without a cut. How can computational induction be used to show that assuming halting, an answer to the query "?- member(a, L)." has the form L = [_ , ... , _ , a | _] where the number of occurrences "_" before "a" is greater than or equal to zero.

Class discussion: Consider the definition of the "append" predicate without a cut. How can computational induction be used to show that the response to the query "?- append($[a_1,...,a_m]$, $[b_1,...,b_n]$, C)." will be C=$[a_1,...,a_m,b_1,...,b_n]$.

Class discussion: Consider the definition of the "append" predicate without a cut. How can computational induction be used to show that the response to the query "?- append(A, B, $[c_1,...,c_k]$)." will be A=$[c_1,...,c_j]$, B=$[c_{j+1},...,c_k]$. (A, B may equal [ ].)

Exercise: Consider the definition of the "sublist" predicate based on the first definition of append without a cut. Use computational induction to show that assuming halting, the "sublist" predicate tests if L1 is a sublist of L2.

Generating Permutations

```
% SPECIFICATION: perm(L1, L2) means that L2 is a permutation of L1.
% L1, L2 are lists.
perm(L, [H | T] ) :- select(H, L, R), perm(R, T).
perm([ ], [ ]).

% SPECIFICATION: select(H, L, R) means that H is an element of the list L
% and R is a list of the remaining elements of L.
select(H, L, R) :- append(Prefix, [H], Temp), append(Temp, Suffix, L),
                   append(Prefix, Suffix, R).
% THIS DEFINITION OF select IS VERY INEFFICIENT.
```

Exercises:

1) The above predicate "select" is similar to a predicate for deleting an element from a list. What happens if H is not in L?
2) How can "select" be written with only two uses of "append" ?
3) Without using the predicate "append" write an efficient definition of the predicate "select" as defined above.
4) Write predicate(s) for putting eight queens on a chessboard such that no two queens may attack each other. That is, no two queens on the same row, column, or diagonal. HINT - base your solution on the predicate "perm" for generating permutations of the list [1,2,3,4,5,6,7,8].

Counting in Prolog

```
% SPECIFICATION: count(L,VALUE) counts the number of elements in a list L
% and returns this number in VALUE.
% L may contain internal lists.

count( [H | T], VALUE) :- listp(H), ! , count(H,M), count(T,N), VALUE is M+N .
count( [H | T], VALUE) :- ! , count(T,N), VALUE is 1+N .
count( [ ], 0).
% NOTE THAT THE CUT IS USED LIKE AN "IF THEN ELSE".

% SPECIFICATION: listp(L) tests if L is a list.
listp( [ ] ).
listp( [ _ | _ ] ).
```

Exercises

1) Write Prolog definitions for list union and list intersection.

2) Write a Prolog definition which will find all values of A and B such that A **directly precedes** B in the list L .

3) As 2) but there may be other elements intervening.

4) Write a prolog program to flatten a nested list.

Note - The predicate append is helpful in solving some of the above questions.

Data Base Lookup (D.Warren)

"Find all countries whose population density differ by at most five percent"

Area in thousands of square miles.	Population in millions.
area(china, 3380).	pop(china, 825).
area(india, 1139).	pop(india, 586).
area(ussr, 9709).	pop(ussr, 252).
area(usa, 3609).	pop(usa, 212).

```
% SPECIFICATION: density(C,D) means D is the population density of country C.
density(C,D) :- pop(C,P), area(C,A), D is (P * 1000) /A .

answer(C1, C2) :- density(C1, D1), density(C2, D2), D2 < D1, 20*D1 < 21*D2
```

(You can also try expressing this in terms of project, select and cartesian product.)

Exercise: [Pereira & Porto] Consider the query :-

"Is there a student such that a professor teaches him two different courses in the same room?" Write this query in terms of the predicates (relations) :-

```
student(Name, Course_name)
course(Course_name, Day, Room)
professor(Name, Course_name)
```

Reverse - with and without Append

Before looking at the program below, try writing recursive programs in lisp for list reversal, with and without append.

With append:

```
% SPECIFICATION: reverse1(L, R) means R is like the list L but reversed.
reverse1([ ], [ ]).
reverse1([H | T], Result) :- reverse1(T, Res1), append(Res1, [H], Result).
```

More efficient without append:

```
% SPECIFICATION: reverse2(L, R) means R is like the list L but reversed.
reverse2(L, Result) :- rev_append (L, [ ], Result).

% SPECIFICATION: rev_append(L1, L2, R) means R is like the list L1 but reversed
% followed by the elements of L2 in their original order.
rev_append([ ], L, L).
rev_append([H | T], L, Result) :- rev_append(T, [H | L], Result).
```

Quick Sort - with and without Append (M. Van Emden)

We leave it to the reader to complete the definition of the predicate split(L,V,HI,LE) and should take care to use the cut properly.

```
% SPECIFICATION: split(L, V, HI, LE) means that the list HI contains all elements
% of the list L larger than the value V and the list LE is contains all elements of the
% list L less than or equal to the value V.
split( [H | T], V, [H | HI], LE) :- .....
split( [H | T], V, HI, [H | LE]) :- .....
split( [ ], V, [ ], [ ]).
```

Quicksort **with** append:

```
% SPECIFICATION: sort1(L1,L2) means that the list L2 contains all the elements
% of the list L1 but L2's elements are arranged in non decreasing order.
sort1( [ ],[ ] )
sort1( [H | T], Sorted_list ) :- split(T, H, HI, LE),
                                 sort1(HI, S_HI),
                                 sort1(LE, S_LE),
                                 append(S_LE, [H | S_HI], Sorted_list).
```

More efficient Quicksort **without** append:

```
% SPECIFICATION: sort2(L1,L2) means that the list L2 contains all the elements
% of the list L1 but L2's elements are arranged in non decreasing order.
sort2(Unsorted, Sorted_list) :- sort_append(Unsorted, [ ], Sorted_list).

% SPECIFICATION: sort_append(L1,L2,R) means that the list R contains all the
% elements of the list L1 but arranged in non decreasing order and these are
% followed by the elements of L2 in their original order.
sort_append( [ ], L, L ).
sort_append( [H | T], L, RESULT ) :- split(T, H, HI, LE),
                                     sort_append (HI, L, S_HI),
                                     sort_append (LE, [H | S_HI],  RESULT) .
```

Queues (Difference Lists)

It is trivial to represent a queue by a list and use append to add a new element to the queue. However, this is not efficient and by using the concept that a variable can be part of a data structure we can get direct access to the "last pointer" in the queue and assign to it (but only once) when we wish to concatenate queues or add an element to a queue.

e.g. a queue of three elements is denoted by a structure such as q([a, b, c | L], L) . The elements in the queue is the difference between the first and second lists inside q(...), hence the term difference list is also used.

The predicate "joinsq" for joining a new element to a queue is defined by:

```
% SPECIFICATION: joinsq(New, Q1, Q2) means Q2 is like Q1 but with
% the element "New" added to its end.
joinsq( New, q(L1,L2), q(L1,L3)) :- L2=[New | L3].
```

or even more briefly as

```
joinsq( New, q(L1, [New | L3] ), q(L1,L3)).
```

This works because we can have variables in a data structure but remember these can be assigned to once. Concatenate is given below but try writing it without "=".

```
% SPECIFICATION: concatenate(Q1, Q2, Q3) means Q3 is a queue whose
% elements are those of Q1 followed by the elements of Q2
concatenate (q(L1,L2), q(L3,L4), q(L1,L4)) :- L2=L3 .
```

Exercises

1) Write a brief definition of concatenate with no right hand side (as a fact).

2) Write an even-odd sort program in PROLOG.

3) Write an insertion sort program in PROLOG.

4) Write a procedure to merge two ordered queues in PROLOG.

Map Colouring (Pereira and Porto)

This and the next example involve almost no programming.
Prolog's search strategy find's the solution from the facts given.

Colour a planar map with at most four colours.
Regions in the map are represented by variables.
The value the variables get are the colours.

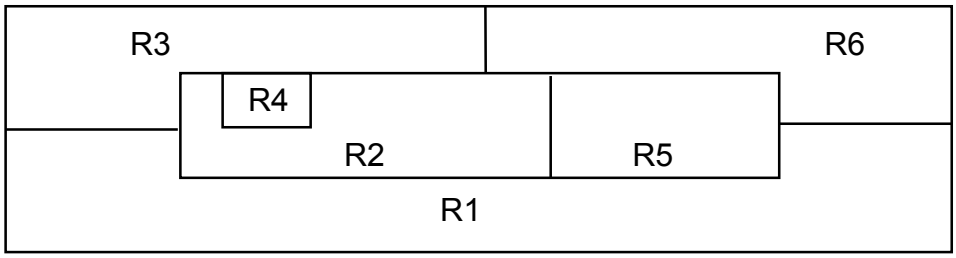
We specify the colours which can be adjacent and the map regions which are adjacent. That is the whole program.

Permitted adjacent four colours :-

```
% SPECIFICATION: adj(C1, C2) means C1 is one of the four colours and
% C2 is one of the four colours and they may be used as adjacent colours
% on the map (that is they are different).
adj(blue,yellow).     adj(blue,red).     adj(blue,green).
adj(yellow,blue).     adj(yellow,red).   adj(yellow,green).
adj(red,blue).        adj(red,yellow).   adj(red,green).
adj(green,blue).      adj(green,yellow). adj(green, red).
```

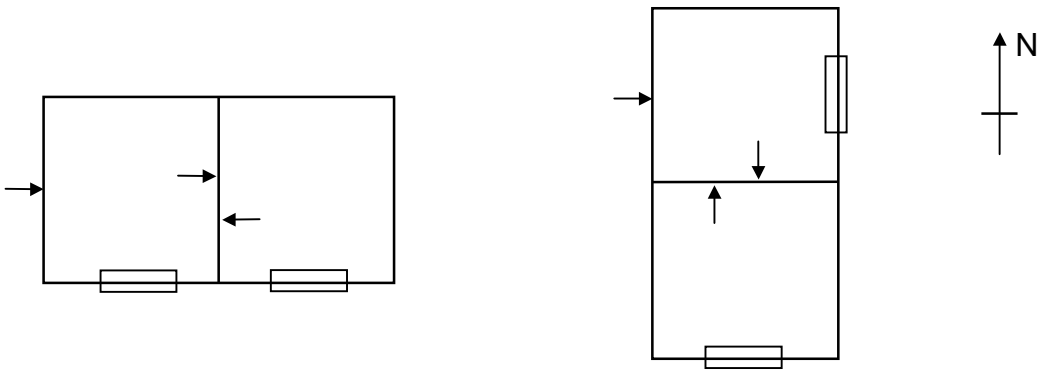
The predicate to colour a map where :-

R1 is adjacent to R2,R3,R5 and R6.
R2 is adjacent to R3,R4,R5 and R6.
R3 is adjacent to R4 and R6.
R5 is adjacent to R6.




```
% SPECIFICATION: colour(R1,R2,R3,R4,R5,R6) means that each of R1, R2, R3,
% R4, R5, R6 are one of the four colours and that adjacent regions on the map
% above are coloured by different colours
colour(R1,R2,R3,R4,R5,R6) :-  adj(R1,R2), adj(R1,R3), adj(R1,R5), adj(R1,R6),
                              adj(R2,R3), adj(R2,R4), adj(R2,R5), adj(R2,R6),
                              adj(R3,R4), adj(R3,R6),
                              adj(R5,R6).
```

Architectural Plans (Markusz)



Here are the planning requirements of the two roomed unit.

Each room has a window and an interior door.
Rooms are interconnected by an interior door.
One room has an exterior door.
A wall can have only one door or window.
No window can face north.
Windows cannot be on opposite sides of the unit.
Here again the trick is to represent what the valid directions are by a predicate. Define a predicate to define opposite directions. Use variables to represent the position of the exterior and interior door, and the windows.

We leave it to the reader to define the predicates direction, opposite, and notopposite. The planning is done as follows.

Key: ED - external door
ID1 - Internal door position in room 1
ID2 - Internal door position in room 2
W1 - Window position in room 1
W2 - Window position in room 2

```
% SPECIFICATION: plan(ED, ID1, W1, ID2, W2) means that ED, ID1, W1, ID2, W2
% are all directions satisfying the planning requirements above.
plan(ED, ID1, W1, ID2, W2) :-  frontroom(ED, ID1, W1), room(ID2, W2),
                               opposite(ID1, ID2), notopposite(W1, W2).

% SPECIFICATION: frontroom(ED, ID, W) means that ED, ID, W
% are all directions satisfying the planning requirements above.
frontroom(ED, ID, W) :- room(ID, W), direction(ED), ED=/=ID, ED =/=W.

% SPECIFICATION: room(ID,W) means that ID, W
% are all directions satisfying the planning requirements above.
room(ID,W) :- direction(ID), direction(W), ID=/=W, W=/=north.
```

Prolog in Prolog (Meta interpreters)

Meta interpreters are Prolog interpreters written in Prolog and they are very short, but why write them.

The default behaviour of depth first left to right search may be unsuitable for some problems. It turns out that it is more convenient to write a breadth first meta interpreter rather than write a specific breadth first program. Meta interpreters have also been written for reasoning about uncertainity. Sterling and Shapiro in their book "The art of Prolog" have many examples of such meta interpreters. The following meta interpreter for pure prolog uses depth first left to right search.

```
% SPECIFICATION: solve(Q) means solve the query Q.
solve(true).
solve(A) :- clause(A,C), solve(C).
solve((A,B)) :- solve(A), solve(B).
```

Here clause(A,C) is a system predicate which is true if there is a clause with <head> A and <tail> C. The empty tail is represented by "true".

It is more efficient and flexible to write this interpreter in the form:-

```
% SPECIFICATION: solve1(Q) means solve the query Q in depth first order.
solve1(((A1,A2),B)) :- !, solve1((A1,(A2,B))).
solve1((true,B)) :- !, solve1(B).
solve1((A,B)) :- !, clause(A,C), solve1((C,B)).
solve1(true) :- !.
solve1(A) :- clause(A,C), solve1(C).
```

Modifying this interpreter for breadth first search is easy - (B,C) replaces (C,B) in the third line and we get:

```
% SPECIFICATION: solve2(Q) means solve the query Q in breadth first order.
solve2(((A1,A2),B)) :- !, solve2((A1,(A2,B))).
solve2((true,B)) :- !, solve2(B).
solve2((A,B)) :- !, clause(A,C), solve2((B,C)).
solve2(true) :- !.
solve2(A) :- clause(A,C), solve2(C).
```

Class discussion: In the third clause of solve1 and solve2, is it better to have a cut before or after clause(A,C)?

Here is an example of output from a more complicated meta interpreter which prints out a formatted proof of a prolog query.

```
?- solve(grandfather(X,Y), CLAUSES_USED).

CLAUSES_USED = [[[[[[[[ ]],[[(parent(abraham,isaac):-true)]]],[[[
]],[[(male(abraham):-true)]]]],[[(father(abraham,isaac):-
parent(abraham,isaac),male(abraham))]]],[[[ ]],[[(parent(isaac,jacob):-
true)]]]],[[(grandfather(abraham,jacob):-
father(abraham,isaac),parent(isaac,jacob))]]]],
X = abraham,
Y = jacob ? ;

no
```

```
?- printproof(grandfather(X,Y)).

   parent(abraham,isaac).  FACT
   male(abraham).  FACT
   father(abraham,isaac):-parent(abraham,isaac),male(abraham).  RULE
 father(abraham,isaac).  CONCLUSION
 parent(isaac,jacob).  FACT
 grandfather(abraham,jacob):-father(abraham,isaac),parent(isaac,jacob).  RULE
grandfather(abraham,jacob).  CONCLUSION

X = abraham,
Y = jacob ? ;

no
```

Here is the prolog program which produced the proof.

```
% SPECIFY WHICH PREDICATES CAN BE PASSED AS PARAMETERS

:- dynamic grandfather/2, father/2, parent/2, male/1.

% SOME SIMPLE FAMILY RELATIONSHIPS IN PROLOG

% SPECIFICATION: grandfather(X,Y) means X is the grandfather of Y.
grandfather(X,Y) :- father(X,Z), parent(Z,Y).

% SPECIFICATION: father(X,Y) means X is the father of Y.
father(X,Y) :- parent(X,Y), male(X).

% SPECIFICATION: parent(X,Y) means X is a parent of Y.
parent(abraham,isaac).
parent(isaac,jacob).

% SPECIFICATION: male(X) means X is male.
male(abraham).
male(isaac).
male(jacob).

%
% A COMPLEX PROLOG PROGRAM WHICH RUNS PROLOG PROGRAMS
% AND PRINTS OUT A FORMATTED PROOF OF THE ANSWER.
%

% SPECIFICATION: printproof(Q) prints a formatted proof of the query Q.
printproof(Q) :-
      solve(Q,CLAUSES_USED),
      nl,indentprint(-4,CLAUSES_USED).
```

```
%
% PRODUCE A NESTED LIST OF CLAUSES USED IN THE DEDUCTION
%

% SPECIFICATION: solve(Q, CLAUSES_USED) returns in CLAUSES_USED
% a nested list of clauses used to solve the query Q.
solve((A,B),[PA | PB]) :-              % CONJUNCTION
      !,solve(A,PA),
      solve(B,PB).

solve(A,[PA]) :-                    % ONE ATOMIC FORMULA
      solveatom(A,PA).

% SPECIFICATION: solveatom(Q, CLAUSES_USED) returns in
% CLAUSES_USED a nested list of clauses used to solve the query Q.
% However, Q must consist of only one atom.
solveatom(true,[ ]) :- !.              % TRIVIAL CASE
solveatom(A,[PC,[[(A:-C)]]]) :- % CLAUSE A:-C.
      clause(A, C),
      solve(C,PC).

%
% PRINT AN INDENTED ANNOTATED PROOF FROM A NESTED LIST
% OF CLAUSES USED
%

% SPECIFICATION: indentprint(L, CLAUSES_USED) prints CLAUSES_USED
% in indented format based on the nesting of lists.
% L denotes the current indentation.
indentprint(L,[ ]) :-                  % EMPTY LIST
      !.
indentprint(L,[A | B]) :-              % NON EMPTY LIST
      !,
      L1 is L+1, indentprint(L1,A),
      indentprint(L,B).
indentprint(L,(A:-true)) :-            % CLAUSE A.
      !,tab(L),
      write(A), write('.  FACT'), nl.
indentprint(L,(A:-C)) :-               % CLAUSE A:-C.
      !, tab(L), tab(2),
      write((A:-C)), write('.  RULE'), nl,
      tab(L),
      write(A), write('.  CONCLUSION'), nl.
```

Exercise: Write a meta interpreter with only one extra parameter for the level, to print a formatted trace of the execution. (Similar to printing a proof but all stages are printed including failures.)

Grammars

Prolog has a special notation "Definite Clause Grammar" (DCG) for parsing context free grammars. For example the context free grammar for $\{a^n cb^n : n \geq 0\}$ is :-

$s \rightarrow c$
$s \rightarrow a\ s\ b$

In DCG notation this is written :-

```
s --> [c].
s --> [a], s, [b].
```

It is conceptually equivalent to the prolog program

```
% SPECIFICATION check if the list L is built according to the rules
% of the above grammar.
s([c]).
s(L) :- append([a],L1,L2), s(L1), append(L2, [b], L).
```

Both this prolog program and the DCG notation permits the addition of additional parameters for example to count the number of a's as follows (Cohen) :-

```
ss([c],0).
ss(L,N1) :- N is N1-1, append([a],L1,L2), ss(L1,N), append(L2, [b], L).
```

(With this form of definition, if n is given then $a^n cb^n$ will be generated.)

In DCG this becomes :-

```
ss(0) --> [c].
ss(N1) --> {N is N1 - 1}, [a], ss(N), [b].
```

Using braces { } in DCG indicates that this part is regular Prolog not connected with parsing.

Note: The above treatment is a little simplified. In practice "append" is not used by PROLOG systems but a technique similar to difference lists is used. An additional parameter is added to s or ss to indicate which part of the input list is yet to be parsed. Here are some queries and their responses.

```
?- s([a,a,c,b,b,d], Unparsed).          %  meaning parse as much as possible.
Unparsed = [d].

?- s([a,a,c,b,b,d], [ ]).               %  meaning parse entire input.
"No".

?- ss(3, L,Unparsed).                   % meaning generate "aaacbbb"
L=[a,a,a,c,b,b,b | _1]
Unparsed=_1
```