Introduction to Prolog

The name *Prolog* stands for *Pro*graming *Log*ic. It is an excellent tool to experiment with databases, algorithms, and programs. It is often used for expert systems. It is unique in its ability to handle logical and algebraic concepts. At CSUSB I support an industrial strength, portable, and free Prolog interpreter called SWI Prolog. You can get a binary executable for MSWindows. The dept also supports another free version of Prolog from the Gnu people called gprolog. These notes were written for SWI Prolog. There are a few notes on Gnu Prolog.

. How to run Prolog on CSUSB Computers

. Start

For a quick experimental run use SWI Prolog and input the UNIX command:

```
$ pl
```

(Your input as a user will be underlined in these notes). If the above fails you may need /share/bin in your PATH. For the Gnu version use:

```
$ gprolog
```

SWI Prolog source files have the suffix `.plg`. Gnu Prolog uses suffix `.pro`. Use your favorite ASCII editor to prepare them. SWI 'pl' acts like the C and C++ compilers on UNIX. To compile files, load them, and start the interpreter input this command:

```
$ pl file1 file2 ...
```

To compile files into "a.out" type:

```
$ pl -c file1 -c file2 ...
OR  $ gplc file1
```

The result is put in an executable program called `a.out` If a syntax error occurs in a file, the error message will include the name of the file and the number of the line in which it occurred. To compile and output an executable program called *p* use:

```
$ pl -o p  -c file ...
OR  $ gplc -o p file ...
```

If the Prolog program is in a single file you can use `Q/quickie` in the usual way

. **Interacting with a Prolog Program**

A Prolog program prompts for *input* like this:

```
?-
```

The user enters *commands* and *queries*. **Each must be terminated by a period.** The period is ***very*** important.

. Commands

To exit from Prolog either input CTRL/D or the query:

```
?- halt.
```

To look at the loaded program/database, input

```
?- listing.
```

To look at a definition of *name* in the database/program, input

```
?- listing(name).
```

To ask for help input

```
?- help(name).
```

To look for help on a topic *word* try inputting

```
?- apropos(word).
```

To load a program from a file, input

```
?- consult('name.plg').
```

and the system finds and compiles the file '*name.* plg' looking in the standard library and then the working directory. To save the current data base in file *name.*plg:

```
?- tell('name.plg'), listing, told.
```

You can add new rules or facts to the program/database by inputting

```
?- assert( rule ).
```

For example:

```
?- assert( event(jun,5,lab('18 Prolog101')) ).
```

places a new fact in the data base. Try it!

. Queries

In addition to commands, a user can input queries. Each *query* terminates with a **period**. A query can be any Prolog statement with variables in it. Variables are capital letters. Prolog then searches for a set of values for the variables that satisfy the statement or *goal*. Prolog then and prints out the values. For example

```
?- X*Y+Z=1*2+3.
```

If you input the above Prolog will tell you that X=1, Y=2, and Z=3, try it!

If the Prolog program/database contains this fact:

```
event(jun,5,lab('18 Prolog101'))
```

then the query

```
?- event(jun,5,lab('18 Prolog101')).
```

would produce the response:

```
Yes.
```

And the query

```
?- event(Moth, Day, Event).
```

would produce a response that matches the data in the database, like this:

```
Day=5
Month=jun
Event=lab('18 Prolog101')
```

This query:

```
?- event(jun,6,lab('18 Prolog101')).
```

would produce the response:

```
No.
```

unless there was another matching 'event' in the program/database. If the file

http://www.csci.csusb.edu/dick/cs320/schedule.plg

is downloaded into your home directory then it can be loaded as a database/program like this:

```
?- consult( 'schedule.plg' ).
```

and searched by commands like this:

```
?- event(jun,5, What), write(What).
```

This command tells us the value of *What* that matches the third argument in the schedule for that day.

. Execution And Interruption

Execution of a goal starts when you input it as a *query*. Only when execution is complete does the program become ready for another directive. However, one may interrupt the normal execution of a directive by typing CTRL/C. This interrupt signal has the effect of terminating the execution of the command. Prolog will then respond with a prompt. Execution may also be terminated if the program runs out of stack space. This usually indicates an error in your program.

. Sample Queries

You can run these without loading a program. Input just the underlined parts. Do not forget the periods.
In Prolog 'X' is a 'variable' but 'x' is a constant:

```
?- X = 1.
?- x = 1.
?- X = x.
?- x = x.
```

One variable can not have two different values at once:

```
?- X = 1, X=2.
No.
```

However we can give two (or more) variables values:

```
?- X = 1, Y=2.
X=1
Y=2
Yes.
```

Prolog *abandons all variable values* when a query is finished. The variables are held on a stack with one level for each definition being executed. Permanent values must be recorded in the program/database -- and perhaps saved in a file before Prolog finishes.

The following command:

```
?- member(3,[1,2,3]), write (ok).
```

makes Prolog check whether 3 belongs to the list [1, 2, 3], and to output "ok" if it does. If no solution can be found, the system simply returns "No" and a '?-' prompt:

```
?- member(4,[1,2,3]), write (ok).
```

If a variable is included in a query then a possible value is displayed:

```
?- member(X, [tom, dick, harry]).
X = tom <Return>
?-
```

Prolog can generate a *series* of alternative answers. After the first answer is given to a query, the program then waits for ';' (Prolog for "or"), and will then search for another solution.

```
?- member(X, [tom, dick, harry]).
X = tom ;
X = dick ;
X = harry ;
No
```

Such a series of answers can be passed into later parts of a query. Prolog will *redo* a predicate if a later predicate fails. For example we can make X equal every digit in turn and then test to see if the value of X*X is 4:

```
?- member(X,[1,2,3,4,5,6,7,8,9] ), write(X), X*X=:=4.
1 2 3 4 5 6 7 8 9
X = 2 ;
No
```

The query below finds digits that are squares of digits:

```
?- Digits=[1,2,3,4,5,6,7,8,9],
|  member(X,Digits),
|  member(Y,Digits), X is Y*Y.
Y = 1
X = 1 ;
Y = 4
X = 2 ;
Y = 9
X = 3 ;
No
```

. Tracing

Prolog interpreter provides a tracing facility. This is turned on by putting the trace command before the query:

```
?- trace, query.
```

Tracing stays on for that one query only. Predicates may be individually traced enabling each call or exit involving the predicate to be displayed on the terminal with the current values of its arguments. The events traced are:

Call Exit Fail

After reporting an event the program waits for one of a number of actions by the user - 'h' gives a list of possibilities, 'a' aborts execution and tapping *return* lets the program continue. `Call` indicates that a predicate is being called for the first time. A line beginning with `Exit` shows the interpreter exiting a clause after all the goals have been satisfied. `Fail` indicates an exit from a clause due to a failure in satisfying a goal. An example of tracing output is shown below:

```
5   ?- trace, member(X, [1,2]).
 Call:  (  5) member(G912, [1,2]) ? <Enter> creep
 Exit:  (  5) member(1, [1,2]) ? <Enter> creep
X = 1 ;
 Exit:  (  5) member(2, [1,2]) ? <Enter> creep
X = 2 ;
 Fail:  (  5) member(G912, [1,2]) ? <Enter> creep
No
```

Explanation

<Enter> is where I tapped the Enter or Return key, **;** is where I tapped the semicolon. The rest is output by Prolog. The G912 is the internal name given to the variable X. Prolog replaces variables names by numbered variables. The first call of member has actual parameter G912, member returns G912=1 -- the first item in [1,2]. We have an in-out mode in Prolog! Prolog then prints its discovery that X=1. I reply with a semicolon and it tries again. This time

member finds G912=2. …. The third time member finds NO more values and so Fails. Prolog prints out "No".

Example with Backtracking:

```
6 ?- trace, member(X,[1,2,3,4,5] ),  X*X=:=4.
   Call:  (  5) member(G1548, [1,2,3,4,5]) ? creep
   Exit:  (  5) member(1, [1,2,3,4,5]) ? creep
   Call:  (  5) 1 * 1 =:= 4 ? creep
   Fail:  (  5) 1 * 1 =:= 4 ? creep
   Exit:  (  5) member(2, [1,2,3,4,5]) ? creep
   Call:  (  5) 2 * 2 =:= 4 ? creep
   Exit:  (  5) 2 * 2 =:= 4 ? creep

X = 2 ;
   Exit:  (  5) member(3, [1,2,3,4,5]) ? creep
   Call:  (  5) 3 * 3 =:= 4 ? creep
   Fail:  (  5) 3 * 3 =:= 4 ? creep
   Exit:  (  5) member(4, [1,2,3,4,5]) ? creep
   Call:  (  5) 4 * 4 =:= 4 ? creep
   Fail:  (  5) 4 * 4 =:= 4 ? creep
   Exit:  (  5) member(5, [1,2,3,4,5]) ? creep
   Call:  (  5) 5 * 5 =:= 4 ? creep
   Fail:  (  5) 5 * 5 =:= 4 ? creep
   Fail:  (  5) member(G1548, [1,2,3,4,5]) ? creep

No
```

Note: The Gnu Prolog trace give different output.

A Useful Predicate

the *member*(Item, List) predicate is useful for finding items in a list. The `select(List, Item, Rest)` predicate gets the Item from the list and sets Rest to the reduced list. In Gnu Prolog you have to write `select(Item, List, Rest)`. Suppose for example we wanted to be sure that we want to find two different digits where one is the square of the other. In SWI Prolog we write:

```
      ?- Digits=[1,2,3,4,5,6,7,8,9],
      |  select(Digits, X, Rest),
      |  member(Y, Rest), X is Y*Y.
...
      Y = 4
      X = 2 ;
...
      Y = 9
      X = 3 ;
      No
```

In Gnu:

```
      ?- Digits=[1,2,3,4,5,6,7,8,9],
      |  select(X, Digits,  Rest),
      |  member(Y, Rest), X is Y*Y.
...
      Y = 4
      X = 2 ;
...
      Y = 9
      X = 3 ;
      No
```

CS320 Introduction to Syntax and Semantics of Prolog

. Lexemes

Prolog is highly case sensitive and spaces are significant.

. Programs

Prolog source code is made up of a sequence of *clauses*, *directives*, and *comments*. The text of a program is normally created separately in a file (or files), using any ASCII editor.

source_code::= #(*clause* | *directive* | *comment*).

Any characters following the `%' character on any line are treated as part of a comment and are ignored.

comment ::= "%" #non(*end_of_line*) *end_of_line*.

Directives are ways of directing the compiler to execute some goal or goals during the compilation process:

directive ::= ":-" *goal*.

A Typical directive is like this:

:-*op*(=>, xfx, 9000).

Typical clauses in source code are either *definitions* or *facts*.

clause::= *definition* | *fact*.

definition::=*predicate* ":-" *goal* "."

fact ::= *predicate* ".".

Example definitions:

```
get_wet:-its_raining, no_umbrella.
get_wet:-swimming.
get_wet:-fixing(sprinklers).
```

(1) You normally can not redefine a *predefined* predicate. (2) Several definitions can all define the same predicate and their sequence is often important. (3) A goal is usually a list of predicates separated by commas.

. Predicates

Syntactically a predicate is just a term. In a clause it comes before the ":-". To act as a predicate it must be called in a query or from a goal.

predicate::=*compound_term*.

Example predicates:

```
it_is_raining
has('Jim', 'red hair')
```

event(mar, 5,X).

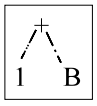
class(cs320, 4, 'Programming Languages', [cs202, m272]).

. Structures

Structures in Prolog do not have to mean anything. The meaning comes from how they are used and what is stored program/database. In a way Prolog is closer to algebra than arithmetic.

Expressions are stored and only evaluated when special predicates are invoked. Prolog parses compound terms as tree structures. So `1+B` has a `+` at the top and `1` and `B` as leaves. Prefix, infix and postfix operators exist and more can be defined. Comma, semicolon, and ":-" are predefined infix operators. Prolog already understands mathematical syntax, precedence and associativity. Internally all structures are converted to the special functorial form defined below:

1 + B +(1, B)



compound_term::= *functorial_form* | *operational_form* | *list*.

operational_form ::=*prefix arg* | *arg infix arg* | *arg postfix* .

functorial_form ::=*functor* "(" *arg*, *arg*, *arg*, ... ")" | *functor* .

Note: Spaces may not occur between the functor and the parenthesis!

functor::= *term*.

Operators are special terms. A *functor* is any atomic term where:

term::=`identifier that starts with a lowercase letter` | *string* | *operator*.

The *arg*uments can be compound terms, constants and variables:

arg ::= *compound_term* | *constant* | *variable*.

. Lists

Prolog lists are shown this:

list::="[ ]" | "[" *element*, *element*,.... "]" | "[" *head* "|" *tail* "]".

The following two expressions are the same:

```
[ 1, 2 ,3 ,4] = [1 | [2,3,4]].
```

. Variables

In Prolog a *variable* has an UPPERCASE letter or an underscore(_) as its first character. All other identifiers, operators, and numbers are atomic *constants*. The variable "_" is a special wild-card variable that can have any value. Each occurrence of "_" is treated as a different variable.

. Constants

Constants are atomic terms. An identifier (starting with a lowercase letter) is an *atom*. So is any defined operator (like + or =:=). Single quoted strings like 'A123' are treated as atoms - even if they begin a capital letter. Double quoted strings indicate an array of integers for the ASCII numbers of the contents of the string.

string::= single_quoted_string | double_quoted_string

Any *compound_term* made up of constants is also a constant -- an item of data that can be stored and manipulated as we wish. The meaning of constants is determined by the program/database plus some predefined terms.

. Semantics

. Database

All permanent data is stored in the *database* as a collection of *clauses*. Programs are stored in the same way ion the same database. Clauses are extracted from the database by matching variables up to particular values or instances. If no matching fact or clause is found then the search *fails*.

In Prolog: data and program are stored using the same notation and structure. A *predicate* has a unique name and a number of arguments (its *arity -- this is not a typo*). Its meaning varies depending on the context - it can be a data structure, program, or an axiom. A form like this

```
has('Jim', 'red hair')
```

is stored as a tree with `has` at the top and `'Jim'`, and `'red hair'` at the bottom.

. **User Interaction**

The user inputs a query, Prolog looks up possible answers in the data base/program and presents them to the user.

. Variables

All variables are **temporary** and **local**. Permanent data must be placed in the database. Prolog searches for values for its variables. Each value is called an *instance*. Prolog binds the instance to its variable. It can bind equal variables to each other. These bindings are temporary. Prolog keeps deducing values until it has processed all the facts(success), or it fails to find a matching definition or fact, or it is about to *change* a value of a variable(failure)!

Once an instance has been found it is fixed. Prolog can not change it without *undo* ing the command that created the instance. It can then *backtrack*, delete the instance(value) and try an alternative. At the end of a search(fail or succeed) all variables loose their values. The only permanent storage is the database.

. Unification

Variables get temporary values by a process called *unification*. Alternative definitions can uncover new temporary values. Each is an *instance* and the variable is then said to be *instanciated*.

Unification forces two expressions to match by (1) instantiating variables as constants, and (2) linking variable together. This is done, piece by piece, throughout the two expressions, in parallel. For example `X+Y*3` matches `1+X*Z` when `X=Y=1` and`Z=3`. Unification does NOT evaluate expressions. If *f1* and *f2* are functors then *f1* (*a1*)=*f2* (*a2*) only if *f1=f2* and *a1=a2*.

Prolog uses unification whenever it calls a predicate. The name of the predicate is found in the database and the arguments in the database are unified with the arguments in the call. So if

```
a(1). a(2).
```

are in the database the query:

```
?- a(X).
```

will generate

```
X=1
```

Tapping ';' (or) gives

```
X=2
```

. Equality

Terms are *equal* (=) only if they can be *unified*. The query

```
?- A=1+B, B=3.
```

unifies A with 1+B. Then B is unified with 3. So A is known to equal 1+3 as well. The 1+3 is not evaluated, So,

```
B=3
A=1+3
```

is output. Notice that A*B=1*2+3 is false but A+B=1*2+3 if A=1*2 and B=3 because of the predefined priorities of * and +.

. Evaluating Expressions

Normally Prolog treats an expression like `1+B` as a data structure and **does not evaluate it**. However, the operator `**is**` does evaluate its right-hand expression:

```
?- B=3, A is 1+B.
```

A is instanciated to 4 rather than 1+3. The same thing happens here:

```
?- C = 1+B, B=3, A is C.
```

Evaluation only occurs in certain predicates:

```
is,<,>,>=,=\=,=:=, >,=<,...
```

In `X is E` the E is evaluated and X becomes the value or else X's value tested vs. result. Evaluation is on the RHS of 'is' only. Both E1 and E2 are evaluated in: E1<E2, E1=<E2, E1 =:= E2(equal value), E1=\=E2(different value), E1>E2, E1>=E2.

Example. Finding all two digit numbers that are the sum of the squares of their digits:

```
?- D=[1,2,3,4,5,6,7,8,9],
|  member(X,D),
|  member(Y,D), 10*X+Y =:= X*X + Y*Y.
```

. Facts

Prolog searches its database for clauses that match the current query. Suppose this is in the database:

```
e(0).
```

Then this is a fact and unifies this with a query like e(0) or e(X) but not e(1) or e(x). Prolog unifies e(X) by instantiating X to 0.

. Clauses

A *clause* has a :- in it, like the following:

```
e(N):- N>0, M is N-2, e(M).
```

This can be read `e(N) **is true if** N >0 **and** M is N-1 **and** e(M) **is true**`. Prolog first matches e(N) vs the query by setting the argument N, then it executes each of the three sub-queries (N>0?, M is N-2, e(M)) in turn. The clause fails the moment any sub-query fails. The clause succeeds and exits only if all succeed in turn. For example, given the above clause is in the database the query:

```
?- e(4).
```

matches it with N=4, and so Prolog executes in turn:

```
4>0
M=2
```

and then executes

```
e(2)
```

In turn, *e(2)* is looked up in the database and a *new N* is matched to 2 and a *new M* is set to 0, thus calling *e(0).* If there are no other definitions in the database this rule never terminates!

. Definitions

A definition declares a series of *alternative* clauses for the same *term*. If there are several different (even conflicting) definitions then any one of them can be true. Prolog searches for the *first* one to match the goal. If it fails Prolog goes on to the next one. For example, if

```
e(0).
e(N):- N>0, M is N-2, e(M).
e(N):- N< 0, fail.
```

is in the program/database, and you ask

```
?- e(0).
```

Prolog immediatley finds `e(0)` and reports success. But this query

```
?- e(-1).
```

skips the e(0) because 0 and -1 are different and next tries the 2nd clause e(N) and sets N=-1. But N>1 fails when N= -1. Prolog next tries the last clause with N=-1, and fails. Since there are no more clauses Prolog treats as e(-1) as false.

The following query

```
?- e(1).
```

does not match `e(0)`. It matches e(N) with N=1. 1 > 0, so M is set to -1. Prolog calls e(-1) which fails. This failure makes the second clause of e(1) fail. Prolog tries the last possibility, setting N=1 but then N is not less than 0... so `e(1)` is false.

Similarly the query:

```
?- e(2).
```

does not match `e(0)` but does match e(N) with N=2 and N>2. So Prolog now sets M=0 and tries to match e(0), and finds that this matches the first clause. So this query succeeds: e(2) is true.

If you try

```
?- e(3).
```

it does not match `e(0)` and does fit e(N) with N=3 and so Prolog tries M=1 and tries to match e(1). But we know already that e(1) will ultimately fail and so Prolog will exit and try the last clause. This also fails. So e(3) is false. You prove this to yourself by inputting

```
?- trace,e(3).
```

If you do more tests you will find out that e(N) is true precisely when N is a positive multiple of two and false otherwise. Thus 'e' is short for 'even'. The meaning of 'e' is determined by the program/database. With a different set of rules `e(N)` could be made to mean other things.

. Resources on the WWW

http://www.csci.csusb.edu/dick/cs320/prolog/

. **Punctuation**

.	End of clause/query
:-	if
,	and then(also in list)
;	or else

. **Predefined Operators**

```
+ A    - A         A * B   A + B
A - B  A / B     A mod B
abs, acos, asin, atan, ceil, cos, cputime, e, exp, float, float_fractional_part,
float_integer_part, floor, integer, log, log10, max, min,  random, rem, round,
truncate, pi, sign, sin, sqrt, tan, xor.
```

. **Predefined Relations**

Equality: E1=E1 if they can be unified, E1\==E2 if they can't.
Evaluation: V is E::= evaluate E and unify value with V.
Arithmetic comparison evaluate both arguments and compare:
<,>,>=,=\=,=:=, >,=<,...
Lexicographic comparison:
@<, @>,...

. **Predicates**

```
atom(X)::=`True if X is an atom`.
atom_length(A,N) ::=`N is length of atom A`.
atomic(X)=`Type check for primitive`.
compound(X)::=`Test for compound term`.
fail::=`Always false`.
ground(T)::=`T has no free variables`.
number(X)::=`Type check for integer or float`.
string(X)::=`Type check for string`.
true::=`Succeed`.
var(X)::=`Type check for unbound variable`.
```

. **List Operations**

```
append(L1,L2,L3) ::=`Concatenating L1, L2 gives L3`.
findall(X,G,L)=`L is List of all the X that solve G`.
forall(G1,G2)::=`For each solution of G1 prove G2`.
last(E,L)::=`E is last element of a list L`.
length(L,N)::=`N is length of a list L`.
maplist(P, L1, L2) ::=`Apply P to pairs in L1 and L2`.
member(E,L)::=`E is a member of a list L`.
nth0(I, L, E)::=`I-th element of L equals E, start at 0`.
nth1(I, L, E)::=`I-th element of L equals E`.
select(L1, E, L2)::=`E is in L1 and L2 is rest of L1`.
setof(X, G, L)=`L is a sorted list without duplicates of all X that solve G`.
sort(L1,L2)::=`Sort L1 giving L2`.
```

. **Atoms**

```
name(A,L)::=`atom A matches list L in ASCII `.
concat(A1,A2,A3)::=`Append two atoms`.
```

. **Strings**

```
atom_char(C,A)::=`atom C is ASCII A`.
atom_chars(A,L)::=`atom A is list L of ASCII`.
string_length(X,N)::=`Determine length of a string`.
string_to_atom(S,A)::=`Convert string S to atom A`.
string_to_list(S,L)=`Convert: string and list of ASCII`.
```

. Database Operations

```
abolish(F,N) ::=`Remove the definition of F with N arguments`.
assert(P)::=`Add clause P to the database`.
clause(H, T)::=`find next clause to match H:-T.`.
consult('F')::=`Read and compile source code file F`.
ed::=`Edit last edited predicate`.
ed(P)::=`Edit a predicate`.
listing::=`List program`.
listing(F)::=`List predicate F`.
retract(P)::=`Match & Remove clause `.
```

. Save Database

```
tell('filename '), listing, told.
```

. System

```
abort::=`Abort execution, return to top level`.
apropos(word)::=`Show related predicates`.
halt::=`Exit from Prolog`.
halt(N)::=`Exit from Prolog with status N`.
help::=`Give help on help`.
help(X)::=`Give help on X`.
ls::=`list current directory'.
op(Symbol, Form, Priority)::=`Declare an operator`.
statistics::=`Display various stats`.
shell::=`Execute interactive subshell`.
shell(Command)::=`Execute OS command`.
time(G)::=`Determine time needed to execute G`.
trace::=`Start the tracer`.
```

. Input and Output

```
get(X)=`Read first non-blank character`.
get0(X) =`Read next character`.
nl::=`print a newline`.
print(X)::=`Print term X`.
put(C)::=`Write character C`.
read(Term)::=`read next term(with period) on input and unify with Term`.
see(File)::=`Change the current input stream`.
seeing(X)::=`Unify X with the current input stream`.
seen::=`Close the current input`.
skip(C)::=`Skip to character C in current input`.
tab(N) ::=`Output N spaces`.
tell(File)::=`Change current output stream`.
telling(X) ::=`X is the current output stream`.
told::=`Close current output`.
write(Term)::=`Write Term`.
write_ln(Term)::=`Write term, followed by a newline`.
```

. Logic and Control

```
V is E::= evaluate E and unify value with X.
E=.. [F|A] ::= `E has form F(A)`. -- constructs/analyze functorial forms.
once(P)::=`don't redo P`.
! ::= `cut off all backtracking here`. -- remove choice points.
P,Q::= P and then Q.
P;Q::= P or else Q.
P->Q::= `If P then Q`, Safer to be explicit: (not(P); Q).
not(P)::=`try P. True if P fails, else false`.
repeat::=`Succeed, with infinite backtracks`.
```

CS320 Examples for Prolog laboratory

Over the page is the beginning of a unusual detective story that stars Sherlock Homes and Babbage's Analytical Engine. Read the story and then try to match up the Prolog code below to the facts that Sherlock Homes has noted in the story.

```
% In Elementary Pascal, Ledgard & Singer have Sherlock Holmes program
% the Analytical Engine to confirm the identity of the murderer of
% a well known art dealer at the Metropolitan Club in London.

% The murderer can be deduced from the following apparently trivial
% clues.

murderer(X):-hair(X,brown). % the murderer had brown hair

attire(mr_holman, ring). % mr_holman had a ring
attire(mr_pope, watch).  % mr_pope had a watch.

 % If sir_raymond had tattered cuffs
 % then mr_woodley wore the pincenez spectacles
 % and vice versa
attire(mr_woodley,pincenez):-attire(sir_raymond,tattered_cuffs).
attire(sir_raymond,pincenez):-attire(mr_woodley,tattered_cuffs).

 % A person has tattered cuffs if they were in room 16.
attire(X,tattered_cuffs):-room(X,16).

hair(X,black):-room(X,14). % A person has black hair if they were in room 14.
hair(X,grey):-room(X,12).
hair(X,brown):-attire(X,pincenez).
hair(X,red):-attire(X,tattered_cuffs).

room(mr_holman, 12). % mr_holman was in room 12
room(sir_raymond,10).
room(mr_woodley,16).
room(X,14):-attire(X, watch).
```

In the Prolog lab you will be able to download and test the above program.

. Review Questions

1. What does a user input to a Prolog interpreter? What does the interpreter do with the input? What is output?

2. How do you terminate our Prolog? Can you edit the database?

3. Is Prolog case sensitive? Is white space significant and if so, where? How many kinds of string? Name some characters can be in a string but not in an atom.

4. Define *functorial form* and express 1+2*3 in this form.

5. One of these two queries is true: `2=1+1;  2 is 1+1.` Which? Why?

6. How do you distinguish an atom, a string, and a variable?

7. One of these two queries is true: `X=1+1;  x=1+1.` Which? Why?

8. List a dozen predefined arithmetic operators of SWI Prolog. When are they evaluated?

9. Which arithmetic operators are prefix? Which are infix? What priorities do you expect?

10. How do you print out the loaded database? Can you print out predefined definitions?

11. Write a query that finds three digit decimal numbers that equal the sum of the cubes of their digits.

12. Name the predicate or command that places a single fact or clause in the database. Name the command/predicate that loads a file of facts and definitions into the database.

13. What command/predicate removes a single matching clause from the database?

13. List the 6 relational operators that evaluate two arithmetic expressions and then compare the results.

14. What do the following mean in Prolog: *atom atomic*, *number*, *var*?

15. What punctuation symbols act like boolean operators in Prolog? Name the equivalent of true and false constants.

16.If `[X | Y] = [1,2,3]` what are X and Y?

17. What are the functor names in: `fun(A,B), A+B, -B, A*B, A*B+C`, respectively?

18. Draw the tree structure of `A*B+C*D`.

19. How does `X=Y+1` differ from `X` **`is`** `Y+1`? If Y=2 what is the value of X in each case?

20. If X=1 and Y=2, then what happens to each of the following: `X=Y, X` **`is`** `Y, X=:=Y, and X=\=Y`?

21. Unify `married(X,Y)` with `married(husband(dick), wife(tricia))`. Can you unify `sqrt(4)` with `1+1` ? Explain

22. Can you unify `A*B` with `1*2+3`? Can you unify `A+B` with `1*2+3`? Explain why not and/or how.