Prolog Binary Search

Unknown macro: {style}

cite

Unknown macro: {font-family}

Valid programs in Prolog are of the form

Unknown macro: {div}

Program -> {Rule}
Rule -> CompoundTerm[:- CompoundTerm|true {, CompoundTerm | true}].
Term -> CompoundTerm | Variable | Atom
CompoundTerm -> Atom[(Term {, Term})]
Variable -> UppercaseLetter {AlphaNumeric}
Atom -> LowercaseLetter {AlphaNumeric} | 'AlphaNumeric {AlphaNumeric}'

In addition, many arithmetic compound terms are written in infix notion, and these may occur in the body of the rule.

The syntax of any file is a list of rules and facts. Then a query may be made of the file for any of the rules.
Two or more rules may have the same name, and optionally the same arity. When the arity is different, then the compound terms are distinct, but if they are the same, then either may be used to satisfy a query. The body of a rule is a list of compound terms which are queried to find if there is a way to make truth for the given inputs.

The following Prolog program provides logic to test whether an element is contained in a sorted list or not. It uses a binary search algorithm that does not keep track of the index.

Unknown macro: {table}

Unknown macro: {tr}

Unknown macro: {td}

Program binarySearch.pl

Unknown macro: {div}

Unknown macro: {span}

```
%checks to see if a value is contained in a list of integers,
%using binary search tactics
```

Unknown macro: {span}

```
%success base case
%A list contains a value if its center is the value

contains(List, Value):- even_division(_, [Value|_], List).
```

Unknown macro: {span}

```
%case that Value is large
%A list contains a value if the value is larger than the center,
%and the second half of the list contains the value

contains(List, Value):- even_division(_, [Center|SecondHalf], List),
```

Unknown macro: {span}

```
Center<Value, SecondHalf \= [],
```

Unknown macro: {span}

```
contains(SecondHalf, Value).
```

Unknown macro: {span}

```
%case that Value is small
%A list contains a value if the value is smaller than the center,
%and the first half of the list contains the value

contains(List, Value):- even_division(FirstHalf, [Center|_], List),
```

Unknown macro: {span}

```
Center>Value, FirstHalf\=[],
```

Unknown macro: {span}

```
contains(FirstHalf, Value).
```

Unknown macro: {span}

```
%even_division(First, Second, Xs) is true when
% Xs is the concatenation of First and Second,
% and First and Second are either the same length
% or Second is one element longer than first.

even_division(First, Second, Xs) :- append(First, Second, Xs),
```

Unknown macro: {span}

```
length(First,F), length(Second,S),
```

Unknown macro: {span}

```
S>=F, S-F=<1.
```

Unknown macro: {td}

Queries

```
?- consult("binarySearch").
?- even_division(First, [Center|Second], [5,7,8,9,3,5]).
First = [5,7,8]
Center = 9
Second = [3,5]
?- even_division(First, [Center|Second], [5,7,8,9,3,5,8]).
First = [5,7,8]
Center = 9
Second = [3,5,8]
?- even_division(First, [Center|Second], [5]).
First = []
Center = 5
Second = []
?-even_division(First, [Center|Second], []).
false

?- contains([1,3,5],2).
no
?- contains([1,3,5],1).
yes
?- contains([1,3,5],X).
X = 3 ;
ERROR: </2: Arguments are not sufficiently instantiated
   Exception: (6) contains([1, 3, 5], _G8401) ? Unknown option (h for help)
```

Notice that this program does not use any facts. The three rules for —*contains* are interchangeable in execution since they all have arity 2 (so they are referred to as —*contains/2*). The additional rule —*even_division/3* plays a role similar to an auxiliary function. By checking all possible ways to divide a list, it can find the correct division to get a first half, center, and second half. Special syntax for representing a list as a head and tail (or, to use Lisp terminology, car and cdr) allows the programmer to extract the first element of the second half of the list.

At this time, a note on execution is due. The idea of Prolog programming is that the programmer can focus on logical relationships without worrying unduly about how to implement an imperative procedure (executable by a machine) to reason through the logic. Nonetheless, as with all languages, it is important to have some understanding of implementation in order to know how to write efficient, or even executable, code.
An initial, abstract understanding is through nondeterminism. Deterministic processes have a single line of execution; there may be conditional statements, but, dependent upon the state, only one path is correct. With nondeterminism, there may be more than one valid path to take. For instance, when asked to find the truth of the statement —*contains([1,3,5],3)*, the program may take any of three paths, one for each definition of —*contains/2*. If it chooses the second path, it tries to find a way to make —*even_division(_, [Center|SecondHalf], [1,3,5]), Center<3, SecondHalf* \= *[],contains(SecondHalf, 3)* true. In order to do so, it must find possible values for —*Center* and —*SecondHalf* by querying —*even_division*. The only way for that condition to be true is for —*Center* to be 3. Since this cannot be true when —*Center*<3, the entire statement fails to be true. Now we back up to the last nondeterministic branch and choose another option. The correct one to choose is the one that appears first in the program. This statement easily evaluates to —*true*, so the entire query is true.

One thing to learn from this description is that if a condition has infinitely many ways to be true where all of them are inconsistent with other conditions, it is entirely possible that the execution of a query will enter an infinite loop. Such considerations are important to keep in mind when designing a Prolog program. In particular, the error from attempting to answer —*X<2* is a defensive mechanism due to the infinite nature nmbers. To solve this problem, move the test of —*Value<Center* to the end of the line. Then the query —*contains([1,3,5],X)* returns the values 3, 5, and 1, in that order.