Ana Balcells and Yuyu Li
CS 251
Final Project - Prolog

home(cs251, X).
X = microfocus.

Unlike other approaches to problem-solving that use functional or object-oriented paradigms, logic programming provides a model dependent on formal first order logic. One such example of a logic programming language is Prolog, whose structure, syntax, and semantics uses this logical paradigm through its constraint-based format. Prolog is a declarative language rather than a procedural one; instead of providing line-by-line directions for the computer to execute, it describes the constraints that need to be satisfied in order to solve a problem. These constraints are expressed as either a statement or a rule, and these are the basic building blocks of the language. Indeed, everything is either a statement or a rule in Prolog, and this structure follows naturally from first order logic to model the human system of reasoning. The statements and rules that make up a program describe what exactly the situation is in order to solve a problem rather than giving directions on how to solve it, which is often more representative of logical problem-solving in real life.

In order to reflect the natural language used in logic problems, Prolog uses several unique language features to establish its statements and rules. A program in Prolog is called a knowledge database since it contains all the rules which will be used in order to solve a given problem.These rules are made up of even smaller units: atoms (or alternatively, constants). An atom is the smallest unit within a program. In logic programming, an atom serves a similar function to an object. Atoms can have several types, with the most common being lowercase words or the empty list ([]). An atom is a single data item which is used to reference only itself. Rules are used to describe atoms and their expected behaviors.Take the following classic Prolog rule for example: `student(john).` In this line,

“john” is an atom. This is essentially binding the string “john” to a single object, which can then be used as a reference later in the programming to change the rules determining the behavior of “john”. Student, followed by its parentheses is known as a predicate. This be thought of as creating a function which moves from “john” to a value of true or false. In this case, student(john) is being set to true (since there is no expression on the right hand side, as we’ll explain in a moment). In the above example, the line student(john) as a whole is a rule, since it is establishing that john is a student. This rule is of a special type, called a fact since it consists of only one line and is considered true as it is written. However, rules can also be written in a longer format. All rules must adhere to the following form- `predicate(arguments) :- body1, body2, …, bodyn`. A given predicate will only return true if all conditions (or predicates) on the right hand side of the :- are satisfied. Variables can be created in this environment by replacing any given constant with any series of letters beginning with a capital. In our previous example, replacing the constant john would lead to the expression student(X). Prolog then resolves X = john, since john is the only condition which satisfies this rule to make it true.

The types of expressions above are used repeatedly to define desired behaviors and data structures in any given Prolog program. This program is called a knowledge database, since it is a collection of the facts and rules which Prolog will use to solve a given problem. In order to use a Prolog program, a user queries the knowledge database. A query appears to be a rule or fact like those defined in the program, and can either contain constants or variables. The knowledge database is then consulted, and if the query is true according to the rules of the database, a value of true will be returned. If any variable definitions are involved in the search for this solution, these variable bindings will also be returned in all possible arrangements which satisfy the knowledge databases rules.

Another important feature in Prolog is its implementation of lists. Prolog lists are recursive structures. Like lists in Racket, they are implemented as right branching trees. This means that while a list may be represented as a set of items within brackets, it can also be represented as a head (which is a

single item) followed by its tail (a list with all remaining list items). This is similar to the Racket concepts of car and cdr, allowing a list to be represented by a single item of any type at its head (which in Prolog means it can be a predicate or another list), followed by a structure in the tail which is always a list. In Prolog, lists can be defined explicitly in a representation such as `[1, 2, 3]`, or their recursive structure can be used to represent a list in the following way: `[Head | Tail]`. This allows all of a lists items to be examined one at a time by the variable Head, while the remaining list items are represented by the variable Tail. Certain predicates which are familiar to us from Racket and ML functions are also present in this language, such as member, append, and reverse.

This list structure is the basis for the implementation of ADTs in the Prolog language. Using lists as ADTs allows a programmer to easily store and retrieve data in a relatively manner due to a lists recursive nature. The predicates which are predefined by Prolog also allow for easy manipulation of data sets without ever necessitating full-knowledge of the workings of these predicates. However, full encapsulation of ADTs is also difficult for these very same reasons since there is transparency in how the data is stored and since the definition of these ADTs must be defined explicitly using the Prolog list structures (Haberman). While full encapsulation is difficult, we can still mimic the actions of ADTs such as trees, stacks, queues etc. It simply requires extra effort on the part of the programmer to work around the declarative nature of Prolog in order to hide information about the inner workings of these ADTs.

Recursion in Prolog takes recursive principles seen previously in other languages and simply adapts them to suit rules instead of function. In practice, this means that any recursive rule will consist of a predicate on the left side of a :-, followed by at least 2 cases. At least one of these cases will be a base, or termination case. This will terminate the recursive calls once a given predicate has been satisfied, indicating the the conditions set for the rule have been met. There may also be one or more recursive calls which call a rule on itself so that the arguments might be changed and providing further

opportunities for the satisfying conditions to be met. As is the case with all rule calls, and similar to local scoping seen in other languages, the variables defined within any single rule call are unique to that rule. In recursion, this means that the variable values created during each level of the recursive call can be different from those values created at a higher or lower level of the recursive call stack. However, the Prolog feature of unification (described in a future section) also means that there is a limited amount of relationship between variables of different levels. Internally, Prolog stores temporary variable bindings so that they might be passed between levels of recursion. If these variable bindings lead to the rule being false, these bindings will be discarded (i.e. backtracking will occur).

Because of the logical constraint-based format of Prolog, the interpreter needs to go through a proof search when it is provided with a query. There are two possible approaches to try, one being the top-down, goal-directed approach and the other being the bottom-up, forward-reasoning approach. Forward-reasoning starts with rules and applies these until a specific goal is found. This is in contrast to the goal-directed approached used by Prolog that starts with a goal and reasons backwards to find the necessary rules. When it determines that a rule must be necessary, that rule's premises become a subgoals. This process continues until the subgoals are fulfilled, in which case the primary goal is satisfied or proven. Given the query, Prolog will search the knowledge base and attempt to either a fact that matches the query or a rule with the same head. The process of determining goals and subgoals that follows is recursive, with the program's complexity determining the depth (for instance, if there are many complex nested rules).

In the process of proof-searching, in order to match goals with facts or heads of rules, unification must occur. To unify the goal and constraint, however, is more complex than simply checking if both have the same constant since variables can present themselves in either one. To unify the goal and constraint, subterms must be matched. If both are constants, this is a simple process, but if variables are present, a unifier equation is used to set one variable to the other term and the given variables are

substituted for new ones. In this way, the matching of subterms to determine if certain goals can be unified is similar to pattern-matching where structures are broken down and a goal pattern or variable must correspond with the given one for evaluation (or determination of whether a goal can be satisfied or not). Unifying variables that can be instantiated with equal terms is crucial in proof searching.

A graphical representation of the process of proof searching is an execution or search tree. For each goal or subgoal, there can be multiple possibilities of matching with a fact or head of a rule. Since Prolog will search the knowledge base from top to bottom, the first goal that is satisfied is dependent on the order in which rules and facts are stated. If one supposes that a goal can be satisfied by a number of possible matches, the one represented by the fact or rule that is expressed first is explored first. If the next goal can also be satisfied by a number of possible matches, the interpreter will continue in the same manner. In this way, if a search tree were drawn out with each goal being a node and levels of goals to fulfill, the Prolog interpreter will first perform a depth-first search, left to right, in order to find one possible match of the goal. If there are more possibilities once the goal is satisfied or if it encounters a rule that cannot be satisfied given a previous match to a "higher" rule, it will go up a level to the higher rule and look at the rest of the possible matches. This process is aptly named backtracking and produces a list of possible results in a systematic way. Each possible match to a rule is a choice point, and by keeping track of which match is chosen, the interpreter can ensure that it finds all possible results, or none if the goal cannot be proven. It should be noted that the programmer must ask Prolog to find more possible answers by using the ";" or "or" command, since Prolog will not backtrack to the latest choice point once the goal is satisfied unless prompted.

Unlike procedural languages, a declarative language like Prolog does not require the programmer to define a sequence of directions for the interpreter to execute. The directional execution model as described above then in some ways takes much of the choices of operations to run out of the programmer's hands. The programmer has control over the way the knowledge base is formed, but

does not have to wrestle with the details of execution the same way as in procedural languages. This means that certain programs might be easier to write; specifically, in programs that simulate situations easily expressed as a system of constraints, Prolog's directional model would make the language an intuitive choice. However, for goals that might have a more sequential aspect or require more steps to their constraints, for instance, if a goal is more naturally expressed with loops, using Prolog would be more complex than simply writing a procedural program.

In term of efficiency, knowledge of how the Prolog model can help the programmer manipulate the knowledge base in order to reduce the time and work it takes to answer a query. Because it is already known that Prolog performs a depth-first search for goals and the order of facts and rules matters for this, the constraints can be changed to yield higher efficiency. The Prolog model does not seem to be inherently more efficient than a procedural one in any way, especially given that each subgoal adds a recursive layer to the search tree. When considering the backtracking usually needed to find multiple matches or to ensure that a goal can either be proven or disproven, it would be much more beneficial to place the more restrictive constraints higher up on the tree. By limiting the possibilities at these shallowest levels, one can ensure that each recursive layer of subgoals after is applied to less potential matches. With the same exact constraints, the orientation of these into goals can yield different processing efficiencies. The top-down search in the knowledge base that is performed by the interpreter means that the programmer's control over the order of constraints stated directly affects the proof tree search and thus the efficiency of the program. (Though this may be powerful when applied, it does take away from the Prolog model of separating the correctness of the program from its efficiency, or the separation between the program's logic and its control.)

Another way programmers are able to manipulate Prolog processing is by using directives like cut, expressed by "!", and negation. When Prolog works through goals in a proof search and encounter a cut in a rule, it cannot backtrack in its current traversal. All choices made at previous choice points are now

set and cannot be changed even if the goal cannot be satisfied given these choices. Also, for any goals with multiple rules, the rules that follow will not be tried. Cuts can be categorized by their functions: green cuts are used to optimize efficiency while red cuts change the semantics of the program. Generally, cuts can be powerful in the language in order to prioritize or weighing certain constraints over others.

Another special directive in Prolog is the negation, or "not". Unlike negation in other languages, "not" in Prolog does not mean that a certain goal is false, but rather that it cannot be proven true. When combining not and cut, negation as failure can be implemented in Prolog since the not directive does not have the same semantic meaning. In term of what these directives mean for the language, "not" in Prolog seems to blur the semantic meaning and make negation less clear. The combination of the cut and "not" cannot necessarily be considered bad language design since both are useful in their own right, but it seems superfluous to use both for negation as failure rather than having a more simple "not" that is actually semantically representative of what many programmers mean when they use negation. This would also make the language clearer to understand. Merging both into one directive would make the language less powerful though and limit the expression of the language.

The nature of the language of Prolog makes it especially well suited for certain tasks. For example, Prolog is particularly popular in artificial intelligence. This is because artificial intelligence is intended to model the human reasoning system. Since first-order logic is a paradigm frequently used to represent reasoning and knowledge, logic programming becomes an ideal choice for development of artificial intelligence systems. Prolog's declarative structure also makes it a model for human reasoning since it does not require the procedural development necessitated by other languages. This makes it an accessible language for less experienced programmers. Prolog is also naturally suited to the definition of rule based systems, since any Prolog program is essentially a system of correlations between rules. Prolog's process of unification also makes it an ideal typing or pattern matching language since its

unification algorithm ensures that all stated equalities still hold for any values determined to be a possible solution. Despite its advantages, in order for this language to be properly utilized, the user must first have a clear understanding of predicate knowledge. This may require some extra effort on the part of the programmer, and may also require additional time spent in the planning steps of any large program (when compared to time spent planning a program in a OOL). This language, when properly used, has the potential to be incredibly useful in problem domain which is currently being addressed by more inefficient language choices.

Sources:


Blackburn, Bos, & Striegnitz, Learn Prolog Now! (Tutorial)
http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse5

Fisher, prolog:- tutorial.
http://www.cpp.edu/~jrfisher/www/prolog_tutorial/intro.html

Friedman, A Characterization of Prolog Execution.
ftp://ftp.cs.wisc.edu/sohi/theses/friedman.pdf

Haberman, Formal and Practical Aspects of Implementing Abstract Data Types in the Prolog Instruction.
www.mii.lt/informatica/pdf/INFO702.pdf

Pfenning, Logic Programming.
http://www.cs.cmu.edu/~fp/courses/lp/lectures/lp-all.pdf

Tamir & Kandel, Logic Programming and the Executional Model of Prolog.
http://ac.els-cdn.com/1069011595900381/1-s2.0-1069011595900381-main.pdf?_tid=134da2b6-fdcb-11e4-b595-00000aacb361&acdnat=1432001117_2b2eff8a12764a0add109b37ddb55bf3

Wikibooks, Prolog.
https://en.wikibooks.org/wiki/Prolog