Alex Gruenstein
Linguistics 238

Prolog in Java

Introduction

For my final project, I implemented an interpreter for a subset of the Prolog language using Java. In addition, I extended the language such that predicates may be defined as Java methods which may be called as Prolog predicates, just as a normal Prolog predicate is called. My general motivations for undertaking this project were as follows. First, discovering the steps involved in implementing Prolog is an interesting and edifying process. Second, the cross-platform nature of Java means that a Prolog interpreter written in Java is extremely portable. Third, Java and Prolog each have strengths and weaknesses – a Prolog interpreter written in Java provides a basis by which the two languages may be freely intermingled in a single executable.

Implementation

The interpreter is written purely in Java. The parser component of the system was created using ANTLR, an open-source parser-generator implemented in Java which compiles grammars into Java classes for lexers and parsers.

Terms and Unification

Each term that appears in the Prolog program is represented by one of three Java classes: `Atom`, `Variable`, or `CompoundTerm`. Each of these implements an interface called `Term` and each inherits from a base class called `BasicTerm`, which defines basic behavior shared by all three classes. `Term` stipulates that a term must be able to return its functor, its arity, and any inner terms it might contain; moreover, it says that a term should know how to unify with another term, and know how to duplicate itself (how to make a "deep" copy). Unification is performed recursively, with variables able to "bind" to atoms, compound terms, and other variables. The "occurs check" is implemented to guard against infinite recursion. Each time a variable binding is made, a reference to a `Valid` object is stored in that variable – if the `Valid` object is "invalidated," then the binding is lost and the variable is no longer bound.

The Predicate Library

Predicates consist of a head, an optional list of subgoals, and a flag indicating if the predicate is a "system" predicate (one implemented in Java), or a standard predicate implemented in Prolog. All of the predicates available to the interpreter are stored in a hash table as part of the class `Library`. The *key* is the predicate functor suffixed by `"/arity"`, where `arity` is the arity of the predicate's head. The *value* is a list of predicates with such a head functor and arity.

The Prolog Machine

Once we have terms which know how to unify with one another (and "un-unify" with one another via invalidation) and a predicate library, we have the necessary components in hand to build the prolog machine, implemented in the class `Prolog`. This class provides a method called `solve`, which takes a list of goals and attempts to prove them. If a solution is found, the variable bindings are printed, and the machine pauses until the user instructs it to either continue looking for more solutions or stop its search (unless there are no variables in the goals, in which case it simply prints "yes" and halts the search). If no solutions are found, the machine prints "no" and waits for more goals.

The basic algorithm for proving a set of goals is the following:

```
Algorithm: solve(S, V)
Returns true iff the search for solutions should continue
S: a stack of goals to prove, with the first goal to prove
on top.
V: A list of variables whose values to print upon finding a
solution


if(empty(S))
     print_solutions(V)
     if(|V| > 0 && queryUserShouldContinueSearching())
          return true
     else
          return false


goal = pop(S)
predicates = lookup_predicate_list(goal)

foreach(predicate in predicates)
     if(unify(head(predicate), goal, valid))
          push(S, goals(predicate))

          if(!solve(S, V))
               invalidate(valid)
               return false

          pop(S, goals(predicate))
          invalidate(valid)


push(S, goal)
return true
```

The algorithm is fairly simple. Essentially, we pop a goal off of the stack and then look it up in the library. For each alternative version in the library that might unify with it, we try to unify the head of the predicate with the goal. If this succeeds, then we replace the goal on the stack with the (possibly empty) list of subgoals provided by the predicate. Next, we do the same thing recursively. If the stack is empty, then we have found a solution and we print it out to the user. If the user wants us to continue searching, then we return true and continue searching; otherwise, we return false which causes us to break each of our unification bindings as we make our way up the call stack.

I didn't implement the *cut* ("!") as part of this algorithm. We could imagine implementing it by giving goals pushed on the stack a reference object that could be "invalided" by a *cut* subgoal. If this object were invalidated, then the machine would not cycle through other alternatives for the goal replaced by that subgoal on the stack.

Writing System Predicates in Java

The above algorithm assumed that all of the predicates would be defined in terms of subgoals (or simply be facts with no subgoals). In order to provide *system* predicates (such as *write*, *nl*, *assertz*, *retract*, *fail*, and so on), some of the predicates must actually be implemented in Java. Such predicates are stored in the library and marked with a special flag. When they are found by the prolog machine, it attempts to invoke their Java methods rather than replacing them with further goals on the stack.

Writing a system predicate is fairly simple. First, it must be declared to the library with a call to method `addPredicate()`, where the predicate's head must be defined. For instance, to add the predicate `write(Term)` to the library, we simply do the following:

```
library.addPredicate(new Predicate(new CompoundTerm("write",
         new Variable("Term"))));
```

Next, we implement a java method matching this prototype as a member of the class `JavaMethods`. This method is as follows:

```
public boolean write(Term t) {
  System.out.println(t);
  return true;
}
```

At runtime, the prolog machine calls a method called `runJavaPredicate()` which makes use of Java's reflection capabilities to call the appropriate method matching the invoked system predicate.

This system makes it easy to develop new predicates in Java, which may be invoked in normal Prolog programs. Since the interpreter's native language is Java, no run-time conversions are necessary – the code flows seamlessly between the prolog machine and the custom java predicate. In order to demonstrate how this interaction can be useful, I wrote a couple of predicates in Java which make use of its built in GUI features. Both predicates display prolog terms graphically, as trees. The first, called

`display(Term)` is capable of displaying any prolog term as a tree where each functor becomes a node and each inner term becomes a child. Lists are a special case – they are displayed sequentially, rather than as embedded terms of the form `'.'(A,B)`, which is how they are really implemented.

The second predicate is called `display_parse` and it is used to display parse trees graphically. These parse trees are produced by a bottom-up parser and a relatively simple grammar I've modified from a class I took last year.

Parsing Prolog

I was a bit surprised to realize that far and away the most difficult task of building the interpreter was building a lexer/parser. Rather than coding one by hand, I chose to use a parser-generator called ANTLR by Terrance Parr (see http://www.antlr.org). This is a parser-generator similar in spirit to *yacc* and *lex*, but geared specifically toward integration in Java programs. To produce a parser, one first writes a *grammar* which defines the syntax of the language to be parsed – in this case, Prolog. This is then compiled into parser/lexer which can be easily invoked as part of a Java program.

Defining the grammar for most of prolog is extremely straightforward. The main difficulty is in dealing with *operators*. In Prolog, it is possible to define prefix, postfix, and infix operators of arbitrary precedence on the fly. This has an impact on both the lexer, which must now pull apart tokens differently, and the parser, which must incorporate the new operator into its semantics. After much work at trying to do this in ANTLR, by hand, and through preprocessing methods, and after much web-searching, I gave up. Instead, I hard coded all of the standard prolog operators, and a few extra ones needed by the bottom up parser and the grammars it can parse ("`--->`" and "`::`"). This unfortunate state of affairs means that in order to add a new operator, at the moment, the grammar for the parser must be modified and recompiled.

Definite Clause Grammars

Built in is the typical prolog ability to define a definite clause grammar (DCG). I support the most basic translation of rules from non-terminals to non-terminals and from non-terminals to terminals. I did not include support for the arbitrary insertion of prolog goals inside of curly braces. Following Clocksin and Mellish, (4th edition, chapter 9), the rules for translation can be demonstrated by the following examples:

np --> det, n.
det --> [the].
n --> [dog].

Becomes:

```
np(S0, S) :- det(S0, S1), n(S1, S).
det([the|S],S).
n([dog|S], S).
```

A slightly more complex example is translated as follows:

```
np(np(Det,N)) -->
     det(Det),
     n(N).

det(det(the)) --> [the].
n(n(dog)) --> [dog].
```

Becomes:

```
np(np(Det,N), S0, S) :-
     det(Det, S0, S1),
     n(N, S1, S).

det(det(the), [the|S], S).
n(n(dog), [dog|S], S).
```

Testing

In order to make sure the system was working, I first tested out some of the functionality. The following sample interaction below shows that variable bindings work correctly, assert and retract work, and that the "occurs check" bars variables from unifying with terms which include them.

```
> java -classpath antlr.jar:. Prolog
Prolog in Java

| ?- A=B,B=C,C=B,C=A,D=C,D=E,C=c.
C = c
D = c
E = c
A = c
B = c
  ? ;
no

| ?- assertz(foo(a)).
yes

| ?- assertz(foo(b)).
yes

| ?- foo(X).
X = a
  ? ;
X = b
  ? ;
no

| ?- listing.
```

```
halt.
consult(File).
nl.
display(Term).
assertz(Term).
display_parse(Term).
fail.
write(Term).
retractall(Term).
=(V,V).
foo(a).
foo(b).
listing.
yes

| ?- retractall(foo(X)).
true  ? ;
no

| ?- foo(X).
no

| ?- A=bar(B).
A = bar(B)
  ?

| ?- A=bar(A).
no

| ?- A=B,A=bar(foo(bar(X, Y, Z), A)).
no
```

I next loaded a relatively simply grammar and a bottom-up parser and ran some sentences through it. Below is a sample interaction with the system, as well as the graphical output it produces using `display_parse`.

```
> java -classpath antlr.jar:. Prolog
Prolog in Java

| ?- consult(grammar1).
consulted: grammar1.pl
yes

| ?- consult(bu_parser).
consulted: bu_parser.pl
yes

| ?- S::cat=s, bu_parse(S, [the,dog,chased,the,cat], Parse),
display_parse(Parse).

Parse =
[s,[np(sg),[det(sg),[the]],[n(sg),[dog]]],[vp(sg,tensed),[v(sg,np(sg),
tensed),[chased]],[np(sg),[det(sg),[the]],[n(sg),[cat]]]]]]
S = s
  ? ;
```

```
no

| ?- S::cat=s,
bu_parse(S,[the,cat,will,have,eaten,the,dog],Parse),display_parse
(Parse).
Parse =
[s,[np(sg),[det(sg),[the]],[n(sg),[cat]]],[vp(sg,tensed),[v(sg,vp(_,inf
),tensed),[will]],[vp(_,inf),[v(_,vp(_,perf),inf),[have]],[vp(_,perf),[
v(_,np(sg),perf),[eaten]],[np(sg),[det(sg),[the]],[n(sg),[dog]]]]]]]]
S = s
  ? ;
no
```

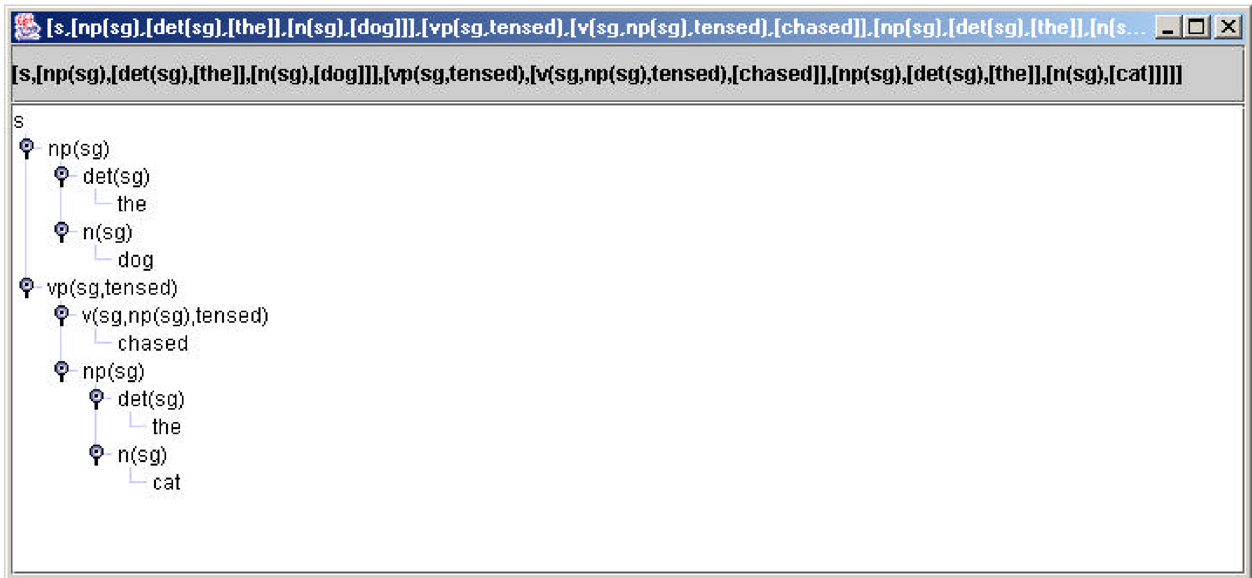

Figure 1 Parse of "The dog chased the cat" using a bottom-up parser, displayed using `display_parse`

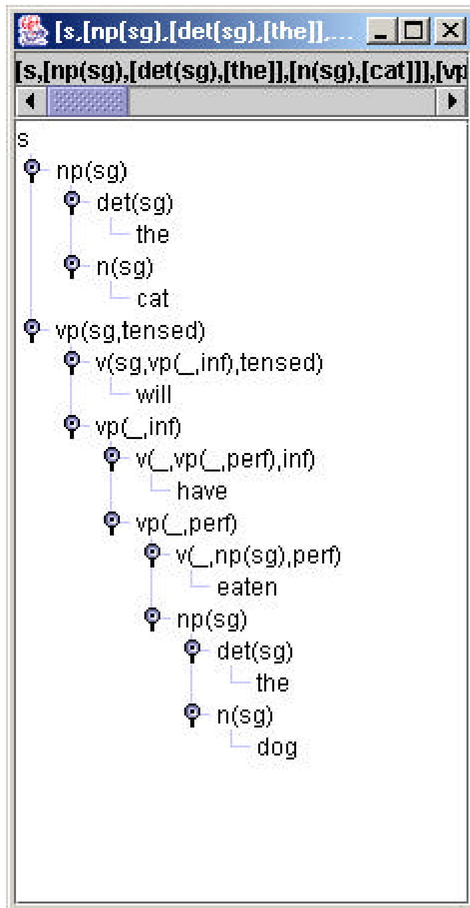

Figure 2 Parse of "The cat will have eaten the dog" using a bottom-up parser, displayed using display_parse

In addition, I also tested two simple definite clause grammars. Here is a sample interaction using one that gives a parse tree. Notice that we can use the "built-in" display predicate, which displays any sort of term, to print out a nice looking parse tree without having to write any special tree-producing code.

```
> java -classpath antlr.jar:. Prolog
Prolog in Java

| ?- consult(dcg2).
consulted: dcg2.pl
yes

| ?- listing.
v(v(chased),[chased|S],S).
np(np(Det,N),S0,S2) :-
        det(Det,S0,S1),
        n(N,S1,S2).
halt.
consult(File).
```

```
n(n(dog),[dog|S],S).
n(n(cat),[cat|S],S).
nl.
display(Term).
s(s(NP,VP),S0,S2) :-
        np(NP,S0,S1),
        vp(VP,S1,S2).
assertz(Term).
display_parse(Term).
fail.
write(Term).
det(det(the),[the|S],S).
retractall(Term).
=(V,V).
vp(vp(V,NP),S0,S2) :-
        v(V,S0,S1),
        np(NP,S1,S2).
listing.
Yes

 | ?- s(Parse, [the,dog,chased,the,cat], []),
display(Parse).
Parse =
s(np(det(the),n(dog)),vp(v(chased),np(det(the),n(cat))))
  ? ;

no
```

Figure 3 Parse of "The dog chased the cat" using built-in DCG features, displayed using `display`

Conclusion

In this project, I have implemented a subset of the Prolog language using Java. Moreover, I have shown how this interaction between Prolog and Java can lead to some interesting interactions, by making use of Java's built-in support for graphical user interfaces. The system has been built in such a way that a Java programmer could implement the interface `Term` in order to send his or her own classes through the prolog machine, or simply use the provided implementations. This means that not only can a prolog program easily call java methods, but that java methods can easily place goals to be proven on the prolog goal stack.

File List

The system can be executed with the command “java –classpath .:antlr.jar Prolog” from its directory.

Java Files	
Atom.java	implementation of Atoms
BasicTerm.java	base class for Atom, CompoundTerm, and Variable
CompoundTerm.java	represents terms with functors and inner terms
JavaMethods.java	here is where java methods which are accessible to the prolog machine should be implemented
Library.java	stores the currently available predicates
Parser.java	uses the ANTLR generated parser to parse files and terms
Predicate.java	represents a predicate
Prolog.java	the main prolog machine
PrologLexer.java/ PrologParser.java/ PrologParserTokenTypes.java	Generated by the ANTLR grammar
Term.java	Interface for all terms
Valid.java	A simple “valid” object which just contains a Boolean
Variable.java	An implementation of the variabless

ANTLR files	
Prolog.g	Grammar for creating the lexer/parser
antlr.jar	Needs to be on the classpath to use the lexer/parser, and hence to use Prolog

Test Files	
dcg.pl	A simple DCG grammar
dcg2.pl	A DCG grammar which yields a parse
grammar1.pl	A grammar which the bottom-up parser can use to parse
bu_parse.pl	The bottom up parser