

Program 1: Prolog Interpreter

CSC 7101, Spring 2007

Due: 8 April 2007

Reading

Please read: “Describing PROLOG by its interpretation and compilation,” by Jacques Cohen. It appeared in *Communications of the ACM*, volume 28, number 12, December 1985, pages 1311–1324.

The Assignment

Your assignment is to design and implement a simple Prolog interpreter in the programming language ML. Your interpreter will implement Prolog with the following note-worthy characteristics:

- The interpreter will print out just the first solution found.
- The interpreter will use a depth-first search, and search the database in the order in which the assertions were made.
- The interpreter will implement the occurs check.
- The interpreter will have a special, predefined relation symbol `init`, but none of the ones found in other implementations, like `atom`, `=`, `fail`, `!` (`cut`), etc.

The main function of your interpreter must be named `Prolog` and take one argument of type `HornClause` (this data structure is defined below). Your interpreter takes one clause at a time and performs one of three actions:

1. adds a clause to a database,
2. determines a solution to a query, or
3. clears the entire database.

All but the second item are easy. Implementing a query requires writing an ML function to perform unification and an ML function to implement the search on the database of facts and rules. An ML algorithm for unification appears in Ryan Stansifer’s *ML Primer*, but it has a minor error.

The interpreter must search the search space in a *depth-first* manner and retrieve the first solution, if one exists. We ask that `Prolog` respond to a query by printing the query and the solution exactly as in the following template:

```
query: ␣␣<clause>  
solution:  
<solution>
```

After the word `query`, two spaces follow the colon after “query.” If no solution exists at all for the query, then print “No” after printing the query. For a solution (if one exists) show the bindings of all the variables occurring in the query (in the order that they appear in the clause from left to right, no duplicates). The format of a solution has the following form:

```

<variable name 1> = <value 1>
<variable name 2> = <value 2>
:

```

If there are no variables in the query, you must still print the heading “**solution:**”.

The function `Prolog` should print “**assert:□□**” followed by the clause whenever a clause is added to the database. When the database is cleared, the string “**Database erased.**” should be printed.

ML

We will be using the implementation of ML known as Standard ML of New Jersey, or SML for short. Version 1.10 of SML is installed on byte, the executable is `/usr/local/bin/sml`. I will also make an executable version available that has all the syntax of Horn clauses predefined. In addition, it has a Prolog interpreter built-in. You can see what your interpreter should do, by running the built-in interpreter.

To make this project easier, a number of functions and data structures are given to you. The SML signature for these are given in figure 1 on page 3. The two key definitions are of types `Term` and `HornClause`:

```

datatype Term =
  Fun of string * Term list |
  Var of string * int      ;

datatype HornClause =
  Headed of (Term * Term list) |
  Headless of Term list      ;

```

`Term` is used to represent Prolog terms. A functor (or function symbol) is represented by a string. (Note that this implies that constants are represented as functions with empty argument lists, for example `Fun("x", [])`.) For the purposes of this assignment do not introduce a new type for relations; represent a relation as a `Term`. This representation is possible since relations are just predicate symbols (which we can represent as strings) and a list of arguments which are just terms. This is the same representation as terms. With this simplification, unification of terms and unification of relations can be performed by the same function.

In Prolog, variables with the same names are different, if they are used in different contexts. The integer part of the `Var` constructor is an easy way to make `Var` objects different, even if they have the same string part. In the article by Cohen it is explained how the recursion level can be used to insure that variables in different contexts are distinct even if they have the same name.

`HornClause` is used to represent Prolog Horn clauses. A fact will be a `Headed` Horn clause with an empty tail. A rule will be a `Headed` Horn clause with more than one term in the list. So, for example, the Prolog rule `a(X,f(X)):-b(X)` would be represented in ML by

```

Headed (
  Fun ("a", [Var("X",0); Fun("f",[Var("X",0)])]), [ Fun ("b", [Var("X",0)]) ]
)

```

A query will be represented as a `Headless` Horn clause.

The Prolog database, called `db`, is a pointer to a list of (headed) clauses. This data structure is predefined by the following definition:

```

val db: (HornClause list) ref = ref [];

```

The query function will need to obtain the value of the database. This is done with the dereferencing operator “`!`” as in `!db`. The SML signature in figure 1 on page 3 lists all the predefined functions and data structures (except for the function `parse` discussed below).

In addition to the declarations above, a number of auxiliary functions are also provided.

```

(* Abstract syntax of Prolog. *)

signature SYN =
  sig
    (*
      Type definition for terms.
      We don't distinguish between function symbols and predicate
      symbols. Variables are the same if they have the same name
      and same instance number.
    *)
    datatype Term =
      Fun of string * Term list |
      Var of string * int      ;

    (*
      Type definition for Horn clauses.
      Headed clauses are used for facts and rules in the database.
      Headless clauses are used for queries.
    *)
    datatype HornClause =
      Headed of Term * Term list |
      Headless of Term list      ;

    (*
      The data base is a variable containing a list of clauses.
    *)
    val db: (HornClause list) ref;

    val PrintTerm: Term -> string;          (* Un-parse Term to string *)
    val PrintClause: HornClause -> string; (* Un-parse HornClause *)
    val OutLine: string -> unit;           (* Output a string on a line *)
    val OutSol: (Term * Term) list -> unit; (* Output one solution *)
    val Init: unit -> unit;                (* Erase the database *)
    val Assert: HornClause -> unit;        (* Add to the database *)
  end;

```

Figure 1: Signature of predefined functions and data structures (except for `parse`).

- `PrintTerm(t)` converts the `Term t` into a string suitable for output. The integer part of a variable is ignored if it is zero, otherwise it is added to the output.
- `PrintClause(c1)` converts the `HornClause c1` into a string suitable for output.
- `OutLine(x)` prints the string `x` on a line by itself.
- `OutSol(1)` prints an individual solution (without the heading) in the manner prescribed by this hand-out.

Parsing

We have also provided the predefined function `parse` which takes strings and parses them into their appropriate `HornClause` form. It is not part of the signature `SYN`, but it is predefined nonetheless.

```
val parse: string -> HornClause;
```

This function is only to make it easier to produce an SML `HornClause`. This is useful in testing your `Prolog` function.

The function `parse` expects the input to be in the syntax of the textbook. Queries are lists of terms terminated by a question mark, and assertions are terminated by a period. Functors and relation symbols are identifiers beginning with a lowercase letter, variables are identifiers beginning with an uppercase letter. For example, the ML expression `parse "a(X,f(X)):-b(X)."` yields the `HornClause` given in the previous section.

Strings that are not correct Prolog clauses will cause an exception to be raised and an error message will be printed.

```
- parse "total garbage!";
Line 1: syntax error found at VAR

uncaught exception ParseError
```

Example

Here is what the dialog with SML should look like when the `Prolog` function is correctly written. Notice that the `parse` function is used to produce the input to the function.

The minus sign is the SML prompt. User input is typeset in `teletype` font. Output from `Prolog` is typeset in *italics*. SML output is in Roman font.

```
- Prolog (parse "init.");
Database erased.
val it = () : unit
- Prolog (parse "p(a,b,c).");
assert: p(a,b,c).
val it = () : unit
- Prolog (parse "p(A,B,C)?");
query: P(a,b,c)?
solution:
A = a
B = b
C = c
val it = () : unit
```

The built-in Prolog interpreter can be useful in determining the correct output for arbitrary Prolog programs. A sample dialog with the Prolog interpreter is given below. The Prolog interpreter is not a standard part of SML, so it is available only when executing the special executable file `pml`.

```
byte--> pml
Special version for CSC 7101 with Prolog definitions preloaded.
  Type 'prolog()' for a Prolog interpreter; type ^D
  to exit the Prolog interpreter. (Type ^D again to exit SML.)
val it = () : unit
- prolog();
PROLOG> init.
Database erased.
PROLOG> p(a,b,c).
assert: p(a,b,c).
PROLOG> p(A,B,C)?
query: p(A,B,C)?
solution:
A = a
B = b
C = c
PROLOG> ^D
- ^D
```

You can develop your Prolog interpreter using only the `pml` executable. In this case, you would only load your file containing the function `Prolog`. You would run your interpreter with calls of the form `Prolog (parse "...");` as described above. The predefined interpreter would then be run with the call `prolog()`;

Alternatively, you can develop your interpreter in regular `sml`. Then you need to load the predefined functions and data structures via use `"load.sml"`; Whenever you reload a new version of your file `prolog.sml`, which contains the function `Prolog()`, you would also need to reload `top-level.sml`. In this case, you can use the top-level function `prolog()` with your own interpreter.

Turning it in

Write your Prolog interpreter as clearly as possible, including reasonable comments. The file `prolog.sml` must define a function `Prolog` of type `HornClause -> unit`.

Put your files in the directory `~/prog1` in your `cs7101xx` account on `byte` and submit it with

```
~cs710100/bin/p_copy 1
```

We will run your function on a sequence of assertions and queries, and check the output against our answers. In some cases, different answers are possible (in the case of unconstrained variables). Note that the order of the bindings is prescribed, and the form of the output is controlled by the given output routines.