CS 131: Programming Languages, Spring 2003

Assignment 7
Mini-Prolog Interpreter

Due: Wednesday, April 9

Goals for this Assignment

1. Recalling ideas of logic programming from CS 60
2. Implementing and using unification
3. Using continuation functions for backtracking search.

Office Hours

The purpose of office hours is to provide you with a chance to discuss class-related topics, to ask questions about material you find particularly interesting or particularly confusing, and to provide comments about the class.

You can drop by Olin 1253 any time to see if the professor is around; times specifically set aside are Tuesday, Thursday, and Friday from 2:30–4pm. Grutors are available in the Terminal Room on Tuesdays 7–9pm and Sundays 2–4pm. Questions can also be sent to `cs131help@cs.hmc.edu` .

Instructions

- Get copies of all the files from `/cs/cs131/src/7` on turing.
- Start by take a look at the files `absyn.sig` and `parse.sig`, and the test files `lists.pl` and `zebra.pl`.
- When done, you should `cs131submit` the files `parse.sml` and `prolog.sml`.

1 Unification (40%)

Metavariables can be thought of as variables whose values are pieces of syntax; they stand for missing or unknown information.

In most formal mathematical treatments, *unification* is the procedure which, given two "phrases" u_1 and u_2 from some fixed language, attempts to find a substitution σ of terms for metavariables such that $u_1\sigma = u_2\sigma$. (Here $u_1\sigma$ means to apply the substitution σ to the term u_1.)

Dealing with substitutions can be difficult to make efficient, so in this assignment, we will take a more "imperative" view, which is commonly used in implementations. Metavariables are mutable (implemented using SML `ref`s), and the goal of the unification procedure is to plug definitions directly into the representation of the metavariable, rather than carrying around a separate mapping associating metavariables with their definitions.

A value of type `Absyn.metavar` can be thought of as a box which is either empty (i.e., it is an unset metavariable) or contains a value of type `Absyn.term`; there is also a string and a number associated with each metavariable that are used only to distinguish different metavariables when they are displayed; a metavariable will look like `M{X3=bob}` (a metavariable with identifiers `"X"` and `3` that has been found to be equal to the atomic term `bob`) or `M{X3=UNSET}` if the metavariable still has no definition.

The function `newMetavar` creates new metavariables (you specify the string identifier; a fresh number is automatically chosen). Then `setMetavar` can be used to specify the definition of an *undefined* metavariable.

The file `absyn.sml` includes a function

```
Absyn.follow : Absyn.term -> Absyn.term
```

which takes a term and returns an equivalent term that is not a metavariable with a definition. Given any non-metavariable term or an unset metavariable, this function simply returns its argument; given a metavariable with a definition, this function takes the definition and recursively applies `follow` (because it might turn out that the definition itself is a metavariable with a definition).

The `prolog.sml` file contains a definition for a function

```
unify : Abysn.term * Absyn.term -> unit
```

which takes its two arguments, calls `follow` on each type and then passes the result to a function `unify'`; this latter function actually does the work, which should gives values to any unset metavariables if necessary to make the two types equal. Complete the definition of `unify'`. This function should raise the exception `Unify` if the arguments cannot be unified.

Remember:

- If asked to unify a metavariable with itself, `unify` can simply return without doing any real work; the two inputs are already equal.

- To unify a metavariable with anything else, just define the metavariable to be that anything.
- Otherwise, for two terms to unify they must have the same "shape", the same strings and integers, and corresponding subterms must unify.

2 Adding Quantifiers (10%)

The concrete syntax of mini-Prolog is shown in Figure 1. [Note: we will not be implementing or parsing arithmetic or comparisons.] The base input is a sequence of `<fact>`s read in from a file; the user is then allowed to enter a query (a `<hypothesis>`) that the system tries to derive from the given facts.

```
<term> ::= <capitalized variable>          // X, Y, List, FooBar, ...
         | <lowercase constant>            // a, bob, kentucky, ...
         | <integer>
         | _
         | <lowercase predicate-name> ( <terms> ) // member(4,Y), ...
         | []
         | [ <terms> ]                     // [3,[4,5],3]
         | [ <terms> | <term> ]            // [X,Y | Z]

<terms>  ::= <term>
           | <term> , <terms>

<fact>   ::= <term> .                      // parent(pat,chris)
             <term> :- <hypothesis> .      // weird(X) :- parent(X,X)

<hypothesis> ::= true .
               | <term> .
               | <hypothesis> , <hypothesis> .
               | <term> = <term> .
```

Figure 1: Mini-Prolog Concrete Syntax

For example, in the fact

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

the comma between the two `parent` specifications means *logical-and*, while the `:-` symbol is *reverse implication*. That is, the fact can be read as *if* `X` is a parent of `Z` *and* `Z` is a parent of `Y` then `X` is a parent of `Y`.

The abstract syntax (see `absyn.sig`) is largely similar, but with several important differences. First, a `<fact>` that is just a `<term>`, such as

```
append([],Y,Y).
```

(which might informally be interpreted as "the result of appending the empty list to a list `Y` is the same list `Y`") represented internally as in implication, as if the user had written

```
append([],Y,Y) :- true.
```

(i.e., "*if* `true` *then* the result of appending ...") This is not critical, but means that all facts boil down to implications.

As one might expect, the abstract syntax boils list notation down to uses of `Cons` and `Nil`. Recall that the prolog notation `[X,Y|Z]` corresponds to the SML notation `X::Y::Z` (not `[X::Y::Z]`).

Next, the abstract syntax is intended to be pickier about exactly how variables are quantified. When we are being very specific about a fact such as

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

what we are really saying is that *for all* values `X` and `Y`, if *there exists some* value `Z` such that `parent(X,Z)` and `parent(Z,Y)` are true, then we can conclude that `grandparent(X,Y)` also holds. Thus what we really want is abstract syntax along the lines

$$\forall \texttt{X}.\forall \texttt{Y}.\ ((\exists \texttt{Z}.\ \texttt{parent(X,Z)} \wedge \texttt{parent(Z,Y)}) \Rightarrow \texttt{grandparent(X,Y)})$$

The rules in prolog are that any variable appearing on the left-hand-side of the `:-` are universally quantified, and all other variables in the hypothesis are existentially quantified.

The parser returns facts with the quantifiers missing; the above input results in

```
Implies(And(Predicate("parent", [Var "X", Var "Z"]),
            Predicate("parent", [Var "X", Var "Y"])),
        Predicate("grandparent", [Var "X", Var "Y"]))
```

You have been supplied with an embarrassingly simple implementation of sets of variable names (of type `varset`).

1. In `parse.sml`, fix the definition of the function

   ```
   quantifyH : Absyn.hypothesis * varset -> Absyn.hypothesis
   ```

   which wraps the given hypothesis in existential quantifiers for all variables in the given hypothesis that are not members of the given set. You may assume the given hypothesis contains no quantifiers.

2. In `parse.sml`, fix the definition of the function

   ```
   quantifyF : Absyn.fact -> Absyn.fact
   ```

   which adds all the implied existential and universal quantifiers to the given fact. You may assume that this fact starts out with no quantifiers.

Main Interpreter (50%)

Finally, you are to complete `prolog.sml` by writing the main backtracking-search routine that is the core of the interpreter. This should be implemented as follows:

The function

```
prove : Absyn.hypothesis -> Absyn.fact list -> (unit->unit) -> unit
```

takes three arguments: the query to be proved, the current database of usable facts, and a so-called *success continuation*, that is a function to call if and only if the current query is proved. The *prove* function should simply return if the query is unsolvable. More specifically

- If the query is an equality test, the query succeeds if and only if the given two terms unify. (We want to *return* if unification fails, and call the success continuation otherwise. You have been supplied with a helper function `unifyWithCont` which does exactly this: it tries to unify two terms, and if successful invokes a success continuation.

- If the query is a conjunction, we want to check if the first sub-query can be satisfied and if so (i.e., iff it invokes whatever success continuation function it was passed!) the second sub-query is then checked. Code of the form

  ```
  prove hyp1 facts sc; prove hyp2 facts sc
  ```

  does not satisfy the requirements, as it (a) calls the success continuation `sc` as soon as only part of the goal is satisfied, and (b) invokes the second `prove` only when the first `prove` gives up (by returning). [Hint: design your own success continuation.]

- If the goal is existentially quantified (we want to know whether there exists a variable X such that ...) we can simply remove the quantifier, replace the variable X with a fresh metavariable (representing the unknown value of the X we're hoping exists) and recursively try to prove. I.e., if the goal is $\exists$`X.parent(pat,X)` then we can reduce this to solving `parent(pat,`M`)` where M is a fresh metavariable.

- If the goal is simply a single predicate, then you should invoke the helper function `provePred` (see below).

- `true` is always true and requires no further work to successfully prove.

The function `provePred` is like `prove`, but concentrates on proving a single predicate. The idea is to search through the database of facts looking for those whose conclusion unifies with the predicate we're trying to prove.

- For each fact, if the fact is universally quantified then we can can again remove the quantifier, replace the quantified variable with a fresh metavariable, and then recursively consider whether the resulting fact is helpful.

- If the fact is an implication, we can check whether the conclusion unifies with the predicate we're trying to prove; if so we want to try to recursively satisfy the hypothesis of this implication (and then call the original success continuation iff this hypothesis was provable). [Hint: use `unifyWithCont` and an appropriate success continuation.] For example, if we want to show `grandparent(sam,pat)` and the fact has become `grandparent(`M_1`,`M_2`) :-` $\exists$ `Z. parent(`M_1`,Z)` $\wedge$ `parent(Z,`M_2`)`, it suffices to try to prove $\exists$ `Z. parent(sam,Z)` $\wedge$ `parent(Z,pat)` — or more precisely, $\exists$ `Z. parent(`M_1`,Z)` $\wedge$ `parent(Z,`M_2`)` where unification has defined M_1 to be `sam` and M_2 to be `pat`].

If none of the facts end up being useful, we want `provePred` to simply return without calling the success continuation.

Further, when we try to use a fact to prove our goal it might start out well but eventually turn out to be a false path (i.e., if the hypothesis of a rule with the right sort of conclusion turns out to be unprovable). Thus, when going on to the next fact in the database, we want to undo all the metavariable definitions we tried on this false path.

For this purpose, there is a very helpful function

```
Absyn.tryThenUndoChanges : (unit->unit) -> unit
```

which runs the given function (which might try to use a given fact, for example) and then undoes *all* the changes to metavariables done by that function — backtracking so that we can try a different fact.

Notes on using the interpreter

The main function `Prolog.consult` takes the name of a file containing the database (e.g., `"lists.pl"`), reads it in, and then starts an interactive loop. At the prompt is `?-` you can type in a goal without quotes but ending in a period (e.g., `append(X,Y,[a,b,c,d]).`).

The `consult` function runs the prover you wrote with a success continuation that first prints out the values of the variables in your input causing the proof search to succeed, and then waits for user input. If you enter a semicolon (followed by the return key) you have rejected this solution, and the system will then continue searching for another solution. Otherwise you are returned back to the prompt.