CS 131 Programming Languages

November 5, 2009

Prolog and Homework 4

There is no plus operation in Prolog so there's nothing to overload. The assignment says that kenken has to work in a certain way; at some point it will have a list of form `[+(3,[A,B])]`.

```
satisfy_constraint(+(A,B)) :- plus(A,B).
```

Prolog is based on logic and a lot of its strengths come from the fact that it's based on logic. But we don't give you classes in logic or at least there's no requirement to take a logic class before taking this class. Let's go over the basics of logic.

Prolog Theory and Mathematical Logic

Depending on notation, different symbols may represent the basic logic operations. AND can be represented in a variety of ways. In Prolog, AND is "," and OR is ";". Logical implication (if P then Q) is written as "<=" in C.

When is it true that P implies Q? Let P be "the freeway is busy" and Q be "it's raining." In standard logic, P implies Q is false *only* when P is true and Q is false. "Implies" symbol doesn't exactly correspond to the idea of implication in natural language. Logical implication is purely an operation on booleans and if it disagrees with intuition, then the intuition is wrong.

```
(P → Q ^ Q → P) <-> (P <-> Q)
(P → Q) <-> (!P v Q)
```

The above statements are known as *tautologies* and don't contain any useful information. We'd like to avoid tautologies in Prolog because they don't give the interpreter any useful information.

A *proposition* is a statement about the world that's either true or false. *Propositional logic* can be solved by an algorithm to find whether any statement is true or not. In that way propositional logic is an easy problem to solve.

All men are mortal. Socrates is a man. Socrates is mortal.

The above statement cannot be captures in propositional logic because there's no way to express "all" in it. We'll need logical variables. First-order logic (aka predicate calculus) contains logical variables and *quantifiers – for all* and *there exists*.

```
for_all(X) { if (man(X) then mortal(X) } ^ man(Socrates) → mortal(Socrates)
```

We add two add-ons to the first-order logic. *Function symbols* is one of them. *Equality* is another. Although function symbols (*structures*) and equality (*unification*) are present in Prolog, quantifiers are absent, and there's no negation.

Mathematicians came up with *clausal form* to automate solving of first-order logic. First, represent any problem as a series of clauses. Each A and B stand for one of the predicates (what looks like a Prolog goal). A's are called the *antecedents* and the B's are called the *consequents*. There is no negation and no quantifiers in clausal form.

```
B1 v B2 v B2 v … v Bn ← A1 ^ A2 ^ A3 ^ … ^ Am   // clausal form

P(X, Y) ← Q(X) ^ R(Y) // for all X for all Y is implied. there exists is just syntactic sugar for for all.
```

Here's a statement in clausal form that can't be expressed in Prolog. In this way, clausal form is more powerful.

```
dog(X) v cat(X) ← licensed(X) ^ allowed_in_apt(X).
```

All statements in predicate calculus can be converted to clausal form without changing their meaning. Prolog doesn't quite do every kind of the clause. Also, it's not possible to solve any statement in clausal form because of the *halting problem*.

Horn clause is a clause in which there's only one implication. A Horn clause of size 0 (n = 1, m = 0) is simply a Prolog *fact*:

```
prereq(cs31, cs131) ← (nothing).     // right side (nothing) implies left side; left side must be true
```

A Horn clause of size 1+ (n = 1, m > 0) is a Prolog *rule*:

```
pr(A, Z) ← prereq(A, B) ^ pr(B, Z).  // left side implies right side
```

A Horn clause with n = 0 is a Prolog *query* or *goal*:

```
(nothing) ← pr(A, B) ^ pr(B, A).     // this implies false so it must be false
```

Prolog is designed to *prove you wrong* so it must come up with a counter-example to your query.

What's Wrong with Prolog?

We started by saying that *algorithm = logic + control*. Even in pure Prolog, which is what we're using (no control advice, no *cuts*) you still have to know something about the control or you'll write code that's way too slow. In many system one must give control advice and use cuts to make the code work. Also, Prolog really works best if one can assume the closed-world assumption (and it's essential if one's going to use *cuts* or *not-provable (\f)*). Therefore, it's not really ready for general-purpose programming.

Storage Management

Simplest storage management system:

- Static allocation via absolute address, at compile time
- Simple, never run out of storage, fast code
- Inflexible because it doesn't respond to user input (static sizes of arrays, etc)
- Can't do recursion because the inner calls will modify the variables of outer calls

The easiest way to support recursion is to have a *stack* and to use stack allocation of *frames* (*activation records*). A frame or an activation record is a piece of storage that represents the current state of a function that's executing. It needs the activation record to keep copies of its local variables, return address, and saved registers of the caller, including return address (instruction pointer) and return activation address (frame pointer). *Frame pointer* points at the activation record, which contains anything that the function needs. Looking at the offset from the frame pointer will retrieve the needed data. *Disadvantage*: the amount of storage for each frame must be fixed and known at the compile time. All arrays have the compile-time size.

We'd like to add a new feature to decide the sizes of arrays at run time – dynamic allocation of arrays on the stack. The frame size is now variable and we'll need a *stack pointer* in addition to the frame pointer, which keeps track of how much more space we have allocated. Allocating and releasing stack storage is just a matter of subtracting and adding to the stack pointer.

Algol, like ML, and other real languages, lets functions define other functions (*nested functions*).

```
f() {
        int x;
        int g(int y) {
                return x + y;
        }
}
```

```
int h(int z) {
        g(12);
}
```

Somehow `g()` must know what the value of variable `x` is even when `g()` is called from outside of `f()`. A new frame pointer is introduced, called *definer's frame pointer*. To retrieve the variable `x`, `g()` must follow this frame pointer.

The technical names for the link lists made by the frame pointer are:

- *Dynamic chain*: linked list to caller's frame, caller caller's frame, etc.
- *Static chain*: linked list to definer's frame, definer definer's frame, etc.

What about *functions that return other functions*? When combined with nested functions, the stack-based approach doesn't work anymore because the stack space is reclaimed by the time the function tried to access its parent's variables. Two possibilities arise: first, don't do that. Second, have storage manager not use a stack for frames and keep the frames until they're garbage-collected. The downside is that the second approach is slower because stack allocation is very cheap and fast.