

## Lecture 32: Declarative Programming (Under the Hood)

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 1

## Review: A "Schemish" Prolog

- Programs in our language define subsets of Scheme expressions that will be considered "true."
- (fact *CONCLUSION*) means that *CONCLUSION* is to be taken as true, for any replacement of its logical variables.
- (fact *CONCLUSION HYPOTHESIS...*) means that *CONCLUSION* is to be taken as true for all replacements of the logical variables that cause each of the the *HYPOTHESES* to be true.
- logical variables*, represented as symbols starting with '?', stand for operands that may be replaced by other expressions (including other logical variables).

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 2

## Another Example: Lists

- In ordinary Scheme, `append` (or `extend` in Python) is a function taking two lists and returning a list.
- In our Scheme Prolog, it is a *relation between three lists*, which we define by writing two facts about it that cover all cases:  

```
;;; (append-to-form A B C) means "appending list B to list A produces  
;;; list C.
```

```
; Fact about the empty list.  
(fact (append-to-form () ?x ?x))  
; Fact about a general non-empty list  
(fact (append-to-form (?a . ?r) ?b (?a . s)) ; assuming that  
      (append-to-form r b s))
```

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 3

## Applying append-to-form

```
logic> (fact (append-to-form () ?x ?x))  
logic> (fact (append-to-form (?a . ?r) ?b (?a . s))  
        (append-to-form r b s))  
logic> (query (append-to-form (a b c) (d e f) (a b c d e f)))  
Success!  
logic> (query (append-to-form (a b c) (d e f) ?x))  
Success!  
x: (a b c d e f)  
logic> (query (append-to-form ?x (d e f) (a b c d e f)))  
Success!  
x: (a b c)  
logic> (query (append-to-form (a b c) ?y (a b c d e f)))  
Success!  
y: (d e f)  
logic> (query (append-to-form (a . ?r) ?x (a b c d e f)))  
???
```

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 4

## Permutations (Anagrams)

- When is list *B* a permutation (reordering) of *A*?
- An obvious fact:  

```
logic> (fact (permutation () ()))
```
- Key fact: every permutation of  $(a . R)$  consists of a permutation of *R* with *a* inserted somewhere in that permutation:  

```
(0 1 2 3 4) ==> (4 3 1 2)  
                  ↑  
                  0
```
- Or, in our logic language:  

```
logic> (fact (permutation (?a . ?r) ?s)  
            (permutation ?r ?t) (insert ?a ?t ?s))
```

  
where we intend `(insert x L0 L1)` to mean that inserting *x* into *L0* (at the right place) gives *L1*:  

```
logic> (fact (insert ?a ?r (?a . ?r)))  
logic> (fact (insert ?a (?b . ?r) (?b . ?s)) (insert ?a ?r ?s))  
1
```

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 5

## Operational and Declarative Meanings

- An assertion  

```
(fact (eats ?P ?F) (hungry ?P) (has ?P ?F) (likes ?P ?F))
```

  
means that for any replacement of *?P* (e.g., 'brian') and *?F* (e.g., 'potstickers') throughout the rule:  
**Declarative Meaning** If brian is hungry and has potstickers and likes potstickers, then brian will eat potstickers.  
**Operational Meaning** To show that brian will eat potstickers, show that brian is hungry, then that brian has potstickers, and then that brian likes potstickers.
- The *declarative meaning* allows us to look at our Scheme-Prolog program as a logical specification of a problem for which the system is to find a solution.
- The *operational meaning* allows us to look at our Scheme-Prolog specification as an executable program for searching for a solution.
- Closed Universe Assumption:** We make only positive statements. The closest we come to saying that something is false is to say that we can't prove it.

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 6

## Unification

- In general, our system, given a target expression involving a predicate to prove, must find a fact that might assert that target, given a suitable replacement of logical variables.
- To do this, we try to pattern-match the conclusions of all our facts against the target expression.
- The pattern matching is called **unification**, [J. A. Robinson].

```
(likes brian potstickers) }  
(likes ?P ?F) } True: {P: brian, F: potstickers}
```

- The substitution itself (the dictionary on the right) is called a **unifier**.

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 7

## Unification (II)

- The substitution has to be uniform:

```
(1e 0 1) }  
(1e ?x ?x) } False
```

- And logical variables may appear in either expression (unification is **symmetric**).

```
(related (a b c) ?x ) }  
(related ?x (a . ?r)) } True: { x: (a b c), r: (b c) }
```

- It is possible for logical variables to be unified with each other:

```
(likes ?P yams) }  
(likes ?Q ?F ) } True: { P: ?Q, F: yams }, or { Q: ?P, F: yams }
```

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 8

## Implementing Unification

- A plain, unbound logical variable will unify with anything. Must record this unification in the unifier we construct.
- Before unifying other (bound) logical variables, first must replace them with their recorded bindings, in order to make sure we bind consistently.
- To unify two atoms (numbers, booleans, symbols that are not logical variables), just compare them.
- To unify two lists: recursively unify their heads and tails.

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 9

## Implementing Unification: Code

A simple tree recursion with side-effects:

```
def unify(e, f, env):  
    """Destructively extend ENV so as to unify (make equal) E and F, returning  
    True if this succeeds and False otherwise. ENV may be modified in either  
    case (its existing bindings are never changed)."""  
    e = lookup(e, env)  
    f = lookup(f, env)  
    if scheme_eqvp(e, f):  
        return True  
    elif isvar(e):  
        env.define(e, f)  
        return True  
    elif isvar(f):  
        env.define(f, e)  
        return True  
    elif scheme_atomp(e) or scheme_atomp(f):  
        return False  
    else:  
        return unify(e.first, f.first, env) and unify(e.second, f.second, env)
```

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 10

## Using Unification to Search for Proofs

- The process of attempting to demonstrate an assertion (answer a query) is a systematic **depth-first search** of facts.

```
def search(clauses, env):  
    if clauses is nil:  
        yield env  
    for fact in fact database:  
        fact = rename_variables(fact, ...)  
        env_head = new environment that extends env  
        if unify(conclusion of fact, first clause, env_head):  
            for env_rule in search(hypotheses of fact, env_head):  
                for result in search(rest of clauses, env_rule):  
                    yield result
```

- In the actual program, we put on a **depth limit**: a limit on how deeply the recursive calls on search may go.
- This prevents us from going down infinite paths when there is a finite path that will work.

Last modified: Fri Apr 18 13:57:42 2014

CS61A: Lecture #32 11