Announcements

- HW12 due tonight
- HW13 out
- Scheme project, contest due Monday

Logic Language Review

Expressions begin with `query` or `fact` followed by relations

Expressions and their relations are Scheme lists

- `logic> (fact (parent eisenhower fillmore))`
- `logic> (fact (parent fillmore abraham))`
- `logic> (fact (parent abraham clinton))`

Success!

who: fillmore
who: eisenhower

If a fact has more than one relation, the first is the conclusion, and it is satisfied if the remaining relations, the hypotheses, are satisfied

If a query has more than one relation, all must be satisfied

The interpreter lists all bindings that it can find to satisfy the query

Hierarchical Facts

Relations can contain relations in addition to atoms

- `logic> (fact (dog (name abraham) (color white)))`
- `logic> (fact (dog (name clinton) (color white)))`
- `logic> (fact (dog (name delano) (color white)))`
- `logic> (fact (dog (name eisenhower) (color tan)))`
- `logic> (fact (dog (name fillmore) (color brown)))`
- `logic> (fact (dog (name grover) (color tan)))`
- `logic> (fact (dog (name herbert) (color brown)))`

Variables can refer to atoms or relations

- `logic> (query (dog (name clinton) ?color))`
  Success!
  color: white

- `logic> (query (dog (name clinton) ?info))`
  Success!
  info: (color white)

Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

- `logic> (query (dog (name ?name) (color ?color))`  
  (ancestor ?ancestor ?name)  
  (dog (name ?ancestor) (color ?color)))

Success!

name: barack  color: tan  ancestor: eisenhower
name: clinton  color: white  ancestor: abraham
name: grover  color: tan  ancestor: eisenhower
name: herbert  color: brown  ancestor: fillmore

Example: Appending Multiple Data Sources

Two lists append to form a third list if:

- The first list is empty and the second and third are the same
  
  () (a b c)
- Both of the following hold:
  
  - List 1 and 3 have the same first element
  - The rest of list 1 and all of list 2 append to form the rest of list 3

- `logic> (fact (append-to-form () ?x ?x))`
- `logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z))
  (append-to-form ?r ?y ?z))`
Logic Example: Anagrams

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list
- The first element of the list inserted into an anagram of the rest of the list

```
(fact (insert `a ?r ([`a . `r])))
(fact (insert `a (`b . `r) [(`b . ?s)])
 (insert `a ![r] ![s]))
(fact (anagram () ()))
(fact (anagram `a ![b])
 (insert `a ![s] ![b])
 (anagram ![r] ![s]))
```

Pattern Matching

The basic operation of the Logic interpreter is to attempt to unify two relations.

Unification is finding an assignment to variables that makes two relations the same.

```
( (a b) c (a b) ) → True, {x: (a b)}
( ?x c ?x )
( (a b) c (a b) ) → True, {y: b, z: c}
( (a ?y) ?z (a b) )
( ?x ?x ?x ) → False
```

Unification

Unification unifies each pair of corresponding elements in two relations, accumulating an assignment.

1. Look up variables in the current environment
2. Establish new bindings to unify elements

```
(a b) (a b)   (a b) (a b)
(?x) (?x)    (?x) (?x)
```

Symbols/relations without variables only unify if they are the same

{ x: (a b) }

Success!

Implementing Unification

```
def unify(e, f, env):
  if e == f:
    return True
  elif isvar(e):
    if e == f:
      f = lookup(f, env)
      return True
    elif scheme_atomp(e) or scheme_atomp(f):
      e = lookup(e, env)
      if e == f:
        return True
      return False
  else:
    unify(e.first, f.first, env) and 
    unify(e.second, f.second, env)
```

Unification recursively unifies each pair of elements

Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true.

```
(fact (app ![1] ![2] ![3]))
(fact (app ![1] ![2] ![3]) ![2])
(query (app ![1] ![2] ![3]))
```

Variables are local to facts and queries

```
left: ([0] ![0]) → ![0]
```
Underspecified Queries

Now that we know about Unification, let’s look at an underspecified query.

What are the results of these queries?

> (fact (append-to-form () ?x ?x))
> (fact (append-to-form (?a . ?r) ?x (?a . ?s)))
> (query (append-to-form (1 2) (3) ?what))

Success!

what: (1 2 3)

> (query (append-to-form (1 2 . ?r) (3) ?what))

Success!

r: () what: (1 2 3)

r: (?s_6) what: (1 2 ?s_6 3)

r: (?s_6 ?s_8 ?s_10) what: (1 2 ?s_6 ?s_8 ?s_10 3)

r: (?s_6 ?s_8 ?s_10 ?s_12) what: (1 2 ?s_6 ?s_8 ?s_10 ?s_12 3)

...

Now that we know about Unification, let’s look at an underspecified query:

What are the results of these queries?

Success!

what: (1 2 3)

Success!

r: (?s_6) what: (1 2 ?s_6 3)

r: (?s_6 ?s_8 ?s_10) what: (1 2 ?s_6 ?s_8 ?s_10 3)

r: (?s_6 ?s_8 ?s_10 ?s_12) what: (1 2 ?s_6 ?s_8 ?s_10 ?s_12 3)

Search for possible unification

The space of facts is searched exhaustively, starting from the query and following a depth-first exploration order.

A possible proof is explored exhaustively before another one is considered.

```python
def search(clauses, env):
    if clauses is nil:
        yield env
    elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:
        for fact in facts:
            fact = rename_variables(fact, get_unique_id())
            env_head = Frame(env)
            if unify(fact.first, clauses.first, env_head):
                for env_rule in search(fact.second, env_head, depth+1):
                    for result in search(clauses.second, env_rule, depth+1):
                        yield result
```

Some good ideas:

- Limiting depth of the search avoids infinite loops
- Each time a fact is used, its variables are renamed
- Bindings are stored in separate frames to allow backtracking

Implementing Search

```python
def search(clauses, env, depth):
    if clauses is nil:
        yield env
    elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:
        for fact in facts:
            fact = rename_variables(fact, get_unique_id())
            env_head = Frame(env)
            if unify(fact.first, clauses.first, env_head):
                for env_rule in search(fact.second, env_head, depth+1):
                    for result in search(clauses.second, env_rule, depth+1):
                        yield result
```

An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```logic
(fact (ints 1 2))
(fact (ints 2 3))
(fact (ints 3 4))
(fact (ints 4 5))
```

Then we define addition:

```logic
(fact (add (ints ?x ?y) (ints ?x ?y)))
```

Finally, we define the evaluator:

```logic
(fact (eval (ints ?x ?y) (ints ?x ?y) ?val))
```

Success!

what: 2
what: (+ 1 (ints 2))
what: (+ 1 1)