The Logic Language

The logic language was invented for Structure and Interpretation of Computer Programs

- Based on Prolog (1972)
- Expressions are facts or queries, which contain relations
- Expressions and relations are Scheme lists
- For example, (likes john dogs) is a relation

Simple Facts

A simple fact expression in the Logic language declares a relation to be true

Let’s say I want to track the heredity of a pack of dogs

Language Syntax:
- A relation is a Scheme list
- A fact expression is a Scheme list of relations

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{delano} \ \text{herbert}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{abraham} \ \text{barack}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{abraham} \ \text{clinton}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{fillmore} \ \text{abraham}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{fillmore} \ \text{delano}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{fillmore} \ \text{grover}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{eisenhower} \ \text{fillmore}))
\]

Relations are Not Procedure Calls

In Logic, a relation is not a call expression.
- Scheme: the expression (abs -3) calls abs on -3. It returns 3.
- Logic: (abs -3) asserts that abs of -3 is 3.

To assert that 1 + 2 = 3, we use a relation:

\[
\text{logic} \ (\text{add} \ 1 \ 2 \ 3)
\]

We can ask the Logic interpreter to complete relations based on known facts.

\[
\text{logic} \ (\text{add} \ ? \ 2 \ 3)
\]

\[
\text{logic} \ (\text{add} \ 1 \ ? \ 3)
\]

\[
\text{logic} \ (\text{add} \ 1 \ 2 \ ?)
\]

\[
\text{logic} \ (\text{add} \ ? \ 2 \ 3)
\]

Queries

A query contains one or more relations that may contain variables.

Variables are symbols starting with ?

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{delano} \ \text{herbert}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{abraham} \ \text{barack}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{abraham} \ \text{clinton}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{fillmore} \ \text{abraham}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{fillmore} \ \text{delano}))
\]

\[
\text{logic} \ (\text{fact} \ (\text{parent} \ \text{fillmore} \ \text{grover}))
\]

\[
\text{logic} \ (\text{query} \ (\text{parent} \ \text{abraham} \ ?\text{puppy}))
\]

Success:
- puppy: barack
- puppy: clinton

Each line is an assignment of variables to values
Compound Queries

An assignment must satisfy all relations in a query.

(query <relation1> <relation2> ... <relationN>)

is satisfied if all the <relation> are true.

Recursive Facts

A fact is recursive if the same relation is mentioned in a hypothesis and the conclusion.

Hierarchical Facts

Relations can contain relations in addition to symbols.

Variables can refer to symbols or whole relations.
Combining Multiple Data Sources

Which dogs have an ancestor of the same fur?

\[
\text{logix} \ (\text{query} \ ((\text{dog} \ (\text{name} \ x) \ (\text{fur} \ y)) \ (\text{ancestor} \ y \ x)) \ (\text{dog} \ (\text{name} \ y) \ (\text{fur} \ y)))]
\]

Success!

\[
\begin{align*}
x: \text{barack} & \quad \text{fur:} \text{short} \\
y: \text{eisenhower} & \\
x: \text{clinton} & \quad \text{fur:} \text{long} \\
y: \text{abraham} & \\
x: \text{grover} & \quad \text{fur:} \text{short} \\
y: \text{eisenhower} & \\
x: \text{herbert} & \quad \text{fur:} \text{curly} \\
y: \text{fillmore} &
\end{align*}
\]

Appending Lists

Lists in Logic

Expressions begin with query or fact followed by relations.

Expressions and their relations are Scheme lists.

\[
\text{fact} \ (\text{append-to-form} \ () \ ?x \ ?x)
\]

\[
\text{fact} \ (\text{append-to-form} \ (?a . ?r) \ ?y \ (?a . ?z))
\]

\[
\text{query} \ (\text{append-to-form} \ ?left \ (c \ d) \ (e \ b \ c \ d))
\]

Success!

\[
\begin{align*}
\text{left:} \ (e \ b)
\end{align*}
\]

The interpreter lists all bindings of variables to values that it can find to satisfy the query.

Define Base Fact of append-to-form So That No Lists Can Be Empty

\[
\text{append-to-form} \ (1 \ 2 \ 3) \ (1 \ 2 \ 3)
\]

\[
\text{append-to-form} \ (1) \ (2 \ 3) \ (1 \ 2 \ 3)
\]

\[
\text{append-to-form} \ (1 \ 2 \ 3) \ () \ (1 \ 2 \ 3)
\]

Recursive fact

\[
\text{fact} \ (\text{append-to-form} \ (?) \ ?x \ (?a \ . \ ?x))
\]

\[
\text{fact} \ (\text{append-to-form} \ (?a \ . \ ?r) \ ?y \ (?a \ . \ ?z))
\]

Anagrams in Logic

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.

\[
\text{fact} \ (\text{insert} \ ?a \ ?x \ (?ax))
\]

\[
\text{fact} \ (\text{anagram} \ () \ ())
\]

\[
\text{fact} \ (\text{anagram} \ ?a \ ?x)
\]

Unification
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.

\[
\begin{align*}
(a \ b) & \ c & & (a \ b) \\
\{ x \} & & & \{ x \} \\
\end{align*}
\]

True, \{x: (a b)\}

\[
\begin{align*}
(a \ b) & \ c & & (a \ b) \\
(a \ ?y) & \ ?z & & (a \ b) \\
\{ y: \ b, z: \ c \} & & & \{ y: \ b, z: \ c \} \\
\end{align*}
\]

True, \{y: b, z: c\}

\[
\begin{align*}
(a \ b) & \ c & & (a \ b) \\
?x & & ?x & & ?x \\
\end{align*}
\]

False

Unification

Unification recursively 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.

**Unifying Variables**

Two relations that contain variables can be unified as well.

\[
\begin{align*}
?x & \ c & & ?x \\
(a \ ?y) & & ?z & & (a \ b) \\
\{ x: (a \ b) \} & & & \{ \text{lookup}(\ ?y) \} \\
\end{align*}
\]

Substituting values for variables may require multiple steps.

**Implementing Unification**

```python
def unify(e, f, env):
e = lookup(e, env)
f = lookup(f, env)
if 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)
```

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

**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.

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

Depth-first search: Each proof approach is explored exhaustively before the next.

```python
def search(clauses, env):
for fact in facts:
    env_head = an environment extending 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 each successful result
```

Limiting depth of the search avoids infinite loops.

Each time a fact is used, its variables are renamed.

Variables are local to facts & queries.

Environment now contains new unifying bindings.

**Addition**

(Demo)