Factbases

Assembly Language for Data Structures

Factbases are to data structures as assembly language is to control flow.

Factbases

A *factbase* is a collection of triples:

```
relation(object,subject).
```

or

```
relation(object,value).
```

Exercise: Create a factbase from the data in the essay "Rectangle Recognition Simplified".

Hint:
A line can be represented as:

```
line(id1,nil).
begin_x(id1,N).
begin_y(id1,M).
end_x(id1,P).
end_y(id1,Q).
```

Where N, M, P and Q are integer values. A line is declared as existing (with subject *nil*). It has a begin point (N,M) and an end-point (P,Q).

Gedanken exercise: Does a line have a width? Does a line have a color? Does a line have other attributes (e.g. dotted, dashed, solid, etc.)? How would the factbase be extended to accomodate such attributes?

Gedanken exercise: Does a rectangle-matcher need to know the color of a line? Can it skip over facts that it finds uninteresting?

Gedanken exercise: Which is simpler - a factbase of atomic facts or a data structure (e.g. a line object that contains points and attributes)? How do you know where to stop adding attributes to lines? If the boss comes by a year later and asks for some

incongruous information about the drawing (e.g. how many lines are there?), is it faster (easier) to use an existing factbase or an existing data structure?

Gedanken exercise: What is an atom? How is it different from a composite item?

Gedanken exercise: If you need to add a feature to a codebase, is it faster to understand the data structures that were used in the codebase, or is it faster to view the list of atoms?

Gedanken exercise: When must we predetermine the structure of data items and use languages that enforce the use of predetermined data structures? Do we have enough computing power to structure data at runtime instead of at compile time? PROLOG and other backtracking / pattern-matching languages make sense of the data and create data "structures" at runtime. Most other languages insist that one define data structures at compile time (which leads to Waterfall thought processes).

Backtracking

Backtracking is exhaustive search.

PROLOG, miniKanren, pattern-matching, etc., can perform searches.

Backtracking[1] makes the searches exhaustive. PROLOG and miniKanren have built-in backtracking.

Pure PROLOG finds all solutions. Practical PROLOG adds mechanisms (e.g. Cut) that allow more efficient searching.

[1] and looping and recursion, etc.

Process

I use the following process: construct *assembly language* primitives as *atoms*, then create structures over the language(s), using, e.g. parsers, as fits the architecture.

See Also

https://www.t3x.org/bits/prolog6.html

OO is an optimization of factbases.

A factbase is an object that contains:

1. data (flat, unordered set of triples)
2. methods (functions that query the data and/or write new data into the factbase).

An OO object is an object that contains:

1. data / state
2. methods.

An OO object is a mini factbase.

The only difference between an OO object and a Factbase, is that the OO object has been optimized to reduce the search space - instead of searching the full factbase, OO methods search/modify only the data that is contained (scoped) within the object.

Factbase methods "skip over" facts that are not need (are uninteresting) to the method.

The above is the simplest form of duck-typing. A method considers only the data triples that it is concerned with.

Duck-typing in OO is the same, but optimized. A group of facts (triples) form a duck-type if a method(s) applies to them.