Prolog Approach

The first "tracer bullet" had at its core Tau Prolog, an opensource Javascript implementation of Prolog. This is an informal diagram of the architecture that was employed:

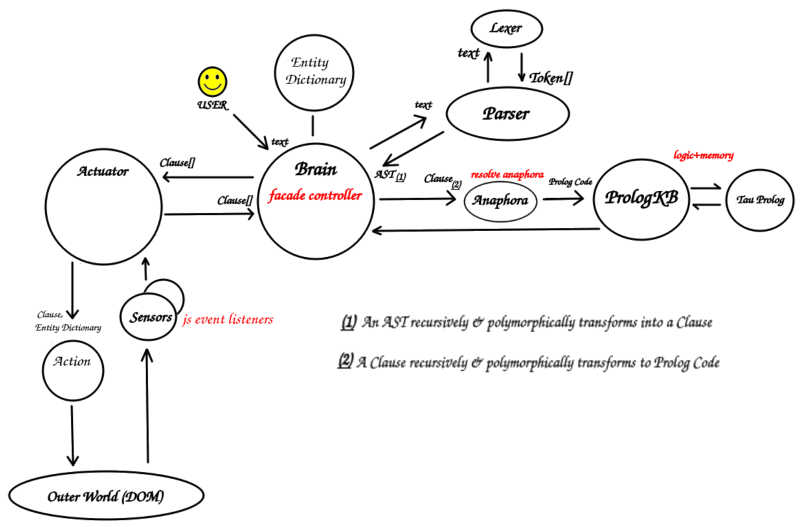

Figure 1: old architecture

How it worked

In summary, the Brain (a facade controller) received "natural language" sentences from the user, and called the Parser, which returned an AST (Abstract Syntax Tree). The AST was recursively converted into a "Clause" (a language-agnostic representation of the equivalent predicate logic), which could in turn be converted into (one or more) valid PrologClauses (see: SWORIER), to be fed to the Prolog engine.

The state of the system was centralized in the Brain, more specifically: in a Prolog object in the Brain, which provided a dynamic knowledge base. New assertions by the user could contain anaphora (references to pre-existing entities) which where resolved by a dedicated class (Anaphora) through a simple algorithm (see: anaphora resolution).

Once new knowledge was asserted in the Brain, the Brain made a "diff" of the old versus the new pieces of knowledge for each single entity, and "pushed" the changes downstream to the Actuator (cf. React Reconciliation). Each entity was identified by a unique global ID, and each ID had a corresponding Javascript object stored in the Entity Dictionary.

The Actuator converted Clauses to Actions, which could run in an asynchronous fashion (waiting for a required but not yet initialized object if need be). These Actions eventually produced the required effect in the underlying javascript environment, like creating a button or changing its color.

Naturalistic Features

The system supported general statements (eg: “every button is red”), through the use of the universal quantifier (in the form of one or more Horn Clauses in Prolog). The system also supported anaphora (“the green button . . . ”).

Support for general statements (or “dynamic sets”), alongside with anaphora, is held to be a highly expressive feature of natural languages, that could be profitably imitated in programming languages (see: Beyond AOP).

Problems of the Prolog Approach

The main problems of this setup were:

- The attributes of an entity had to be declared beforehand and then mapped to the actual attributes on the Javascript object, when they would otherwise be already available in Javascript. This is duplication of effort.
- More generally, the state of the Brain has to be kept in sync with the state of the UI.
- At least on a first approximation, attributes of an entity had to be treated as equal entities in the global namespace. This doesn’t scale.
- Dealing with mutually exclusive values of a property (eg: the color of a button) is tricky, you need a general way of defining and spotting contraddictions in Prolog, which complicates the style of the Prolog clauses. (see: SWORIER)
- It’s also tricky to make sure these be all paraphrases of each other: *“the color of the button is red”* (1), *“the button is red”* (2), *“the background of the style of the button is red”* (3). . .
- More generally, Prolog isn’t the best at building ontologies and dealing with mutating knowledge bases out of the box (see: SWORIER).

No Prolog Approach

An alternative idea was motivated by the following characteristics of the Javascript programming language:

- Prototypes.
- Polymorphism.

- The possibility to extend prototypes (even native ones) **dynamically** (cf. Decorator Pattern).

A high-level overview of the current architecture is this:

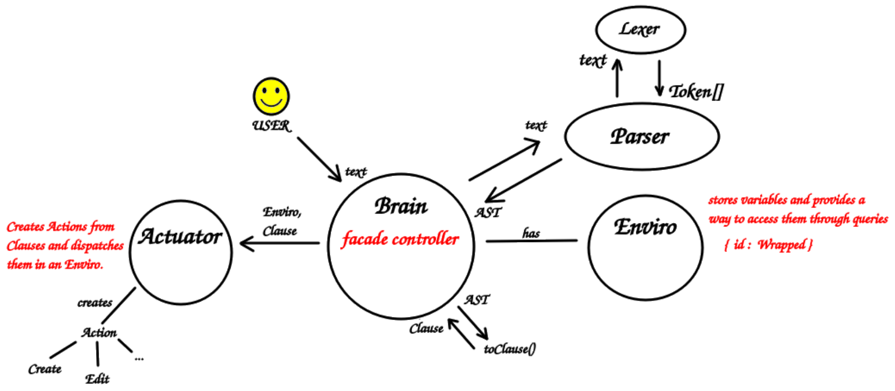

Figure 2: new architecture

The Parser and the AST-to-Clause conversion mechanism remain more or less unchanged. The Actuator is also still present, but is seen in a very different light. The `Prolog` and `Entity Dictionary` components merge into the single `Enviro` object, which holds the variable bindings, and is more akin to the "environment/scope" structure of a conventional programming language (see: Lisperator), although it can be queried through Clauses, for the purpose of resolving anaphora.

The "knowledge base" is now handled in a decentralized manner: each object knows exactly what predicates apply to itself, and the same predicate can result in different (polymorphic) behaviors if applied to different objects.

Concepts

The following mechanisms were introduced:

- Property aliasing (eg: 'style.background' is simply 'color').
- Concept-based grouping of predicates (eg: 'red', 'green', 'blue' refer to the 'color' Concept). A concept, relative to an object, is mapped to a single property or alias (let's start simple).

The addition of these mechanisms was motivated by the need for flexibility in paraphrasing a sentence, as well as the need for the definition of polymorphic behavior across different classes of things (eg: a paragraph object could default to applying colors to its foreground rather than its background).

These ideas were inspired by general notions from ontology modelling (see: ontology 101).

Generalization

Generalization could be handled by defining a Concept or applying a predicate on a Javascript prototype (eg: `HTMLElement.prototype`), instead of on a regular object.

Current State

Custom Concepts

Right now the goal is to define and implement a general syntax for the definition of custom Concepts and predicates.

Bootstrapping

The idea is to create a minimal subset of the language as a natural language sugar-coat over Javascript (cf. Natural Java). It would correspond to how some programmer could "declaim" a line of code, in correct (albeit dull) English. It has a minimum level of flexibility.

Examples:

Minimal Subset	Javascript
background of style of x is red.	x.style.background = 'red'
invoke push on x with 1.	x.push(1)
invoke appendChild on x with y.	x.appendChild(y)

There would then be a set of mechanisms (like Concept-defining statements) to create higher level abstractions, bootstrapped onto the basic js-like subset of the language, eg:

`background of style of any x is color of x.`

or even:

`background of style of any x is its color.`

Bibliography

Tracer Bullets

- https://www.cin.ufpe.br/~cavmj/104The%20Pragmatic%20Programmer,%20From%20Journeyman%20To%20Master%20-%20Andrew%20Hunt,%20David%20Thomas%20-%20Addison%20Wesley%20-%201999.pdf
- offline

SWORIER Paper

- https://arxiv.org/pdf/0711.3419.pdf
- offline

SWORIER Dealing with Contraddictions

- https://arxiv.org/pdf/0711.3419.pdf
- offline

Limitations on Prolog Syntax SWORIER

- https://arxiv.org/pdf/0711.3419.pdf
- offline

Anaphora Resolution Algorithm

Based on a theme-rheme (aka: Topic & Comment) subdivision of a sentence. Entities in theme should be looked up omitting their relation to entities in rheme. And vicecersa, entities in rheme should be looked up omitting their relation to entities in theme. This ensures that only "old" information is used to look up potential anaphoric references. The nexus between theme and rheme entities is new information, and should be omitted in the lookup.

Theme-Rheme

https://en.wikipedia.org/wiki/Topic_and_comment

React Reconciliation

https://reactjs.org/docs/reconciliation.html

Beyond AOP

- https://www2.ccs.neu.edu/research/demeter/papers/oopsla-onward/beyondAOP.pdf
- offline

Natural Java Paper

- https://www.cs.utah.edu/~riloff/pdfs/iui2000.pdf
- offline

Ontology

- https://protege.stanford.edu/publications/ontology_development/ontology101.pdf
- offline

Decorator Pattern

- https://it.wikipedia.org/wiki/Decorator
- https://sourcemaking.com/design_patterns/decorator

Lisperator

https://lisperator.net/pltut/eval1/

Bootstrapping

https://en.wikipedia.org/wiki/Bootstrapping_(compilers)

Tau Prolog

http://tau-prolog.org/

Facade SW Design Pattern

https://en.wikipedia.org/wiki/Facade_pattern