Derivative rules

$$\frac{dk}{dx} = 0 \quad k \text{ is constant}$$

$$\frac{dx}{dx} = 1$$

$$\frac{d}{dx}(ku) = k\frac{du}{dx}$$

$$\frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = \frac{du}{dx}v + u\frac{dv}{dx}$$

Approaches to computing derivatives

Numerical approximation

$$f'(x) \approx \frac{\Delta f}{\Delta x} = \frac{f(x+h) - f(x)}{h}$$

```clojure
(defn derivative [f x h]
  (/ (- (f (+ x h))
        (f x))
     h))
```

Pros

- Works for any function
- simple

Cons

- inexact
- point by point
- far from the formulas we wanted to program

Dual numbers

Extend the real numbers, $\mathbb{R}$, to include ε where

$$\varepsilon^2 = 0.$$

Redefine basic math functions to take dual numbers.

$$f(x + t\varepsilon) = f(x) + tf'(x)\varepsilon$$

Pros

- exact
- (somewhat) simple

Cons

- point by point
- still far from the formulas we wanted to program

Used for training neural networks

Symbolic calculus

Represent symbolic algebraic expressions as data structures.

Pros

- Closest to the formulas
- Exact
- You get the whole expression

Cons

- more complex

Representing symbolic expressions in the computer

Lisp $\Longrightarrow$ Clojure

Lisp was created by John McCarthy, an artificial intelligence researcher, in 1958–1960.

> [M]y own research in artificial intelligence…involved representing information about the world by sentences in a suitable formal language and a reasoning program that would decide what to do by making logical inferences. Representing sentences by list structure seemed appropriate—it still is—and a list processing language also seemed appropriate for programming the operations involved in deduction—and still is…
>
> [T]he point of the exercise was not the differentiation program itself, several of which had already been written, but rather clarification of the operations involved in symbolic computation.
>
> —McCarthy (History of Lisp, 1979)

The symbol datatype

```
'x ;; => x
```

Represents a name

Like a string

But for programming names, not natural language

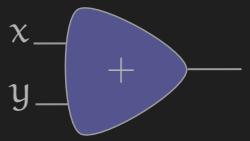
Symbol table—data structure for storing names. If the name is already in the table, just use its index. Otherwise, add new name

Java VM has a string constant pool—identical string literals only create one string object

Clojure uses this as a symbol table (I think?)

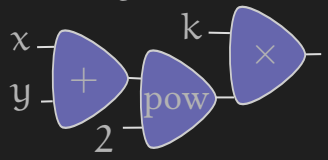
Expressions are trees

$x + y$



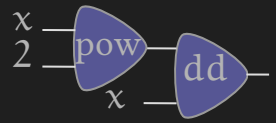

```
(+ x y)
```

$k(x + y)^2$




```
(* k
   (pow (+ x y)
        2))
```

$\frac{d}{dx}x^2$




```
(dd (pow x 2)
    x)
```

Trees in clojure

Nested sequences

First element is the node, rest are children

Shortcut for a tree of symbols:

```
'(dd (pow x n) x) ;; => (dd (pow x n) x)
```

The same as

```
(list 'dd (list 'pow 'x 'n) 'x)
```

Derivative rules in clojure

```
(def deriv-rules
  '{(dd c?k ?v)          0,
    (dd ?x ?x)           1,

    (dd (+ ?u ?v) ?x)    (+ (dd ?u ?x) (dd ?v ?x)),
    (dd (* ?u ?v) ?x)    (+ (* (dd ?u ?x) ?v        )
                            (* ?u         (dd ?v ?x))),
    (dd (pow ?u c?n) ?x) (* (* ?n (pow ?u (- ?n 1)))
                            (dd ?u ?x))})
```

? a name to be matched

c? must match a constant

Strategy

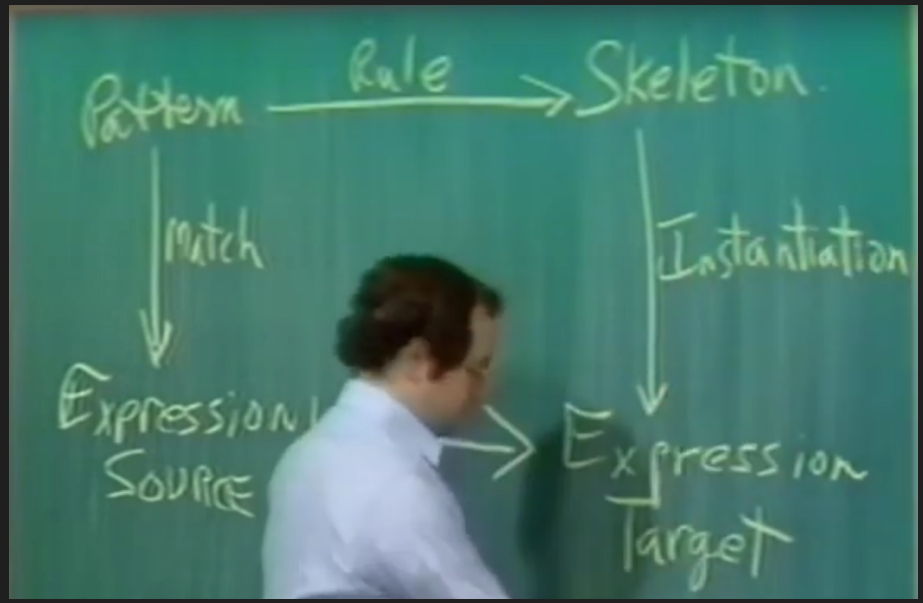
In order to make a system that's robust, it has to be insensitive to small changes…A small change in the problem should lead to a small change in the solution…Instead of solving a particular problem at every level of decomposition…you solve a class of problems that are a neighborhood of the particular problem…by producing a language…in which the solution…is representable. Therefore, when you make small changes to the problem you're trying to solve, you generally have to make only small local changes to the solution

—Gerald Sussman, 6.001 lecture 3b, 1986

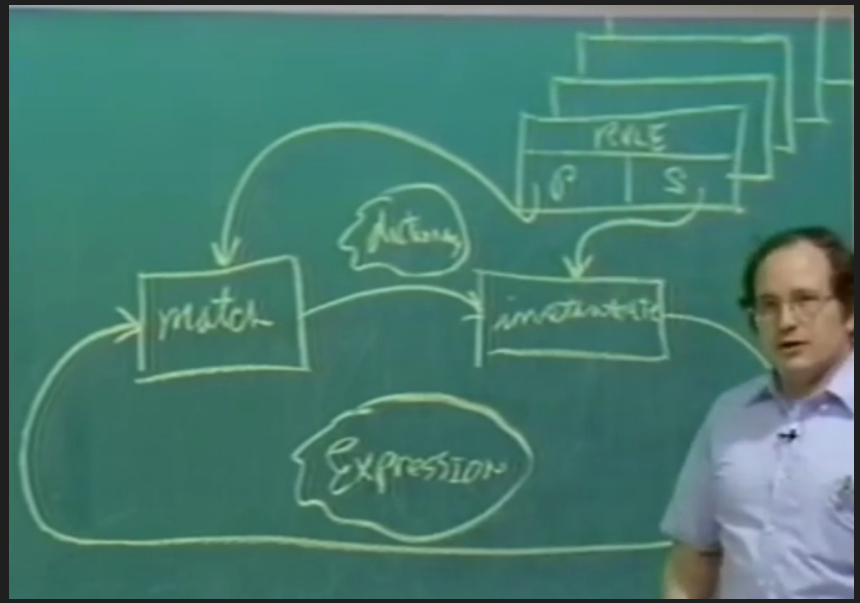
Instead of bringing the rules to the level of the computer by writing a program that is those rules in the computer's language, we're going to bring the computer to the level of us by writing a way by which the computer can understand rules of this sort.

—Gerald Sussman, 6.001 lecture 4a, 1986

Matching and substitution



Dictionaries and instantiation



Pattern vars

?name kind is plain (matches anything)

c?name kind is constant (matches a constant)

```
(defn parse-pattern-var [var]
  (and (symbol? var)
       (let [[prefix name] (.split (str var) "\\?" 2) ]
         (case prefix
           ""  [:plain    name]
           "c" [:constant name]
           false))))
```

Merging dictionaries

Combine results of matching parts of a pattern

Matches must be consistent!

If the same key has different values, the match fails.

```
(def fail ...) ; to represent match failure

(defn merge-unique [& dicts]
  (if (some? #{fail} dicts) fail          ; bail on fail
    (let [fail-if-not= #(if (= %1 %2) %1 fail)
          merged       (apply merge-with fail-if-not= dicts)]
      (if (contains? (vals merged) fail)  ; propagate failure
        fail
        merged))))
```

Matching

```
(defn match [pattern expression]
  (if (seqable? pattern)
    (if (and (seqable? expression)
             (same-count? pattern expression))
      (apply merge-unique (map match pattern expression))
      fail)
    (if-let [[kind name] (parse-pattern-var pattern)]
      (case kind
        :plain    {name expression}
        :constant (if-not (constant? expression) fail
                          {name expression}))
      (if (= pattern expression) {} fail))))
```

Instantiation

```clojure
(defn instantiate [skeleton dict]
  {:pre [(not= dict fail)]}
  (if (seqable? skeleton)
    (map #(instantiate % dict) skeleton)
    (if-let [[_ name] (parse-pattern-var skeleton)]
      (dict name)
      skeleton)))
```

Simplify

```
(defn simplify1 [rules expression]
  (let [dict-skeletons (for [[pattern skeleton] rules]
                         [(match pattern expression) skeleton])
        success?       (fn [[dict _]] (not= dict fail))]
    (if-let [[dict skeleton] (find1 success? dict-skeletons)]
      (instantiate skeleton dict)
      expression)))

(defn simplify-deep1 [rules expression]
  (if (seqable? expression)
    (simplify1 rules
               (map #(simplify-deep1 rules %) expression))
    expression))
```

Simplify as much as possible

Fixed point: $f(x) = x$

```
(defn zip [& cols] (apply map vector cols))
(defn adjacents [col] (zip col (rest col)))
(defn adjacent-filter [bipred? col]
  (filter (fn [[x y]] (bipred? x y))
          (adjacents col)))
(defn adjacent-find1-pair [bipred? col]
  (first (adjacent-filter bipred? col)))

(defn fix [f init]
  (first (adjacent-find1-pair = (iterate f init))))

(defn simplify [rules expression]
  (fix #(simplify-deep1 rules %) expression))
```

Possible improvements

Rules for algebraic simplification

Compile patterns instead of interpreting

How to handle ordering the rules?