A Short and Sweet Dive

Khinshan Khan

Infix Notation

1 + 2

(1 + 2)

(+ 1 2)

3

2 - 1

(2 - 1)

(- 2 1)

1

Benefit of Infix

1 + 2 + 3 + 4 + 5

(+ 1 (+ 2 (+ 3 (+ 4 5))))

(+ 1 2 3 4 5)

Diff Operators

5 + 6 - 7

(5 + 6 - 7)

5 + (6 - 7)

(+ 5 (- 6 7))

4

Lets evaluate some simple ones

```
(+ 9 7)
```

```
(+ 9 (- 7 2))
```

```
(+ (- 9 7) 2)
```

A Hard one

```
(+ 1 (+ 2 3 (- 8 9)) (* (+ 9 10) 6))
```

Analysis

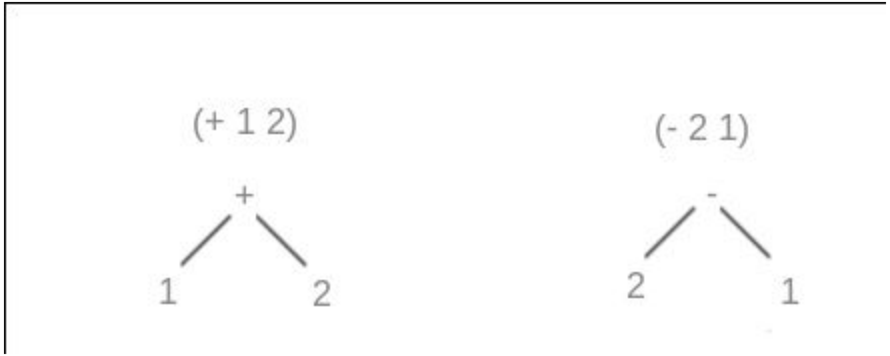
(+ 1 5)

(+ 1 (+ 2 3))

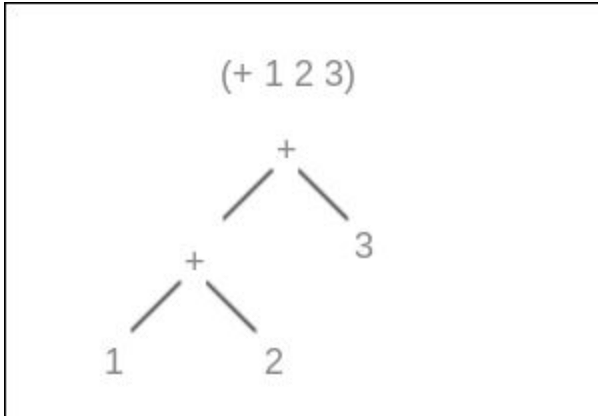
sexp (Symbolic Expressions)

```
type t =
  | Atom of [ `Int of int | `Float of float | `Bool of bool | `String of string
            | `Char of char | `Sym of string | `Tuple of t list]
  | Cons of t list
```

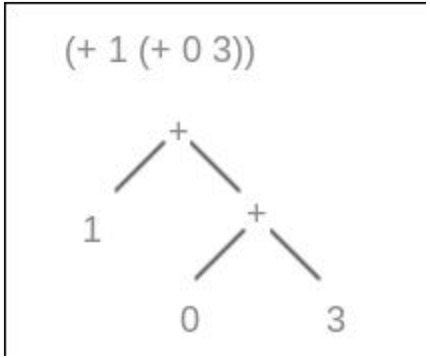
Lets see some trees



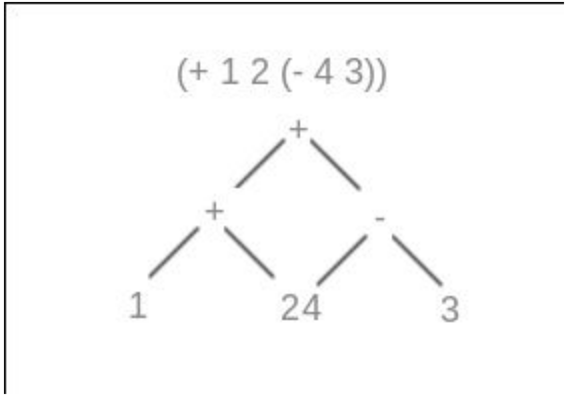
Let's add in one more arg



Lets try an inner cons



One more example



Why is this important?

When we evaluate, we can move in a left to right fashion. The function is usually applied on one or a series of atoms. Whenever we encounter a cons, we evaluate that cons until it becomes an atom, and then we can apply the function onto that. This way, we don't really need to worry about nested parentheses being in the middle of an expression and tackling that first, we'll simply get to it eventually when we move left to right.

The Code

```
let rec eval_sexp = function
  | Sexp.Cons t ->
  begin
    match t with
    | (Sexp.Atom h)::t ->
      let a = List.map atomizer t in
      sym_ops a h
    | _ -> Error._failwith "cons fail"
  end
  | _ -> Error._failwith "sexp fail"
and atomizer = function
  | Sexp.Atom t -> t
  | t -> eval_sexp t
```

Some more code

```
let rec sym_ops a sym =
  let s = sym_extract sym in
  match s with
  | ("+" | "-" | "*" | "/") ->
    begin
      match a with
      | h::t -> List.fold_left (sym_lookup sym) h t
      | _ -> Error._failwith "unaccounted for"
    end
  | _ -> Error._failwith "unaccounted for"
```

Remember sum?

```
let sum n1 n2 =
  match n1, n2 with
  | `Int x, `Int y -> `Int (x+y)
  | `Int x, `Float y -> `Float (float x +. y)
  | `Float x, `Int y -> `Float (x +. float y)
  | `Float x, `Float y -> `Float (x +. y)
```

Generalized for any operator

```
let op f1 f2 n1 n2 =
  match n1, n2 with
  | `Int x, `Int y -> `Int (f1 x y)
  | `Int x, `Float y -> `Float (f2 (float x) y)
  | `Float x, `Int y -> `Float (f2 x (float y))
  | `Float x, `Float y -> `Float (f2 x y)
  | _ -> Error._failwith "invalid num"
```

Print Function

```
let rec sym_ops a sym =

 let s = sym_extract sym in

 match s with

 | "print" -> List.iter print a; sym

 | ("+" | "-" | "*" | "/") ->

  begin

   match a with

   | h::t -> List.fold_left (sym_lookup sym) h t

   | _ -> Error._failwith "unaccounted for"

  end

 | _ -> Error._failwith "unaccounted for"
```