Parsing (simple, functional)

- language = set of sentence
- sentence = token sequence
- recursion, concatenation, disjunction | BNF
- backtracking (hint: matcher)

Matcher form

1. input sequence -> bool
2. grammar -> matcher
3. matcher: fragment -> acceptor -> bool
4. matcher: acceptor -> fragment - bool
5. matcher -> acceptor -> frag -> frag

```
let accept_none = fun frag = None
let accept_none frag = None
let accept_a;;_frag = Some (frag)
let accept_nonempty = function [[]] -> None | F -> Some F
let match_empty acc frag= acc frag
let match_empty = fun acc -> (fun frag) -> (acc frag)
let match_nothing = fun acc -> fun frag -> None
let match_anything _ frag = Some frag
make_matcher -> matcher

let make_ts_matcher ts acc = function
| h :: t -> if h == ts then acc else None
| [] -> None
```

Assume we built a matcher for smaller thing A, B. Now build a matcher for bigger S

- S -> A
- S -> B

```
let make_matcher = function
| [] -> match_empty
| Tx -> make_ts_matcher x
| Or (a, b) -> make_disj_matcher (a, b)
```

```
let make_disj_matcher (a, b) =
    let ma = make_matcher a
    and mb = make_matcher b

    in fun acc -> fun frag ->
    match (ma acc frag) with
    | None -> mb frag acc
    | x -> x

let rec make_or_matcher = function
| [] -> match_nothing
| h :: t ->
    let hm = make_matcher h
    and tm = make_or_matcher t

    in ...(i forgot)

let rec make_concat_matcher (a, b) =
    let ma = make_matcher a
    and mb = make_matcher b

    in fun acc -> ma (mb acc)

3 pieces: [1] ma matched; [2] mb matched; [3] unmatched suffix (acceptable)

let make_appended_matches &s =
    let rec mans = function
    | [] -> match_empty
    | h :: t -> append_matcher (make_matcher h) (mans t)

    in mans &s

let append_matchers m1 m2 acc = m1 (m2 acc)
```

Translation stages in compilers (GCC)

1. Translation
2. Parsing
3. Semantic analysis

- parse tree -> attribute tree
  - check declaration before use
  - check types
  - static checking - check before runtime
- attributed tree
  - more info per node
  - we checked everything static we can

4. Code generation
   - attribute tree -> ASM (assembly)
   - `ps -ef ... ccl`
   - `gcc -S foo.c foo.s` (text file) -> `foo.o` (binary file)
   - `foo.o + libsco -> foo -> ./foo`
5. ASM
6. Linker
7. Loader - load into RAM

IDE

- in main memory:
  - compiler: char -> machine code in ram, jump to code
- Hyper environment (Java, JS)
  - interpreter:
    - compiles into tree., bytecodes
    - interpreter executes these safely + just in time compiler