```
open Printf
open Str

type s_exp =
  | SNum of int
  | SName of string
  | SList of s_exp list
```

(start_char, end_char)

```
let pats = [
● (regexp "[0-9]+", fun str -> ("num", str));
● (regexp "[a-zA-Z][a-zA-Z0-9]*", fun str -> ("name", str));
● (regexp "(", fun str -> ("LPAREN", str));
● (regexp ")", fun str -> ("RPAREN", str));
● (regexp "[ \n\t\r]*", fun str -> ("WS", str));
]
```

Regexp, (string -> (string, string))

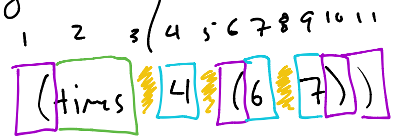
what's the return type?

```
let rec tok str start pats : (string * string) list =
  if String.length str = start then []
  else
    let rec first_match pats =
      match pats with
        | [] -> failwith (sprintf "Tokenizer error at character %d" start)
        | (reg, f)::restpats ->
          if string_match reg str start then
            f (matched_string str)
          else
            first_match restpats
    in
    let (tok_type, content) = first_match pats in
    (tok_type, content)::(tok str (start + (String.length content)) pats);;
```

greedily matches reg w/ str from start

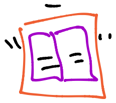
```
let rec str_of_toks toks =
  match toks with
    | [] -> ""
    | (tok_type, str)::rest -> (sprintf "(%s, \"%s\")" tok_type str) ^ "; " ^ (str_of_toks rest);;

let rec str_of_expr e =
  match e with
    | SName(n) -> sprintf "SName(%s)" n
    | SNum(n) -> sprintf "SNum(%d)" n
    | SList(exprs) -> "SList(" ^ (String.concat "," (List.map str_of_expr exprs)) ^ ")"
```

2 steps: 1. Tokenize
2. Parsing



What could it look like for token descriptions to conflict?

- different kinds of names (def vs. let)
- " = "

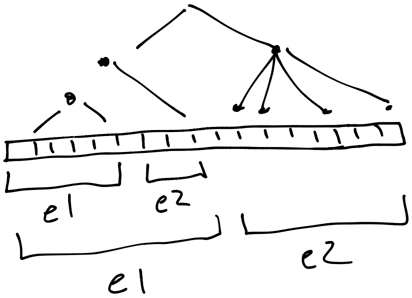


A longest match
B choose order
C Both - but prioritizing order
D neither

```
let rec parse_expr toks : (s_exp option * (string * string) list) = (* returns the expression at the front
                                                                       of toks and the remaining toks *)
```

actual result

tokens assoc'd w/sexp



e := (e e)

```
match toks with
  | [] -> (None, [])
  | ("LPAREN", "(") :: rest ->
      match parse_expr rest with
        | Some(e), remaining ->
            match parse-expr remaining with
              | Some(e2), ("RPAREN", _) :: rem ->
                  Some(SList([e1;e2])), rem
              | _ -> parse error!
            end
        | _ -> parse error!
| NAME  | ("name", s) :: rest -> (Some(SName(s)), rest)
| NUM   | ("int", n) :: rest -> (Some(SNum(n)), rest)
        | ("WS", _) :: rest -> parse_expr rest
        | ("RPAREN", _) :: rest -> (None, toks)
```

Recursive Descent Parsing

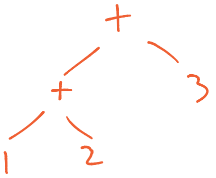
Bottom Up

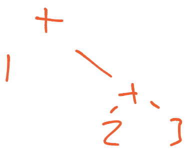
```
let parse (toks : (string * string) list) : s_exp =
  match parse_expr toks with
    | Some(e), [] -> e
    | Some(e), lst -> failwith (sprintf "Extra tokens at end: %s" (str_of_toks lst))
    | None, lst -> failwith (sprintf "Parse error, remaining toks were: %s" (str_of_toks lst))


let () =
  begin
    printf "%s\n" (str_of_toks (tok "(5 6 xyz (11 30))" 0 pats));
    printf "%s\n" (str_of_expr (parse (tok "(5 6 xyz (11 30))" 0 pats)));
  end
```

e := e + e | e * e | n

1 + 2 + 3

```
match toks with
  | ("num", n) :: rest ->   SNum(n), rest
```





e := lhs + rhs | lhs + rhs

lhs := n | (e)

rhs := e

1 - 2 - 3 (2 or -4?)

Left Recursion

```
let rec parse_list toks : (s_exp list option * (string * string) list) =
  let (first_expr, remaining) = parse_expr toks in
  match first_expr with
    | None -> (Some([]), toks)
    | Some(first_expr) ->
      let (rest_list, remaining_after) = parse_list remaining in        (* <expr list> := <expr>             *)
      (match rest_list with                                             (*              |  <expr> <expr list> *)
        | Some(rest_list) ->
          (Some(first_expr::rest_list), remaining_after)
        | None -> None, remaining_after)

and parse_expr toks : (s_exp option * (string * string) list) =
  begin
    let ans = match toks with
      | [] -> failwith "Empty program?"
      | ("WS", _)::rest -> parse_expr rest
      | ("num", n)::rest -> (Some(SNum(int_of_string n)), rest)         (* <expr> := <number>                *)
      | ("name", n)::rest -> (Some(SName(n)), rest)                     (*        |  <name>                  *)
      | ("LPAREN", _)::rest ->                                          (*        |  LPAREN                  *)
        (match parse_list rest with                                     (*            <expr list>            *)
          | Some(exprs), ("RPAREN", _)::rest ->                         (*           RPAREN                  *)
            Some(SList(exprs)), rest
          | _, remaining -> None, remaining)
      | _ -> None, toks
    in
    match ans with
      | Some(e), _ ->
        begin printf "Producing: %s\n" (str_of_expr e); ans end
      | None, _ -> ans
  end

let parse (toks : (string * string) list) : s_exp =
  match parse_expr toks with
    | Some(e), [] -> e
    | Some(e), lst -> failwith (sprintf "Extra tokens at end: %s" (str_of_toks lst))
    | None, lst -> failwith (sprintf "Parse error, remaining toks were: %s" (str_of_toks lst))


let () =
  begin
    printf "%s\n" (str_of_toks (tok "(5 6 xyz (11 30))" 0 pats));
    printf "%s\n" (str_of_expr (parse (tok "(5 6 xyz (11 30))" 0 pats)));
  end
```

parse : string → sexp → expr

.bnf
.grammar
.antlr

parser generator → DO THIS IN PRODUCTION

```
expr ::= NUMBER |
         STRING |
         ( expr * )
```

→ (string → trees)