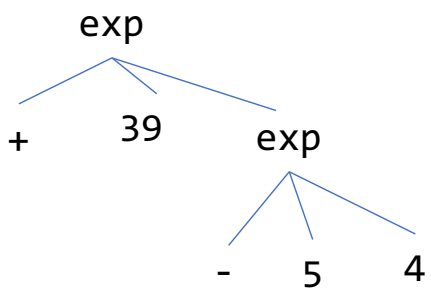
```
"(+ 39 (- 5 4))"
```

```
["(", "+", "39", "(", "-", "5", "4", ")"]
```




```
type Sexp.t =
  | List of Sexp.t list
  | Atom of string
```

Recursive Datatypes and Functions – Templates

```
type 'a lst =
  | Empty
  | Link of 'a * 'a lst
```

```
let rec lf (l : 'a lst) ... : ... =
  match l with
    | Empty -> ... base case ...
    | Link(fst, rest) ->
      ... fst ...
      ... (lf rest ...) ...
```

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

```
let rec tf (t : 'a lst) ... : ... =
  match t with
    | Leaf -> ... base case ...
    | Node(val, left, right) ->
      ... fst ...
      ... (tf left ...) ...
      ... (tf right ...) ...
```

```
type op = Inc | Dec

type expr =
  | ENum of int
  | EOp of op * expr

let rec ef (e : expr) ... : ... =
  match e with
    | ENum(n) -> ... base case ...
    | EOp1(op, arg) ->
      ... op ...
      ... (ef arg ...) ...
```

```
let rec expr_to_instrs (e : expr) =
  match e with
    | ENum(i) -> [sprintf "mov rax, %d" i]
    | EOp(op, e) ->
      let arg_instrs = expr_to_instrs e in
      match op with
        | Inc -> arg_instrs @ ["add rax, 1"]
        | Dec -> arg_instrs @ ["sub rax, 1"]
```

```
open Sexplib.Sexp
module Sexp = Sexplib.Sexp

(*
expr := <number>
     |  (<op> <expr>)
op   := inc | dec
*)

type op =
  | Inc
  | Dec

type expr =
  | ENum of int
  | EOp of op * expr

let rec sexp_to_expr (se : Sexp.t) : expr =
  match se with
    | Atom(s) -> ENum(int_of_string s)
    | List(sexps) ->
      match sexps with
        | [Atom("inc"); arg] -> EOp(Inc, sexp_to_expr arg)
        | [Atom("dec"); arg] -> EOp(Dec, sexp_to_expr arg)
        | _ -> failwith "Parse error"

let parse (s : string) : expr =
  sexp_to_expr (Sexp.of_string s)
```

```
open Printf

let rec expr_to_instrs (e : expr) : string list =
  match e with
    | ENum(i) -> [sprintf "mov rax, %d" i]
    | EOp(op, e) ->
      let arg_exprs = expr_to_instrs e in
      match op with
        | Inc -> arg_exprs @ ["add rax, 1"]
        | Dec -> arg_exprs @ ["sub rax, 1"]

(* Compiles a source program string to an x86 string *)
let compile (program : string) : string =
  let ast = parse program in
  let instrs = expr_to_instrs ast in
  let instrs_str = (String.concat "\n" instrs) in
  sprintf "
section .text
global our_code_starts_here
our_code_starts_here:
  %s
  ret\n" instrs_str;;

let () =
  let input_file = (open_in (Sys.argv.(1))) in
  let input_program = (input_line input_file) in
  let program = (compile input_program) in
  printf "%s\n" program;;
```

"(inc (dec 4))" → EOp(Inc, EOp(Dec, ENum(4))) →

_____

_____

_____

```
open Sexplib.Sexp
module Sexp = Sexplib.Sexp

type op =
  | Inc
  | Dec

type expr =
  | ENum of int
  | EOp of op * expr
  (* Add the cases for ELet and EId! *)



let rec sexp_to_expr (se : Sexp.t) : expr =
  match se with
    | Atom(s) ->




    | List(sexps) ->
      match sexps with
        | [Atom("inc"); arg] -> EOp(Inc, sexp_to_expr arg)
        | [Atom("dec"); arg] -> EOp(Dec, sexp_to_expr arg)
          (* Add the case for ELet! *)



        | _ -> failwith "Parse error"
```

```
(*
expr := <number>
     |  (<op> <expr>)
     |  (let (<name> <expr>) <expr>)
     |  <name>

op   := inc | dec
*)


open Printf

(* FILL the ELet case and anything else for the header! *)

let rec expr_to_instrs
  match e with




    | ENum(i) -> [sprintf "mov eax, %d" i]
    | EOp(op, e) ->
      let arg_exprs = expr_to_instrs e                in
      match op with
        | Inc -> arg_exprs @ ["add rax, 1"]
        | Dec -> arg_exprs @ ["sub rax, 1"]
```

- What assembly code is generated for this input program (on worksheet)?

```
(inc (dec 4))
```

  - A:
```
mov rax, 4
add rax, 1
sub rax, 1
```

  - B:
```
add rax, 1
sub rax, 1
mov rax, 4
```

  - C:
```
mov rax, 4
sub rax, 1
add rax, 1
```

- Which of these is a good definition for the ELet variant of the expr definition?
  - A: `| ELet of expr * expr * expr`
  - B: `| ELet of string * expr`
  - C: `| ELet of string * expr * expr`
  - D: `| ELet of string * int * expr`

- Which of these correctly parses let expressions?

  - A:

```
| [Atom("let"); bind; body]->
  match bind with
    | [name; e] -> ELet(name, e, body)
```

  - B:

```
| [Atom("let"); bind; body]->
  match bind with
    | [name; e] ->
      ELet(name, sexp_to_expr e, sexp_to_expr body)
```

  - C:

```
| [Atom("let"); name; e; body]->
  ELet(name, sexp_to_expr e, sexp_to_expr body)
```

  - D:

```
| [Atom("let"); Atom(name); bind; body]->
   ELet(name, sexp_to_expr e, sexp_to_expr body)
```

  - E:

```
| [Atom("let"); List(bind); body]->
  match bind with
    | [Atom(name); e] ->
      ELet(name, sexp_to_expr e, sexp_to_expr body)
```

- Which of these matches the grammar extended with let and identifiers?

  - A: `(let x 5 x)`
  - B: `(let (x 5) x)`
  - C: `(let x 10)`
  - D: `(let 10 x 10)`
  - E: `(let 10 x)`

- Which of these matches the grammar on the left?

  - A: `(+ 1 2)`
  - B: `(inc 3 3)`
  - C: `(inc (inc 4))`
  - D: `(inc dec 3)`
  - E: `(inc x)`