```
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 eax, %d" i]
    | EOp(op, e) ->
      let arg_exprs = expr_to_instrs e in
      match op with
        | Inc -> arg_exprs @ ["add eax, 1"]
        | Dec -> arg_exprs @ ["sub eax, 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" instructions_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))" $\longrightarrow$ EOp(Inc, EOp(Dec, ENum(4))) $\longrightarrow$

____

____

____

```
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 eax, 1"]
        | Dec -> arg_exprs @ ["sub eax, 1"]
```

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

$$\begin{tikzcd}\texttt{"(inc (dec 4))"} \arrow[r] & \texttt{EOp(Inc, EOp(Dec, ENum(4)))} \arrow[r] & \begin{array}{c}\rule{4cm}{0.4pt}\\ \rule{4cm}{0.4pt}\\ \rule{4cm}{0.4pt}\end{array}\end{tikzcd}$$