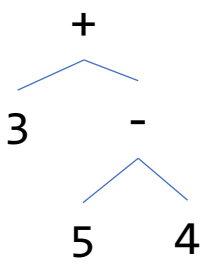
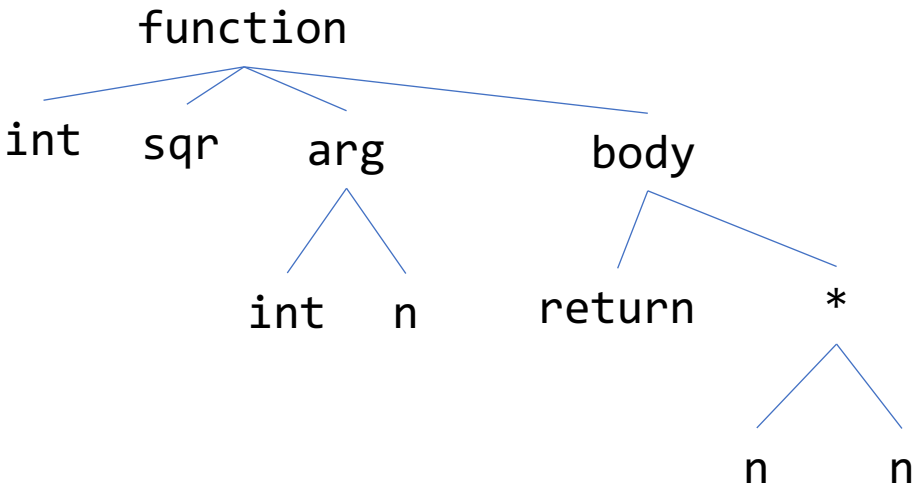
```
3 + (5 - 4)

(+ 3 (- 5 4))

3 5 4 - +
```




```
int sqr(int n) {
  return n * n;
}

(int sqr (int n)
  (return (* n n)))
```




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

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

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

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

```
type expr =
  | ENum of int
  | EOp1 of string * expr
```

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

```
let rec compile (e : expr) : string list =
  match e with
    | ENum(n) ->
      ["mov eax, " ^ (string_of_int n)]
    | EOp1(op, arg) ->
      let argis = compile arg in
      if op == "add1" then
        argis @ ["add eax, 1"]
      else if op == "sub1" then
        argis @ ["sub eax, 1"]
```

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

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

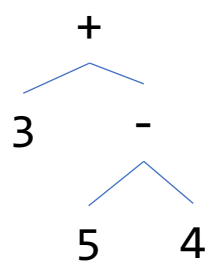
```
let rec lf (l : 'a lst) ... : ... =
  match l with
    | Empty -> ... base case ...
    | Link(fst, rest) ->
      ... fst ...
      ... (lf rest ...) ...
```

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

```
type expr =
  | ENum of int
  | EOp1 of string * expr
```

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

```
3 + (5 - 4)

(+ 3 (- 5 4))

3 5 4 - +
```




```
int sqr(int n) {
  return n * n;
}

(int sqr (int n)
  (return (* n n)))
```