



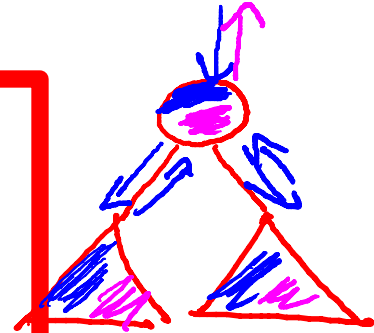
# Compilers

---

## Symbol Tables

- Much of semantic analysis can be expressed as a recursive descent of an AST

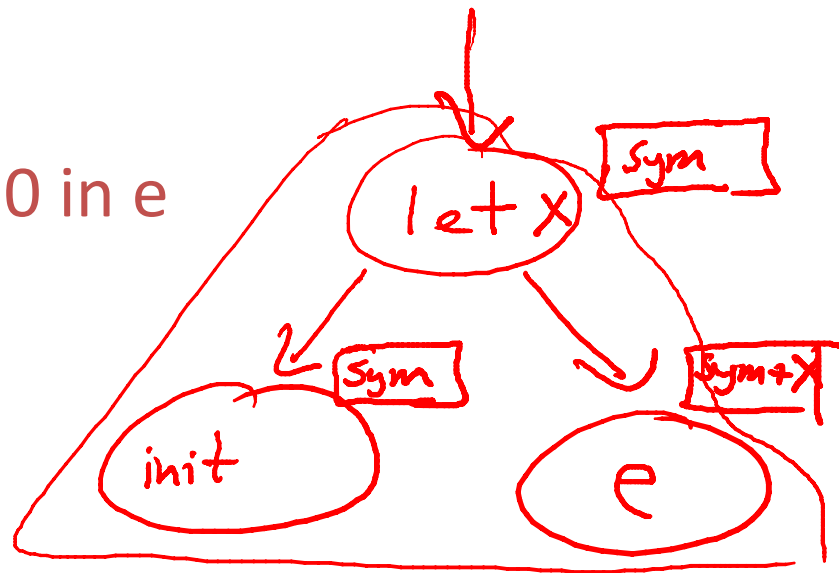
- *Before*: Process an AST node  $n$
- *Recurse*: Process the children of  $n$
- *After*: Finish processing the AST node  $n$



- When performing semantic analysis on a portion of the the AST, we need to know which identifiers are defined

- Example: the scope of **let** bindings is one subtree of the AST:

let x: Int  $\leftarrow$  0 in e



- x** is defined in subtree **e**

- Recall: `let x: Int  $\leftarrow$  0 in e`
- Idea:
  - Before processing `e`, add definition of `x` to current definitions, overriding any other definition of `x`
  - Recurse
  - After processing `e`, remove definition of `x` and restore old definition of `x`
- A symbol table is a data structure that tracks the current bindings of identifiers

- For a simple symbol table we can use a stack
- Operations
  - add\_symbol(x) push **x** and associated info, such as **x**'s type, on the stack
  - find\_symbol(x) search stack, starting from top, for **x**. Return first **x** found or NULL if none found
  - remove\_symbol() pop the stack



- The simple symbol table works for **let**
  - Symbols added one at a time
  - Declarations are perfectly nested

$f(\underline{x}:\text{Int}, \underline{x}:\text{Int}) \{$   
  
 $\}$



- enter\_scope() start a new nested scope
- find\_symbol(x) finds current  $x$  (or null)
- add\_symbol(x) add a symbol  $x$  to the table
- check\_scope(x) true if  $x$  defined in current scope
- exit\_scope() exit current scope

$f(\underline{x:T}, \underline{x:T})$

A symbol table manager is supplied with the project.

- Class names can be used before being defined
- We can't check class names
  - using a symbol table
  - or even in one pass
- Solution
  - Pass 1: Gather all class names
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes
  - Probably more than two