Type Checking, AST traversal

```
abstract class Exp {
  Option<Type> type;
  abstract void typeCheck() throws SemanticException;
  ...
}
class BinExp extends Exp {
  Op op; Exp left, right;

  void typeCheck(C) throws ... {
    left.typeCheck(C); t1 = left.type()
    right.typeCheck(C); t2 = right.type()
    switch (op) {
     case PLUS:
       if (t1.equals(IntType) && t2.equals(IntType)) {
          this.type = Option.of(IntType)
          return;
       } else if (t1.equals(StringType) || t2.equals(StringType)) {
          this.type = Option.of(StringType)
          return;
       } else {
          throw new SemanticException("...")
       }
     ...
    }
  }
```

```
void f() {
  int i, n;
  for (i=0; i<n; i++) {
    boolean b ...
  }
}
```

Γ = ∅

Γ = { i : int, n : int }

Γ = { i : int, n : int, b : boolean }

Γ = { i : int, n : int }

Γ = ∅

scope of a variable

```
class Context {
  Type get(String name) throws Unbound...

  Type put(String name, Type type) throws AlreadyBound...
}
class LocalVar extends Exp {
  String name;
  void typeCheck(Context c) ... {
    c.get(name);
  }
  ...
}
class LocalDecl extend Stmt {
  String name; TypeNode tn;
  void typeCheck(Context c) ... {
    tn.typeCheck(c);
    c.put(name, tn.type())
  }
  ...
}

class Block extends Stmt {
  List<Stmt> stmts;
  void typeCheck(Context c) ... {
    c.push()                          C = c.clone()
    for (Stmt s : stmts) {
      s.typeCheck(c)
    }
    c.pop()                           // discard c
  }
}
```

Stack of hash tables

push	O(1)	put	O(1)
pop	O(1)	get	O(n)

AST Traversal

OO style.

\+ all code for a data type in one place

\- boilerplate for traversal

\- awkward to add new passes

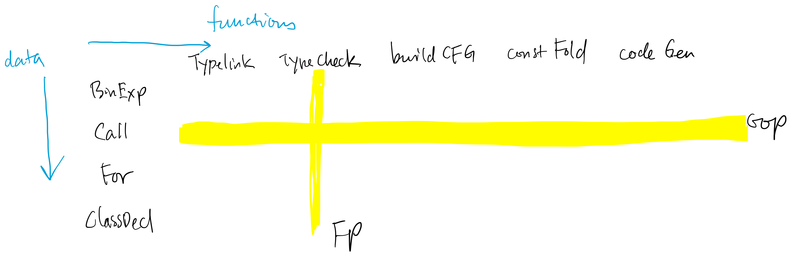
Functional style

```
fun typeCheck(e: Exp, c: Context) :=
  match e with
  | BinExp(op, e1, e2) => typeCheck(e1)
  ...
```

\+ all code for a pass in one place

\- boilerplate for traversal

\- awkward to add new data



- language-based solution
- design pattern

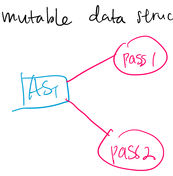
Visitor

```
class Visitor<T> {
  void visitBinExp()
  void visitCall(C)
  ;
}
class Node {
  void accept(Visitor v)
}
```

```
class TypeChecker extends Visitor {
  Context c;
  void visitBinExp() {
    ... // no traversal logic.
  }
}
class BinExp extends Exp {
  void accept(Visitor v) {
    left.accept(v)
    right.accept(v)        } traversal logic.
    v.visitBinExp()
  }
}
```

AST node mutability

mutable data structures



less copying

error prone

immutable data structures