Lecture 19:

- Semantic Analysis (cont)

Announcements:

- HW-3 due Wed.
- HW-4 out
- Proj Part 1 due Mon after Spring Break

Scopes and Environments

`MyPL` uses static (i.e., lexical or block) scoping

- we associate to each block an environment (set of bindings)
- blocks (i.e., environments) can be nested
- bindings are found at a location by looking through all containing blocks

Note:

- a slight abuse of the notion of an "environment"
- where an "environment" is typically all of the containing blocks

For example: "sub environments" created through **`while`** and **`if`** statements:

```
# outer environment        +--------------------+
int x = 1;                 |      x -> int      |
while (x < 10) {           | +----------------+ |
   # sub environment 1     | |                | |
   x = x * 2;              | |                | |
}                          | +----------------+ |
if (x == 10) {             | +----------------+ |
   # sub environment 2     | |   f -> double  | |
   double f = 3.14;        | |                | |
}                          | +----------------+ |
elseif (x > 10) {          | +----------------+ |
  # sub environment 3      | |      ...       | |
  ...                      | +----------------+ |
}                          +--------------------+
```

To find the type bound to a given name ...

1. look at the names defined in the current environment first
2. then the parent environment
3. and so on

In our implementation, the symbol table maintains the environment information

- which is updated as you navigate the AST

Symbol Table

Stores variable state in a "stack" of environments as program is being checked

```python
class SymbolTable:

    # initializes environment ``stack''
    def __init__(self):

    # number of environments (e.g., len(st)
    def __len__(self):

    # print current environments (for debugging)
    def __repr__(self):

    # add new environment (to stack)
    def push_environment(self):

    # remove current environment (from stack)
    def pop_environment(self):

    # add a binding (name -> info) to current environment
    def add(self, name, info):

    # check if name is in the symbol table
    def exists(self, name):

    # check if name is local (in current environment)
    def exists_in_curr_env(self, name):

    # return info for name
    def get(self, name):
```

New environments created/removed when we visit new "blocks"

```python
self.symbol_table.push_environment()
for stmt in stmts:
    stmt.accept(self)
self.symbol_table.pop_environment()
```

The plan:

- basic idea of the type checker implementation for HW-4
- kinds of semantic errors and typing rules for MyPL
- hints on type checker (visitor) implementation

Summary – Things to Know

1. Purpose of the Symbol Table in semantic checking
2. The basic data structures behind a Symbol Table
3. The basic operations of a Symbol Table and what they are used for