Today

- Quiz 6
- Wrap up MyPL Interpreter (HW 7)

Assignments

- HW 6 due
- HW 7 out

Note on Revised Proposals

Add Details ...

- If doing a PL, convince me the app will be complex enough to require a wide range of language features (the goal)
- For all projects, provide some design description (potentially including some sketches of how your program will work/behave)
- Provide resources for learning PL you will use
MyPL Interpretation (cont)

Symbol Table Extensions

- New ability to jump between environments on the stack
- Including inserting and removing from a specific environment

```python
class SymbolTable(object):

    def __init__(self):
        self.scopes = []  # list of {id_name:info}
        self.env_id = None  # current environment in use

    def __get_env_index(self)  # get index in scopes list
    def __environment(self, name)  # get environment for name
    def __str__(self)  # to print symbol table

    # new functions:
    def id_exists_in_env(self, identifier, env_id)
    def get_env_id(self)
    def set_env_id(self, env_id)

    # modified to work from the current environment
    def id_exists(self, identifier)
    def add_id(self, identifier)
    def get_info(self, identifier)
    def set_info(self, identifier, info)
    def push_environment(self)
    def pop_environment(self)
```
Using the new features ...

Creating an environment e1

```python
sym_tbl = SymbolTable()
sym_tbl.push_environment()
e1 = sym_tbl.get_env_id()
sym_tbl.add_id('x')
sym_tbl.set_info('x', 0)
```

Pushing a new environment e2

```python
sym_tbl.push_environment()
e2 = sym_tbl.get_env_id()
sym_tbl.add_id('x')
sym_tbl.set_info('x', 2)
```

Pushing a new environment between e1 and e2

```python
sym_tbl.set_env_id(e1)
sym_tbl.push_environment()
e3 = sym_tbl.get_env_id()
sym_tbl.add_id('x')
sym_tbl.set_info('x', 3)
v = sym_tbl.get_info('x'))
print('x =', v) # prints 3
```

Popping an environment ...

```python
sym_tbl.pop_environment()
v = sym_tbl.get_info('x')
print('x =', v) # prints 1
```
NEW: Disallow variable shadowing in same scope/block ...

Currently allowed in our MyPL implementation:

```plaintext
var x = 0;
struct T
    var y = x + 1;
end
var x = 1;
var t = new T;  # what should t.y be?
```

The new restriction changes your type checker:

```python
def visit_var_decl_stmt(self, var_decl):
    var_decl.var_expr.accept(self)
    exp_type = self.current_type
    var_id = var_decl.var_id
    curr_env = self.sym_table.get_env_id()

    # check that variable isn't already defined
    if self.sym_table.id_exists_in_env(var_id.lexeme, curr_env):
        msg = 'variable already defined in current environment'
        self.__error(msg, var_id)

...
```

This change makes the above variable shadowing fail type checking
Interpreting Struct Types and Struct Objects

The Basic Idea

- A struct object is stored as a dictionary \( \{a_1:v_1, \ldots, a_n:v_n\} \)
- Each struct object is represented by an object id (e.g., 12034501)
- The oid (object id) is the “reference” to the struct object
- The oid is what is stored in variables
  - Implies \( x == y \) compares oid's
  - Also means we are passing struct objects by reference
- Use a heap to store struct object values
  - The heap is represented as a dictionary \( \{oid: \text{struct}_\text{obj}\} \)
  - Struct objects live beyond their “current environment”
  - Since no garbage collection or delete, they live for remaining runtime
- Path expressions dereference the oid ... like the struct name in type checking

Implementation within the Interpreter

1. When you visit a struct declaration:
   - record the definition (AST node) in the symbol table
   - and the current environment id that the definition was made in

```python
self.sym_table.add_id('T')
self.sym_table.set_info('T', [env_id, struct_decl])
```
(2) When you visit a `new` statement:

- Get the struct info from above:
  ```python
  struct_info = self.sym_table.get_info('T')
  ```

- Save the current environment, go to the struct def’s environment
  ```python
  curr_env = self.sym_table.get_env_id()
  self.sym_table.set_env_id(struct_info[0])
  ```

- Create empty struct object, initialize vars, reset environment
  ```python
  struct_obj = {}
  self.push_environment()
  ... initialize struct_obj w/ vars in struct_info[1] ...
  self.pop_environment()
  ... return to starting environment
  ```

- Create the oid, add the struct object to the heap, assign current value
  ```python
  oid = id(struct_obj)
  ... add to the heap ...
  self.current_value = oid
  ```

(3) lvalues and rvalues work in a similar way to the type checker
Interpreting Functions and Function Calls

Handling function declarations is similar to structs ...

- record the definition (AST node) in the symbol table
- and the current environment id that the definition was made in

```python
    self.sym_table.add_id('f')
    self.sym_table.set_info('f', [env_id, fun_decl])
```

Function calls are a bit more involved (but similar) ...

1. get the function information
2. store the current environment
3. compute and store the argument values
4. go to the function declarations environment id
5. add a new environment (for the function to run in)
6. initialize the parameters with the argument values
7. visit the function’s statement list ... (⋆) see handling return statements
8. remove the new environment
9. return to the caller’s environment

⇒ compare this to using a function call stack (in textbook)
Only thing left is handling return statements ...

Return statements are tricky with visitor pattern navigation

- since a return statement abruptly “jumps” out of function/program execution
- we’ll use exceptions (with `try–except`) to force a jump out of the navigation

(1) Create a special type of exception for return:

```python
class ReturnException(Exception): pass  # in mypl_interpreter.py
```

(2) From within the return statement visitor:

```python
def visit_return_stmt(self, return_stmt):  
    ... set the current value to return expression ...  
    raise ReturnException()
```

(3) Add method to start interpreter (instead of calling accept on statement list):

```python
# starts the interpreter (handles return from global scope)  
def run(self, stmt_list):  
    try:  
        stmt_list.accept(self)  
    except ReturnException:  
        pass
```

(4) Finally, when performing function calls, need to catch return exceptions (∗)