## Today

- Intro to Haskell (cont)
- Exam 2
- Finishing the MyPL Interpreter

## Assignments

- HW5 & HW6 due
- Quiz 6
- HW7 out (due Tues, Apr 7)
On to Haskell ...

Some of the major features of Haskell

1. A purely functional language
   - Only “pure” functions
     - In general, functions do not have side effects (do not modify state)
     - some nice features: memoization, recursion
   - Values (variables) are immutable
   - Functions (and operations) always produce entirely new values
   - This is very different than most other PLs

2. Static typing
   - All type checking done at compile time (statically)
   - Employs type inference (unobtrusive—w/out type annotations)

3. “Strong” typing
   - Guarantees a program cannot contain certain type errors
   - Haskell places limits on type conversion (implicit/explicit)
4. Functions are “first-class” objects
   - i.e., used just like any other kind of value in the language
   - e.g., functions can be defined that take functions as parameters (and call them in the function body)
   - Can create new functions during program execution
   - Can store functions in data structures
   - Can pass functions as arguments to other functions
   - Can return functions as values of other functions

5. Lazy evaluation
   - Defer computation until the result is needed
   - One benefit: possible performance gain (no needless computations)
     - e.g., using quicksort, can ask for first (first two, etc.) values, without sorting entire list
   - Another benefit: “infinite” data structures
     - and in particular, the ability to compute with them
     - somewhat similar to iterators (or streams)
   - Another benefit: programmer-defined control structures
     - e.g., short circuit evaluation of if-then-else
     - this means you don’t need special constructs for control flow

6. Expression-oriented
   - All statements return values (e.g., even if statements!)
Exam 2 Overview

Basics

- Open book and notes
- 10% of final grade
- 60 minutes, fixed time, one sitting
- Administered via Blackboard

Topics to Study

- AST creation (e.g., something like quiz 5)
- Grammars wrt associativity and precedence issues (and how to deal with them)
- Scopes, environments, symbol tables
- Static and dynamic scoping
- Static analysis and type checking – understanding rules, simple examples (coding)
- Interpretation – how it works, simple examples (coding)
- Lambda calculus – see quiz 6
MyPL Interpretation (cont)

Symbol Table Features for HW7

- Ability to jump between environments on the stack
- Inserting and removing from a specific environment

```java
public class SymbolTable {
    ...

    // current environment identifier
    public Integer getEnvironmentId() {...}

    // set current environment to environment id
    public void setEnvironmentId(Integer envId) {...}

    ...
}
```
Examples ...

Creating an environment e1

```java
symbolTable.pushEnvironment();
e1: +--------+ <= top
int e1 = symbolTable.getEnvironmentId();
| x -> 0 |
symbolTable.addName("x");
+--------+
symbolTable.setInfo("x", 0);
```

Pushing a new environment e2

```java
symbolTable.pushEnvironment();
e2: +--------+ <= top
int e2 = symbolTable.getEnvironmentId();
| x -> 2 |
symbolTable.addInfo("x");
+--------+
symbolTable.setInfo("x", 2);
e1: +--------+
| x -> /zero.alt1 |
+--------+
```

Pushing a new environment between e1 and e2

```java
symbolTable.setEnvironmentId(e1);
symbolTable.pushEnvironment();
e2: +--------+
int e3 = symbolTable.getEnvironmentId();
| x -> 2 |
+--------+
symbolTable.addInfo("x");
symbolTable.setInfo("x", 3);
e3: +--------+ <= top
| x -> 3 |
+--------+
int v = (int)symbolTable.getInfo("x");
System.out.println(v); // prints 3
| x -> 1 |
+--------+
```

Removing an environment ...

```java
// assuming we are in e3 (from above)
symbolTable.popEnvironment();
e2: +--------+
int v = (int)symbolTable.getInfo("x");
| x -> 2 |
+--------+
System.out.println("x"); // prints 1
e1: +--------+ <= top
| x -> 1 |
+--------+
```
Interpreting user-defined types and objects

The Basic Idea

• An object (instance of type) is stored as a Map \{a1:v1, ..., an:vn\}
• Each object is represented by an object id (oid) (e.g., 12034501)
• The oid 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 objects by reference
• Use a heap to store object values (i.e., attribute-value Map objects)
  – The heap is also represented as a Map \{oid: obj_val\}
  – 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
   - with the current environment id the definition occurred in
     
     // envId = environment id of T, node = T's TypeDeclStmt
     List<Object> typeInfo = List.of(envId, node);
     symbolTable.addInfo("T");
     symbolTable.setInfo("T", typeInfo)

(2) When you visit a new statement:
   - Get the struct info from symbol table:
     
     List<Object> typeInfo = (List<Object>)symbolTable.getInfo("T");

   - Save current environment, go to the type's defining environment
     
     int currEnv = symbolTable.getEnvironmentId();
     symbolTable.setEnvironmentId((int)typeInfo.get(0));

   - Create empty object, initialize vars, reset environment
     
     Map<String, Object> obj = new HashMap<>();
     int oid = System.identityHashCode(obj);
     symbolTable.pushEnvironment();
     ... initialize obj from TypeDeclStmt in typeInfo
     symbolTable.popEnvironment();
     symbolTable.setEnvironmentId(currEnv);
• Add the struct object to the heap, assign current value

    heap.put(oid, obj);

• Note that `heap` is a **new** member of **Interpreter**:

    // mapping of oid's -> objects
    private Map<Integer, Map<String, Object>> heap = new HashMap<>();

(3) lvalues and rvalues work in a similar way to the type checker
Implementing 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

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 (Ch. 10 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-catch) to force a jump out of the navigation

(1) Modified MyPLEException for return control:

```java
public MyPLEException(Object value) {...} // value being return
public boolean isReturnException() {...}
public Object getReturnValue() {...}
```

(2) From within the return statement visitor:

```java
public void visit(ReturnStmt node) throws MyPLEException {
    ... set returnVal to the current value ...
    throw new MyPLEException(returnVal);
}
```

(3) Extend the run function (see below)

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

- can happen when visiting statements in the function’s statement list
- if exception is thrown, check if it is a return exception (see above)
- if so, get return value and set as currentValue
- otherwise throw the exception (since it was an actual error)
public Integer run(StmtList stmtList) throws MyPLEException {
    try {
        // evaluate the stmtList
        stmtList.accept(this);
        // default return
        return 0;
    }
    catch (MyPLEException e) {
        if (!e.isReturnException())
            throw e;
        Object returnVal = e.getReturnValue();
        if (returnVal == null)
            return 0;
        return (Integer) returnVal;
    }
}