Today

- Basic Evaluation (HW 6)
- Programming Paradigms
- Continue Evaluation Next Time (HW 7)

Assignments

- HW5 due
- HW 6 out (due next Tuesday)
- HW 7 out (due Tuesday after Spring break)

Note regarding Projects ...

Every team will need to do a different project (within a section)

- if two projects are the same, I’ll pick one of them
- and the other team will have to come up with a different project
- this might influence the project you pick
Interpretation ... HW 6

Writing a “pure (AST) interpreter” for MyPL:

- Overall similar to type checking
- We’ll again use the visitor pattern ... over AST nodes
- We’ll also use the symbol table ... this time, var -> value
- Instead of a current type, we’ll keep track of the current value

The basic class structure for HW 6

class Interpreter(Visitor):

    def __init__(self):
        self.sym_table = sym_tbl.SymbolTable()  # var_name -> value
        self.current_value = None               # last eval result

... In HW 6, evaluating everything but user-defined functions and struct creation

... HW 7 extends HW 6 to finish the interpreter

Our plan:

- start with basic evaluation (HW 6)
- go over hints for structs and functions (HW 7)
- move on to functional programming while you finish up the interpreter
Homework 6 – Basic Evaluation

Some examples to get you started ...

Statement Lists (same as before ...)

```python
def visit_stmt_list(self, stmt_list):
    self.sym_table.push_environment()
    for stmt in stmt_list.stmts:
        stmt.accept(self)
    self.sym_table.pop_environment()
```

Simple RValues

```python
def visit_simple_rvalue(self, simple_rvalue):
    if simple_rvalue.val.tokentype == token.INTVAL:
        self.current_value = int(simple_rvalue.val.lexeme)
    elif simple_rvalue.val.tokentype == token.FLOATVAL:
        self.current_value = float(simple_rvalue.val.lexeme)
    elif simple_rvalue.val.tokentype == token.BOOLVAL:
        self.current_value = True
        if simple_rvalue.val.lexeme == 'false':
            self.current_value = False
    elif simple_rvalue.val.tokentype == token.STRINGVAL:
        self.current_value = simple_rvalue.val.lexeme
    elif simple_rvalue.val.tokentype == token.NIL:
        self.current_value = None
```
Simple ID RValues

```python
def visit_id_rvalue(self, id_rvalue):
    var_name = id_rvalue.path[0].lexeme
    var_val = self.sym_table.get_info(var_name)
    for path_id in id_rvalue.path[1:]:
        ... handle path expressions ...
    self.current_value = var_val
```

Simple LValues

```python
def visit_lvalue(self, lval):
    identifier = lval.path[0].lexeme
    if len(lval.path) == 1:
        self.sym_table.set_info(identifier, self.current_value)
    else:
        ... handle path expressions ...
```

Variable Declarations

```python
def visit_var_decl_stmt(self, var_decl):
    var_decl.var_expr.accept(self)
    exp_value = self.current_value
    var_name = var_decl.var_id.lexeme
    self.sym_table.add_id(var_decl.var_id.lexeme)
    self.sym_table.set_info(var_decl.var_id.lexeme, exp_value)
```
Handling built-in function calls ...

```python
def visit_call_rvalue(self, call_rvalue):
    # handle built in functions first
    built_ins = ['print', 'length', 'get', 'readi', 'reads',
                 'readf', 'itof', 'itos', 'ftos', 'stoi', 'stof']
    if call_rvalue.fun.lexeme in built_ins:
        self.__built_in_fun_helper(call_rvalue)
    else:
        ... handle user-defined function calls ...
```
Basic structure of the built-in function call helper ...

```python
def __built_in_fun_helper(self, call_rvalue):
    fun_name = call_rvalue.fun.lexeme
    arg_vals = []
    ... evaluate each call argument and store in arg_vals ...
    # check for nil values
    for i, arg in enumerate(arg_vals):
        if arg is None:
            ... report a nil value error ...
    # perform each function
    if fun_name == 'print':
        arg_vals[0] = arg_vals[0].replace(r'\n', '\n')
        print(arg_vals[0], end='')
    elif fun_name == 'length':
        self.current_value = len(arg_vals[0])
    elif fun_name == 'get':
        if 0 <= arg_vals[0] < len(arg_vals[1]):
            self.current_value = arg_vals[1][arg_vals[0]]
        else:
            ... report index out of range error ...
    elif fun_name == 'reads':
        self.current_value = input()
    elif fun_name == 'readi':
        try:
            self.current_value = int(input())
        except ValueError:
            self.__error('bad int value', call_rvalue.fun)
    ... etc ...
```
Imperative vs Declarative Languages

**Imperative Languages:** Programmers specify how to solve the problem and the system carries out the steps

**Declarative Languages:** Programmers specify what the solution should look like and the system determines how best to compute the solution

Logic and Functional languages are generally considered (more) declarative
- compared to object-oriented & procedural languages (C/C++/Python/Java/etc.)
- largely has to do with the underlying models of computation used
From Turing Machines to Imperative Programming

Turing Machines:

1. infinite tape divided into memory cells (one symbol per cell)
2. read/write head that can move left/right one cell at a time
3. state register that stores the current state of the machine
4. state transition table: 
   state + curr head symbol → write symbol + move head + new state

Example: replace a’s with b’s

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↑

s

the tape head (with the machine in state s_i)

Transition Table:

<table>
<thead>
<tr>
<th>Current State</th>
<th>Current Symbol</th>
<th>New Symbol</th>
<th>New State</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_1</td>
<td>Blank</td>
<td>Blank</td>
<td>s_2</td>
<td>Right</td>
</tr>
<tr>
<td>s_2</td>
<td>a</td>
<td>b</td>
<td>s_2</td>
<td>Right</td>
</tr>
<tr>
<td>s_2</td>
<td>b</td>
<td>b</td>
<td>s_2</td>
<td>Right</td>
</tr>
</tbody>
</table>

Turing Machines are imperative ...

• they specify how the computation should be carried out (very low level)
• close to how computers function and to many imperative languages
• tape stores data, state determines what operations occur, etc.
From $\lambda$-calculus to functional programming

Basic idea of $\lambda$-calculus

1. Unnamed, single-variable functions (“$\lambda$ functions”)
   - $\lambda x. x$ takes an $x$ and returns an $x$
   - $\lambda x. (\lambda y. x)$ takes $x$ and returns a function that takes $y$ and returns $x$
   - shorthand for multi-argument functions: $\lambda xy. x$

2. Function application
   - $(\lambda x. x) 0$ applies the identity function to 0 (resulting in 0)
   - $(\lambda x. (\lambda y. x)) a b$ reduces to $a ... (\lambda x. (\lambda y. x)) a b \Rightarrow (\lambda y. a) b \Rightarrow a$

3. Expressions
   - Either a function, an application, or a name (like $x$, $a$, 0, etc.)
   - A function has the form: $\lambda x. e$ where $x$ is a name and $e$ an expression
   - An application has the form: $e_1 e_2$ where both $e$'s are expressions

Computation in $\lambda$-calculus is via function application

- Given a function application such as:
  
  $$(\lambda x. x) y$$

- An application is evaluated by substituting $x$'s in the function body with $y$:
  
  $$(\lambda x. x) y = [y/x] x = y$$

Substitutions give a way to simplify λ-expressions:

\[ T \equiv \lambda xy.x \quad \text{(True)} \]
\[ F \equiv \lambda xy.y \quad \text{(False)} \]

We can use these to define basic logical operators (AND, OR, NOT):

\[ \land \equiv \lambda xy.xy(\lambda uv.v) \equiv \lambda xy.xyF \]
\[ \lor \equiv \lambda xy.x(\lambda uv.u)y \equiv \lambda xy.xTy \]
\[ \neg \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a) \equiv \lambda x.xFT \]

For Example:

\[ T \land T = (\lambda xy.xyF)TT = (\lambda y.TyF)T = TTF = (\lambda xy.x)TF = T \]
\[ T \land F = (\lambda xy.xyF)TF = (\lambda y.TyF)F = TFF = (\lambda xy.x)FF = F \]

Exercise 1: Evaluate the expressions:

\[ F \land T = (\lambda xy.xyF)FT = (\lambda y.FyF)T = FTF = (\lambda xy.y)TF = F \]
\[ F \land F = (\lambda xy.xyF)FF = (\lambda y.FyF)F = FFF = (\lambda xy.x)FF = F \]
\[ F \lor T = (\lambda xy.xTy)FT = FTT = (\lambda xy.y)TT = T \]
\[ \neg T = (\lambda x.xFT)T = TFT = (\lambda xy.x)FT = F \]
\[ \neg F = (\lambda x.xFT)F = FFT = (\lambda xy.y)FT = T \]
You can even express recursion using $\lambda$-calculus ...

$$R \equiv (\lambda y. (\lambda x. y(xx))(\lambda x. y(xx)))$$

- The basic idea is that $R$ calls a function $y$ then “regenerates” itself
- For example, applying $R$ to a function $F$ yields:

$$R_F = (\lambda y. (\lambda x. y(xx))(\lambda x. y(xx)))F$$  \hspace{1cm} (1)

$$= (\lambda x. F(xx))(\lambda x. F(xx))$$  \hspace{1cm} (2)

$$= F((\lambda x. F(xx))(\lambda x. F(xx)))$$  \hspace{1cm} (3)

$$= F(R_F)$$  \hspace{1cm} (4)

$$= F(F(R_F))$$  \hspace{1cm} (5)

$$= \text{ and so on}$$  \hspace{1cm} (6)

- Note in (4) that $F(R_F)$ since $R_F = (\lambda x. F(xx))(\lambda x. F(xx))$ from (2)
- We can stop recursion using conditional functions (similar to Boolean ops)

**Different paradigms, same power ...**:

$\lambda$-calculus and Turing Machines have the same expressive power!