Today

- Interpretation (cont)
- On to Functional Programming: Paradigms

Assignments

- R-5, HW-5 out (due)
- R-6, HW-6 out (due next Thurs)
Interpretation ... HW 6

Writing a “pure interpreter” for MyPL:

- Overall very similar to type checking ... in fact, “easier”
- 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

class Interpreter(Visitor):

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

    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()
Some examples to get you started ...

Simple expressions

```python
def visit_simple_expr(self, simple_expr):
    if simple_expr.term.tokentype == ID:
        var_name = simple_expr.term.lexeme
        var_val = self.sym_table.get_variable_value(var_name)
        self.current_value = var_val
    elif simple_expr.term.tokentype == INT:
        self.current_value = int(simple_expr.term.lexeme)
    elif simple_expr.term.tokentype == BOOL:
        if simple_expr.term.lexeme == "true":
            self.current_value = True
        else:
            self.current_value = False
    elif simple_expr.term.tokentype == STRING:
        self.current_value = simple_expr.term.lexeme
```
Handling print statements

```python
def __write(self, msg):
    sys.stdout.write(str(msg))

def visit_print_stmt(self, print_stmt):
    print_stmt.expr.accept(self)
    if type(self.current_value) == bool:
        if self.current_value == True:
            self.__write("true")
        else:
            self.__write("false")
    else:
        self.__write(self.current_value)
    if print_stmt.is_println:
        self.__write(\n)
```

• requires you to “import sys”
Handling read expressions

```python
def visit_read_expr(self, read_expr):
    val = raw_input(read_expr.msg.lexeme)
    if read_expr.is_read_int:
        try:
            self.current_value = int(val)
        except ValueError:
            self.current_value = 0
    else:
        self.current_value = val
```

- if the value entered isn’t an int, then return 0


**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 \(\rightarrow\) write symbol + move head + new state

Example: replace a’s with b’s

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
a & b & b & a & b & b & \cdots
\end{array}
\]

the infinite tape

\[\uparrow\]

the tape head (with the machine in state \(s_i\))

<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)
   - \((\lambda x.(\lambda y.x))ab\) reduces to \( a \)... \((\lambda x.(\lambda y.x))ab \Rightarrow (\lambda y.a)b \Rightarrow a\)

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: \( e_1e_2 \) 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 $\lambda$-expressions:

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

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

\[
\land \equiv \lambdaxy.xy(\lambdauv.v) \equiv \lambdaxy.xyF \\
\lor \equiv \lambdaxy.x(\lambdauv.u)y \equiv \lambdaxy.xTy \\
\neg \equiv \lambdax.x(\lambdauv.v)(\lambdaab.a) \equiv \lambdax.xFT
\]

For Example:

\[
T \land T = (\lambdaxy.xyF)TT = (\lambday.TyF)T = TTF = (\lambdaxy.x)TF = T \\
T \land F = (\lambdaxy.xyF)TT = (\lambday.TyF)F = TFF = (\lambdaxy.x)FF = F
\]

**Exercise 1:** Evaluate the expressions:

\[
F \land T = (\lambdaxy.xyF)FT = (\lambday.FyF)T = FTF = (\lambdaxy.y)TF = F \\
F \land F = (\lambdaxy.xyF)FF = (\lambday.FyF)F = FFF = (\lambdaxy.y)FF = F \\
F \lor T = (\lambdaxy.xTy)FT = FTT = (\lambdaxy.y)TT = T \\
\neg T = (\lambdax.xFT)T = TFT = (\lambdaxy.x)FT = F \\
\neg F = (\lambdax.xFT)F = FFT = (\lambdaxy.y)FT = T
\]
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:

$$RF = (\lambda y. (\lambda x. y(xx))(\lambda x. y(xx)))F$$

(1)

$$= (\lambda x. F(xx))(\lambda x. F(xx))$$

(2)

$$= F((\lambda x. F(xx))(\lambda x. F(xx)))$$

(3)

$$= F(RF)$$

(4)

$$= F(F(RF))$$

(5)

$$= \text{and so on}$$

(6)

- Note in (4) that $F(RF)$ since $RF = (\lambda x. F(xx))(\lambda x. F(xx))$ 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!