Today

- Exam 1 overview
- ASTs and Visitors
- Type checking

Assignments

- R-4, HW-4 out (due Thurs)
- R-5, HW-5 out (due next Thurs)
Exam 1 Overview

Basics ...
- Closed book, notes, computer, etc.
- 4–5 multipart questions
- Worth 15% of final grade

Reading topics to study ...
- readability, writability, reliability
- simplicity and orthogonality
- why type checking is a good thing
- compiler vs pure interpreter vs hybrid implementation
- r-value vs l-value, aliases
- static vs dynamic bindings
- explicit vs implicit type declarations
- stack vs heap dynamic variables
- static scoping, variable visibility

Exam topics ...
- Anything we’ve discussed in class and on the above reading list is fair game
- lexical analysis (tokens, lexemes, token streams)
- syntactic analysis (CFGs, derivations, LL(1) and rewritings, recursive descent parsing)
- semantic analysis (ASTs, issues with associativity and precedence)
The Visitor Design Pattern

The Visitor pattern allows:

1. functions over an object structure (like an AST) to be decoupled from the object structure itself
2. this means you can have many different functions, without having to change the object structure

• an object (node) in the structure “accepts” a visitor
• which means the node simply passes itself to the visitor
• the visitor then “visits” the node (e.g., prints, evaluates, etc.)
• and then “navigates” to child nodes (repeating the process)
A simple/hypothetical example

In `PrintVisitor`:

```java
public class PrintVisitor {
    public void visit(ValueNode v) {
        System.out.println(v.value);
    }
    public void visit(PlusNode v) {
        v.leftExpr.accept(this);
        System.out.println(" + ");
        v.rightExpr.accept(this);
    }
    public void visit(TimesNode v) {
        v.leftExpr.accept(this);
        System.out.println(" * ");
        v.rightExpr.accept(this);
    }
}
```
Implementing Visitors in Python ...

- The previous examples relied on function overloading
- But a given function can have at most one definition in Python
- Plus no explicit types

The general solution is to define the classes as follows ...

```python
class PrintVisitor(object):
    def __init__(self):
        pass
    def visit_value_node(self, value_node):
        print value_node.value
    def visit_plus_node(self, plus_node):
        plus_node.leftExpr.accept(self)
        print ' + '
        plus_node.rightExpr.accept(self)
    def visit_times_node(self, plus_node):
        plus_node.leftExpr.accept(self)
        print ' * '
        plus_node.rightExpr.accept(self)

class ExprNode(object): pass

class ValueNode(ExprNode):
    def __init__(self):
        self.value = None
    def accept(visitor):
        visitor.visit_value_node(self)
```

class PlusNode(ExprNode):
    def __init__(self):
        self.leftExpr = None
        self.rightExpr = None
    def accept(visitor):
        visitor.visit_plus_node(self)

class TimesNode(ExprNode):
    def __init__(self):
        self.leftExpr = None
        self.rightExpr = None
    def accept(visitor):
        visitor.visit_times_node(self)

How we’ll use the Visitor Pattern:

- In HW4 we use the visitor pattern to print nodes
- In HW5 we’ll use it to do type checking of an AST
- In HW6 we’ll use it to interpret (“run”) an AST

All without having to modify the AST classes!
Type Checking ... HW 5

The goal of type checking is to:

- Detect errors due to type issues, e.g.:
  
  ```
  x = 0 + "1";                  # assuming + requires two numbers
  if 42 <= true then x = 1; end # int and bool comparison
  ```

- Also includes “use before def” errors, e.g.:
  
  ```
  x = 42 + y;                   # y not defined
  if x < 10 then
    y = x + 1;
  else
    x = y;                     # y not defined in this block
  end
  ```

Type errors are based on a set of typing rules

- lots of possibilities, different languages have different rules
- we define a set of “strict” typing rules in HW4 (informally)
Basic Approach

For HW 5 ...

- navigate the AST using the Visitor pattern
- as we go, we infer types and look for type errors
- keep track of last inferred type using a “current type” variable ...

Given this code ...

```plaintext
x = 3;
if x < 25 then
  println(x);
end
```

1. Type check the assignment statement
   (a) Type check the RHS (current_type assigned to INT)
   (b) Since x isn’t defined, assign INT as x’s type

2. Type check the if statement
   (a) Type check the conditional (both operands are of type INT)
   (b) Ensure each statement in the body is type safe

Note we have to keep track of variables and their types!

- we do this using a “symbol table” (var -> type mappings)
Scopes and Environments

An environment stores the state of variables within a scope (block)

- We use static (block) scoping in MyPL
- Blocks can be nested (e.g., if-then or while statements)
- The “visibility” of a variable is roughly the current and “ancestor” blocks

“Sub environments” created through while and if statements:

```plaintext
# global environment
x = 1;
while x < 10 do
    # sub environment 1
    x = x * 2;
end
if x == 10 then
    # sub environment 2
elseif x > 10 then
    # sub environment 3
end
```

To find the state of a variable ...

- look in the current environment first
- then the parent environment
- and so on
Symbol Table

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

class SymbolTable(object):
    def push_environment(self)
    def pop_environment(self)
    def variable_exists(self, var_name)
    def add_variable(self, var_name)
    def set_variable_type(self, var_name, var_type)
    def get_variable_type(self, var_name)
    def set_variable_value(self, var_name, var_value)
    def get_variable_value(self, var_name)

New environments created/removed when we visit statement lists ...

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()

Checking that a print statement is type safe

def visit_print_stmt(self, print_stmt):
    print_stmt.expr.accept(self)

Inferring the type of a read statement

def visit_read_expr(self, read_expr):
    if read_expr.is_read_int:
        self.current_type = token.INT
    else:
        self.current_type = token.STRING
Checking/inferring the type of a simple expression ...

```python
# a simple expression is an id or primitive value
def visit_simple_expr(self, simple_expr):
    term = simple_expr.term
    if term.tokentype == token.ID:
        var_name = term.lexeme
        if self.sym_table.variable_exists(var_name):
            var_type = self.sym_table.get_variable_type(var_name)
            self.current_type = var_type
        else:
            # "use before def" error
            err_msg = term.lexeme + " is undefined"
            raise error.Error(err_msg, term.line, term.column)
    else:
        # primitive value
        self.current_type = term.tokentype
```

Most of the work is done in:

- assignment statements
- complex expressions and complex bool expressions
- handling lists

Sometimes finding the line and column is tricky ...

- I defined a "first_token(self, expr)" helper function
- which finds the line and column of the first token in an expression