Today

- Quiz 5
- Exam 1 overview
- Type checking basics

Assignments

- HW4 due
- HW5 out, due in 2 Tuesdays
Exam Overview

Basics

• closed notes, book, etc.
• 4 multi-part questions
• worth 15% of final grade

Possible Topics

• Anything from class, readings, and assignments is fair game
• Lexical analysis (tokens, lexemes, token streams)
• Syntactic analysis including CFGs, derivations, LL(1) and rewritings, recursive descent parsing
• Semantic analysis including generating ASTs, issues with associativity and precedence, and type checking (today and Tuesday)
• General understanding of MyPL syntax
• Reading specific:
  – static vs dynamic
  – static vs dynamic scoping (name visibility)
  – program blocks and environments
Type Checking

The goal of type checking is to:

• Detect errors due to type issues, e.g.:
  
  ```plaintext
  set x = 0 + "1";   # + requires same-typed operands
  if 42 <= true then # int can't be compared to bool
    set x = 1;
  end
  
  var x = 42 + y;   # y not defined
  if x > 42 then
    var y = x + 1;
  else
    set x = y;       # y not defined in this block
  end
  
  fun int add(x: int, y: int)
    return x + y;
  end
  
  var r1 = add(1, 2, 3); # wrong number of args
  var r2 = add(3.14, 1); # wrong argument types
  ```

• Also includes “use before def” errors, e.g.:

• Function call errors, e.g.:

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 HW5 (more later)
Basic Approach for HW5

- navigate the AST using the Visitor pattern
- during navigation infer types and look for errors

Given this code ...

1: var x: int = 1/zero.alt1;
2: var r = /zero.alt1;
3: while x > /zero.alt1 do
4:   set r = r + x;
5:   set x = x - 1;
6: end

1. For “var x: int = 10”
   - check and infer rhs type, compare against declared type, store x’s type
2. For “var r = 0”, infer rhs type, store as r’s type
3. For “while x > 0 do”
   - ensure x is defined and compatible with 1/zero.alt1 (both ints)
   - check each body statement ...
4. For “set r = r + x”
   - ensure in rhs that r and x are defined and types are compatible for +
   - ensure lhs is defined and result type (int) is compatible with lhs (r) type
5. etc.

Note we have to keep track of variables and their types!
- we do this using a “symbol table” data structure (var -> type mappings)
Scope and Environment

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 name is roughly the current and “ancestor” blocks

“Sub environments” created through while and if statements:

```
# global environment +-------------------+
var x = 1;       | x -> int |
while x < 10 do  | +-------------------+ |
    # sub environment 1 |
    set x = x * 2;    |
end             | +-------------------+ |
if x == 10 then  | +-------------------+ |
    # sub environment 2 |
    var f = 3.14;     |
elseif x > 10 then | +-------------------+ |
    # sub environment 3 |
    ...                |
end             +-------------------+
```

To find the type of a given name ...

1. look in the current environment first
2. then the parent environment
3. and so on

The “symbol table” maintains the environment information
Symbol Table

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

```python
class SymbolTable(object):
    def push_environment(self):
    def pop_environment(self):
    def id_exists(self, identifier):
    def add_id(self, identifier):
    def set_info(self, identifier, info):
    def get_info(self, identifier):
```

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

```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()
```

Record inferred types in a `self.current_type` instance variable

```python
def visit_simple_rvalue(self, simple_rvalue):
    if simple_rvalue.val.tokentype == token.INTVAL:
        self.current_type = token.INTTYPE
    elif simple_rvalue.val.tokentype == token.FLOATVAL:
        self.current_type = token.FLOATTYPE
    elif simple_rvalue.val.tokentype == token.BOOLVAL:
        self.current_type = token.BOOLTYPE
    elif simple_rvalue.val.tokentype == token.STRINGVAL:
        self.current_type = token.STRINGTYPE
    elif simple_rvalue.val.tokentype == token.NIL:
        self.current_type = token.NIL
```
We use the inferred type to help in more complex statements and expressions

```python
def visit_assign_stmt(self, assign_stmt):
    # check and infer rhs type
    assign_stmt.rhs.accept(self)
    rhs_type = self.current_type
    # check and obtain lhs type
    assign_stmt.lhs.accept(self)
    lhs_type = self.current_type
    # error if rhs and lhs types don't match
    if rhs_type != token.NIL and rhs_type != lhs_type:
        msg = 'mismatch type in assignment'
        self.__error(msg, assign_stmt.lhs.path[0])
```

Checking simple lvalues ...

```python
def visit_lvalue(self, lval):
    # check the first id in the path
    var_token = lval.path[0]
    if not self.sym_table.id_exists(var_token.lexeme):
        msg = 'undefined variable "%s"' % var_token.lexeme
        self.__error(msg, var_token)
    self.current_type = self.sym_table.get_info(var_token.lexeme)
    # check if struct for a longer path expression
    ...
```
Complex Expression Example ...

AST fragment for the complex expression $3 + 4 \times 5$

The basic steps:

1. \texttt{visit\_complex\_expr(c1)} is called
2. call \texttt{visit\_complex\_expr(c2)}, which sets \texttt{current\_type} to \texttt{INT}
3. store \texttt{current\_type} in temporary var \texttt{right\_hand\_type}
4. call \texttt{visit\_simple\_expr(s1)}, which sets \texttt{current\_type} to \texttt{INT}
5. check that \texttt{current\_type} == \texttt{right\_hand\_type}
6. set \texttt{current\_type} to \texttt{INT} (really, leave unchanged)

* Note that \texttt{visit()} functions called indirectly through \texttt{accept()}

Exercise: What happens in \texttt{visit\_complex\_expr(c2)} call?
For HW5, you need to finish the visitor implementation ...

- given your parser w/ AST generation
- given the symbol table implementation
- given MyPL “typing rules” (which I’ll give you)

Some aspects are tricky ...

- how to represent struct and function types?
- what to do about nil values?
- what to do about return statements?
- how to report error messages? ... I won’t be picky about this
Understanding Type (Inference) Rules

Basics

- “e : t” states that expression e has type t ... e.g., 42 : int
- \( \Gamma \) denotes the typing context (the environment)
- \( \vdash \) stands for “implies”
- \( \Gamma \vdash e : t \) means it is implied from the given typing context that e has type t

An example typing rule (not from MyPL) ...

\[
\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 + e_2 : t}
\]

“If expressions \( e_1 \) and \( e_2 \) have type \( t \) in the current context, then expression \( e_1 + e_2 \) has the type \( t \) as well

- typing rules allow us to infer the types of complex expressions
- which help us to assign types to names
- and type check statements
Exercise: Go through the MyPL rules and provide examples ... Note that:

1. function types take the form \( f(t_1, \ldots, t_n) \rightarrow t \)
2. struct types take the form \( s\{a_1 : t_1, \ldots, a_n : t_n\} \)
3. \( \text{nil} : \text{nil} \)
4. \( \Gamma, stmt \vdash \ldots \) means “the current context plus the given statement implies”
5. \( \Gamma \vdash f(t_1, \ldots, t_n) \rightarrow t \) is shorthand for \( f \) has the given type (also structs)

An example would be inferring \( x \)'s type in \( \text{var } x = 10 \% (4 / 2) \)

Or given an assignment from a function call, ensure the assignment is well-typed

At this point for HW5 ...

1. Get started on written part
2. Implement basic expression and statement checking
3. Holding off on functions and structs