Today

- Type checking (cont)
- Reminder: Exam 1 on Thur

Assignments

- HW5 out, due next Tuesday
Type Checking (Review)

The goal of type checking for HW5 is to:

- Detect errors due to type issues
- Detect “use before def” errors
- Detect function call errors

Type errors (for HW5) 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
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:

```plaintext
# global environment
var x = 1;
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):
        pass  # implementation is omitted for brevity
    def pop_environment(self):
        pass  # implementation is omitted for brevity
    def id_exists(self, identifier):
        pass  # implementation is omitted for brevity
    def add_id(self, identifier):
        pass  # implementation is omitted for brevity
    def set_info(self, identifier, info):
        pass  # implementation is omitted for brevity
    def get_info(self, identifier):
        pass  # implementation is omitted for brevity
```

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 "{}"'.format(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 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 INT
5. check that \texttt{current\_type} == \texttt{right\_hand\_type}
6. set \texttt{current\_type} to 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., \( 42 : \text{int} \)
- \( \Gamma \) 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) ...

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

“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:

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

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

Or given an assignment from a function call, ensure the assignment is well-typed
Struct Types in MyPL

Four places where struct types are used ...

1. Struct declarations  ... e.g., `struct T var a = ... end`
2. Struct object creation  ... e.g., `var s = new T`
3. Rvalues (path expressions)  ... e.g., `set x = s.a;`
4. Lvalues (path expressions)  ... e.g., `set s.a = v;`

For simplicity, we represent a struct type using a Python dictionary:

- For a struct:
  
  ```
  struct T
  var a1 = 0;
  var a2 = "";
  end
  ```

- We use the type:
  
  ```python
  {'a1':INTTYPE, 'a2':STRINGTYPE}
  ```

During a declaration, add struct type to symbol table ...

```python
self.sym_table.add_id('T')
self.sym_table.set_info('T', {'a1':INTTYPE, 'a2':STRINGTYPE})
```

For object creation, store the struct type name as the type ...

```python
self.sym_table.add_id('s')
self.sym_table.set_info('s', 'T')
```

For rvalues and lvalues, we have two steps ...

```python
   struct_name = self.sym_table.get_info('s')
   struct_type = self.sym_table.get_info(struct_name)
```
Function Types in MyPL

Two places where function types are used

1. Function declarations  
   ... e.g., \texttt{fun int f(x: int) \ldots end}

2. Function calls  
   ... e.g., \texttt{var r = f(0);}

For simplicity we represent function types using nested lists:

- For the function

  \begin{verbatim}
  fun float add(int: x, int: y)
  \hspace{1em} var r = x + y;
  \hspace{1em} return itof(r);
  \hspace{1em} end
  \end{verbatim}

- We use the type

  \textbf{[[INTTYPE, INTTYPE[, FLOATTYPE]]}

To add a function type to the symbol table:

\begin{verbatim}
self.sym_table.add_id('add')
self.sym_table.set_info('add', [[INTTYPE, INTTYPE[, FLOATTYPE]])
\end{verbatim}
To type check the body of a function ...

    # add the function type first
    ... as above but from AST nodes ...

    # add a new environment and a "special" return type
    self.sym_table.push_environment()
    self.add_id('return')
    self.set_info('return', return_type)

    # add parameters to environment
    for param in fun_decl.params:
        param.accept(self)

    # finally, check the body
    fun_decl.stmt_list.accept(self)
    self.sym_table.pop_environment()

    • in the return statement visitor, ensure types align

Function calls ...

    fun_name = call_rvalue.fun.lexeme
    fun_type = self.sym_table.get_info(fun_name)

    • check that function was defined
    • check that parameter and argument types align
    • set the current type to the function return type