Today

- Type checking cont

Assignments

- HW4 due
- HW5 out (due Tues)
- Quiz on Thurs
REVIEW: 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 "above" it in file and in an "ancestor" block

“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 | |      f -> double    |
    var f := 3.14     | +---------------------+      |
elif x > 10 then | +---------------------+      |
    # sub environment 3 | |      ...               |
    ...               | +---------------------+      |
end +---------------------+
```

To find the type of a given name ...
1. look at the names defined in the current environment first
2. then the parent environment
3. and so on

The “symbol table” maintains the environment information
- which is updated as you navigate the AST
Symbol Table

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

```java
public class SymbolTable {
    public void pushEnvironment();
    public void popEnvironment();
    public bool nameExists(String name)
    public bool nameExistsInCurrEnv(String name);
    public void addName(String name);
    public void setInfo(String name, Object info);
    public Object getInfo(String name);
}
```

- Note that actual implementation will be a bit more complicated ...
- We can add and remove environments (via environment ids) within stack

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

```java
public class TypeChecker implements Visitor {
    private SymbolTable symbolTable = new SymbolTable();
    private String currType = null;

    ...

    public void visit(StmtList node) throws MyPLEException {
        symbolTable.pushEnvironment();
        for (Stmt s : node.stmts)
            s.accept(this);
        symbolTable.popEnvironment();
    }

    ...
}
```
Record inferred types in a `currType` instance variable

```java
public void visit(SimpleRValue node) throws MyPLException {
    if (node.val.type() == TokenType.INT_VAL)
        currType = "int";
    else if (node.val.type() == TokenType.DOUBLE_VAL)
        currType = "double";
    else if (node.val.type() == TokenType.BOOL_VAL)
        currType = "bool";
    else if (node.val.type() == TokenType.CHAR_VAL)
        currType = "char";
    else if (node.val.type() == TokenType.STRING_VAL)
        currType = "string";
    else if (node.val.type() == TokenType.NIL)
        currType = "nil";
}
```

We use the inferred type to help in more complex statements and expressions

```java
public void visit(AssignStmt node) throws MyPLException {
    // check and infer rhs type
    node.rhs.accept(this);
    String rhsType = currType;
    // check and obtain lhs type
    node.lhs.accept(this);
    String lhsType = currType;
    // error if rhs and lhs types don't match
    if (!rhsType.equals("nil") && !rhsType.equals(lhsType)) {
        String msg = "mismatched type in assignment";
        error(msg, node.lhs.path.get(0));
    }
}
```
Checking simple lvalues ...

```java
public void visit(LValue node) throws MyPLEException {
    // check the first id in the path
    Token varToken = node.path.get(0);
    if (!symbolTable.nameExists(varToken.lexeme())) {
        String msg = "undefined variable \\
             " + varToken.lexeme() + \\
             ";
        error(msg, varToken);
    }
    // ...
    currType = (String)symbolTable.getInfo(varToken.lexeme());
    // check if structured type for a longer path expression
    ...
}
```
Complex Expression Example ...

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

The basic steps (*):

1. visit(e1) is called
2. call visit(s1), which (eventually) sets currType to "int"
3. store currType in temporary var rhsType
4. call visit(e2), which (eventually) sets currType to "int"
5. check that currType == rhsType (less nil types)
6. check that operator type is compatible with rhsType
7. set currType to "int" (really, leave unchanged)

* Note that visit() functions called indirectly through accept()
Exercise: What happens in visit(e2) call?

For HW5, you need to finish the visitor implementation ...

- given your parser w/ AST generation
- given the symbol table implementation (provided)
- given MyPL “typing rules” (also provided)

Some aspects are tricky ...

- what to do about nil values? ... type rules
- what to do about return statements? .. type rules
- how to represent structured type and function types? ... more soon
- variable “shadowing” ... more soon
- how to report error messages? ... We 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) ...

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

“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:

- function types take the form $f(t_1, \ldots, t_n) \rightarrow t$
- struct types take the form $s\{a_1 : t_1, \ldots, a_n : t_n\}$
- nil : 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

Recall: Some aspects are tricky ...

- how to represent structured type and function types?
- variable “shadowing”