Lecture 20:

- Semantic Analysis (cont)

Announcements:

- HW-3 due
- HW-4 out
- Proj Part 1 due Mon after Spring Break
- Quiz 5 Fri (visitor pattern, type checking basics)
The SemanticChecker implements the visitor pattern

- includes a symbol table and the “current” inferred type (as DataType)

Basic layout:

```python
class SemanticChecker(Visitor):
    def __init__(self):
        self.structs = {}  # struct name -> StructDef
        self.functions = {}  # fun name -> FunDef
        self.symbol_table = SymbolTable()
        self.curr_type = None  # AST DataType object

... additional helpers ...
```

Inferred types recorded in curr_type member variable

Recall the AST DataType class:

```python
@dataclass
class DataType:
    is_array: bool
    type_name: Token
    def accept(self, visitor):
        visitor.visit_data_type(self)
```

Example for simple (literal) rvalues:

```python
def visit_simple_rvalue(self, simple_rvalue):
    val = simple_rvalue.value
    line = val.line
    col = val.column
    type_token = None
    if val.token_type == TokenType.INT_VAL:
        type_token = Token(TokenType.INT_TYPE, 'int', line, col)
    elif val.token_type == TokenType.DOUBLE_VAL:
        type_token = Token(TokenType.DOUBLE_TYPE, 'double', line, col)
    elif val.token_type == TokenType.STRING_VAL:
        type_token = Token(TokenType.STRING_TYPE, 'string', line, col)
    elif val.token_type == TokenType.BOOL_VAL:
        type_token = Token(TokenType.BOOL_TYPE, 'bool', line, col)
    elif val.token_type == TokenType.NULL_VAL:
        type_token = Token(TokenType VOID_TYPE, 'void', line, col)
    self.curr_type = DataType(False, type_token)
```
Inferred types help check more complex statements and expressions

For example, part of expression checking:

```python
def visit_expr(self, expr):
    # check the first term
    expr.first.accept(self)
    # record the lhs type
    lhs_type = self.curr_type
    # check if more to expression
    if expr.op:
        # check rest of expression
        expr.rest.accept(self)
        # record the rhs type
        rhs_type = self.curr_type

        # ... check lhs and rhs against op, set new curr_type ...
    
    # check not operation
    if expr.not_op:
        # ... ensure bool type ...
```
More Involved Expression Example ...

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

High-level overview of the basic steps:

1. **accept** called on e1, which calls **visit** function
2. **accept** called on s1, calls **visit**, eventually sets **curr_type** (as int)
3. store **curr_type** in temporary **lhs_type**
4. **accept** called on e2, calls **visit**, eventually sets **curr_type** (as int)
5. store **curr_type** in temporary **rhs_type**
6. check that **rhs_type** and **rhs_type** are compatible with operator
7. check if the expression is logically negated (requires **bool** expression)
8. update **curr_type** to new inferred type (in this case, **int**)

CPSC 326 (Spring 2024)  S. Bowers  4 of 5
Type Inference Rules

Purpose

• like grammar rules, give rules for inferring types
• the “legal” inferences (from which implies type errors)
• not all semantic errors captured (e.g., shadowing, use-before-def)

Basics

• “e : t” states that expression e has type t ... e.g., 42 : int
• Γ denotes the typing context (the environment)
• ⊢ stands for “implies”
• Γ ⊢ 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

For MyPL: www.cs.gonzaga.edu/bowers/courses/cpsc326/type-rules.pdf