

CS 432
Fall 2024

Mike Lam, Professor

Γ
 τ λ

```
public class WhileLoopCounter extends  
    private int numWhileLoops = 0;  
    @Override  
    public void preVisit(ASTWhileLoop  
    {  
        numWhileLoops++;  
    }  
    @Override  
    public void postVisit(ASTProgram  
    {  
        System.out.println("Number of  
            numWhileLoops);  
    }  
}
```

Type Systems and the Visitor Design Pattern

General theme

- *Pattern matching* over a tree is very useful in compilers
 - Debug output (P2)
 - **Type checking & other static analysis (P3)**
 - Code generation (P4)
 - Instruction selection
- Theory and practice
 - **Type systems** describe correctly-typed program trees
 - **Visitor design pattern** allows clean implementation in a non-functional language
 - Generalization of **tree traversal** (CS 240 approach)

Types

- A **type** is an abstract category characterizing a range of data values
 - Base types: integer, character, boolean, floating-point
 - Enumerated types (finite list of constants)
 - Pointer types (“address of X”)
 - Array or list types (“list of X”)
 - Compound/record types (named collections of other types)
 - Function types: (type1, type2, type3) → type4

Not all of these will be necessary for Decaf

Type Systems

- A **type system** is a set of **type rules**
 - Rules: valid types, type compatibility, and how values can be used
 - A **type judgment** is an assertion that expression **x** has type **t**
 - Written as “**x : t**” (e.g., “**3 : int**” and “**true : bool**”)
 - Often requires the context of a **type environment** (i.e., symbol table)
 - “**Strongly typed**” if every expression can be assigned an unambiguous type
 - “**Statically typed**” if all types can be assigned at compile time
 - “**Dynamically typed**” if some types can only be discovered at runtime
- Benefits of a robust type system
 - Earlier error detection
 - Better documentation
 - Increased modularization

Formal Type Theory

- A **formal type system** is a set of **type rules**
 - Each rule has a **name**, zero or more **premises** (above the line), and a **conclusion** (below the line)
 - Premises and conclusions are **type judgments** ($\mathbf{A} \vdash \mathbf{x} : \mathbf{t}$)
 - “ \vdash ” is a ternary operator connecting **type environments**, expressions, and types
 - Omit type for statements (“ $\mathbf{A} \vdash \mathbf{s}$ ” means “s is well-typed in environment A”)

$$\text{TDec} \frac{}{\vdash \text{DEC} : \mathbf{int}}$$

$$\text{TTrue} \frac{}{\vdash \text{true} : \mathbf{bool}}$$

$$\text{TLoc} \frac{\text{ID} : \tau \in \Gamma}{\Gamma \vdash \text{ID} : \tau}$$

$$\text{TAdd} \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 \text{ '+' } e_2 : \mathbf{int}}$$

$$\text{TAssign} \frac{\Gamma \vdash \text{ID} : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{ID} \text{ '=' } e \text{ ';'}}$$

$$\text{TFuncCall} \frac{\text{ID} : (\tau_1, \tau_2, \dots, \tau_n) \rightarrow \tau_r \in \Gamma \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \text{ID} \text{ '(' } e_1, e_2, \dots, e_n \text{ ')' : } \tau_r}$$

Formal Type Theory

- **Type proofs** consist of **composing** multiple type rules
 - Apply **rule instances** recursively to form **proof trees**
 - **Type environments** (e.g., symbol tables) provide type context
 - Proof structure is based on the AST structure (“**syntax-directed**”)
 - **Curry-Howard correspondence** (“proofs as programs”)

$$\begin{array}{c}
 \text{TFuncCall} \frac{\text{foo} : (\text{int}) \rightarrow \text{int} \in A}{\text{TFuncCall}} \quad \frac{\text{y} : \text{int} \in A}{A \vdash \text{y} : \text{int}} \text{TVar} \quad \text{TDec} \\
 \frac{\text{TFuncCall} \quad \frac{\text{A} \vdash \text{foo}(\text{y}) : \text{int} \quad \text{A} \vdash 1 : \text{int}}{\text{TAdd}}}{\text{A} \vdash \text{foo}(\text{y}) + 1 : \text{int}} \text{TAssign} \\
 \frac{\text{x} : \text{int} \in A \quad \text{A} \vdash \text{foo}(\text{y}) + 1 : \text{int}}{\text{A} \vdash \text{x} = \text{foo}(\text{y}) + 1}
 \end{array}$$

$$A = \{ \text{foo} : \text{int} \rightarrow \text{int}, \text{x} : \text{int}, \text{y} : \text{int} \}$$

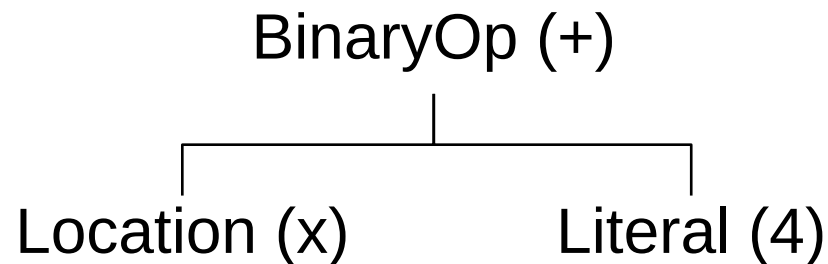
Formal Type Theory

- Is the following Decaf expression well-typed in the given environment?
 - If so, what is its type?

$x + 4$

$A = \{ x : \text{int} \}$

AST:



Formal Type Theory

$$\text{TLoc} \frac{\text{ID} : \tau \in \Gamma}{\Gamma \vdash \text{ID} : \tau}$$

$$\text{TDec} \frac{}{\vdash \text{DEC} : \text{int}}$$

$$\text{TAdd} \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ '+' } e_2 : \text{int}}$$

$$\frac{\text{TLoc} \frac{x : \text{int} \in A}{A \vdash x : \text{int}} \quad \frac{}{A \vdash 4 : \text{int}} \text{TDec}}{A \vdash x + 4 : \text{int}} \text{TAdd}$$

$$A = \{ x : \text{int} \}$$

P3: Static Analysis

- Language and project specifications provide rules to check at each type of AST node while traversing the AST
 - E.g., at WhileLoop, make sure the conditional has a boolean type
 - E.g., at BinaryOp, if it's an add make sure both operands are integers (or if it's an equals make sure the operand types match)

$$\text{TDec} \frac{}{\vdash \text{DEC} : \text{int}} \quad \text{THex} \frac{}{\vdash \text{HEX} : \text{int}} \quad \text{TStr} \frac{}{\vdash \text{STR} : \text{str}}$$

$$\text{TTrue} \frac{}{\vdash \text{true} : \text{bool}} \quad \text{TFalse} \frac{}{\vdash \text{false} : \text{bool}} \quad \text{TSubExpr} \frac{\Gamma \vdash e : t}{\Gamma \vdash \text{'(' } e \text{')} : t}$$

$$\text{TAdd} \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{'+' } e_2 : \text{int}} \quad (\text{similar for TSub } (-), \text{TMul } (*), \text{TDiv } (/) \text{ and TMod } (\%))$$

$$\text{TEq} \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \text{'==' } e_2 : \text{bool}} \quad (\text{similar for TNeq } (!=)) \quad \text{TWhile} \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash b}{\Gamma \vdash \text{while '(' } e \text{') } b}$$

P3: Static Analysis

- General idea: traverse AST and reject invalid programs
 - Need to traverse the tree multiple times
 - Print debug output
 - Build symbol tables
 - Perform type checking
 - Later compiler passes
 - We could write the tree traversal code every time, but that would be tedious w/ a lot of code duplication
 - Software engineering provides a better way in the form of the **visitor design pattern**

A brief digression ...

- What are "design patterns"?

(remember them from CS 345?)

A brief digression ...

- What are "design patterns"?
 - A reusable "template" or "pattern" that solves a common design problem
 - "Tried and true" solutions
 - Main reference: Design Patterns: Elements of Reusable Object-Oriented Software
 - "Gang of Four:" Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

*(excerpt scanned as
PDF in Canvas)*



Common Design Patterns

- **Adapter** – Converts one interface into another
- **Factory** – Allows clients to create objects without specifying a concrete class
- **Flyweight** – Manages large numbers of similar objects efficiently via sharing
- **Iterator** – Provides sequential access to a collection
- **Monitor** – Ensures mutually-exclusive access to member variables
- **Null Object** – Prevents null pointer dereferences by providing "default" object
- **Observer** – Track and update multiple dependents automatically on events
- **Singleton** – Provides global access to a single instance object
- **Strategy** – Encapsulate interchangeable algorithms
- **Thread Pool** – Manages allocation of available resources to queued tasks
- **Visitor** – Provides an iterator over a (usually recursive) structure

Design Patterns

- Pros
 - Faster development
 - More robust code (if implemented properly)
 - More readable code (for those familiar with the patterns)
 - Improved maintainability
- Cons
 - Increased abstraction
 - Increased complexity
 - Philosophical: Suggests language deficiencies
 - Consider a more appropriate language if many patterns are needed

Visitor Pattern

- **Visitor design pattern**: don't mix data and actions
 - Separates the **representation** of an object structure from the definition of **operations** on that structure
 - Keeps data class definitions cleaner
 - Allows the creation of new operations without modifying all data classes
 - Solves a general issue with most OO languages
 - Lack of **multiple dispatch** (choosing a concrete method based on two objects' data types)
 - NOTE: This is stronger than single dispatch + overloading alone
 - Less useful in functional languages with more robust pattern matching
 - In C, we'll handle this manually with function pointers

General Form

- Data: **AbstractElement** (ASTNode)
 - ConcreteElement1 (Program)
 - ConcreteElement2 (VarDecl)
 - ConcreteElement3 (FuncDecl)
 - (etc.)
 - All elements define "Accept()" method that recursively calls "Accept()" on any child nodes (this is the actual tree traversal code!)
- Actions: **AbstractVisitor** (NodeVisitor)
 - ConcreteVisitor1 (PrintVisitor)
 - ConcreteVisitor2 (SetParentVisitor)
 - ConcreteVisitor3 (CalcDepthVisitor)
 - (etc.)
 - All visitors have "previsit_X()" and "postvisit_X()" methods for each element type (i.e., AST node type)

Benefits

- Adding new operations is easy
 - Just create a new concrete visitor
 - In our compiler, create a new `NodeVisitor` struct
- No wasted space for state in data classes
 - Just maintain state in the visitors (e.g, `AnalysisData`)
 - In our compiler, we will make a few exceptions for state that is shared across many visitors (e.g., symbol tables)
 - These are stored as “attributes” in the AST

Drawbacks

- Adding new data classes is hard
 - This won't matter for us, because our AST types are dictated by the grammar and won't change
- Breaks encapsulation for data members
 - Visitors often need access to all data members
 - This is ok for us, because our data objects are just structs anyway (all data is public)

Minor Modifications

- "Accept()" → "traverse()"
- "Visit()" → "previsit_X()" and "postvisit_X()"
 - previsit_X() allows preorder operations
 - postvisit_X() allows postorder operations
 - Also, a single inorder method: `invisit_binaryop()`
- NodeVisitor struct
 - Function pointers for all visitor methods
 - CS 430 note: this is a manual implementation of **virtual method tables!**
 - No type checking – be careful when building the struct!
 - NULL pointers for unneeded methods
 - Allows subclasses to define only the relevant visit methods

Visitor example

```
typedef struct {
    int loop_count;
} CountLoopsData;

#define DATA ((CountLoopsData*)(visitor->data))

void CountLoopsVisitor_previsit_program
    (NodeVisitor* visitor, ASTNode* node)
{
    DATA->loop_count = 0;
}

void CountLoopsVisitor_previsit_whileloop
    (NodeVisitor* visitor, ASTNode* node)
{
    DATA->loop_count++;
}

void CountLoopsVisitor_postvisit_program
    (NodeVisitor* visitor, ASTNode* node)
{
    printf("%d\n", DATA->loop_count);
}
```

Visitor example

```
NodeVisitor* CountLoopsVisitor_new ()
{
    NodeVisitor* v = NodeVisitor_new();
    v->data = malloc(sizeof(CountLoopsData));
    v->dtor = free;
    v->previsit_program    = CountLoopsVisitor_previsit_program;
    v->previsit_whileloop = CountLoopsVisitor_previsit_whileloop;
    v->postvisit_program   = CountLoopsVisitor_postvisit_program;
    return v;
}
```

In main.c:

```
NodeVisitor_traverse_and_free(CountLoopsVisitor_new(), tree);
```

Decaf Project

- Project 2 (parser)
 - NodeVisitor (blank)
 - PrintVisitor
 - GenerateASTGraph
 - SetParentVisitor
 - CalcDepthVisitor
- Project 3 (analysis)
 - PrintSymbolsVisitor
 - BuildSymbolTablesVisitor
 - Your static analysis (custom NodeVisitor)
- Project 4 (code gen)
 - Your code generator (custom NodeVisitor)