public class WhileLoopCounter extends DefaultASTVisitor {
    private int numWhileLoops = 0;
    @Override
    public void preVisit(ASTWhileLoop node)
    {
        numWhileLoops++;
    }
    @Override
    public void postVisit(ASTProgram node)
    {
        System.out.println("Number of while loops = "+numWhileLoops);
    }
}
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)
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., “$3 : \text{int}$” 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
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 \((A ⊢ x : t)\)
- \(\vdash\) is a ternary operator connecting type environments, expressions, and types
- Omit type for statements ("A ⊢ s" means "s is well-typed in environment A")

\[
\begin{align*}
\text{TDec} & \quad \vdash \text{DEC} : \text{int} \\
\text{TTrue} & \quad \vdash \text{true} : \text{bool} \\
\text{TLoc} & \quad \Gamma \vdash \text{ID} : \tau \in \Gamma \\
\text{TAdd} & \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
& \quad \Gamma \vdash e_1 + e_2 : \text{int} \\
\text{TAssign} & \quad \Gamma \vdash \text{ID} : \tau \quad \Gamma \vdash e : \tau \\
& \quad \Gamma \vdash \text{ID} \mathbin{=} e \mathbin{;} \\
\text{TFuncCall} & \quad \text{ID} : (\tau_1, \tau_2, \ldots, \tau_n) \to \tau_r \in \Gamma \\
& \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \\
& \quad \Gamma \vdash \text{ID} \mathbin{\langle} e_1, e_2, \ldots, e_n \mathbin{\rangle} : \tau_r
\end{align*}
\]
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")

\[
\begin{align*}
    A & = \{ \text{foo : int → int, x : int, y : int} \} \\
    A \vdash x = \text{foo}(y) + 1 & \quad \text{TAssign}
\end{align*}
\]
Is the following Decaf expression well-typed in the given environment?

- If so, what is its type?

\[
A = \{ x : \text{int} \}
\]

\[
x + 4
\]

**AST:**

- BinaryOp (+)
- Location (x)
- Literal (4)
Formal Type Theory

\[
\begin{align*}
\text{TLoc} & \quad \frac{\text{ID} : \tau \in \Gamma}{\Gamma \vdash \text{ID} : \tau} \\
\text{TDec} & \quad \frac{\vdash \text{DEC} : \text{int}}{}
\end{align*}
\]

\[
\begin{align*}
\text{TAdd} & \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
\end{align*}
\]

\[
\begin{align*}
\text{TLoc} & \quad \frac{x : \text{int} \in A}{A \vdash x : \text{int}} \\
\text{TDec} & \quad \frac{A \vdash 4 : \text{int}}{A \vdash x + 4 : \text{int}}
\end{align*}
\]

\[
A = \{ x : \text{int} \}
\]
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)

\[
\begin{align*}
TDec & \quad \vdash DE\!C : int \\
THex & \quad \vdash HEX : int \\
TStr & \quad \vdash STR : str \\
TTrue & \quad \vdash true : bool \\
TFalse & \quad \vdash false : bool \\
TSubExpr & \quad \Gamma \vdash e : t \\
TAdd & \quad \Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \\
& \quad \Gamma \vdash e_1 + e_2 : int \\
\text{similar for } TSub & \quad (\text{for } TSub (-), \text{TMul } (\ast), \text{TDiv } (/) \text{ and } TMod \%)
\end{align*}
\]

\[
\begin{align*}
TEq & \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \\
& \quad \Gamma \vdash e_1 \text{ == } e_2 : \text{bool} \\
\text{similar for } TNeq \quad & \quad (\text{for } TNeq \text{ (!=)})
\end{align*}
\]

\[
\begin{align*}
TWhile & \quad \Gamma \vdash e : \text{bool} \quad \Gamma \vdash b \\
& \quad \Gamma \vdash \text{while } (\text{'C e'} \text{) b}
\end{align*}
\]
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
- **ThreadPool** – 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:** Abstract\textbf{Element} (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:** Abstract\textbf{Visitor} (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
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)