CS 432
Fall 2017

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 (P3)
  - **Type checking & other static analysis (P4)**
  - Code generation (P5)
  - Instruction selection
- Theory and practice
  - Type systems describe "correct" tree structures
  - Visitor design pattern allows clean implementation in a non-functional language
    - Generalization of tree traversal (i.e., 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

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
    - Often made in 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

- Type systems are expressed formally as 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 judgements ($A \vdash x : t$)

Lambda calculus:

$$\begin{array}{lll} \mathbf{E} \to \mathbf{x} & & \textit{(name/variable)} \\ \quad\;\; \mid \ \boldsymbol{\lambda}\mathbf{x.E} & & \textit{(function)} \\ \quad\;\; \mid \ \mathbf{E\ E} & & \textit{(application)} \end{array}$$

$$\text{TInt}\ \frac{}{A \vdash n : int} \qquad\qquad \frac{x : t \in A}{A \vdash x : t}\ \text{TVar}$$

$$\text{TFun}\ \frac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \frac{A \vdash e : t \to t' \qquad A \vdash e' : t}{A \vdash e\ e' : t'}\ \text{TApp}$$

Formal Type Theory

- Type systems are expressed formally as a set of type rules
  - Apply rules recursively in specific environments (e.g., symbol tables, marked in rules with ⊢ operator) to form proof trees
  - Curry-Howard correspondence (“proofs as programs”)

Lambda calculus:

E →	**x**	*(name/variable)*
|	**λx.E**	*(function)*
|	**E E**	*(application)*

$$\text{TInt}\ \frac{}{A \vdash n : \text{int}} \qquad \frac{x : t \in A}{A \vdash x : t}\ \text{TVar}$$

$$\text{TFun}\ \frac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad \frac{A \vdash e : t \to t' \qquad A \vdash e' : t}{A \vdash e\ e' : t'}\ \text{TApp}$$

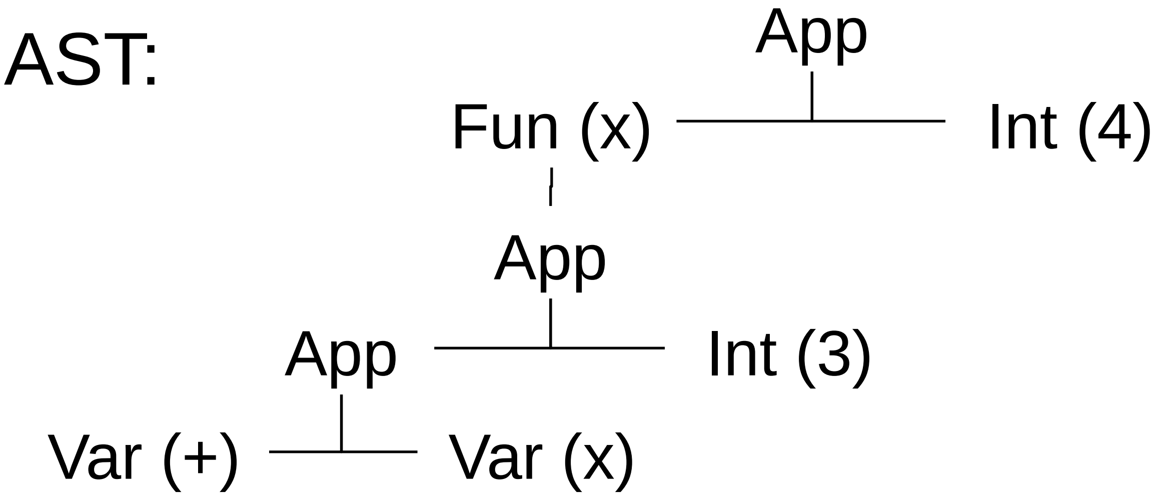
Formal Type Theory

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

$$(\lambda x{:}\mathrm{int}.+\ x\ 3)\ 4$$

$$A = \{\ +\ :\ \mathrm{int} \rightarrow \mathrm{int} \rightarrow \mathrm{int}\ \}$$

AST:



Formal Type Theory

$$\frac{}{A \vdash n : \mathrm{int}}\ \textbf{TInt} \qquad \frac{x : t \in A}{A \vdash x : t}\ \textbf{TVar} \qquad \frac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'}\ \textbf{TFun} \qquad \frac{A \vdash e : t \to t' \qquad A \vdash e' : t}{A \vdash e\ e' : t'}\ \textbf{TApp}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{TVar}\ \dfrac{+ : \qquad\qquad \in B}{B \vdash + : } \qquad \dfrac{x : \qquad \in B}{B \vdash x : }\ \text{TVar}}{B \vdash +\ x : }\ \text{TApp} \qquad B \vdash 3 : }{B \vdash +\ x\ 3 : }\ \text{TApp}}{A \vdash (\lambda x{:}\mathrm{int}.+\ x\ 3) : }\ \text{TFun} \qquad \dfrac{}{A \vdash 4 : }\ \text{TInt}}{A \vdash (\lambda x{:}\mathrm{int}.+\ x\ 3)\ 4 : }\ \text{TApp}$$

$A = \{\ + : \mathrm{int} \to \mathrm{int} \to \mathrm{int}\ \}$ $\qquad$ $B = A, x : \mathrm{int}$

Formal Type Theory

$$\frac{}{A \vdash n : int}$$

TInt

$$\frac{x : t \in A}{A \vdash x : t}$$

TVar

$$\frac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'}$$

TFun

$$\frac{A \vdash e : t \to t' \qquad A \vdash e' : t}{A \vdash e\ e' : t'}$$

TApp

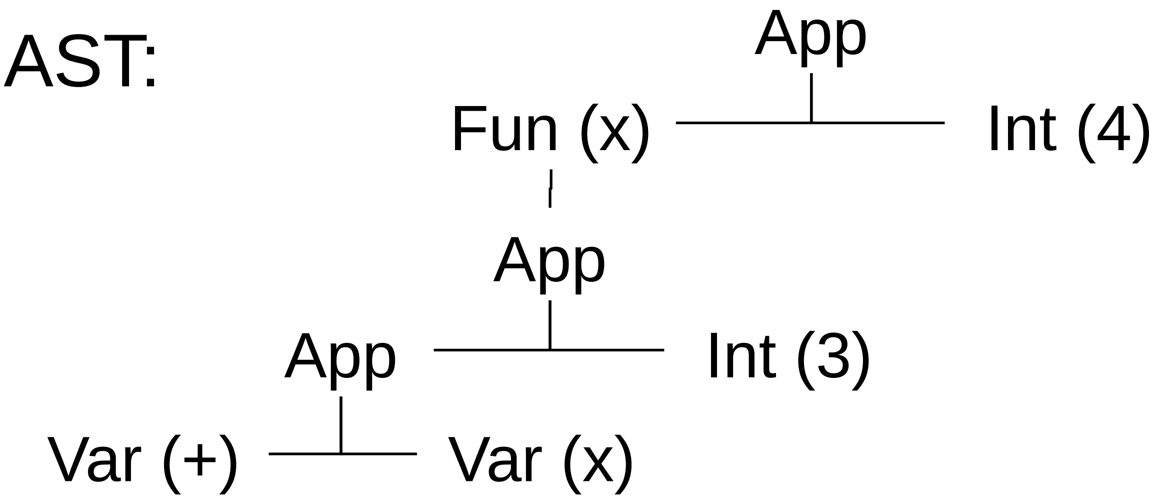
$$\dfrac{\text{TFun}\ \dfrac{\dfrac{\text{TApp}\ \dfrac{\text{TVar}\ \dfrac{+ : i \to i \to i \in B}{B \vdash + : i \to i \to i} \qquad \dfrac{x : int \in B}{B \vdash x : int}\ \text{TVar}}{B \vdash +\ x : int \to int} \qquad B \vdash 3 : int}{B \vdash +\ x\ 3 : int}\ \text{TApp}}{A \vdash (\lambda x{:}int.+\ x\ 3) : int \to int} \qquad \dfrac{}{A \vdash 4 : int}\ \text{TInt}}{A \vdash (\lambda x{:}int.+\ x\ 3)\ 4 : int}\ \text{TApp}$$

$A = \{ + : int \to int \to int \}$

$B = A, x : int$

Formal Type Theory

AST:



App

Fun (x)

Int (4)

App

App

Int (3)

Var (+)

Var (x)

$A = \{ + : \text{int} \to \text{int} \to \text{int} \}$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{+ : i \to i \to i \in B}{B \vdash + : i \to i \to i}\;\text{TVar} \qquad \dfrac{x : \text{int} \in B}{B \vdash x : \text{int}}\;\text{TVar}}{B \vdash + \, x : \text{int} \to \text{int}}\;\text{TApp} \qquad B \vdash 3 : \text{int}}{B \vdash + \, x \, 3 : \text{int}}\;\text{TApp}}{A \vdash (\lambda x{:}\text{int}. + x \; 3) : \text{int} \to \text{int}}\;\text{TFun} \qquad \dfrac{}{A \vdash 4 : \text{int}}\;\text{TInt}}{A \vdash (\lambda x{:}\text{int}. + x \; 3) \; 4 : \text{int}}\;\text{TApp}$$

P4: Static Analysis

- General idea: traverse AST and check for invalid programs
  - Language and project specifications provide rules to check at each type of AST node
    - E.g., at ASTProgram, make sure there is a “main” function
    - E.g., at ASTWhileLoop, make sure the conditional has a boolean type

$$\text{TDec}\,\frac{}{\vdash \mathtt{DEC} : \mathbf{int}} \qquad \text{THex}\,\frac{}{\vdash \mathtt{HEX} : \mathbf{int}} \qquad \text{TStr}\,\frac{}{\vdash \mathtt{STR} : \mathbf{str}}$$

$$\text{TTrue}\,\frac{}{\vdash \mathtt{true} : \mathbf{bool}} \qquad \text{TFalse}\,\frac{}{\vdash \mathtt{false} : \mathbf{bool}} \qquad \text{TSubExpr}\,\frac{\Gamma \vdash e : t}{\Gamma \vdash \text{‘(‘}\ e\ \text{‘)‘} : t}$$

$$\text{TAdd}\,\frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1\ \text{‘+‘}\ e_2 : \mathbf{int}}$$

(similar for TSub (-), TMul (*), TDiv (/) and TMod (%))

$$\text{TEq}\,\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ \text{‘==‘}\ e_2 : \mathbf{bool}}$$

(similar for TNeq (!=))

$$\text{TWhile}\,\frac{\Gamma \vdash e : \mathbf{bool} \qquad \Gamma \vdash b}{\Gamma \vdash \mathtt{while}\ \text{‘(‘}\ e\ \text{‘)‘}\ b}$$

P4: Static Analysis

- General idea: traverse AST and check for invalid programs
  - Need to traverse the tree multiple times
    - Build symbol tables
    - Perform type checking
    - Later compiler passes
  - We could write the tree traversal code every time, but that would get tedious and would result in 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"?

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

Common Design Patterns

- **Adapter** – Converts one interface into another
- **Factory** – Allows clients to create objects without exactly specifying their concrete class
- **Flyweight** – Manages large numbers of similar objects efficiently via sharing
- **Iterator** – Provides sequential access to a collection without exposing its implementation details
- **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** – Iterator over a structure (usually a recursive structure)

Design Patterns

- Pros
  - Faster development
  - More robust code (if implemented properly)
  - More readable code (for those familiar with patterns)
  - Improved maintainability
- Cons
  - Increased abstraction
  - Increased complexity
  - Philosophical: Suggests language deficiencies
    - Solution: Consider using a different language

Visitor Pattern

- **Visitor**: 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 OO languages
    - Lack of multiple dispatch (choosing a concrete method based on two objects' data types)
      - NOTE: Parametric polymorphism != multiple dispatch
    - Less useful in functional languages with more robust pattern matching

General Form

- Data: Abstract**Element** (ASTNode)
  - ConcreteElement1 (ASTProgram)
  - ConcreteElement2 (ASTVariable)
  - ConcreteElement3 (ASTFunction)
  - etc.
  - All elements define "Accept()" method that recursively calls "Accept()" on any child nodes (this is the actual tree traversal code!)
- Actions: Abstract**Visitor** (DefaultASTVisitor)
  - ConcreteVisitor1 (BuildParentLinks)
  - ConcreteVisitor2 (CalculateNodeDepths)
  - ConcreteVisitor3 (StaticAnalysis)
    - BuildSymbolTables
    - MyDecafAnalysis
  - All visitors have "VisitX()" methods for each element type

Benefits

- Adding new operations is easy
  - Just create a new concrete visitor
  - In our compiler, create a new DefaultASTVisitor subclass
- No wasted space for state in data classes
  - Just maintain state in the visitors
  - In our compiler, we will make a few exceptions for state that is shared across many visitors (e.g., symbol tables)

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 basically just structs anyway (all data is public)

Minor Modifications

- "Accept()" → "traverse()"
- "Visit()" → "preVisit()" and "postVisit()"
  - preVisit corresponds to a preorder traversal
  - postVisit corresponds to a postorder traversal
- DefaultASTVisitor class
  - Implements ASTVisitor interface
  - Contains empty implementations of all "visit" methods
  - Allows subclasses to define only the visit methods that are relevant

Visitor example

```
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);
    }
}
```

In DecafCompiler.java:

```
ast.traverse(new WhileLoopCounter());
```

Decaf Project

- Project 3
  - `ASTVisitor`
  - `DefaultASTVisitor` (implements ASTVisitor)
    - `PrintDebugTree`
    - `ExportTreeDOT`
    - `BuildParentLinks` (activity)
    - `CalculateNodeDepths` (activity)
- Project 4
  - `PrintDebugSymbolTables` (extends DefaultASTVisitor)
  - `StaticAnalysis` (extends DefaultASTVisitor)
    - `BuildSymbolTables`
    - `DecafAnalysis` + **`MyDecafAnalysis`**
- Project 5
  - `ILOCGenerator` + **`MyILOCGenerator`**