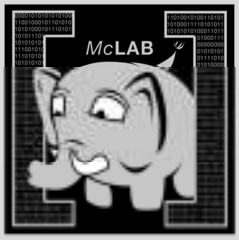
McLab Tutorial
www.sable.mcgill.ca/mclab



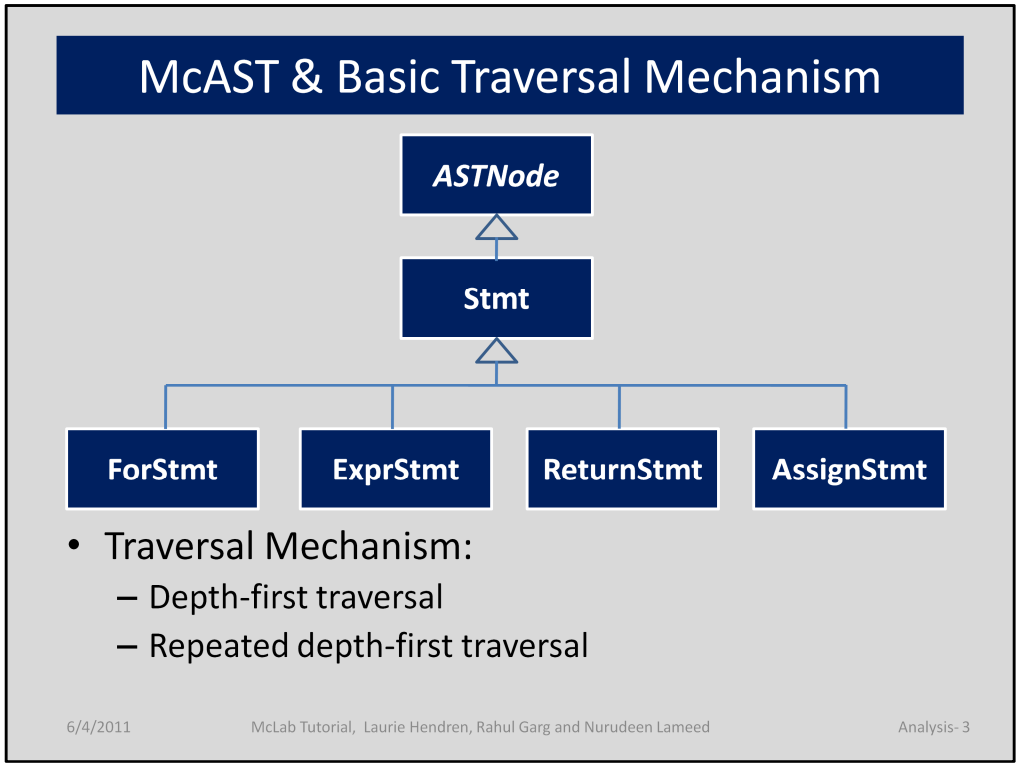
Part 5 – Introduction to the McLab Analysis Framework

- Exploring the Main Components
- Creating a Simple Analysis
- Depth-first and Structural Analyses
- Example: Reaching Definition Analysis

McLab Analysis Framework

- A simple static flow analysis framework for MATLAB-like languages
- Supports the development of intra-procedural forward and backward flow analyses
- Extensible to new language extensions
- Facilitates easy adaptation of old analyses to new language extensions
- Works with McAST and McLAST (a simplified McAST)

The Mclab analysis framework is designed to be simple to use. It is easily extensible for developing analyses for new language extensions. It is written in Java programming language



A simplified and incomplete McLab Abstract Syntax Tree; ASTNode class is the root node of McAST. With Repeated depth-first traversal, some nodes are visited repeatedly to compute a fixed point for an analysis.

Exploring the main components for developing analyses

The interface *NodeCaseHandler*

- Declares all methods for the action to be performed when a node of the AST is visited:

```
public interface NodeCaseHandler {
   void caseStmt(Stmt node);
   void caseForStmt(ForStmt node);
   void caseWhileStmt(WhileStmt node);
   …
}
```

The class *AbstractNodeCaseHandler*

```
public class AbstractNodeCaseHandler implements
    NodeCaseHandler {
    …
    void caseStmt(Stmt node) {
        caseASTNode(node);
    }
    …
}
```

- Implements the interface *NodeCaseHandler*
- Provides default behaviour for each AST node type except for the root node (*ASTNode*)

AbstractNodeCaseHandler provides default implementation for all node types except the root node. The default behaviour of a node case hander is to forward to the node case handler of its parent.

The analyze method

- Each AST node also implements the method *analyze* that performs an analysis on the node:

```
public void analyze(NodeCaseHandler handler)
        handler.caseAssignStmt(this);
}
```

Creating a simple analysis

Creating a Traversal/Analysis:

- Involves 3 simple steps:

1. Create a concrete class by extending the class *AbstractNodeCaseHandler*
2. Provide an implementation for *caseASTNode*
3. Override the relevant methods of *AbstractNodeCaseHandler*

An Example: StmtCounter

- Counts the number of statements in an AST

Analysis development Steps:

1. Create a concrete class by extending the class *AbstractNodeCaseHandler*
2. Provide an implementation for *caseASTNode*
3. Override the relevant methods of *AbstractNodeCaseHandler*

An Example: StmtCounter

1. Create a concrete class by extending the class *AbstractNodeCaseHandler*

```
public class StmtCounter extends
    AbstractNodeCaseHandler {
    private int count = 0;
    … // defines other internal methods
}
```

An Example: StmtCounter --- Cont'd

2. Provide an implementation for *caseASTNode*

```
public void caseASTNode( ASTNode node){
    for(int i=0; i<node.getNumChild(); ++i) {
        node.getChild(i).analyze(this);
    }
}
```

An Example: StmtCounter --- Cont'd

3. Override the relevant methods of *AbstractNodeCaseHandler*

```
public void caseStmt(Stmt node) {
    ++count;
    caseASTNode(node);
}
```

An Example: StmtCounter --- Cont'd

```
public class StmtCounter extends AbstractNodeCaseHandler {
    private int count = 0;
    private StmtCounter() { super(); }
    public static int countStmts(ASTNode tree) {
        tree.analyze(new StmtCounter());
    }
     public void caseASTNode( ASTNode node){
        for(int i=0; i<node.getNumChild(); ++i) {
            node.getChild(i).analyze(this);}
     }
    public void caseStmt(Stmt node) {
        ++count; caseASTNode(node);
    }
}
```

Tips: Skipping Irrelevant Nodes

For many analyses, not all nodes in the AST are relevant; to skip unnecessary nodes override the handler methods for the nodes. For Example:

```
public void caseExpr(Expr node) {
    return;
}
```

Ensures that all the children of *Expr* are skipped

Analyses Types: Depth-first and Structural Analyses

Flow Facts: The interface *FlowSet*

- The interface *FlowSet* provides a generic interface for common operations on flow data

```
public  interface FlowSet<D> {
  public FlowSet<D> clone();
  public void copy(FlowSet<? super D> dest);
  public void union(FlowSet<? extends D> other);
  public void intersection(FlowSet<? extends D> other);
    …
}
```


A typical analysis computes a set of flow facts or data

The *Analysis* interface

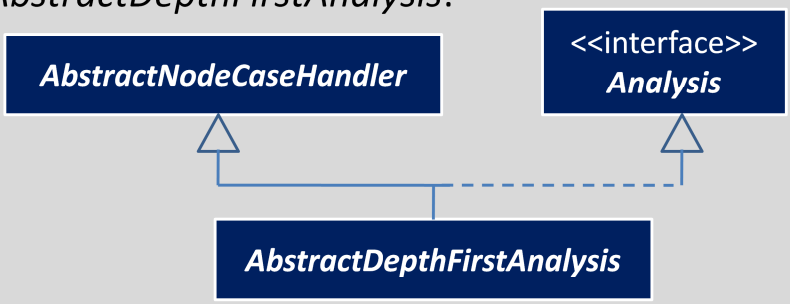
- Provides a common API for all analyses
- Declares additional methods for setting up an analysis:

```
public interface Analysis<A extends FlowSet> extends
    NodeCaseHandler {
    public void analyze();
    public ASTNode getTree();
    public boolean isAnalyzed();
    public A newInitialFlow();
    …
}
```


The method analyze executes the analysis; getTree returns the AST that is being analysed; isAnalyzed tests whether the analysis has been performed; newInitialFlow gives the initial approximation for flow facts.

Depth-First Analysis

- Traverses the tree structure of the AST by visiting each node in a depth-first order
- Suitable for developing flow-insensitive analyses
- Default behavior implemented in the class *AbstractDepthFirstAnalysis*:



Creating a Depth-First Analysis:

- Involves 2 steps:

1. Create a concrete class by extending the class *AbstractDepthFirstAnalysis*
   a) Select a type for the analysis's data
   b) Implement the method *newInitialFlow*
   c) Implement a constructor for the class

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

Depth-First Analysis: NameCollector

- Associates all names that are assigned to by an assignment statement to the statement.
- Collects in one set, all names that are assigned to
- Names are stored as strings; we use *HashSetFlowSet<String>* for the analysis's flow facts.
- Implements *newInitialFlow* to return an empty *HashSetFlowSet<String>* object.

Depth-First Analysis: NameCollector --- Cont'd

1. Create a concrete class by extending the class *AbstractDepthFirstAnalysis*

```
public class NameCollector extends
    AbstractDepthFirstAnalysis
    <HashSetFlowSet<String>> {
    private int HashSetFlowSet<String> fullSet;

    public NameCollector(ASTNode tree) {
        super(tree); fullSet = newInitialFlow();
    }
    … // defines other internal methods
}
```

Depth-First Analysis: NameCollector --- Cont'd

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

```
private boolean inLHS = false;

public void caseName(Name node) {
    if (inLHS)
        currentSet.add(node.getID());
}
```

Depth-First Analysis: NameCollector --- Cont'd

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

```
public void caseAssignStmt(AssignStmt node) {
    inLHS = true;
    currentSet = newInitialFlowSet();
    analyze(node.getLHS());
    flowSets.put(node, currentSet);
    fullSet.addAll(currentSet);
    inLHS = false;
}
```

Depth-First Analysis: NameCollector --- Cont'd

2. Override the relevant methods of *AbstractDepthFirstAnalysis*

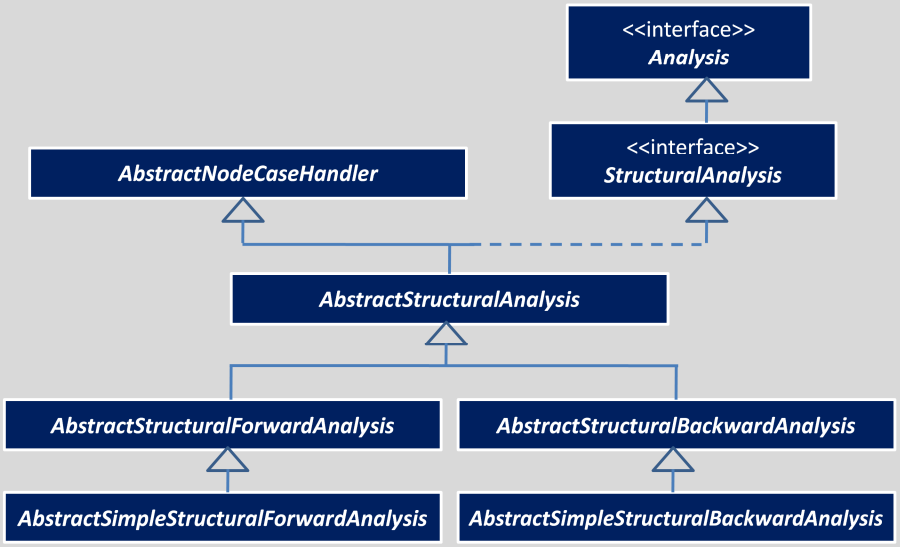
```
public void caseParameterizedExpr
(ParameterizedExpr node) {
    analyze(node.getTarget());
}

...
```

Structural Analysis

- Suitable for developing flow-sensitive analyses
- Computes information to approximate the runtime behavior of a program.
- Provides mechanism for:
  - analyzing control structures such as *if-else*, *while* and *for* statements;
  - handling *break* and *continue* statements
- Provides default implementations for relevant methods
- May be forward or backward analysis

Structural Analysis Class Hierarchy



The interface *StructuralAnalysis*

- Extends the *Analysis* interface
- Declares more methods for structural type analysis:

```
public interface StructuralAnalysis<A extends
    FlowSet> extends Analysis<A> {
    public Map<ASTNode, A> getOutFlowSets();
    public Map<ASTNode, A> getInFlowSets();
    public void merge(A in1, A in2, A out);
    public void copy(A source, A dest);
    …
}
```

Developing a Structural Analysis

- Involves the following steps:
  1. Select a representation for the analysis's data
  2. Create a concrete class by extending the class: *AbstractSimpleStructuralForwardAnalysis* for a forward analysis and *AbstractSimpleStructuralBackwardAnalysis* for a backward analysis
  3. Implement a suitable constructor for the analysis and the method *newInitialFlow*
  4. Implement the methods *merge* and *copy*
  5. Override the relevant node case handler methods and other methods

Example: Reaching Definition Analysis

Example: Reaching Definition Analysis

For every statement *s*, for every variable *v* defined by the program, compute the set of all definitions or assignment statements that assign to *v* and that *may* reach the statement *s*

A definition *d* for a variable *v* reaches a statement *s*, if there exists a path from *d* to *s* and *v* is not re-defined along that path.

Reach Def Analysis: An Implementation Step 1

Select a representation for the analysis's data:

HashMapFlowSet<String, Set<ASTNode>>

We use a map for the flow data: An entry is an ordered pair (*v*, *defs*)

where *v* denotes a variable and

defs denotes the set of definitions for *v* that may reach a given statement.

Reach Def Analysis: An Implementation Step 2

Create a concrete class by extending the class: *AbstractSimpleStructuralForwardAnalysis* for a forward analysis:

```
public class ReachingDefs extends
  AbstractSimpleStructuralForwardAnalysis
 <HashMapFlowSet<String, Set<ASTNode>>> {
  …
}
```


The analysis is a forward analysis so we extend *AbstractSimpleStructuralForwardAnalysis*

Reach Def Analysis: An Implementation Step 3

Implement a suitable constructor and the method *newInitialFlow* for the analysis:

```
public ReachingDefs(ASTNode tree) {
  super(tree);
  currentOutSet = newInitialFlow(); }

public HashMapFlowSet<String, Set<ASTNode>>
  newInitialFlow() {
  return new
  HashMapFlowSet<String,Set<ASTNode>>(); }
```

Reach Def Analysis: An Implementation Step 4a

Implement the methods *merge* and *copy*:

```
public void merge
(HashMapFlowSet<String, Set<ASTNode>> in1,
    HashMapFlowSet<String, Set<ASTNode>> in2,
    HashMapFlowSet<String, Set<ASTNode>> out) {
        union(in1, in2, out);
 }
public void
copy(HashMapFlowSet<String, Set<ASTNode>> src,
     HashMapFlowSet<String, Set<ASTNode>> dest) {
     src.copy(dest);
}
```

Reach Def Analysis: An Implementation Step 4b

```
public void
union (HashMapFlowSet<String, Set<ASTNode>> in1,
   HashMapFlowSet<String, Set<ASTNode>> in2,
   HashMapFlowSet<String, Set<ASTNode>> out) {
   Set<String> keys = new HashSet<String>();
    keys.addAll(in1.keySet()); keys.addAll(in2.keySet());
    for (String v: keys) {
       Set<ASTNode> defs = new HashSet<ASTNode>();
       if (in1.containsKey(v)) defs.addAll(in1.get(v));
       if (in2.containsKey(v)) defs.addAll(in2.get(v));
       out.add(v, defs);
    }
}
```


The helper method *union* is shown on the next slide.

Reach Def Analysis: An Implementation Step 5a

Override the relevant node case handler methods and other methods :

override caseAssignStmt(AssignStmt node)

```
public void caseAssignStmt(AssignStmt node) {
    inFlowSets.put(node, currentInSet.clone() );
    currentOutSet =
        new HashMapFlowSet<String, Set<ASTNode> > ();

    copy(currentInSet, currentOutSet);
    HashMapFlowSet<String, Set<ASTNode>> gen =
        new  HashMapFlowSet<String, Set<ASTNode>> ();
    HashMapFlowSet<String, Set<ASTNode> > kill =
        new HashMapFlowSet<String, Set<ASTNode>> ();
```

Reach Def Analysis: An Implementation Step 5b

```
// compute out = (in - kill) + gen
    // compute kill
     for( String s : node.getLValues() )
      if (currentOutSet.containsKey(s))
         kill.add(s, currentOutSet.get(s));
      // compute gen
     for( String s : node.getLValues()){
       Set<ASTNode> defs = new HashSet<ASTNode>();
       defs.add(node);
       gen.add(s, defs);
     }
```

Reach Def Analysis: An Implementation Step 5c

```
    // compute (in - kill)
    Set<String> keys = kill.keySet();
    for (String s: keys)
      currentOutSet.removeByKey(s);
    // compute (in - kill) + gen
    currentOutSet = union(currentOutSet, gen);

    // associate the current out set to the node
    outFlowSets.put( node, currentOutSet.clone() );
}
```