COMPILER DESIGN

Symbol table structure and generation

Type checking

Visitor pattern

Symbol tables: processing declarations

- One of the objectives of the compiler is to verify that identifiers are properly used according to their specifications:
  - Variables used according to their declarations (type, dimensionality)
  - Functions called according to their declarations (return type, number and type of parameters)
  - Objects are using members as defined in their class declaration
  - Variables/functions are only used in the scope in which they were declared
- In **manifestly-typed languages**, all identifiers have a declaration that specifies the types that they are bound to:
  - Class and/or data structure declaration (global or embedded). Represent user-defined data types.
  - Variable declaration (global/local variables, class/data structure members)
  - Function declaration (free function, member function, embedded function)
- Other languages minimize declarations by omitting types
  - Often referred to as “dynamic languages”
  - Limited analysis can still be done at compile-time, e.g. type inference
  - Some of the analysis has to be delayed to run-time
- In the project, our language is manifestly-typed.

Symbol tables: processing declarations: tree traversal

- Declarations are processed as a tree is traversed, either:
  - As a parse tree as the parse phase is proceeding, using Syntax-Directed Translation.
  - As an Abstract Syntax Tree intermediate representation previously generated by the parse phase is traversed.
- The processing of the declarations is made by aggregating and processing semantic information that is present at the leaves of the tree, and migrated/processed by intermediate nodes of the tree.
- This migration/aggregation/processing is done using **semantic actions** that are triggered as the tree is traversed
- Specific semantic actions should be triggered when certain specific kind of nodes are reached as the tree is traversed.
- If the processing is done upon an intermediate representation (e.g. an AST), further processing can be done in successive phases, where each phase is responsible for a specific part of the analysis/translation process.

Symbol table structure

Symbol tables: scopes

- Generally, every scope defined in the language/program requires its own symbol table.
  - The global scope contains an entry for all identifiers not declared in any scope, e.g. global variables, globally-declared classes or data structures, free functions.
  - Some scopes (e.g. classes, functions) allow some identifiers to be declared, e.g. local variables, or even inner classes or inner functions – which we don't have in the project.
  - Some scopes are not named, e.g. a for loop's statement block – these can also define their own sub-scope and corresponding sub-symbol table.

Symbol table: procedures to implement

- **`CreateNewTable`** – when hen a new scope is entered, a new empty symbol table is created, as well as the symbol table entry to be recorded in the higher-level symbol table.
- **`Insert`** – while in a scope, for every declaration encountered, a symbol table record is created and inserted.
- **`Search`** – when an identifier is referred to in a scope, the compiler needs to check if it has been previously defined, either in the current scope or one of the higher-level scope, requiring a search method that searches across a table hierarchy. If information hiding descriptors are part of the language, they must be taken into consideration here.
- **`Print`** – for utility, a facility to output a symbol table, along with its sub-tables.
- **`Delete`** – some scopes cannot be referred to outside of their scopes. In which case their symbol tables have to be deleted, for efficiency. Some scopes, e.g. classes can be referred to from outside their own scope, These should not be kept until the end of the compilation process.

Symbol tables: example

Symbol table: Global

name	kind	type	link
f1	function	float : int[2][2], float	•
f2	function	int : nil	•
MyClass1	class		•
MyClass2	class		•
program	function		•

Symbol table: f1

name	kind	type	link
fp1	parameter	int[2][2]	
fp2	parameter	float	
fv1	variable	MyClass1[3]	
fv2	variable	int	

Symbol table: f2

name	kind	type	link

Symbol table: MyClass1

name	kind	type	link
mc1v1	variable	int[2][4]	
mc1v2	variable	float	
mc1v3	variable	MyClass2[3]	
mc1f1	function	int : int, MyClass2[3]	•
f2	function	int : MyClass1[3]	•

Symbol table: MyClass1:mc1f1

name	kind	type	link
p1	parameter	int	
p2	parameter	MyClass2[3]	
fv1	variable	MyClass2[3]	

Symbol table: MyClass1:f2

name	kind	type	link
f2p1	parameter	MyClass1[3]	
mc1v1	variable	int	

Symbol table: MyClass2

name	kind	type	link
mc1v1	variable	int[2][4]	
fp1	variable	float	
m2	variable	MyClass2[3]	

Symbol table: program

name	kind	type	link
m1	variable	int	
m2	variable	float[3][2]	
m3	variable	MyClass2[2]	

```
class MyClass1 {
  int mc1v1[2][4];
  float mc1v2;
  MyClass2 mc1v3[3];
  int mc1f1(int p1, MyClass2 p2[3]) {
    MyClass2 fv1[3];
    …
  }
  int f2(MyClass1 f2p1[3]) {
    int mc1v1;
    …
  }
}

class MyClass2 {
  int mc1v1[2][4];
  float fp1;
  MyClass2 m2[3];
  …
}

program {
  int m1;
  float[3][2] m2;
  MyClass2[2] m3;
  ...
}
float f1(int fp1[2][2], float fp2) {
  MyClass1[3] fv1;
  int fv2;
  …
}

int f2() {
...
}
```

Symbol table generation

Symbol table generation: implementation

- Two avenues are possible:
  - Integrate generation of the symbol table in the parse, using syntax-directed translation
  - Have the parser generate an Abstract Syntax Tree intermediate representation, then implement a tree traversal phase on the AST to generate the symbol table

Symbol table generation using syntax-directed translation

Symbol tables: generation using syntax-directed translation

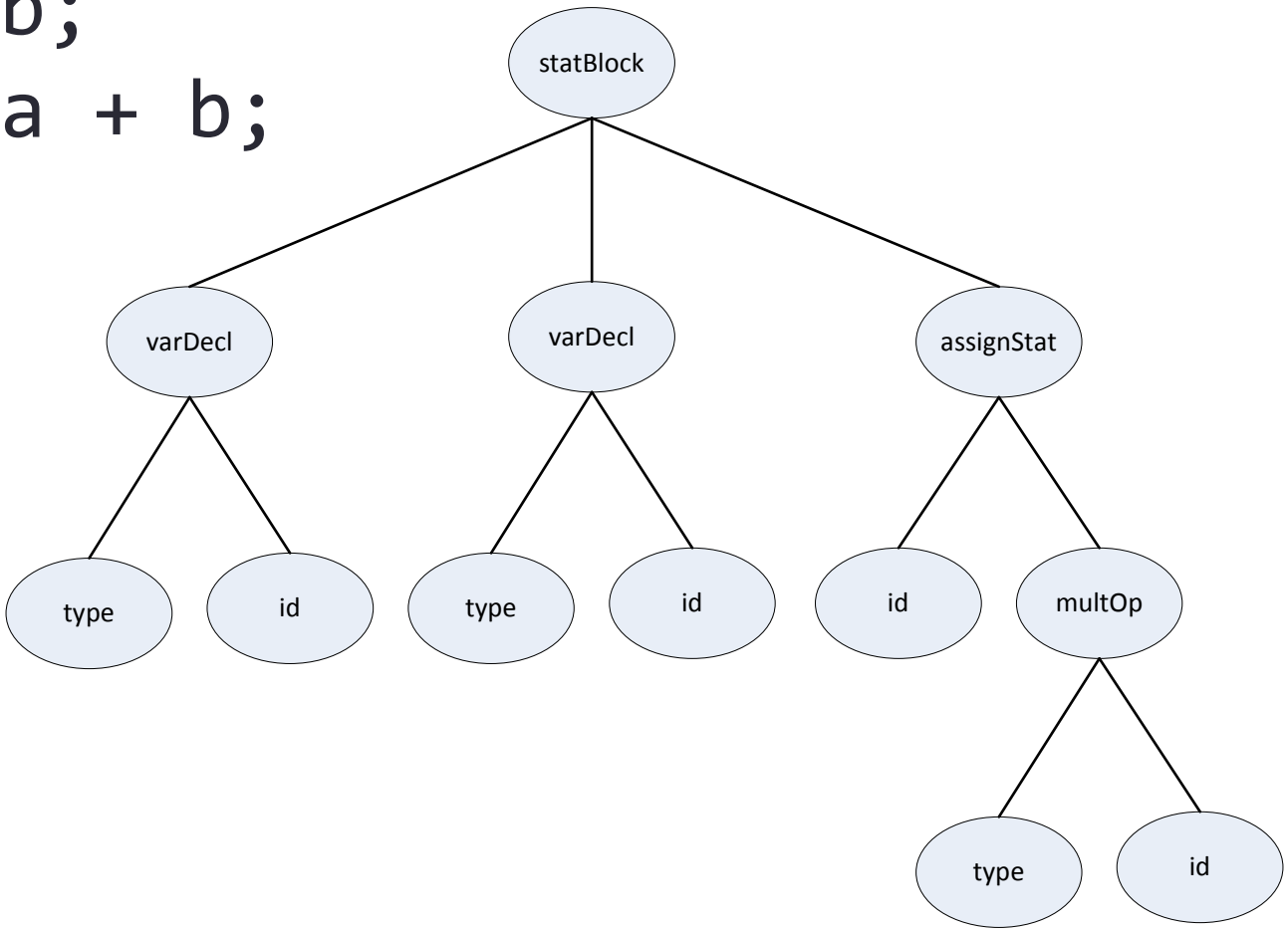
```
          <prog> ::= #createGlobalTable# <classDecl>*<progBody>
     <classDecl> ::= class id #createClassEntryAndTable# {<varDecl>*<funcDef>*};
      <progBody> ::= program #createProgramTable# <funcBody>;<funcDef>*
      <funcHead> ::= <type>id(<fParams>) #createFunctionEntryAndTable#
       <funcDef> ::= <funcHead> <funcBody>;
      <funcBody> ::= {<varDecl>*<statement>*}
       <varDecl> ::= <type>id<arraySize>*; #createVariableEntry#
     <statement> ::= <assignStat>;
                   | if(<expr>)then<statBlock>else<statBlock>;
                   | for(<type>id<assignOp><expr>;<relExpr>;<assignStat>)<statBlock>;
                   | get(<variable>);
                   | put(<expr>);
                   | return(<expr>);
      <assignStat> ::= <variable><assignOp><expr>
      <statBlock> ::= {<statement>*} | <statement> | ε
          <expr> ::= <arithExpr> | <relExpr>
       <relExpr> ::= <arithExpr><relOp><arithExpr>
      <arithExpr> ::= <arithExpr><addOp><term> | <term>
          <sign> ::= + | -
          <term> ::= <term><multOp><factor> | <factor>
        <factor> ::= <variable>
                   | <idnest>*id(<aParams>)
                   | num
                   | (<arithExpr>)
                   | not<factor>
                   | <sign><factor>
      <variable> ::= <idnest>*id<indice>*
        <idnest> ::= id<indice>*.
        <indice> ::= [<arithExpr>]
      <arraySize> ::= [ int ]
          <type> ::= int | float | id
       <fParams> ::= <type>id<arraySize>* #createParameterEntry# <fParamsTail>* | ε
       <aParams> ::= <expr><aParamsTail>* | ε
    <fParamsTail> ::= ,<type>id<arraySize>* #createParameterEntry#
    <aParamsTail> ::= ,<expr>
```

- Insert semantic actions (function calls) that create tables/entries when an identifier is declared.
- Attribute migration must be used when the information necessary to create an entry is distributed over different rules.
- The same process is used to do further semantic processing:
  - Expression type inference
  - Type checking
  - Code generation
- Pitfalls:
  - Many semantic actions have to be inserted in the parser's operation, leading to increasing potential confusion.
  - All the phases including and after syntax analysis are "piled-up" on the parser.

Symbol table generation using Abstract Syntax Tree traversal using the visitor pattern

Tree traversal and semantic actions: example

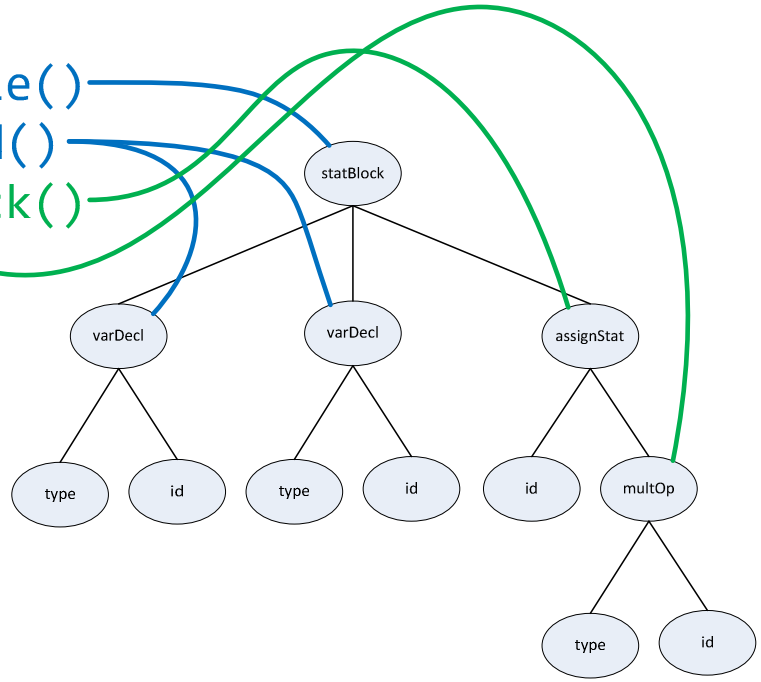
```
{
   int a;
   int b;
   a = a + b;
}
```



Tree traversal and semantic actions: example

Semantic actions

```
statBlock::buildtable()
varDecl::buildrecord()
assignStat::typeCheck()
multOp::typeCheck()
```



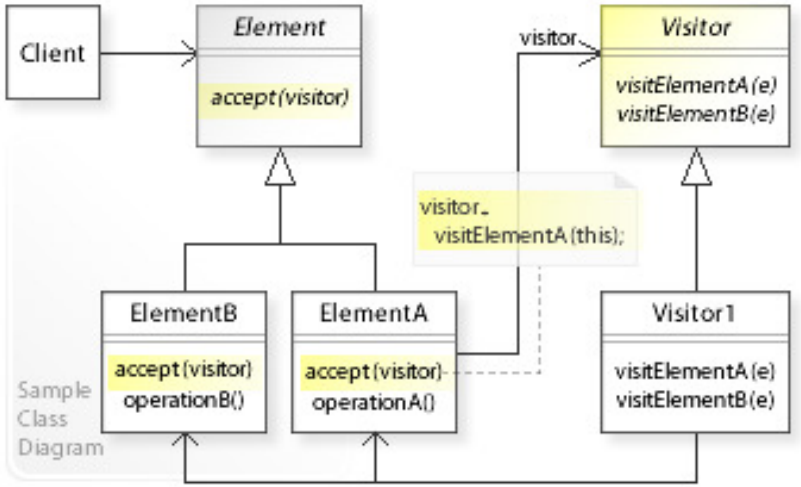
- Semantic actions should be executed as the tree is traversed, as soon as each node's subtree has been fully traversed.
- Each node may potentially have to execute zero to several semantic actions. It may seem appropriate to define semantic actions as members of node classes.
- However, some semantic actions require actions from other semantic actions to have taken place before.
- Thus, semantic actions have to be grouped into phases, AND sill mapped to specific nodes.
- The visitor pattern is a structure that achieves all of this.

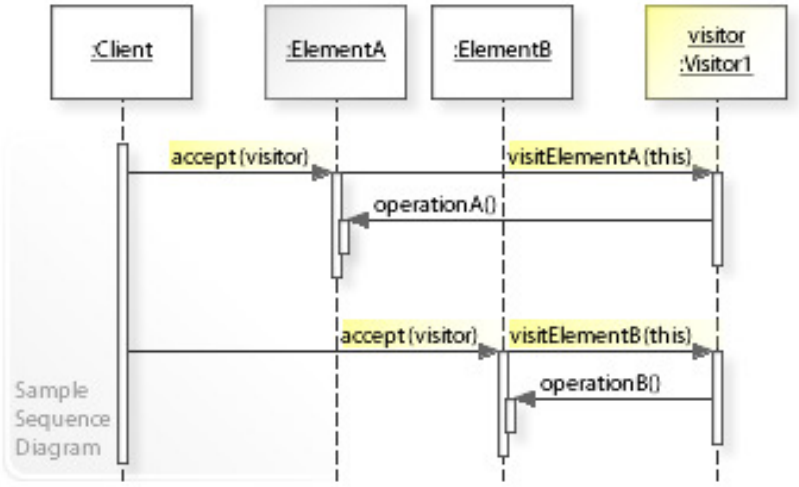
Visitor pattern

- Motivation
  - When many new operations are needed and the object structure consists of many unrelated classes, it is inflexible to add new subclasses each time a new operation is required.
  - Distributing all these new operations across the various classes leads to a system that's hard to understand, maintain, and change.
  - Often, the new operations form groups that are related to specific kinds of operations.
    - Adding all of them in the same existing classes creates low cohesion.
    - Often, only specific sub-groups of operations should be triggered.
- Intent
  - To create a structure/mechanism by which new operations are injected into existing classes, with minimal changes to be applied to these classes.
  - To provide modularity and cohesion by creating groups of related new operations.
  - Be able to inject/execute only a specific group of operations at a time.

Visitor pattern: structure

- The classes/objects participating in adapter pattern:
  - **Element** – superclass of all the objects that are to be injected with new behavior.
  - **ConcreteElement** – all the specific classes into which the new behavior is to be injected.
  - **Visitor** – superclass of all injectable behavior groups
  - **ConcreteVisitor** – specific injectable behavior group
  - **Client** – piece of code that uses a group/structure of elements and wants to inject/execute new behavior in them





Visitor pattern: implementation

- The visitor pattern is made necessary when using a programming language that only supports **single dispatch**, as opposed to multiple dispatch. Under this condition, consider two objects, each of some class type; one is termed the element, and the other is visitor.
- The `Visitor` class *declares* a `visit` method, which takes the `Element` as an argument, for each class of `Element` that may be subject to new behavior injection, and provides the implementation of the injected behavior.

```
public class Visitor {
  public void visit(Node node)        {};
  public void visit(IdNode node)      {};
  public void visit(NumNode numNode)  {};
  public void visit(TypeNode node)    {};
  public void visit(AddOpNode node)   {};
  public void visit(MultOpNode node)  {};
  public void visit(VarDeclNode node) {};
  public void visit(DimListNode node) {};
}
```

- The `Visitor` class can be abstract, or an interface.
- In order to provide default empty behavior, it can be a regular class with empty methods for every `Element` type to be processed.

Visitor pattern: implementation

- **`ConcreteVisitors`** classes are derived from the **`Visitor`** class and *implement* these **`visit`** methods, each of which implements part of the behavior group operating on the object structure. The state of the behavior group is maintained locally by the **`ConcreteVisitor`** class.

```
public class SymTabCreationVisitor extends Visitor {
  …
  public void visit(VarDeclNode node){
    System.out.println("visiting VarDeclNode");
    // aggregate information from the subtree
    String declrecstring;
    // identify what kind of record that is
    declrecstring = "localvar:";
    // get the type from the first child node and aggregate here
    Declrecstring += node.getChildren().get(0).getData() + ':';
    // get the id from the second child node and aggregate here
    declrecstring += node.getChildren().get(1).getData() + ':';
    // loop over the list of dimension nodes and aggregate here
    for (Node dim : node.getChildren().get(2).getChildren())
      declrecstring += dim.getData() + ':';
    // create the symbol table entry for this variable
    // it will be picked-up by another node above later
    node.symtabentry = new SymTabEntry(declrecstring, null);
  }
  …
}
```

Visitor pattern: implementation

- The `Element` declares an `accept` method to accept the injection of the applicable methods of a `Visitor`, by taking the `Visitor` as an argument.
- `ConcreteElements`, derived from the `Element` class, implement the `accept` method, these effectively implement tree traversal.
  - For atomic `Elements`, this is a simple call to the visitor's `visit` method.

```
public class IdNode extends Node {
…
  public void accept(Visitor visitor) {
  visitor.visit(this);}
}
```

  - For composite `Elements` (i.e. elements that contain references to other elements), each referenced element`s `accept` method is called, propagating the injection of the new behavior throughout the structure and doing tree traversal.

```
public class MultOpNode extends Node {
…
  public void accept(Visitor visitor) {
    for (Node child : this.getChildren() )
      child.accept(visitor);
    visitor.visit(this);
  }
}
```

Visitor pattern: implementation

- The `Client` creates the `Element` object structure and instantiates one `ConcreteVisitor` for each of the behavior group it wants to inject into the structure.
- When it wants to execute the behavior implemented by a specific `ConcreteVisitor`, it calls the `accept` method of the element that is the entry point of the structure, passing the specific `ConcreteVisitor` as a parameter.
- The accept/visit methods calls are then propagated in the structure, effectively implementing the chosen behavior.

```
ConstructExpressionStringVisitor CEVisitor = new ConstructExpressionStringVisitor();
TypeCheckingVisitor TCVisitor = new TypeCheckingVisitor();
Node d      = new IdNode("d", "int");
Node e      = new IdNode("e", "int");
Node f      = new IdNode("f", "float");
Node times  = new MultOpNode("*", e, f);
Node plus   = new AddOpNode("+", d, times1);
plus.accept(TCVisitor);
plus.accept(CEVisitor);
```

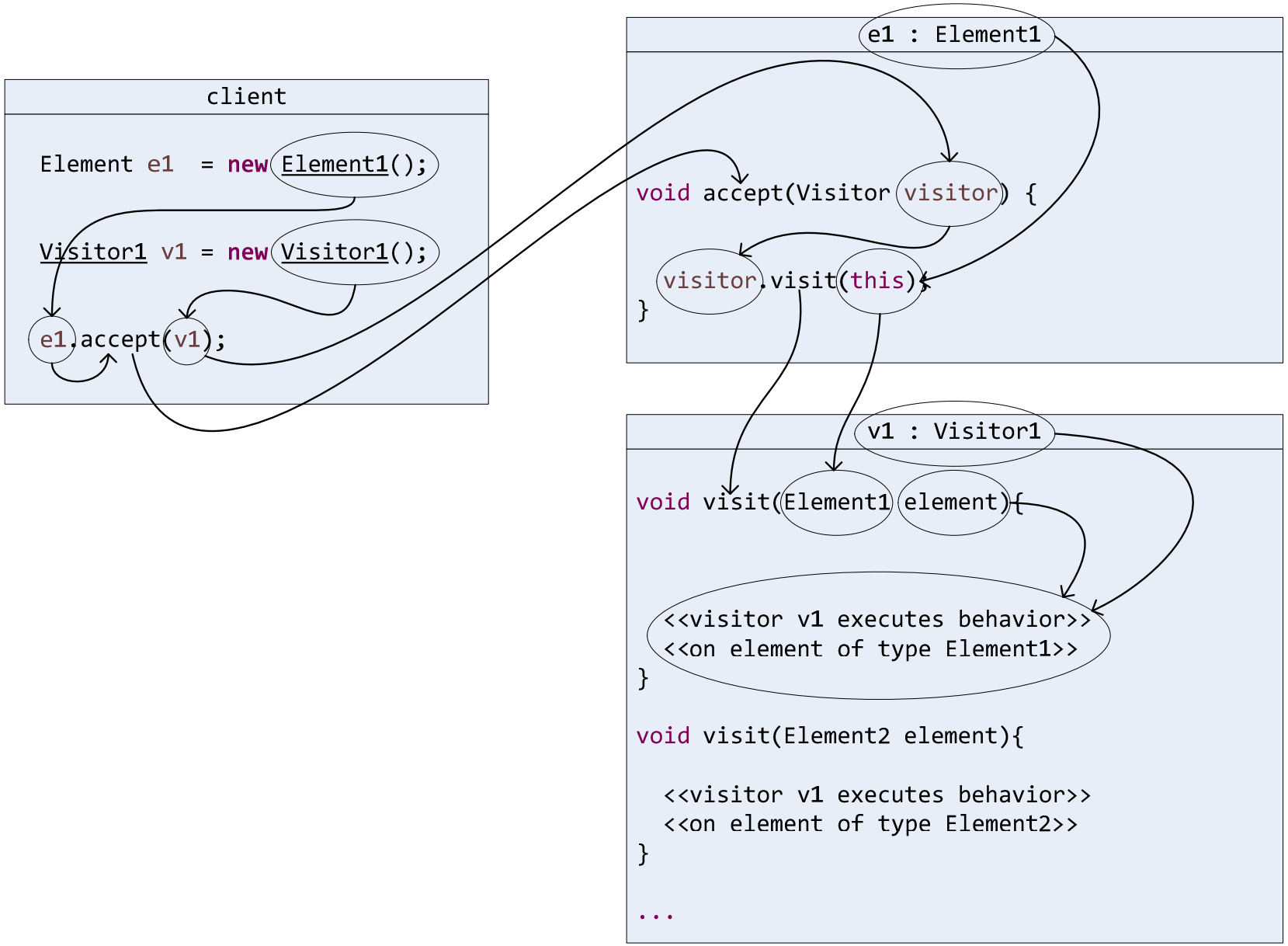
Visitor pattern: mechanism: double dispatch

- When the `accept` method is called, its specific implementation is chosen based on both the dynamic type of the `Element` upon which it is called, and the static type of the `Visitor` that is passed to it as a parameter.

- When the `visit` method is called in the `accept` method, its specific implementation is chosen based on both the dynamic type of the `Visitor` and the static type of the `Element`, as known from within the implementation of the `accept` method, which is the same as the dynamic type of the element.

- Thus, the implementation of the `visit` method is chosen based on both the dynamic type of the element and the dynamic type of the visitor. This effectively implements a *double dispatch mechanism*.

Visitor pattern: mechanism: double dispatch



Visitor pattern: use in compiler design

- In this way, abehavior can be implemented and applied as a graph composed of `Elements` is traversed.
- Many different kinds of separate behaviors can be performed during that traversal by supplying different `ConcreteVisitors` to interact with the elements based on the dynamic types of both the elements and the `Visitors`.

- In a compiler, the `Elements` are AST nodes, which are interconnected to form an Abstract Syntax Tree.
- Different `ConcreteVisitors` can be designed to implement the different semantic analysis and translation phases, e.g.
  - Construction of symbol table
  - Type checking
  - Translation into executable code
  - Various optimization phases
- Using the `Visitor` pattern, the implementation of all these phases can effectively be separated, thus decreasing the overall complexity of the compiler and increasing the flexibility of its implementation.

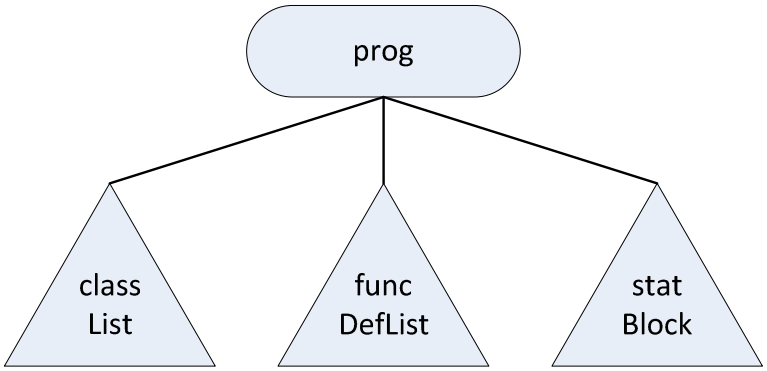
AST visitors for symbol table creation

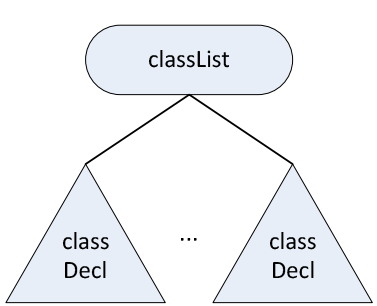
AST visitor: symbol tables creation

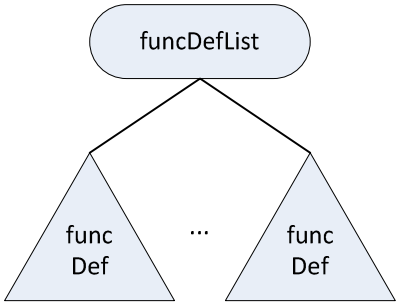
- Each scope in the program should have its own symbol table
  - global, class, function body: member function, free function, main function
  - When we visit one of these node types:
    - Create a new local symbol table that is a new data member of this node
    - Aggregate id declarations' symbol table records from its children subtrees

AST Visitor pattern : example

```
public class SymTabCreationVisitor extends Visitor {
  …
  public void visit(ProgNode node){
    node.symtab = new SymTab("global");
    // for classes, loop over all class declaration nodes
    for (Node classelt : node.getChildren().get(0).getChildren())
      //add the symbol table entry of each class in the global symbol table
      node.symtab.addEntry(classelt.symtabentry);
    // for function definitions, loop over all function definition nodes
    for (Node fndefelt : node.getChildren().get(1).getChildren())
      //add the symbol table entry of each function definition in the global symbol table
      node.symtab.addEntry(fndefelt.symtabentry);
    // for the program function, get its local symbol table from node 2 and create
    // an entry for it in the global symbol table
    // first, get the table and change its name
    SymTab table = node.getChildren().get(2).symtab;
    table.m_name = "program";
    node.symtab.addEntry("function:program", table);
  };
  …
}
```





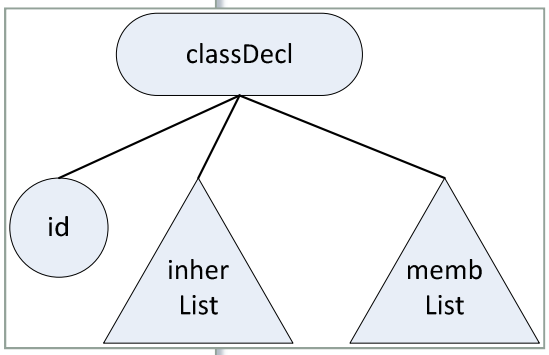


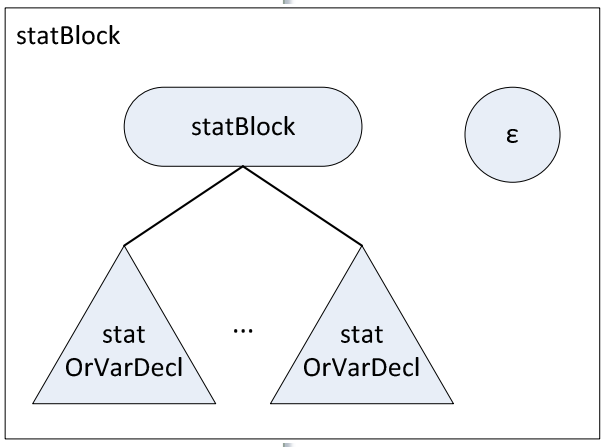
AST Visitor pattern : example

```
public class SymTabCreationVisitor extends Visitor {
…
  public void visit(ClassNode node){
    // get the class name from node 0
    String classname = node.getChildren().get(0).getData();
    // create a new table with the class name
    node.symtab = new SymTab(classname);
    // loop over all children of the class and migrate their
    // symbol table entries in class table
    for (Node member : node.getChildren()){
      if (member.symtabentry != null)
      node.symtab.addEntry(member.symtabentry);
    }
    // create the symbol table entry for the class
    node.symtabentry = new SymTabEntry("class:" + classname, node.symtab);
  }

…

  public void visit(StatBlockNode node){
    node.symtab = new SymTab();
    // add the symbol table entries of all the variables
    // declared in the statement block
    // they will later be picked-up by the higher-level
    // table creation
    for (Node stat : node.getChildren()){
      if (stat.symtabentry != null)
    node.symtab.addEntry(stat.symtabentry);
  }
…
}
```



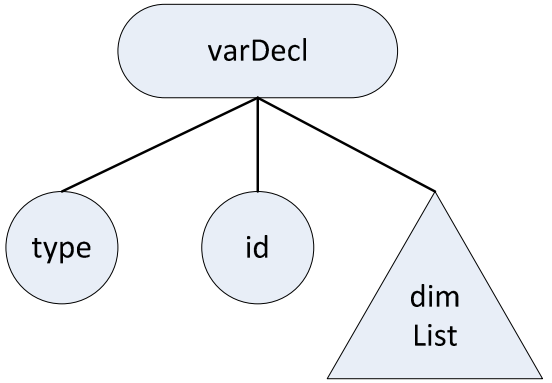


AST Visitor pattern : example

- Inside each scope, identifiers are declared
  - class: data/function members, function body: local variables, parameters
  - When we visit one of these node types:
    - Create a symbol table record
    - It will be picked-up by the parent node's semantic action when all its subtree will nave been traversed/processed.

AST Visitor pattern : example

```
public class SymTabCreationVisitor extends Visitor {
…
  public void visit(VarDeclNode node){
// aggregate information from the subtree
  String declrecstring;
  // identify what kind of record that is
  declrecstring = "localvar:";
  // get the type from the first child node and aggregate here
  declrecstring += node.getChildren().get(0).getData() + ':';
  // get the id from the second child node and aggregate here
  declrecstring += node.getChildren().get(1).getData() + ':';
  // loop over the list of dimension nodes and aggregate here
  for (Node dim : node.getChildren().get(2).getChildren())
    declrecstring += dim.getData() + ':';
  // create the symbol table entry for this variable
  // it will be picked-up by another node above later
  node.symtabentry = new SymTabEntry(declrecstring, null);
  }
```



AST visitors for semantic checking

Semantic checking

- Multiply declared identifiers
  - detected when we inserted the symbol table records
- Undeclared identifiers
  - As identifiers (variable or function) are encountered during the parsing of expressions, check if the identifier is declared in the current scope, or in the class it should be in
  - For function calls, 3 cases:
    - Function id does not exist in the current scope (or higher-level scope)
    - Number of parameters does not match the function defined in this scope
    - Type of expressions passes as parameters does not match the function defined in this scope
  - For variables, 3 cases:
    - Variable id does not exits in the current scope
    - Number of dimensions used does not match the number of dimensions as declared
    - An id uses the . operator on a member that is undefined in its class type
- Type checking
  - Operators' operand types are invalid
  - Type of expressions passed as parameters do not match the function's parameter type
  - Type of an assignment statement's right and left hand side do not match
  - Return statement does not match return type of the function

AST Visitor pattern : example

```
public class TypeCheckingVisitor extends Visitor {
…
  public void visit(AddOpNode node){
    String leftOperandType  = node.getChildren().get(0).getType();
    String rightOperandType = node.getChildren().get(1).getType();
    if( leftOperandType == rightOperandType )
      node.setType(leftOperandType);
    else{
      node.setType("typeerror");
      System.out.println("TYPE ERROR DETECTED between "
                         + node.getChildren().get(0).getData()
                         +  " and “
                         + node.getChildren().get(1).getData() );
    }
}

  public void visit(MultOpNode node){
    String leftOperandType  = node.getChildren().get(0).getType();
    String rightOperandType = node.getChildren().get(1).getType();
    if( leftOperandType == rightOperandType )
      node.setType(leftOperandType);
    else{
      node.setType("typeerror");
      System.out.println("TYPE ERROR DETECTED between "
                         + node.getChildren().get(0).getData()
                         +  " and "
                         + node.getChildren().get(1).getData());
  }
}
```

AST Visitor pattern: separate visitors

- You may separate various steps or phases as separate `ConcreteVisitors`.
- In the example code, there are two visitors:
  - `SymTabCreationVisitor`
    - Creates symbol tables and its entries
  - `TypeCheckingVisitor`
    - Implements a limited form of type inference on expression and assignment subtrees
  - `ConstructAssignmentAndExpressionStringVisitor`
    - Traverses an assignment subtree and re-creates its original form. Not required for project, but similar to what needs to be done for code generation later.
- When you implement the visitors, the semantic information you are gathering must be stored. In the example:

```
public abstract class Node {
  …
  // The following data members have been added
  // during the implementation of the visitors
  // These could be added using a decorator pattern
  // triggered by a visitor
  public String      type          = null;
  public String      subtreeString = null;
  public SymTab      symtab        = null;
  public SymTabEntry symtabentry   = null;
  …
}
```

References

- Wikipedia. *Symbol Table.*
  Wikipedia. *Visitor Pattern.*
- C.N. Fischer, R.K. Cytron, R.J. LeBlanc Jr., *Crafting a Compiler*, Adison-Wesley, 2009. Chapter 8.