VISITORS AND CASTING

CMPT 379 Lecture 9

Lecture Overview

- Visitors
- Casting as a Binary Operator

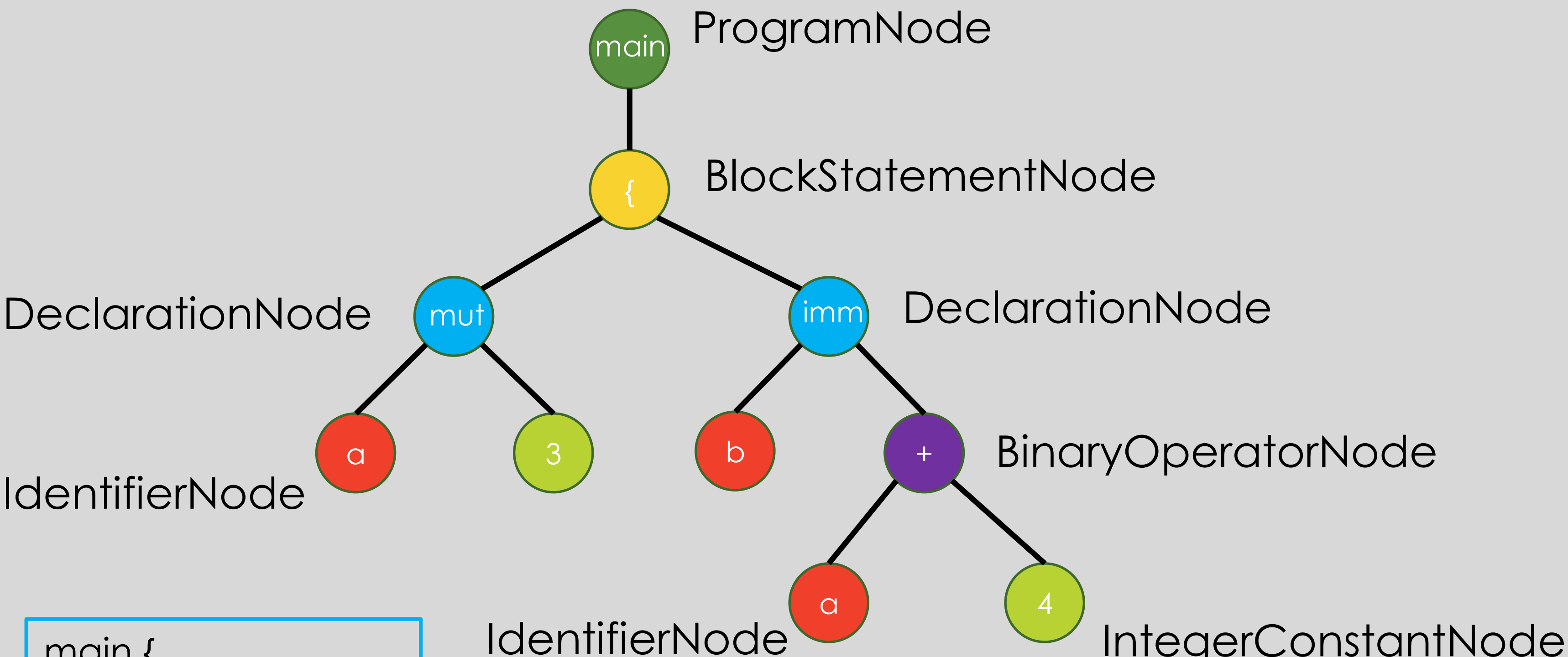
Visitors

- **Visitor** is a design pattern that is often used in applications that have to perform different operations on trees.
- It provides a way to traverse a tree, allowing the programmer to intervene (visit nodes) at different points along the way.
- In compilers, visitors are used to traverse parse trees or ASTs.
- In our compiler, we have a visitor for semantic analysis and a visitor for ASM code generation. In later checkpoints, we may add visitors in semantic analysis, so that it can do more than one traversal of the AST.

Visitors in Bilby

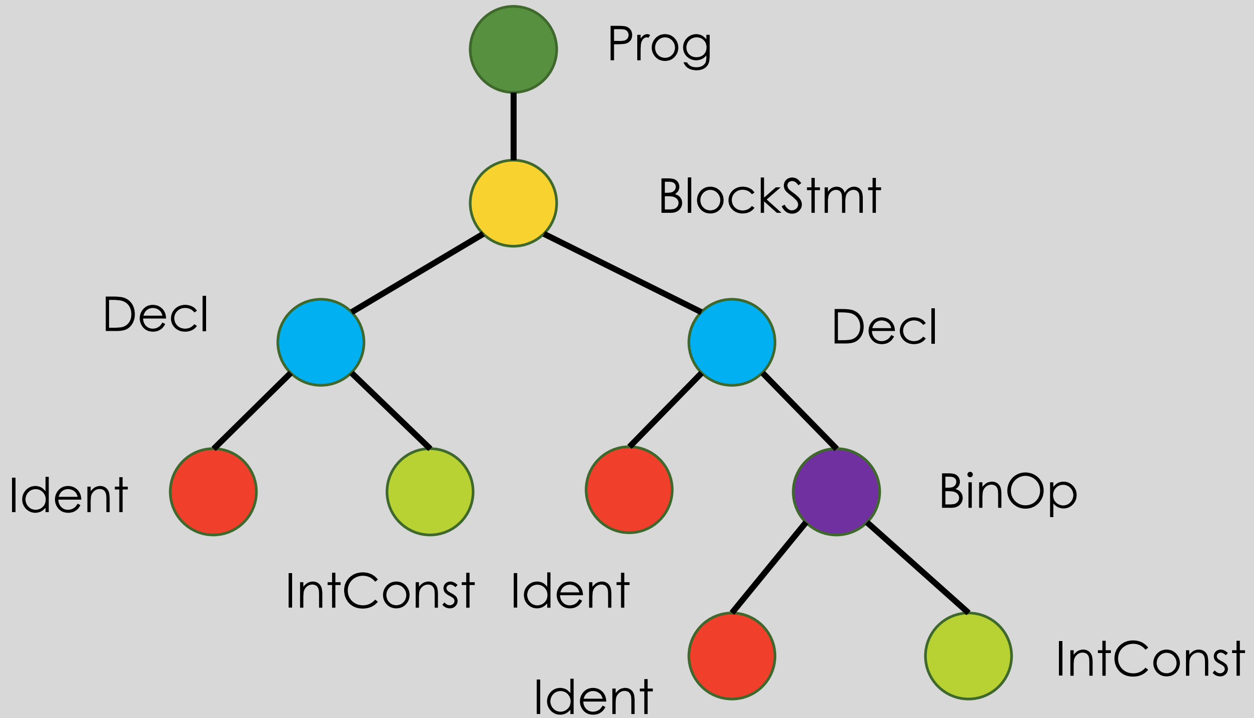
- When a Visitor's visit function is called on the root node of the AST, it initiates a depth-first left-to-right traversal of the AST.
- Hooks (in this case, functions) are present that allow you to visit a tree node either before its children or after its children (or both).
- Typically a node needs a visit before its children if it has something to communicate to its children. This occurs infrequently in compilers.
- Typically a node needs a visit after its children if its children have something to communicate to it. This happens frequently.
- In uses of the Visitor pattern in other applications, the traversal might be a different one.

A Bilby AST

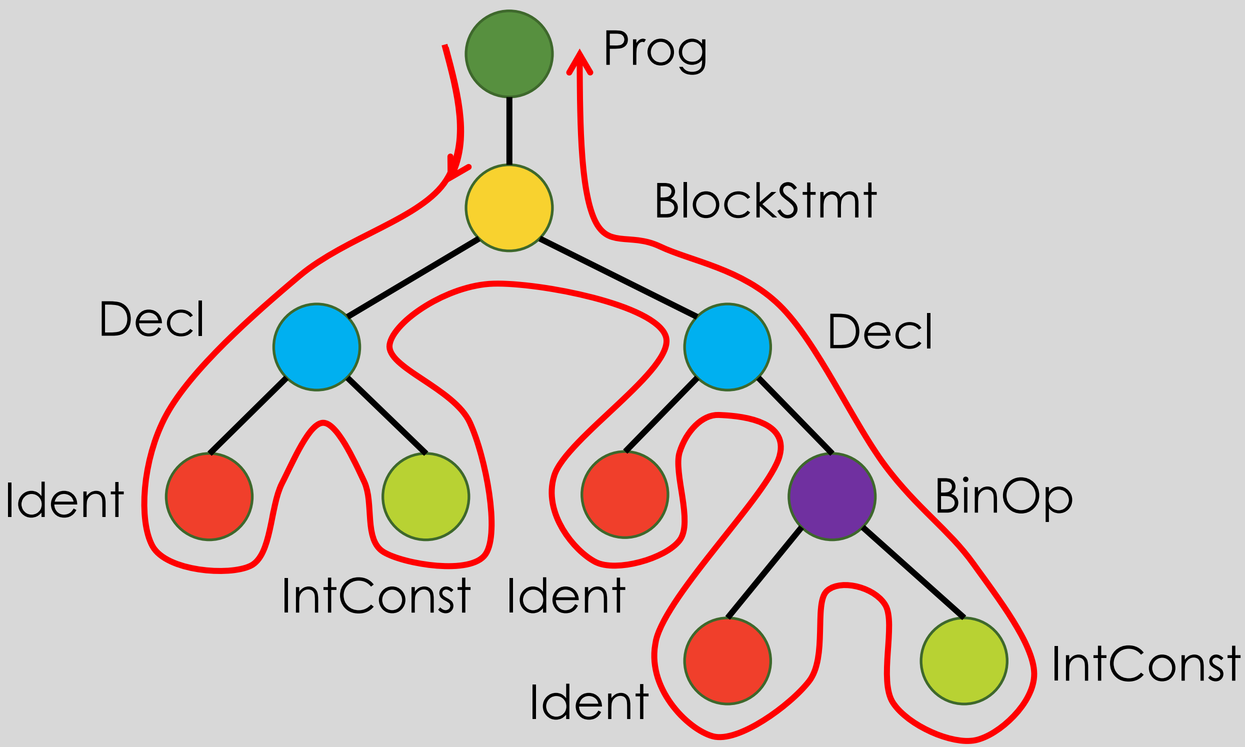



```
main {
   mut a := 3;
   imm b := a + 4;
}
```

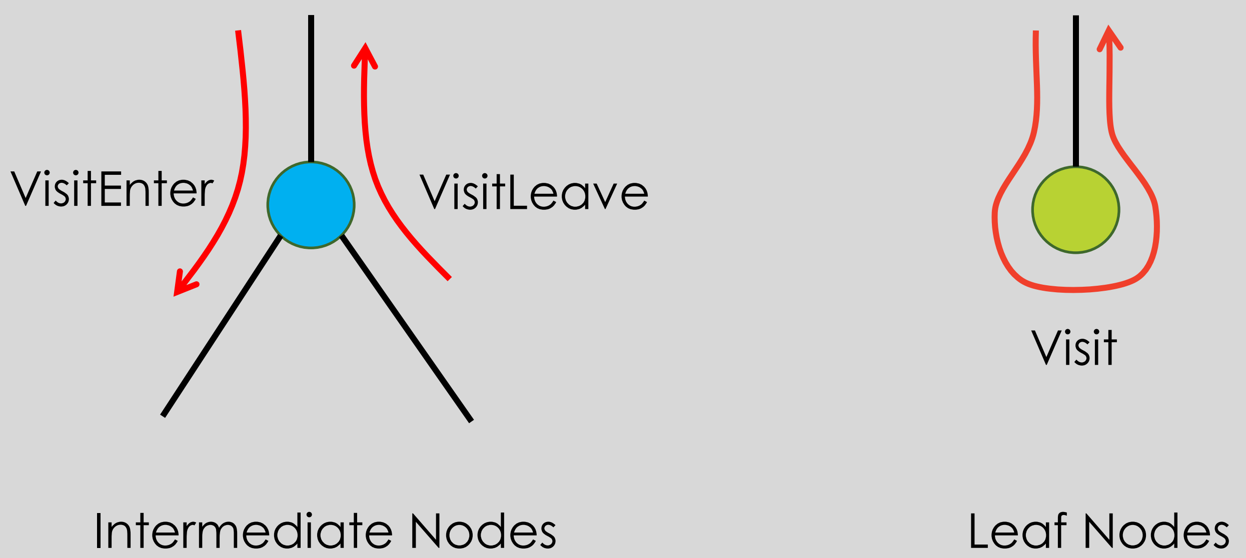
A Bilby AST


Prog
BlockStmt
Decl
Decl
Ident
BinOp
IntConst
Ident
Ident
IntConst

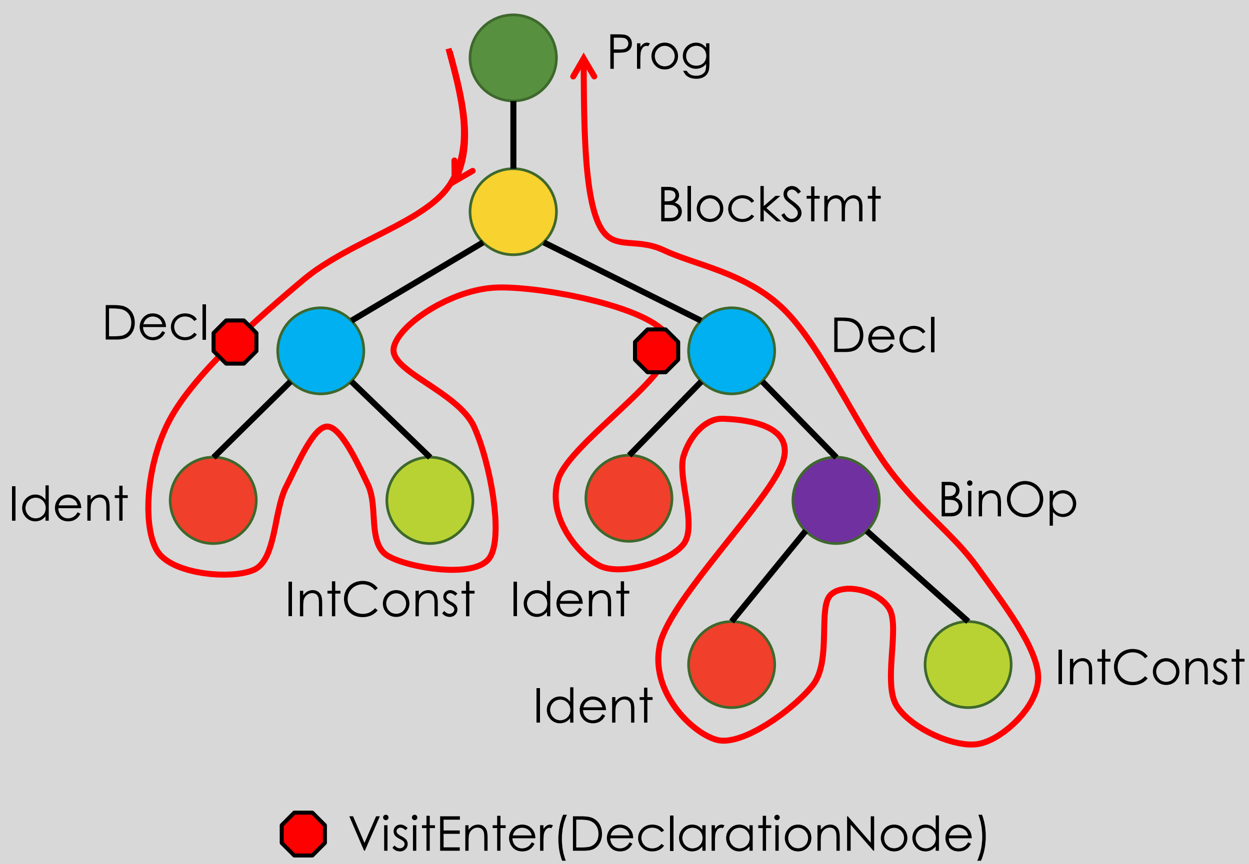
A depth-first left-to-right traversal



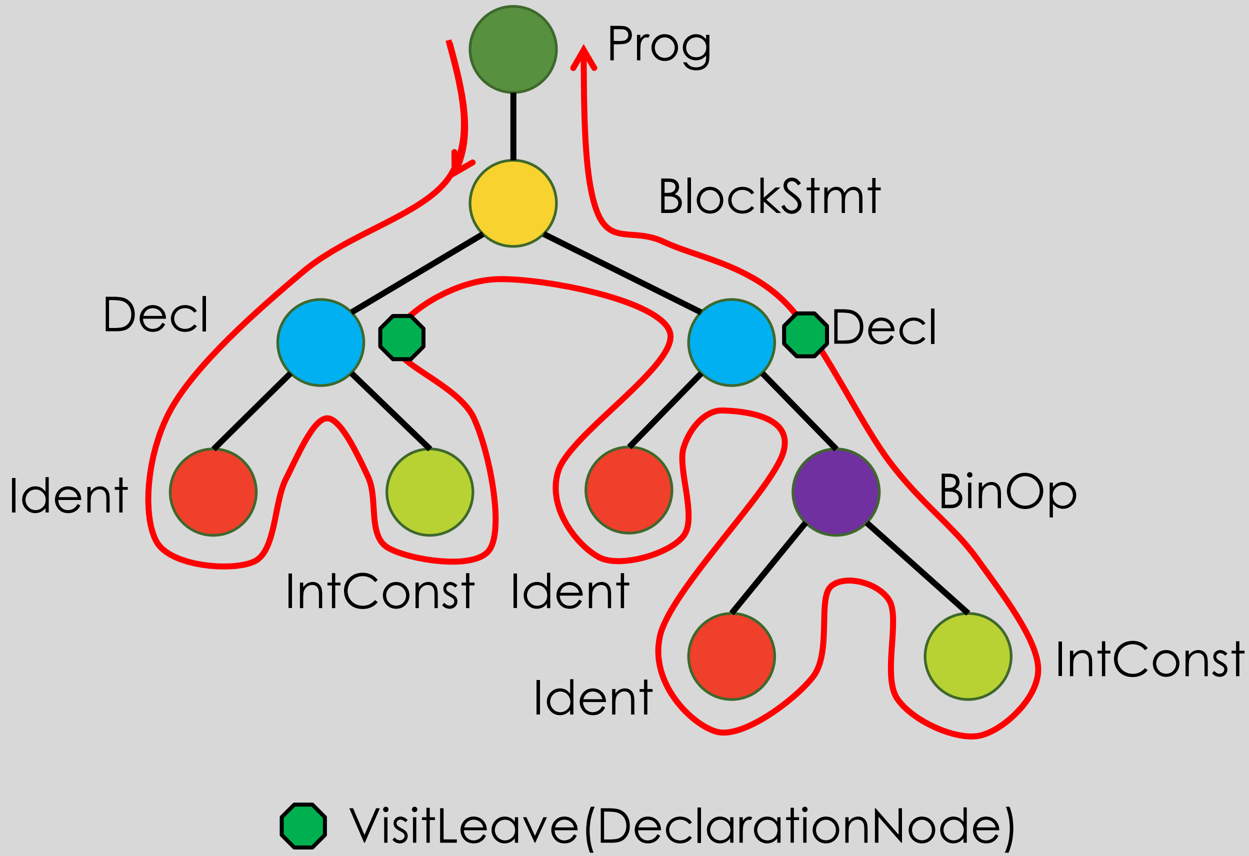
Visiting nodes



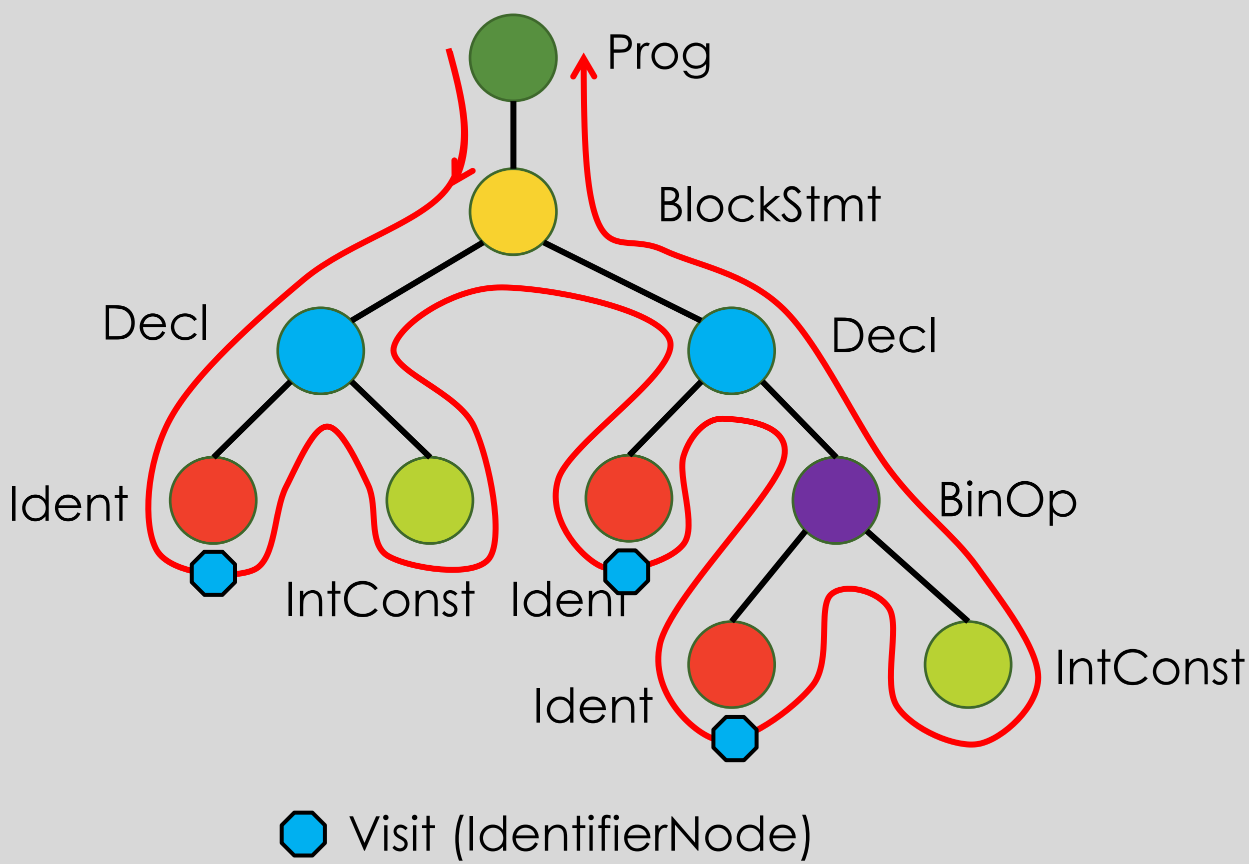
Visiting Node Types



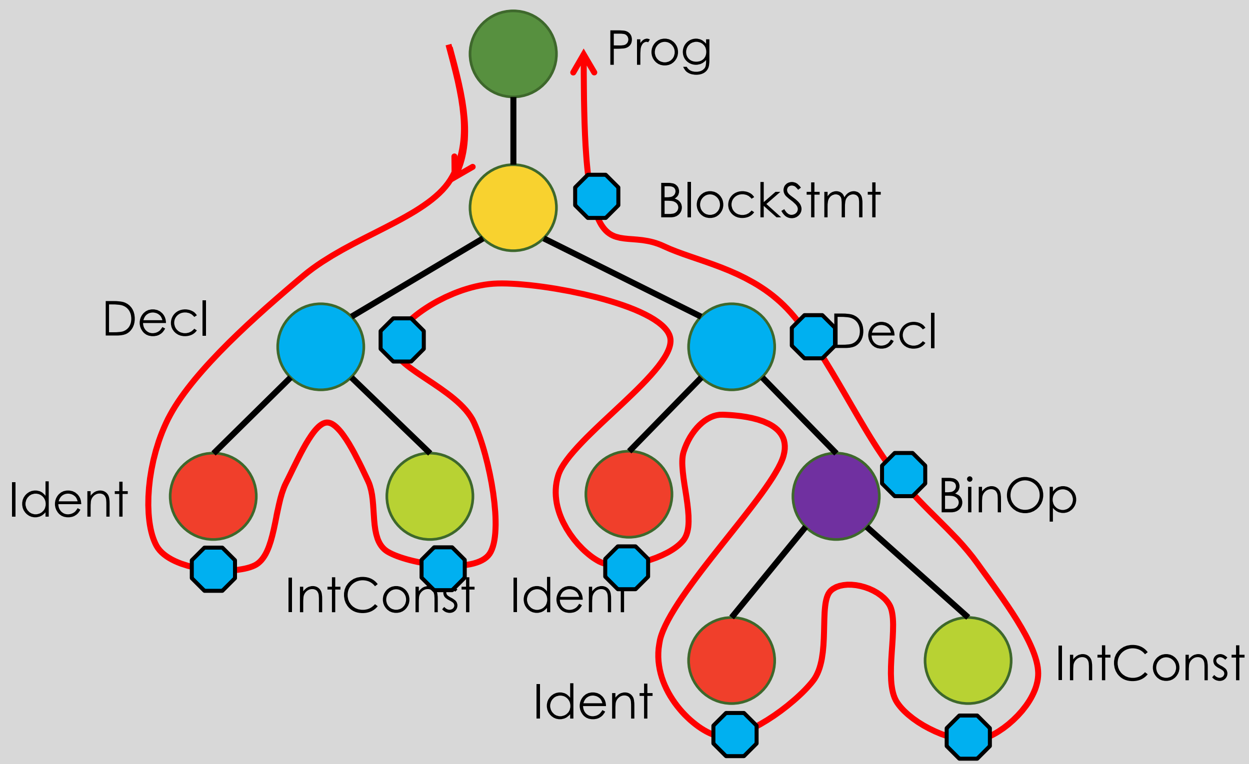
Visiting Node Types



Visiting Node Types

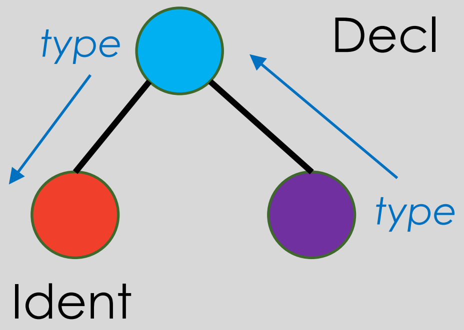


Typical Visitor



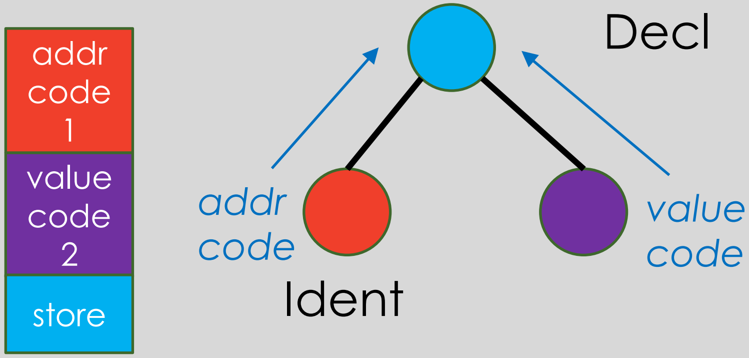
Sample VisitLeave: SemanticAnalyzer

```java
@Override
public void visitLeave(DeclarationNode node) {
    IdentifierNode identifier = (IdentifierNode) node.child(0);
    ParseNode initializer = node.child(1);

    Type declarationType = initializer.getType();
    node.setType(declarationType);

    identifier.setType(declarationType);
    addBinding(identifier, declarationType);
}
```



Sample VisitLeave: ASMCodeGenerator

```
public void visitLeave(DeclarationNode node) {
    newVoidCode(node);
    ASMCodeFragment lvalue = removeAddressCode(node.child(0));
    ASMCodeFragment rvalue = removeValueCode(node.child(1));

    code.append(lvalue);
    code.append(rvalue);

    Type type = node.getType();
    code.add(opcodeForStore(type));
}
```



The three types of code in the Bilby compiler

Void Code: This is code that does not change the stack from what it was before the code. It may use the stack for its computations, but it removes anything it puts on the stack. [...] -> [...]

Address Code: This is code that adds the address of a variable (a location where something is stored) to the top of the stack.
[...] -> [... addr]

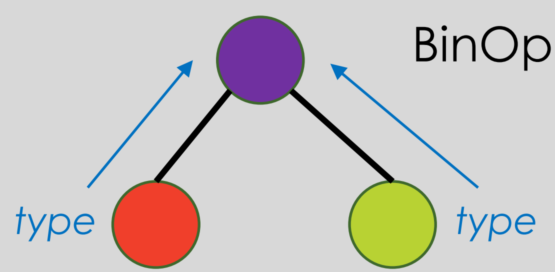
Value Code: This is code that adds the value of something to the top of the stack. [...] -> [... val]

Address Code can be converted to Value Code by asking for its value code. But no other conversions are possible.

Note: the value of a string is the address where its record is stored. So a string constant node creates value code.

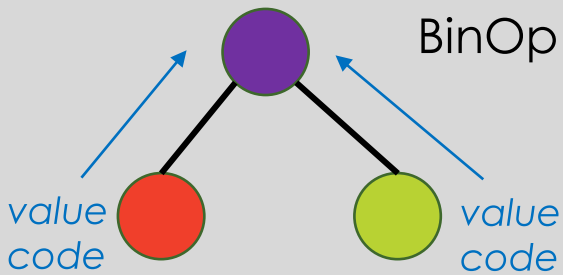
Casting as a BinaryOperator

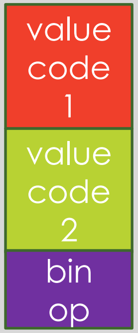
```
public void visitLeave(BinaryOperatorNode node) {
    assert node.nChildren() == 2;
    ParseNode left  = node.child(0);
    ParseNode right = node.child(1);
    List<Type> childTypes = Arrays.asList(left.getType(), right.getType());

    Lextant operator = operatorFor(node);

    FunctionSignatures signatures = FunctionSignatures.signaturesOf(operator);
    FunctionSignature signature = signatures.acceptingSignature(childTypes);

    if(!signature.isNull()) {
        node.setType(signature.resultType());
    }
    else {
        typeCheckError(node, childTypes);
        node.setType(PrimitiveType.ERROR);
    }
}
```



Casting as a BinaryOperator

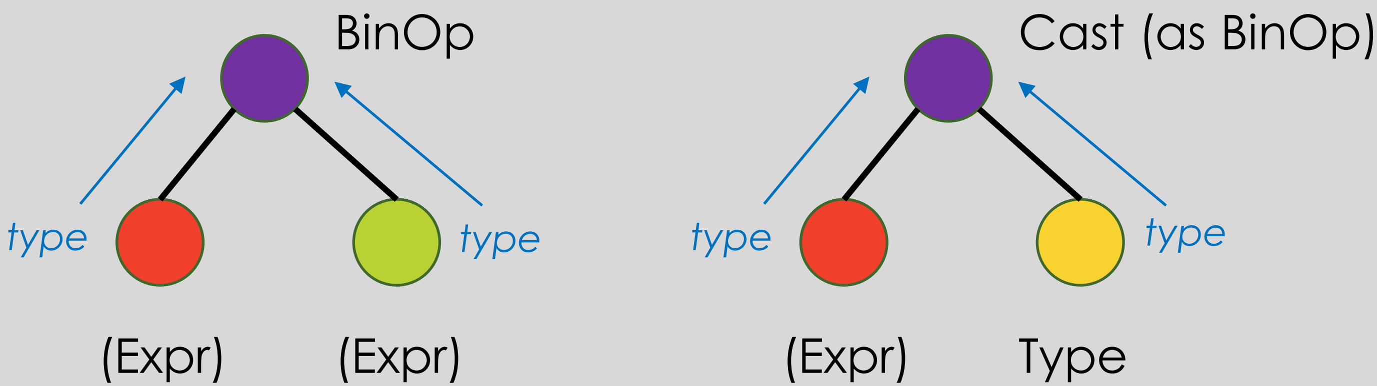
```
private void visitNormalBinaryOperatorNode(BinaryOperatorNode node) {
    newValueCode(node);
    ASMCodeFragment arg1 = removeValueCode(node.child(0));
    ASMCodeFragment arg2 = removeValueCode(node.child(1));

    code.append(arg1);
    code.append(arg2);

    Object variant = node.getSignature().getVariant();
    if(variant instanceof ASMOpcode) {
        ASMOpcode opcode = (ASMOpcode)variant;
        code.add(opcode);
    }
    else if(variant instanceof SimpleCodeGenerator) {
        SimpleCodeGenerator generator = (SimpleCodeGenerator)variant;
        ASMCodeFragment fragment = generator.generate(node);
        code.append(fragment);

        if(fragment.isAddress()) {
            code.markAsAddress();
        }
    }
}
```





Semantic Analysis: Casting

- We want to use FunctionSignatures to encode all the casting rules, because FunctionSignatures are easy (declarative programming).



- To do this, we should have TypeNodes set their type according to the token they have for type. (We'll have to generalize this a bit in future milestones.)

Semantic Analysis: Casting

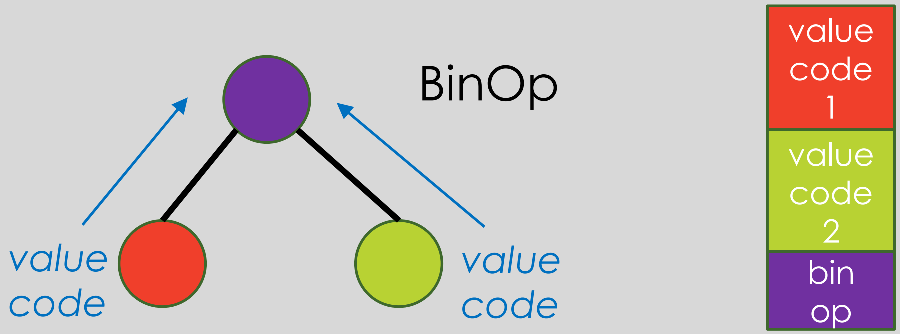
- In SemanticAnalysisVisitor:

```
public void visitLeave(TypeNode node) {
    node.setType(PrimitiveType.fromToken(node.typeToken()));
}
```

- I'll leave you to figure out PrimitiveType.fromToken(token). It can be a simple as a switch statement. A more robust way is to add a constructor argument to PrimitiveType which is the Lextant or Keyword associated with that type and search for the Lextant from the token.

Code Generation: Casting

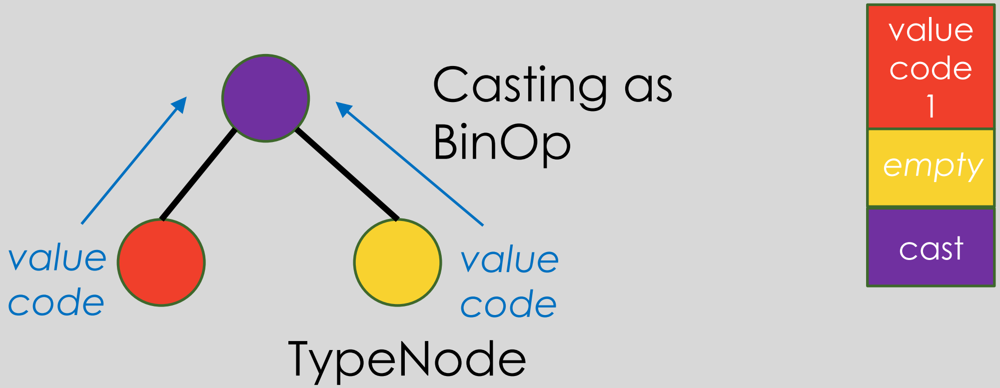
- Recall code generation for binary operators:



- For casting, we'll simply let the value code of a type node be no code at all. This is a violation of the meaning of Value Code, so it qualifies as a trick. We must be careful if in the future we get other uses of the nonterminal type (TypeNodes in places other than under a cast).

Code Generation: Casting

```
public void visitLeave(TypeNode node) {
    newValueCode(node);
}
```



- Now we just use the whichVariant field of the FunctionSignature for a cast to hold the operation(s) required for the cast.

FunctionSignatures: Casting

```
new FunctionSignatures(Punctuator.CAST,
    new FunctionSignature(ASMOpcode.Nop,
              BOOLEAN,   BOOLEAN,   BOOLEAN),
    new FunctionSignature(ASMOpcode.Nop,
              CHARACTER, CHARACTER, CHARACTER),
    new FunctionSignature(ASMOpcode.Nop,
              CHARACTER, INTEGER,   INTEGER),
    new FunctionSignature(new IntToBoolCodeGenerator(),
              INTEGER,   BOOLEAN,   BOOLEAN),
    new FunctionSignature(new IntToCharCodeGenerator(),
              INTEGER,   CHARACTER, CHARACTER),
    new FunctionSignature(ASMOpcode.ConvertF,
              INTEGER,   FLOATING,  FLOATING),
    …
```

Declarative Programming

- FunctionSignatures are a great example of Declarative Programming.
- In declarative programming, we declare things that work with general logic rather than building specific logic (procedural programming).
- Here, we're declaring FunctionSignatures. We're not writing logic as on the next slide:

Nondeclarative Programming

```
visitLeave(BinaryOperatorNode node)  {
    if(node.getToken.isLextant(Punctuator.ADD)) {
         ...
    }
     ...
    else if(node.getToken.isLextant(Punctuator.CAST))  {
        if(child[0].getType() == PrimitiveType.BOOLEAN) {
             if(child[1].getType()==PrimitiveType.BOOLEAN) {
                  node.setType(PrimitiveType.BOOLEAN);
             }
             else {
                  ...
             }
        }
        else if(child[0].getType() == PrimitiveType.CHARACTER) {
             if(child[1].getType() == PrimitiveType.CHARACTER) {
             ....
        }
```

This would become a huge morass of if-then-else statements.

It's difficult to understand and difficult to modify.