IFCC

Generated by Doxygen 1.9.8

Chapter 1

Welcome to the Documentation

1.1 Overview

This project implements a C compiler with a focus on generating Intermediate Representation (IR), performing code analysis, and generating assembly code. It includes functionalities such as syntax checking, control flow graph (CFG) generation, and code optimization.

1.2 Features

- **C Syntax Analysis:** Parses C source code and performs syntax checks.
- **Intermediate Representation (IR):** Generates and manipulates IR for code analysis.
- **Assembly Generation:** Converts IR into assembly code for various platforms.
- **CFG Generation:** Generates control flow graphs for visualizing program execution.
- **Code Checking:** Validates the code for errors and potential optimizations.

1.3 How to Use

To compile and run the compiler:

1. Clone or download the repository.
2. Set up the build environment.
3. Compile the source code using `make` or the appropriate build command.
4. Run the compiler with the C source file as an argument:
   `./ifcc path/to/file.c`

1.4 Dependencies

- **ANTLR:** For parsing C source code.
- **Graphviz:** For generating CFG visualizations.

1.5 Project Structure

- **src/:** Source code for the compiler.
- **include/:** Header files defining the compiler's functionality.
- **test/:** Test files for validating the compiler's correctness.
- **docs/:** Documentation for the project.
- **README.md:** Project overview and setup instructions.

1.6 License

This project is licensed under the BJAQPIG License.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

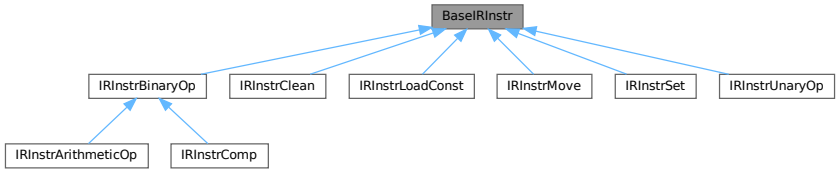
Class Documentation

4.1 BaseIRInstr Class Reference

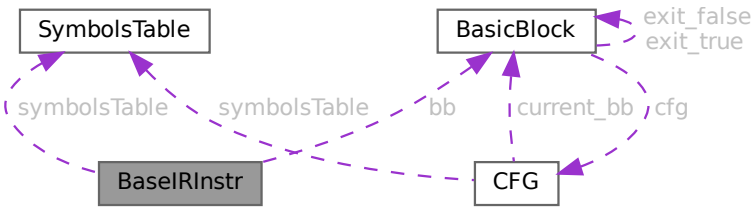
Represents a base class for intermediate representation instructions.

`#include <BaseIRInstr.h>`

Inheritance diagram for BaseIRInstr:



Collaboration diagram for BaseIRInstr:



Public Member Functions

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.
- virtual void gen_asm (ostream &o)=0

  Generates the assembly code for this instruction.

Protected Attributes

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.1.1 Detailed Description

Represents a base class for intermediate representation instructions.

This class serves as the base for all instruction types in the intermediate representation (IR). It provides a basic structure for handling assembly generation and access to the basic block that the instruction belongs to.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 BaseIRInstr()

```
BaseIRInstr::BaseIRInstr (
            BasicBlock * bb_ )
```

Constructs an instruction for a given basic block.

Initializes the instruction with the basic block it belongs to.

Parameters

*bb_*	The basic block to which this instruction belongs.

4.1.3 Member Function Documentation

4.1.3.1 gen_asm()

```
virtual void BaseIRInstr::gen_asm (
            ostream & o )  [pure virtual]
```

Generates the assembly code for this instruction.

This is a pure virtual function that must be implemented by derived classes to generate the specific assembly code for each type of instruction.

Parameters

o	The output stream where the generated assembly code will be written.

Implemented in IRInstrArithmeticOp, IRInstrUnaryOp, IRInstrComp, IRInstrLoadConst, IRInstrMove, and IRInstrBinaryOp.

4.1.3.2 getBB()

```
BasicBlock * BaseIRInstr::getBB ( )
```

Gets the basic block that this instruction belongs to.

Returns

The basic block associated with this instruction.

4.1.4 Member Data Documentation

4.1.4.1 bb

```
BasicBlock* BaseIRInstr::bb  [protected]
```

The basic block that this instruction belongs to.

4.1.4.2 symbolsTable

```
SymbolsTable* BaseIRInstr::symbolsTable  [protected]
```

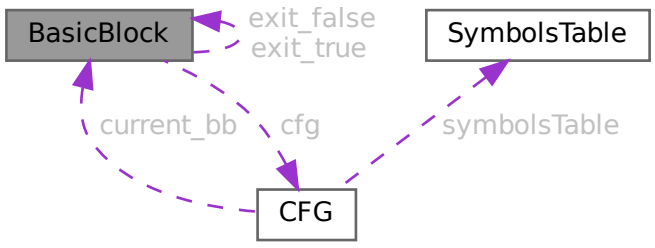
The symbol table for the current scope.

4.2 BasicBlock Class Reference

Represents a basic block in the control flow graph (CFG).

```
#include <BasicBlock.h>
```

Collaboration diagram for BasicBlock:



Public Member Functions

- BasicBlock (CFG *cfg, string entry_label)

  Constructs a BasicBlock with the given CFG and entry label.
- void gen_asm (ostream &o)

  Generates the assembly code for this basic block.
- void add_IRInstr (BaseIRInstr *instr)

  Adds an instruction to the basic block.
- CFG * getCFG ()

  Gets the CFG associated with this basic block.
- string getLabel ()

  Retrieves the label associated with the current block.
- vector< BaseIRInstr * > getInstr ()

  Retrieves the list of instructions within the current block.
- void setExitTrue (BasicBlock *bb)

  Sets the "true" exit point for the current block.
- void setExitFalse (BasicBlock *bb)

  Sets the "false" exit point for the current block.
- BasicBlock * getExitTrue ()

  Retrieves the "true" exit point of the current block.
- BasicBlock * getExitFalse ()

  Retrieves the "false" exit point of the current block.

Protected Attributes

- BasicBlock * exit_true
- BasicBlock * exit_false
- string label

  The label for the basic block, also used as the label in the generated assembly code.
- CFG * cfg

  The control flow graph to which this basic block belongs.
- vector< BaseIRInstr * > instrs

  A vector of instructions that belong to this basic block.

4.2.1 Detailed Description

Represents a basic block in the control flow graph (CFG).

A basic block is a sequence of instructions with a single entry point and a single exit point. This class is responsible for managing the instructions in the block and generating the assembly code.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 BasicBlock()

```
BasicBlock::BasicBlock (
            CFG * cfg,
            string entry_label )
```

Constructs a BasicBlock with the given CFG and entry label.

Initializes a new basic block with a label and associates it with a specific control flow graph (CFG).

Parameters

cfg	The CFG where this basic block belongs.
entry_label	The entry label for the basic block.

4.2.3 Member Function Documentation

4.2.3.1 add_IRInstr()

```
void BasicBlock::add_IRInstr (
            BaseIRInstr * instr )
```

Adds an instruction to the basic block.

This method adds a new intermediate representation instruction (IRInstr) to the basic block.

Parameters

instr	A pointer to the instruction to add.

4.2.3.2 gen_asm()

```
void BasicBlock::gen_asm (
            ostream & o )
```

Generates the assembly code for this basic block.

This method generates the x86 assembly code for all the instructions in the basic block.

Parameters

o	The output stream where the assembly code will be written.

4.2.3.3 getCFG()

```
CFG * BasicBlock::getCFG ( )
```

Gets the CFG associated with this basic block.

This method retrieves the CFG to which this basic block belongs.

Returns

A pointer to the CFG associated with this basic block.

4.2.3.4 getExitFalse()

```
BasicBlock * BasicBlock::getExitFalse ( )
```

Retrieves the "false" exit point of the current block.

This function returns the basic block representing the "false" exit point in the control flow. It is used to identify where the program flow should continue if a condition evaluates to false.

Returns

A pointer to the `BasicBlock` representing the "false" exit.

4.2.3.5 getExitTrue()

```
BasicBlock * BasicBlock::getExitTrue ( )
```

Retrieves the "true" exit point of the current block.

This function returns the basic block representing the "true" exit point in the control flow. It is used to identify where the program flow should continue if a condition evaluates to true.

Returns

A pointer to the `BasicBlock` representing the "true" exit.

4.2.3.6 getInstr()

```
vector< BaseIRInstr * > BasicBlock::getInstr ( )
```

Retrieves the list of instructions within the current block.

This function returns a vector containing all the instructions present in this block. Each instruction represents a basic operation in the program's flow, used in program analysis or code generation.

Returns

A vector of `BaseIRInstr *` representing the instructions in the block.

4.2.3.7 getLabel()

```
string BasicBlock::getLabel ( )
```

Retrieves the label associated with the current block.

This function returns the label associated with this particular control flow block. The label is typically used to uniquely identify this block in control flow analysis.

Returns

A string representing the label of this control flow block.

4.2.3.8 setExitFalse()

```
void BasicBlock::setExitFalse (
            BasicBlock * bb )
```

Sets the "false" exit point for the current block.

This function sets the block that serves as the "false" exit in the control flow of the program. This is typically used for conditional branches, where the program flow follows one path if a condition is true, and another path if it is false.

Parameters

bb	A pointer to the BasicBlock that represents the "false" exit of the current block.

4.2.3.9 setExitTrue()

```
void BasicBlock::setExitTrue (
            BasicBlock * bb )
```

Sets the "true" exit point for the current block.

This function sets the block that serves as the "true" exit in the control flow of the program. This is often used when there is a conditional branch and the flow of execution diverges depending on the outcome of a condition.

Parameters

bb	A pointer to the BasicBlock that represents the "true" exit of the current block.

4.2.4 Member Data Documentation

4.2.4.1 cfg

```
CFG* BasicBlock::cfg  [protected]
```

The control flow graph to which this basic block belongs.

4.2.4.2 exit_false

```
BasicBlock* BasicBlock::exit_false  [protected]
```

Pointer to the basic block representing the "false" exit. This is used for the branch or conditional statement when the condition is false. It can be null if no "false" exit exists.

4.2.4.3 exit_true

```
BasicBlock* BasicBlock::exit_true  [protected]
```

Pointer to the basic block representing the "true" exit. This is used for the branch or conditional statement when the condition is true. It can be null if no "true" exit exists.

4.2.4.4 instrs

```
vector<BaseIRInstr *> BasicBlock::instrs  [protected]
```

A vector of instructions that belong to this basic block.

4.2.4.5 label

```
string BasicBlock::label  [protected]
```

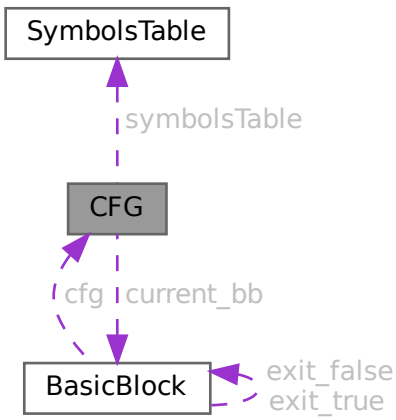
The label for the basic block, also used as the label in the generated assembly code.

4.3 CFG Class Reference

Represents a Control Flow Graph (CFG) in an Intermediate Representation (IR).

```
#include <CFG.h>
```

Collaboration diagram for CFG:



Public Member Functions

- CFG (string label, SymbolsTable *symbolsTable, int initialNextFreeSymbolIndex)

  Constructs a CFG with the given label and symbol index information.
- void add_bb (BasicBlock *bb)

  Adds a basic block to the control flow graph.
- void gen_asm (ostream &o)

  Generates the assembly code for the entire control flow graph.
- string create_new_tempvar (Type t)

  Creates a new temporary variable with a unique name.
- int get_var_index (string name)

  Retrieves the index of a variable by its name.
- Type get_var_type (string name)

  Retrieves the type of a variable based on its name.
- BasicBlock * getCurrentBasicBlock ()

  Retrieves the current basic block in the control flow graph.

- void setCurrentBasicBlock (BasicBlock *bb)

  Sets the current basic block in the control flow graph.
- void resetNextFreeSymbolIndex ()

  Resets the next free symbol index to its initial value.
- void gen_cfg_graphviz (ostream &o)

  Generates the Graphviz representation of the control flow graph (CFG).
- string getLabel ()

  Retrieves the label associated with the control flow graph (CFG).
- void setSymbolsTable (SymbolsTable *symbolsTable)
- SymbolsTable * getSymbolsTable ()

Protected Attributes

- SymbolsTable * symbolsTable

  The symbols table containing information about variables and their types.
- int nextFreeSymbolIndex

  The next available symbol index.
- const int initialTempPos

  The initial value for the next free symbol index.
- vector< BasicBlock * > bbs

  A vector containing all the basic blocks in the control flow graph.
- BasicBlock * current_bb

  A pointer to the current basic block being processed.
- string label

  The label associated with the control flow graph.

4.3.1 Detailed Description

Represents a Control Flow Graph (CFG) in an Intermediate Representation (IR).

A CFG consists of basic blocks and represents the flow of control in a program. This class is responsible for managing the basic blocks and generating assembly code corresponding to the control flow graph.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 CFG()

```
CFG::CFG (
            string label,
            SymbolsTable * symbolsTable,
            int initialNextFreeSymbolIndex )
```

Constructs a CFG with the given label and symbol index information.

Initializes the control flow graph with a label and symbol index, and sets the initial index for the next free symbol.

Parameters

label	The label for the CFG.
SymbolIndex	A map that maps symbol names to their respective indices.
SymbolType	A map that maps symbol names to their respective types.
initialNextFreeSymbolIndex	The initial value for the next free symbol index.

4.3.3 Member Function Documentation

4.3.3.1 add_bb()

```
void CFG::add_bb (
            BasicBlock * bb )
```

Adds a basic block to the control flow graph.

This method adds a new basic block to the vector of basic blocks that make up the CFG.

Parameters

bb	A pointer to the basic block to add.

4.3.3.2 create_new_tempvar()

```
string CFG::create_new_tempvar (
            Type t )
```

Creates a new temporary variable with a unique name.

This method generates a temporary variable of the specified type and returns its unique name.

Parameters

t	The type of the new temporary variable.

Returns

The name of the newly created temporary variable.

4.3.3.3 gen_asm()

```
void CFG::gen_asm (
            ostream & o )
```

Generates the assembly code for the entire control flow graph.

This method generates the assembly code for all basic blocks in the CFG.

Parameters

o	The output stream where the assembly code will be written.

4.3.3.4 gen_cfg_graphviz()

```
void CFG::gen_cfg_graphviz (
            ostream & o )
```

Generates the Graphviz representation of the control flow graph (CFG).

This function generates a Graphviz-compatible `.dot` file that visualizes the control flow of the program. The `.dot` file describes the nodes (representing basic blocks or instructions) and edges (representing the flow of control between the blocks) of the control flow graph.

The generated Graphviz representation is written to the provided output stream.

Parameters

o	The output stream to which the Graphviz `.dot` representation of the CFG is written. This is typically a file stream (e.g., `ofstream`) that writes to a `.dot` file.

4.3.3.5 get_var_index()

```
int CFG::get_var_index (
            string name )
```

Retrieves the index of a variable by its name.

This method retrieves the index of the variable in the symbol table.

Parameters

name	The name of the variable.

Returns

The index of the variable, or -1 if the variable does not exist.

4.3.3.6 get_var_type()

```
Type CFG::get_var_type (
            string name )
```

Retrieves the type of a variable based on its name.

This method looks up the variable's type from the symbols table using the provided variable name.

Parameters

name	The name of the variable whose type is to be retrieved.

Returns

The type of the specified variable.

4.3.3.7 getCurrentBasicBlock()

```
BasicBlock * CFG::getCurrentBasicBlock ( )
```

Retrieves the current basic block in the control flow graph.

This method returns a pointer to the current basic block being processed.

Returns

A pointer to the current basic block.

4.3.3.8 getLabel()

```
string CFG::getLabel ( )
```

Retrieves the label associated with the control flow graph (CFG).

This function returns a string label that represents the name or identifier associated with the current control flow graph. The label can be used to identify different parts of the program, such as functions or basic blocks.

The label is typically used for naming the nodes and edges in the Graphviz `.dot` representation or for other program analysis purposes.

Returns

A string representing the label of the control flow graph. This could be a function name, block identifier, or any other relevant label.

4.3.3.9 getSymbolsTable()

```
SymbolsTable * CFG::getSymbolsTable ( )  [inline]
```

Retrieves the symbols table associated with the control flow graph.

Returns

A pointer to the symbols table associated with the control flow graph.

4.3.3.10 resetNextFreeSymbolIndex()

```
void CFG::resetNextFreeSymbolIndex ( )
```

Resets the next free symbol index to its initial value.

This method resets the index for the next free symbol to its initial state.

4.3.3.11 setCurrentBasicBlock()

```
void CFG::setCurrentBasicBlock (
            BasicBlock * bb )
```

Sets the current basic block in the control flow graph.

This method sets the basic block that is currently being processed in the CFG.

Parameters

bb	A pointer to the basic block to set as the current block.

4.3.3.12 setSymbolsTable()

```
void CFG::setSymbolsTable (
            SymbolsTable * symbolsTable )
```

Sets the symbols table for the control flow graph.

Parameters

symbolsTable	A pointer to the symbols table to be set for the control flow graph.

4.3.4 Member Data Documentation

4.3.4.1 bbs

```
vector<BasicBlock *> CFG::bbs  [protected]
```

A vector containing all the basic blocks in the control flow graph.

4.3.4.2 current_bb

```
BasicBlock* CFG::current_bb  [protected]
```

A pointer to the current basic block being processed.

4.3.4.3 initialTempPos

```
const int CFG::initialTempPos  [protected]
```

The initial value for the next free symbol index.

4.3.4.4 label

```
string CFG::label  [protected]
```

The label associated with the control flow graph.

4.3.4.5 nextFreeSymbolIndex

```
int CFG::nextFreeSymbolIndex  [protected]
```

The next available symbol index.

4.3.4.6 symbolsTable

```
SymbolsTable* CFG::symbolsTable  [protected]
```

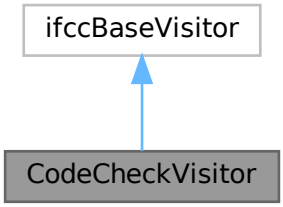
The symbols table containing information about variables and their types.

4.4 CodeCheckVisitor Class Reference

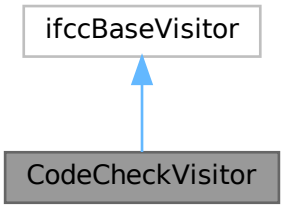
A visitor class for checking code correctness.

```
#include <CodeCheckVisitor.h>
```

Inheritance diagram for CodeCheckVisitor:



Collaboration diagram for CodeCheckVisitor:



Public Member Functions

- CodeCheckVisitor ()

  Constructs a new instance of the CodeCheckVisitor.
- virtual antlrcpp::Any visitProg (ifccParser::ProgContext *ctx) override

  Visits the program context in the parsed code.
- virtual antlrcpp::Any visitAssign_stmt (ifccParser::Assign_stmtContext *ctx) override

Visits an assignment statement in the parsed code.

- virtual antlrcpp::Any visitDecl_stmt (ifccParser::Decl_stmtContext *ctx) override

  Visits a declaration statement in the parsed code.
- virtual antlrcpp::Any visitExpr (ifccParser::ExprContext *expr)

  Visits any expression in the parsed code.
- virtual antlrcpp::Any visitAddsub (ifccParser::AddsubContext *ctx) override

  Visits an addition or subtraction expression.
- virtual antlrcpp::Any visitMuldiv (ifccParser::MuldivContext *ctx) override

  Visits a multiplication or division expression.
- virtual antlrcpp::Any visitBitwise (ifccParser::BitwiseContext *ctx) override

  Visits a bitwise operation expression.
- virtual antlrcpp::Any visitComp (ifccParser::CompContext *ctx) override

  Visits a comparison expression.
- virtual antlrcpp::Any visitUnary (ifccParser::UnaryContext *ctx) override

  Visits a unary expression.
- virtual antlrcpp::Any visitPre (ifccParser::PreContext *ctx) override

  Visits a pre-unary operation (e.g., prefix increment/decrement).
- virtual antlrcpp::Any visitPost (ifccParser::PostContext *ctx) override

  Visits a post-unary operation (e.g., postfix increment/decrement).
- virtual antlrcpp::Any visitBlock (ifccParser::BlockContext *ctx) override

  Visits a block of code in the parsed code.
- SymbolsTable * getCurrentSymbolsTable ()

  Gets the current symbols table.
- SymbolsTable * getRootSymbolsTable ()

  Gets the root symbols table.
- int **getCurrentOffset** ()

4.4.1 Detailed Description

A visitor class for checking code correctness.

This class extends the `ifccBaseVisitor` and is responsible for checking various aspects of code correctness, including variable declarations, assignments, and expressions, during the parsing phase of the compiler.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 CodeCheckVisitor()

```
CodeCheckVisitor::CodeCheckVisitor ( )
```

Constructs a new instance of the CodeCheckVisitor.

4.4.3 Member Function Documentation

4.4.3.1 getCurrentSymbolsTable()

```
SymbolsTable * CodeCheckVisitor::getCurrentSymbolsTable ( )  [inline]
```

Gets the current symbols table.

Returns

The current symbols table.

4.4.3.2 getRootSymbolsTable()

```
SymbolsTable * CodeCheckVisitor::getRootSymbolsTable ( )  [inline]
```

Gets the root symbols table.

Returns

The root symbols table.

4.4.3.3 visitAddsub()

```
antlrcpp::Any CodeCheckVisitor::visitAddsub (
            ifccParser::AddsubContext * ctx )  [override], [virtual]
```

Visits an addition or subtraction expression.

This method processes expressions involving addition or subtraction and checks for correctness in terms of variable usage.

Parameters

ctx	The context for the addition or subtraction expression.

Returns

A result of the visit, typically unused.

4.4.3.4 visitAssign_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitAssign_stmt (
            ifccParser::Assign_stmtContext * ctx )  [override], [virtual]
```

Visits an assignment statement in the parsed code.

This method checks whether variables are assigned values correctly and whether the variables involved are defined.

Parameters

ctx	The context for the assignment statement.

Returns

A result of the visit, typically unused.

4.4.3.5 visitBitwise()

```
antlrcpp::Any CodeCheckVisitor::visitBitwise (
            ifccParser::BitwiseContext * ctx )  [override], [virtual]
```

Visits a bitwise operation expression.

This method processes bitwise operations like AND, OR, XOR, etc., and ensures that all variables in these operations are declared.

Parameters

ctx	The context for the bitwise operation expression.

Returns

A result of the visit, typically unused.

4.4.3.6 visitBlock()

```
antlrcpp::Any CodeCheckVisitor::visitBlock (
            ifccParser::BlockContext * ctx )  [override], [virtual]
```

Visits a block of code in the parsed code.

This method ensures correct variable scoping and handles nested blocks properly.

Parameters

ctx	The context for the block of code.

Returns

A result of the visit, typically unused.

4.4.3.7 visitComp()

```
antlrcpp::Any CodeCheckVisitor::visitComp (
            ifccParser::CompContext * ctx )  [override], [virtual]
```

Visits a comparison expression.

This method processes comparison operations like equality, inequality, greater than, less than, etc., and checks for any correctness issues regarding the variables used.

Parameters

ctx	The context for the comparison expression.

Returns

A result of the visit, typically unused.

4.4.3.8 visitDecl_stmt()

```
antlrcpp::Any CodeCheckVisitor::visitDecl_stmt (
            ifccParser::Decl_stmtContext * ctx )  [override], [virtual]
```

Visits a declaration statement in the parsed code.

This method ensures that the variable being declared is correctly handled, ensuring no variable is declared multiple times, or is used before being declared.

Parameters

ctx	The context for the declaration statement.

Returns

A result of the visit, typically unused.

4.4.3.9 visitExpr()

```
antlrcpp::Any CodeCheckVisitor::visitExpr (
            ifccParser::ExprContext * expr )  [virtual]
```

Visits any expression in the parsed code.

This method processes expressions and checks whether variables in expressions are valid and properly declared.

Parameters

expr	The expression context to check.

Returns

A result of the visit, typically unused.

4.4.3.10 visitMuldiv()

```
antlrcpp::Any CodeCheckVisitor::visitMuldiv (
            ifccParser::MuldivContext * ctx )  [override], [virtual]
```

Visits a multiplication or division expression.

This method processes multiplication and division expressions and checks whether variables involved are correctly declared.

Parameters

ctx	The context for the multiplication or division expression.

Returns

A result of the visit, typically unused.

4.4.3.11 visitPost()

```
antlrcpp::Any CodeCheckVisitor::visitPost (
            ifccParser::PostContext * ctx )  [override], [virtual]
```

Visits a post-unary operation (e.g., postfix increment/decrement).

This method processes post-unary operations and checks for correctness in usage.

Parameters

ctx	The context for the post-unary expression.

Returns

A result of the visit, typically unused.

4.4.3.12 visitPre()

```
antlrcpp::Any CodeCheckVisitor::visitPre (
            ifccParser::PreContext * ctx )  [override], [virtual]
```

Visits a pre-unary operation (e.g., prefix increment/decrement).

This method checks for correctness in expressions involving pre-unary operations.

Parameters

ctx	The context for the pre-unary expression.

Returns

A result of the visit, typically unused.

4.4.3.13 visitProg()

```
antlrcpp::Any CodeCheckVisitor::visitProg (
            ifccParser::ProgContext * ctx )  [override], [virtual]
```

Visits the program context in the parsed code.

This method performs a global check of the program, ensuring all variables are declared and used properly.

Parameters

ctx	The context for the program.

Returns

A result of the visit, typically unused.

4.4.3.14 visitUnary()

```
antlrcpp::Any CodeCheckVisitor::visitUnary (
            ifccParser::UnaryContext * ctx )  [override], [virtual]
```

Visits a unary expression.

This method processes unary expressions (e.g., negation or logical NOT) and ensures that all variables in the expression are valid.

Parameters

ctx	The context for the unary expression.

Returns

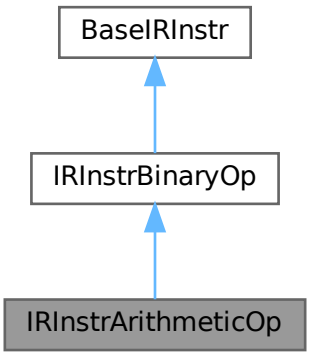
A result of the visit, typically unused.

4.5 IRInstrArithmeticOp Class Reference

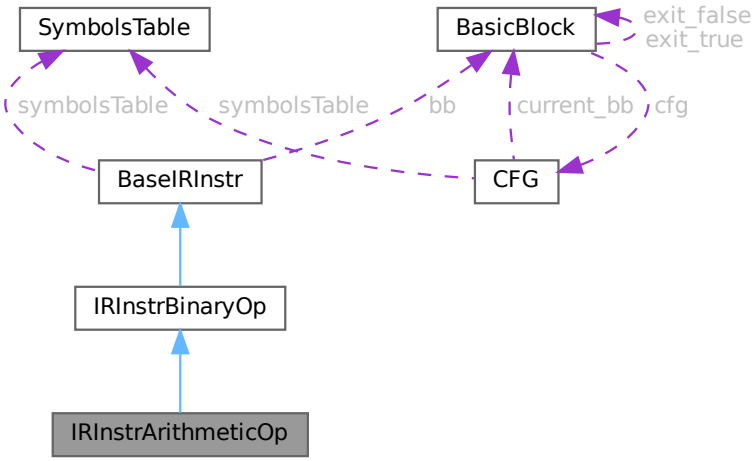
Represents an arithmetic operation instruction in the intermediate representation.

`#include <IRInstrArithmeticOp.h>`

Inheritance diagram for IRInstrArithmeticOp:



Collaboration diagram for IRInstrArithmeticOp:



Public Member Functions

- IRInstrArithmeticOp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs an arithmetic operation instruction.
- virtual void gen_asm (ostream &o)

  Generates the assembly code for this arithmetic operation instruction.

Public Member Functions inherited from IRInstrBinaryOp

- IRInstrBinaryOp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs a binary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from IRInstrBinaryOp

- string firstOp

  The first operand for the binary operation.
- string secondOp

  The second operand for the binary operation.
- string op

  The binary operation (e.g., '+', '-', '', '/',").*

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.5.1 Detailed Description

Represents an arithmetic operation instruction in the intermediate representation.

This class handles the generation of intermediate representation instructions for arithmetic operations, such as addition, subtraction, multiplication, division, modulo, and bitwise operations. It extends the `IRInstrBinaryOp` class and provides specialized methods for handling these operations.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 IRInstrArithmeticOp()

```
IRInstrArithmeticOp::IRInstrArithmeticOp (
            BasicBlock * bb_,
            string firstOp,
            string secondOp,
            string op )  [inline]
```

Constructs an arithmetic operation instruction.

Initializes the instruction with a basic block, two operands, and the arithmetic operation.

Parameters

*bb_*	The basic block to which the instruction belongs.
firstOp	The first operand of the arithmetic operation.
secondOp	The second operand of the arithmetic operation.
op	The arithmetic operation (e.g., '+', '-', '*', '/', '').

4.5.3 Member Function Documentation

4.5.3.1 gen_asm()

```
void IRInstrArithmeticOp::gen_asm (
            ostream & o )  [virtual]
```

Generates the assembly code for this arithmetic operation instruction.

This method generates the appropriate assembly code based on the specific arithmetic operation (e.g., addition, subtraction, multiplication, division, modulo, or bitwise operations).

Parameters

o	The output stream where the generated assembly code will be written.

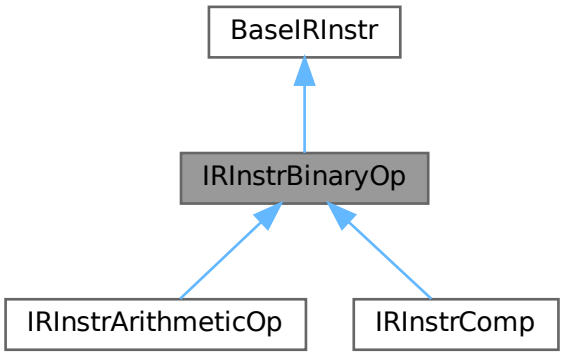
Implements IRInstrBinaryOp.

4.6 IRInstrBinaryOp Class Reference

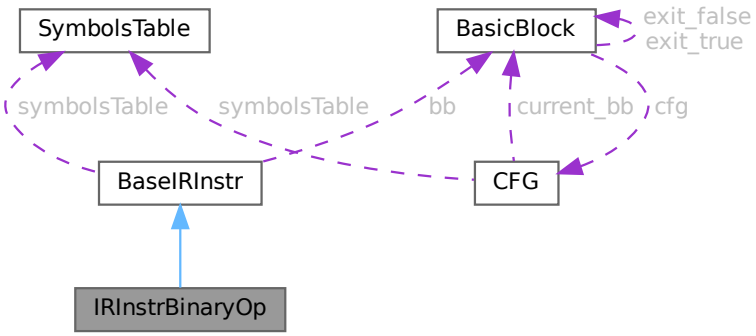
Represents a binary operation instruction in the intermediate representation.

`#include <IRInstrBinaryOp.h>`

Inheritance diagram for IRInstrBinaryOp:



Collaboration diagram for IRInstrBinaryOp:



Public Member Functions

- IRInstrBinaryOp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs a binary operation instruction.
- virtual void gen_asm (ostream &o)=0

  Generates the assembly code for this binary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Protected Attributes

- string firstOp

  The first operand for the binary operation.
- string secondOp

  The second operand for the binary operation.
- string op

  The binary operation (e.g., '+', '-', '', '/',").*

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.6.1 Detailed Description

Represents a binary operation instruction in the intermediate representation.

This class serves as the base class for binary operation instructions such as addition, subtraction, multiplication, etc. It provides a structure for managing two operands and the operation itself, and it also handles the generation of the corresponding assembly code for binary operations.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 IRInstrBinaryOp()

```
IRInstrBinaryOp::IRInstrBinaryOp (
            BasicBlock * bb_,
            string firstOp,
            string secondOp,
            string op )  [inline]
```

Constructs a binary operation instruction.

Initializes the instruction with a basic block, two operands, and the binary operation.

Parameters

*bb_*	The basic block to which the instruction belongs.
firstOp	The first operand of the binary operation.
secondOp	The second operand of the binary operation.
op	The binary operation (e.g., '+', '-', '*', '/').

4.6.3 Member Function Documentation

4.6.3.1 gen_asm()

```
virtual void IRInstrBinaryOp::gen_asm (
            ostream & o )  [pure virtual]
```

Generates the assembly code for this binary operation instruction.

This method must be implemented by derived classes to generate the appropriate assembly code based on the specific binary operation.

Parameters

o	The output stream where the generated assembly code will be written.

Implements BaseIRInstr.

Implemented in IRInstrArithmeticOp, and IRInstrComp.

4.6.4 Member Data Documentation

4.6.4.1 firstOp

```
string IRInstrBinaryOp::firstOp  [protected]
```

The first operand for the binary operation.

4.6.4.2 op

```
string IRInstrBinaryOp::op  [protected]
```

The binary operation (e.g., '+', '-', '*', '/','').

4.6.4.3 secondOp

```
string IRInstrBinaryOp::secondOp  [protected]
```

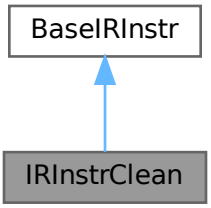
The second operand for the binary operation.

4.7 IRInstrClean Class Reference

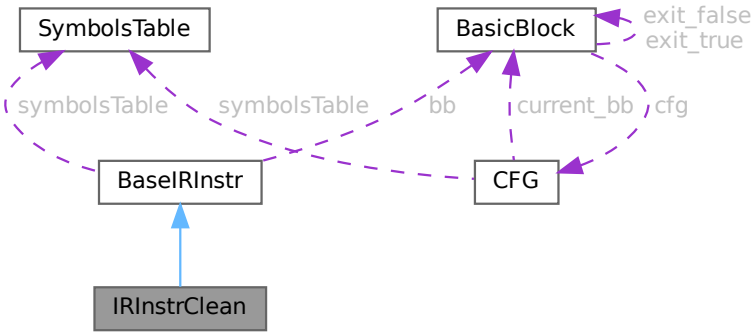
Represents a clean-up instruction in the intermediate representation.

`#include <IRInstrClean.h>`

Inheritance diagram for IRInstrClean:



Collaboration diagram for IRInstrClean:



Public Member Functions

- IRInstrClean (BasicBlock *bb_)

  Constructs an IRInstrClean object.
- virtual void gen_asm (std::ostream &o) override

  Generates assembly code for the IRInstrClean instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.
- virtual void gen_asm (ostream &o)=0

  Generates the assembly code for this instruction.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.7.1 Detailed Description

Represents a clean-up instruction in the intermediate representation.

The `IRInstrClean` class is a subclass of `BaseIRInstr`. It represents a clean-up instruction, typically used to deallocate resources or reset values within a basic block during intermediate representation generation. The main responsibility of this class is to generate the corresponding assembly code for the clean-up operation.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 IRInstrClean()

```
IRInstrClean::IRInstrClean (
            BasicBlock * bb_ )  [inline]
```

Constructs an IRInstrClean object.

This constructor initializes an `IRInstrClean` instance with a reference to the basic block where this clean-up instruction resides.

Parameters

*bb_*	A pointer to the `BasicBlock` to which this instruction belongs.

4.7.3 Member Function Documentation

4.7.3.1 gen_asm()

```
void IRInstrClean::gen_asm (
            std::ostream & o )  [override], [virtual]
```

Generates assembly code for the IRInstrClean instruction.

This function generates the assembly code corresponding to the clean-up operation represented by this instruction and writes it to the provided output stream.

The generated assembly code typically involves operations to reset, deallocate, or clean up resources associated with the instruction.

Parameters

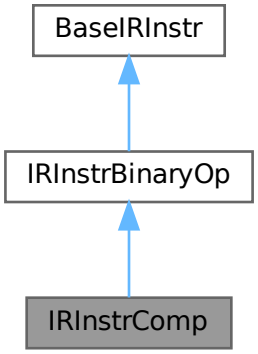
o	The output stream to which the generated assembly code will be written.

4.8 IRInstrComp Class Reference

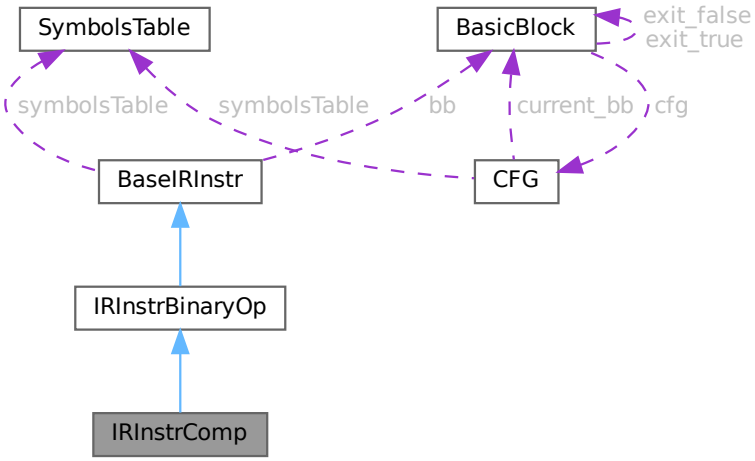
Represents a comparison operation instruction in the intermediate representation.

`#include <IRInstrComp.h>`

Inheritance diagram for IRInstrComp:



Collaboration diagram for IRInstrComp:



Public Member Functions

- IRInstrComp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs a comparison operation instruction.
- virtual void gen_asm (ostream &o) override

  Generates the assembly code for this comparison operation instruction.

Public Member Functions inherited from IRInstrBinaryOp

- IRInstrBinaryOp (BasicBlock *bb_, string firstOp, string secondOp, string op)

  Constructs a binary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from IRInstrBinaryOp

- string firstOp

  The first operand for the binary operation.
- string secondOp

  The second operand for the binary operation.
- string op

  The binary operation (e.g., '+', '-', '', '/',").*

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.8.1 Detailed Description

Represents a comparison operation instruction in the intermediate representation.

This class handles the generation of intermediate representation instructions for comparison operations, such as equality, inequality, greater than, greater than or equal to, less than, and less than or equal to.

4.8.2 Constructor & Destructor Documentation

4.8.2.1 IRInstrComp()

```
IRInstrComp::IRInstrComp (
            BasicBlock * bb_,
            string firstOp,
            string secondOp,
            string op )  [inline]
```

Constructs a comparison operation instruction.

Initializes the instruction with a basic block, two operands, and the comparison operation.

Parameters

*bb_*	The basic block to which the instruction belongs.
firstOp	The first operand of the comparison operation.
secondOp	The second operand of the comparison operation.
op	The comparison operation (e.g., '==', '!=', '>', '<', '>=', '<=').

4.8.3 Member Function Documentation

4.8.3.1 gen_asm()

```
void IRInstrComp::gen_asm (
            ostream & o )  [override], [virtual]
```

Generates the assembly code for this comparison operation instruction.

This method generates the appropriate assembly code based on the specific comparison operation (e.g., equality, inequality, greater than, greater than or equal to, less than, or less than or equal to).

Parameters

o	The output stream where the generated assembly code will be written.

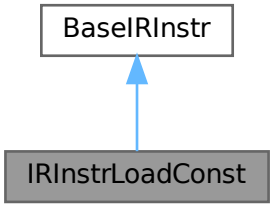
Implements IRInstrBinaryOp.

4.9 IRInstrLoadConst Class Reference

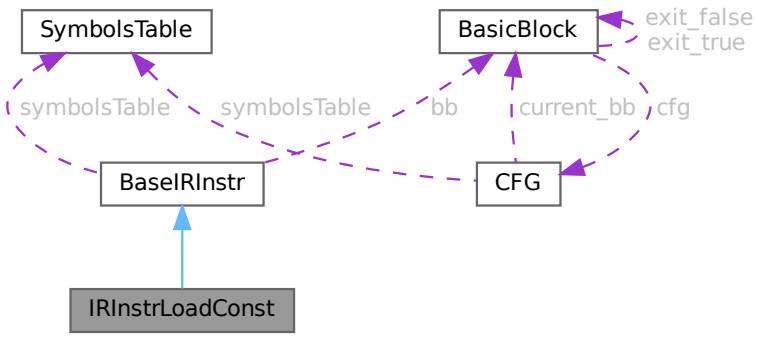
Represents an IR instruction for loading a constant into memory or a register.

```
#include <IRInstrLoadConst.h>
```

Inheritance diagram for IRInstrLoadConst:



Collaboration diagram for IRInstrLoadConst:



Public Member Functions

- IRInstrLoadConst (BasicBlock *bb_, int value, string dest)

  Constructor for the IRInstrLoadConst instruction.
- virtual void gen_asm (ostream &o) override

  Generates assembly code to load a constant.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.9.1 Detailed Description

Represents an IR instruction for loading a constant into memory or a register.

This instruction is used to assign an immediate value to a register or memory location.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 IRInstrLoadConst()

```
IRInstrLoadConst::IRInstrLoadConst (
            BasicBlock * bb_,
            int value,
            string dest )
```

Constructor for the IRInstrLoadConst instruction.

Initializes the instruction with the basic block, the constant value to load, and the destination register or memory variable where the value should be stored.

Parameters

*bb_*	Pointer to the basic block containing this instruction.
value	The constant value to load.
dest	The name of the target register or memory variable.

4.9.3 Member Function Documentation

4.9.3.1 gen_asm()

```
void IRInstrLoadConst::gen_asm (
            ostream & o )  [override], [virtual]
```

Generates assembly code to load a constant.

Generates assembly code to load a constant into memory or a register.

Generates assembly code for loading a constant value into a register or memory.

Parameters

o	Output stream where the assembly code will be written.

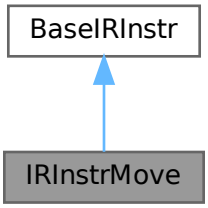
Implements BaseIRInstr.

4.10 IRInstrMove Class Reference

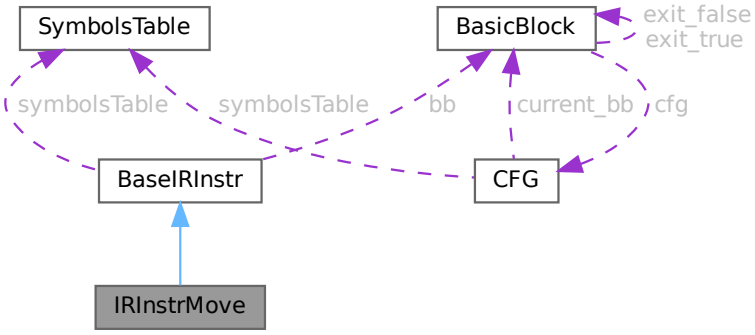
Represents an IR instruction for moving a value between registers and memory.

```
#include <IRInstrMove.h>
```

Inheritance diagram for IRInstrMove:



Collaboration diagram for IRInstrMove:



Public Member Functions

- IRInstrMove (BasicBlock *bb_, string src, string dest)

  Constructor for the IRInstrMove instruction.
- virtual void gen_asm (ostream &o) override

  Generates the assembly code corresponding to the move instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.10.1 Detailed Description

Represents an IR instruction for moving a value between registers and memory.

This class handles data transfers between registers and the stack but does not manage constants.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 IRInstrMove()

```
IRInstrMove::IRInstrMove (
            BasicBlock * bb_,
            string src,
            string dest )
```

Constructor for the IRInstrMove instruction.

Initializes the instruction with the basic block, source, and destination variables.

Parameters

*bb_*	Pointer to the basic block containing this instruction.
src	The name of the source variable (register or memory location).
dest	The name of the destination variable (register or memory location).

4.10.3 Member Function Documentation

4.10.3.1 gen_asm()

```
void IRInstrMove::gen_asm (
            ostream & o )  [override], [virtual]
```

Generates the assembly code corresponding to the move instruction.

Generates assembly code for moving a value between registers and memory.

This method generates the appropriate assembly code for moving a value from the source variable to the destination variable.

Parameters

o	Output stream where the assembly code will be written.

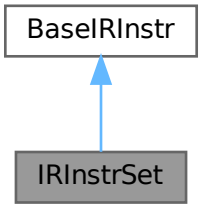
Implements BaseIRInstr.

4.11 IRInstrSet Class Reference

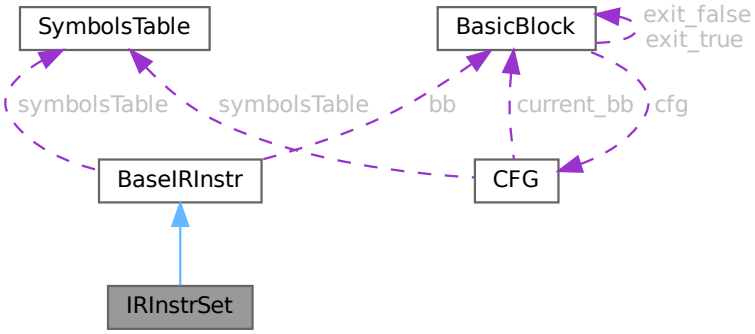
Represents an instruction that sets a value in the intermediate representation.

`#include <IRInstrSet.h>`

Inheritance diagram for IRInstrSet:



Collaboration diagram for IRInstrSet:



Public Member Functions

- IRInstrSet (BasicBlock *bb_)

  Constructs an IRInstrSet object.
- virtual void gen_asm (std::ostream &o) override

  Generates assembly code for the IRInstrSet instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.
- virtual void gen_asm (ostream &o)=0

  Generates the assembly code for this instruction.

Additional Inherited Members

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.11.1 Detailed Description

Represents an instruction that sets a value in the intermediate representation.

The `IRInstrSet` class is a subclass of `BaseIRInstr`. It represents an instruction that performs an assignment or setting of a value within a basic block during intermediate representation generation. The primary function of this class is to generate the assembly code for the instruction.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 IRInstrSet()

```
IRInstrSet::IRInstrSet (
            BasicBlock * bb_ )  [inline]
```

Constructs an IRInstrSet object.

This constructor initializes an `IRInstrSet` instance with a reference to the basic block in which this instruction resides.

Parameters

*bb_*	A pointer to the `BasicBlock` to which this instruction belongs.

4.11.3 Member Function Documentation

4.11.3.1 gen_asm()

```
void IRInstrSet::gen_asm (
```

```
std::ostream & o )  [override], [virtual]
```

Generates assembly code for the IRInstrSet instruction.

This function generates the corresponding assembly code for the `IRInstrSet` instruction and writes it to the provided output stream.

The generated assembly code typically includes an instruction for setting a value in a register or memory location, depending on the context.

Parameters

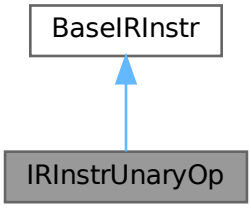
o	The output stream to which the generated assembly code will be written.

4.12 IRInstrUnaryOp Class Reference

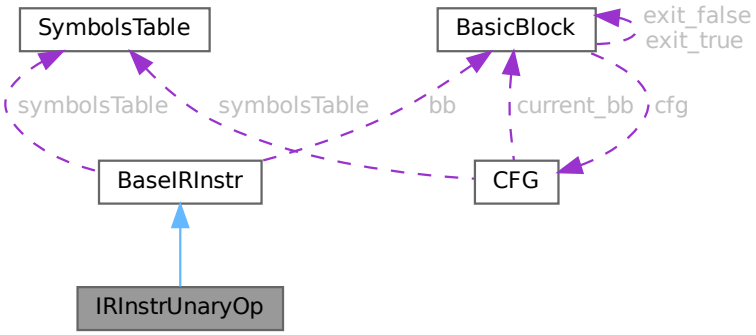
Represents a unary operation instruction in the intermediate representation.

`#include <IRInstrUnaryOp.h>`

Inheritance diagram for IRInstrUnaryOp:



Collaboration diagram for IRInstrUnaryOp:



Public Member Functions

- IRInstrUnaryOp (BasicBlock *bb_, string uniqueOp, string op)

  Constructs a unary operation instruction.
- virtual void gen_asm (ostream &o)

  Generates the assembly code for this unary operation instruction.

Public Member Functions inherited from BaseIRInstr

- BaseIRInstr (BasicBlock *bb_)

  Constructs an instruction for a given basic block.
- BasicBlock * getBB ()

  Gets the basic block that this instruction belongs to.

Protected Attributes

- string uniqueOp

  A unique identifier for the unary operation.
- string op

  The actual unary operation (e.g., '!', '-', '~').

Protected Attributes inherited from BaseIRInstr

- BasicBlock * bb

  The basic block that this instruction belongs to.
- SymbolsTable * symbolsTable

  The symbol table for the current scope.

4.12.1 Detailed Description

Represents a unary operation instruction in the intermediate representation.

This class handles the generation of intermediate representation instructions for unary operations, such as negation, logical negation, and bitwise complement.

4.12.2 Constructor & Destructor Documentation

4.12.2.1 IRInstrUnaryOp()

```
IRInstrUnaryOp::IRInstrUnaryOp (
            BasicBlock * bb_,
            string uniqueOp,
            string op )  [inline]
```

Constructs a unary operation instruction.

Initializes the instruction with a basic block, a unique operation identifier, and the operation itself.

Parameters

*bb_*	The basic block to which the instruction belongs.
uniqueOp	A unique identifier for the operation.
op	The actual unary operation (e.g., '!', '-', '∼').

4.12.3 Member Function Documentation

4.12.3.1 gen_asm()

```
void IRInstrUnaryOp::gen_asm (
            ostream & o )  [virtual]
```

Generates the assembly code for this unary operation instruction.

This method generates the appropriate assembly code based on the specific unary operation (e.g., negation, logical NOT, bitwise complement).

Parameters

o	The output stream where the generated assembly code will be written.

Implements BaseIRInstr.

4.12.4 Member Data Documentation

4.12.4.1 op

```
string IRInstrUnaryOp::op  [protected]
```

The actual unary operation (e.g., '!', '-', '∼').

4.12.4.2 uniqueOp

```
string IRInstrUnaryOp::uniqueOp  [protected]
```

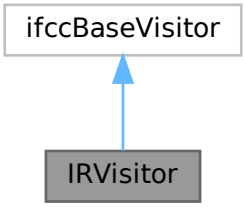
A unique identifier for the unary operation.

4.13 IRVisitor Class Reference

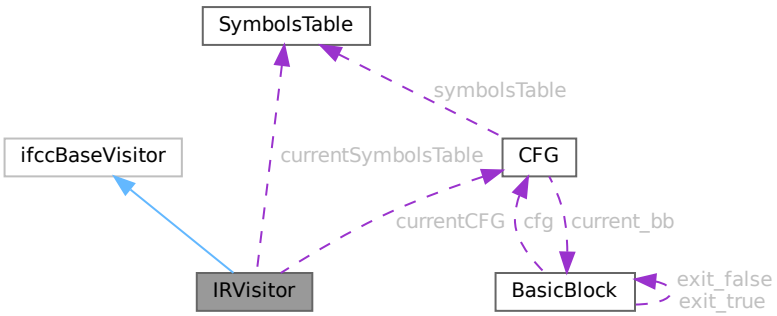
A visitor class for generating Intermediate Representation (IR) during parsing.

```
#include <IRVisitor.h>
```

Inheritance diagram for IRVisitor:



Collaboration diagram for IRVisitor:



Public Member Functions

- IRVisitor (SymbolsTable ∗symbolsTable, int baseStackOffset)

  Constructs an IRVisitor.
- virtual antlrcpp::Any visitProg (ifccParser::ProgContext ∗ctx) override

  Visits the program and starts the IR generation process.
- virtual antlrcpp::Any **visitBlock** (ifccParser::BlockContext ∗ctx) override
- virtual antlrcpp::Any visitReturn_stmt (ifccParser::Return_stmtContext ∗ctx) override

  Visits a return statement and generates the IR.
- virtual antlrcpp::Any visitAssign_stmt (ifccParser::Assign_stmtContext ∗ctx) override

  Visits an assignment statement and generates the IR.
- virtual antlrcpp::Any visitAssign (ifccParser::AssignContext ∗ctx) override

  Visits an assignment expression in the parse tree.
- virtual antlrcpp::Any visitDecl_stmt (ifccParser::Decl_stmtContext ∗ctx) override

  Visits a declaration statement and generates the IR.
- antlrcpp::Any visitExpr (ifccParser::ExprContext ∗expr, bool isFirst)

  Visits an expression and generates the IR.
- virtual antlrcpp::Any visitAddsub (ifccParser::AddsubContext ∗ctx) override

Visits an addition or subtraction expression and generates the IR.

- virtual antlrcpp::Any visitMuldiv (ifccParser::MuldivContext *ctx) override

  Visits a multiplication or division expression and generates the IR.
- virtual antlrcpp::Any visitBitwise (ifccParser::BitwiseContext *ctx) override

  Visits a bitwise operation expression and generates the IR.
- virtual antlrcpp::Any visitComp (ifccParser::CompContext *ctx) override

  Visits a comparison expression and generates the IR.
- virtual antlrcpp::Any visitUnary (ifccParser::UnaryContext *ctx) override

  Visits a unary expression and generates the IR.
- virtual antlrcpp::Any visitPre (ifccParser::PreContext *ctx) override

  Visits a pre-unary operation (e.g., prefix increment/decrement) and generates the IR.
- virtual antlrcpp::Any visitPost (ifccParser::PostContext *ctx) override

  Visits a post-unary operation (e.g., postfix increment/decrement) and generates the IR.
- void gen_asm (ostream &o)

  Generates the assembly code for the IR.
- void setCurrentCFG (CFG *currentCFG)

  Sets the current control flow graph (CFG).
- CFG * getCurrentCFG ()

  Retrieves the current control flow graph (CFG).
- map< string, CFG * > getCFGS ()

  Retrieves the map of Control Flow Graphs (CFGs).
- SymbolsTable * getCurrentSymbolsTable ()

  Retrieves the current symbols table. This method retrieves the current symbols table that is being used for the IR generation.
- void setCurrentSymbolsTable (SymbolsTable *currentSymbolsTable)

  Sets the current symbols table. This method sets the current symbols table that is being used for the IR generation.

Protected Attributes

- map< string, CFG * > cfgs

  A map of variable names to their corresponding Control Flow Graphs (CFGs).
- map< SymbolsTable *, int > childIndices

  A map of symbols tables to their corresponding indices in theirs scope.
- CFG * currentCFG

  The current control flow graph (CFG) being used.
- SymbolsTable * currentSymbolsTable

  The current symbols table being used.

4.13.1 Detailed Description

A visitor class for generating Intermediate Representation (IR) during parsing.

This class extends the `ifccBaseVisitor` and is responsible for traversing the parsed code to generate the Intermediate Representation (IR), such as the Control Flow Graph (CFG), for the code being compiled.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 IRVisitor()

```
IRVisitor::IRVisitor (
            SymbolsTable * symbolsTable,
            int baseStackOffset )
```

Constructs an IRVisitor.

Initializes the IRVisitor with a symbols table and base stack offset.

Parameters

symbolsTable	A map containing variable names and their associated stack offsets.
baseStackOffset	The base offset for the stack.

4.13.3 Member Function Documentation

4.13.3.1 gen_asm()

```
void IRVisitor::gen_asm (
            ostream & o )
```

Generates the assembly code for the IR.

This method generates assembly code from the Intermediate Representation (IR) for output.

Parameters

o	The output stream where the assembly code will be written.

4.13.3.2 getCFGS()

```
map< string, CFG * > IRVisitor::getCFGS ( )
```

Retrieves the map of Control Flow Graphs (CFGs).

This function returns a map where the keys are string labels (e.g., function names or block labels) and the values are pointers to the corresponding `CFG` objects. These `CFG` objects represent the control flow graphs of different sections of the parsed program.

The returned map can be used for further analysis, visualization (e.g., by generating Graphviz `.dot` files), or manipulation of the program's control flow.

Returns

A `std::map` where the key is a string label representing the section of the program (such as a function name) and the value is a pointer to the corresponding `CFG` object.

4.13.3.3 getCurrentCFG()

```
CFG * IRVisitor::getCurrentCFG ( )
```

Retrieves the current control flow graph (CFG).

This method retrieves the current CFG that is being used for the IR generation.

Returns

A pointer to the current CFG.

4.13.3.4 getCurrentSymbolsTable()

```
SymbolsTable * IRVisitor::getCurrentSymbolsTable ( )  [inline]
```

Retrieves the current symbols table. This method retrieves the current symbols table that is being used for the IR generation.

Returns

The current symbols table.

4.13.3.5 setCurrentCFG()

```
void IRVisitor::setCurrentCFG (
            CFG * currentCFG )
```

Sets the current control flow graph (CFG).

This method sets the current CFG that is being used for the IR generation.

Parameters

currentCFG	A pointer to the current CFG.

4.13.3.6 setCurrentSymbolsTable()

```
void IRVisitor::setCurrentSymbolsTable (
            SymbolsTable * currentSymbolsTable )
```

Sets the current symbols table. This method sets the current symbols table that is being used for the IR generation.

Parameters

currentSymbolsTable	A pointer to the current symbols table.

4.13.3.7 visitAddsub()

```
antlrcpp::Any IRVisitor::visitAddsub (
            ifccParser::AddsubContext * ctx )  [override], [virtual]
```

Visits an addition or subtraction expression and generates the IR.

This method processes addition and subtraction operations and generates the corresponding IR.

Parameters

ctx	The context of the addition or subtraction expression.

Returns

A result of the visit, typically unused.

4.13.3.8 visitAssign()

```
antlrcpp::Any IRVisitor::visitAssign (
            ifccParser::AssignContext * ctx )  [override], [virtual]
```

Visits an assignment expression in the parse tree.

This method processes an assignment (=) in the input C code. It retrieves the variable being assigned, evaluates the right-hand side expression, and generates the necessary Intermediate Representation (IR) instructions.

Parameters

ctx	The context of the assignment node in the parse tree.

Returns

The result of processing the assignment, wrapped in `antlrcpp::Any`.

4.13.3.9 visitAssign_stmt()

```
antlrcpp::Any IRVisitor::visitAssign_stmt (
            ifccParser::Assign_stmtContext * ctx )  [override], [virtual]
```

Visits an assignment statement and generates the IR.

This method processes assignment statements and generates the corresponding IR for the variable assignment.

Parameters

ctx	The context of the assignment statement.

Returns

A result of the visit, typically unused.

4.13.3.10 visitBitwise()

```
antlrcpp::Any IRVisitor::visitBitwise (
            ifccParser::BitwiseContext * ctx )  [override], [virtual]
```

Visits a bitwise operation expression and generates the IR.

This method processes bitwise operations like AND, OR, XOR, etc., and generates the corresponding IR.

Parameters

ctx	The context of the bitwise operation expression.

Returns

A result of the visit, typically unused.

4.13.3.11 visitComp()

```
antlrcpp::Any IRVisitor::visitComp (
            ifccParser::CompContext * ctx )  [override], [virtual]
```

Visits a comparison expression and generates the IR.

This method processes comparison operations (e.g., equality, greater-than) and generates the corresponding IR.

Parameters

ctx	The context of the comparison expression.

Returns

A result of the visit, typically unused.

4.13.3.12 visitDecl_stmt()

```
antlrcpp::Any IRVisitor::visitDecl_stmt (
            ifccParser::Decl_stmtContext * ctx )  [override], [virtual]
```

Visits a declaration statement and generates the IR.

This method processes declaration statements and generates the corresponding IR for the variable declaration.

Parameters

ctx	The context of the declaration statement.

Returns

A result of the visit, typically unused.

4.13.3.13 visitExpr()

```
antlrcpp::Any IRVisitor::visitExpr (
            ifccParser::ExprContext * expr,
            bool isFirst )
```

Visits an expression and generates the IR.

This method processes expressions and generates the corresponding IR for the expression.

Parameters

expr	The expression context to generate IR for.
isFirst	A flag indicating whether this is the first expression in a sequence.

Returns

A result of the visit, typically unused.

4.13.3.14 visitMuldiv()

```
antlrcpp::Any IRVisitor::visitMuldiv (
            ifccParser::MuldivContext * ctx )  [override], [virtual]
```

Visits a multiplication or division expression and generates the IR.

This method processes multiplication and division operations and generates the corresponding IR.

Parameters

ctx	The context of the multiplication or division expression.

Returns

A result of the visit, typically unused.

4.13.3.15 visitPost()

```
antlrcpp::Any IRVisitor::visitPost (
            ifccParser::PostContext * ctx )  [override], [virtual]
```

Visits a post-unary operation (e.g., postfix increment/decrement) and generates the IR.

This method processes post-unary operations and generates the corresponding IR.

Parameters

ctx	The context of the post-unary expression.

Returns

A result of the visit, typically unused.

4.13.3.16 visitPre()

```
antlrcpp::Any IRVisitor::visitPre (
            ifccParser::PreContext * ctx )  [override], [virtual]
```

Visits a pre-unary operation (e.g., prefix increment/decrement) and generates the IR.

This method processes pre-unary operations and generates the corresponding IR.

Parameters

ctx	The context of the pre-unary expression.

Returns

A result of the visit, typically unused.

4.13.3.17 visitProg()

```
antlrcpp::Any IRVisitor::visitProg (
            ifccParser::ProgContext * ctx )  [override], [virtual]
```

Visits the program and starts the IR generation process.

This method starts the process of visiting the program node, generating the IR for the entire program.

Parameters

ctx	The context of the program.

Returns

A result of the visit, typically unused.

4.13.3.18 visitReturn_stmt()

```
antlrcpp::Any IRVisitor::visitReturn_stmt (
            ifccParser::Return_stmtContext * ctx )  [override], [virtual]
```

Visits a return statement and generates the IR.

This method processes return statements and generates the corresponding IR for the return operation.

Parameters

ctx	The context of the return statement.

Returns

A result of the visit, typically unused.

4.13.3.19 visitUnary()

```
antlrcpp::Any IRVisitor::visitUnary (
            ifccParser::UnaryContext * ctx )  [override], [virtual]
```

Visits a unary expression and generates the IR.

This method processes unary operations (e.g., negation or logical NOT) and generates the corresponding IR.

Parameters

ctx	The context of the unary expression.

Returns

A result of the visit, typically unused.

4.13.4 Member Data Documentation

4.13.4.1 cfgs

```
map<string, CFG *> IRVisitor::cfgs  [protected]
```

A map of variable names to their corresponding Control Flow Graphs (CFGs).

4.13.4.2 childIndices

```
map<SymbolsTable *, int> IRVisitor::childIndices  [protected]
```

A map of symbols tables to their corresponding indices in theirs scope.

4.13.4.3 currentCFG

```
CFG* IRVisitor::currentCFG  [protected]
```

The current control flow graph (CFG) being used.

4.13.4.4 currentSymbolsTable

```
SymbolsTable* IRVisitor::currentSymbolsTable  [protected]
```

The current symbols table being used.

4.14 SymbolsTable Class Reference

Stores information about variables,including their names, types, usage status, and indexes.

`#include <SymbolsTable.h>`

Public Member Functions

- SymbolsTable (int currentOffset=-4)
  Construct a new Symbols Table object with an optional initial offset.
- void addSymbol (string name, Type type)
  Add the symbol to the current symbol table and update the offset.
- void addChild (SymbolsTable *child)
  Add a child symbol table to the current one.
- int getSymbolIndex (string name)
  Get the symbol index.
- Type getSymbolType (string name)
  Get the symbol type.
- bool symbolIsUsed (string name)
  Returns true if the symbol is used.
- bool symbolHasAValue (string name)
  Returns true if the symbol has been assigned a value.
- void setSymbolUsage (string name, bool isUsed)
  Sets the usage status of a symbol.
- void setSymbolDefinitionStatus (string name, bool hasValue)
  Sets the definition status of a symbol (whether it has been assigned a value).
- bool containsSymbol (string name)
  Returns true if the symbol table contains the symbol.
- map< string, int > getSymbolsIndex ()
  Returns the symbols index mapping variable names to their offsets.
- map< string, Type > getSymbolsType ()
  Returns the symbols type mapping variable names to their types.
- map< string, bool > getSymbolsUsage ()
  Returns the symbols usage mapping variable names to their usage status.
- map< string, bool > getSymbolsDefinitionStatus ()
  Returns the symbols definition status mapping variable names to their assignment status.
- vector< SymbolsTable * > getChildren ()
  Returns the child symbol tables.
- SymbolsTable * getParent ()
  Returns the parent symbol table.

4.14.1 Detailed Description

Stores information about variables,including their names, types, usage status, and indexes.

4.14.2 Constructor & Destructor Documentation

4.14.2.1 SymbolsTable()

```
SymbolsTable::SymbolsTable (
            int currentOffset = -4 )  [inline]
```

Construct a new Symbols Table object with an optional initial offset.

Parameters

currentOffset	The initial offset (default is -4).

4.14.3 Member Function Documentation

4.14.3.1 addChild()

```
void SymbolsTable::addChild (
            SymbolsTable * child )
```

Add a child symbol table to the current one.

Parameters

child	The child symbol table to be added.

4.14.3.2 addSymbol()

```
void SymbolsTable::addSymbol (
            string name,
            Type type )
```

Add the symbol to the current symbol table and update the offset.

Parameters

name	The name of the symbol.
type	The type of the symbol.

4.14.3.3 containsSymbol()

```
bool SymbolsTable::containsSymbol (
            string name )
```

Returns true if the symbol table contains the symbol.

Parameters

name	The name of the symbol.

Returns

bool - Whether the symbol exists in the table.

4.14.3.4 getChildren()

```
vector< SymbolsTable * > SymbolsTable::getChildren ( )  [inline]
```

Returns the child symbol tables.

Returns

vector<SymbolsTable *> - The list of child symbol tables.

4.14.3.5 getParent()

```
SymbolsTable * SymbolsTable::getParent ( )  [inline]
```

Returns the parent symbol table.

Returns

SymbolsTable * - The parent symbol table.

4.14.3.6 getSymbolIndex()

```
int SymbolsTable::getSymbolIndex (
            string name )
```

Get the symbol index.

Parameters

name	The name of the symbol.

Returns

int - The index of the symbol.

4.14.3.7 getSymbolsDefinitionStatus()

```
map< string, bool > SymbolsTable::getSymbolsDefinitionStatus ( )  [inline]
```

Returns the symbols definition status mapping variable names to their assignment status.

Returns

map<string, bool> - The symbols definition status.

4.14.3.8 getSymbolsIndex()

```
map< string, int > SymbolsTable::getSymbolsIndex ( )  [inline]
```

Returns the symbols index mapping variable names to their offsets.

Returns

map<string, int> - The symbols index.

4.14.3.9 getSymbolsType()

```
map< string, Type > SymbolsTable::getSymbolsType ( )  [inline]
```

Returns the symbols type mapping variable names to their types.

Returns

map<string, Type> - The symbols type.

4.14.3.10 getSymbolsUsage()

```
map< string, bool > SymbolsTable::getSymbolsUsage ( )  [inline]
```

Returns the symbols usage mapping variable names to their usage status.

Returns

map<string, bool> - The symbols usage.

4.14.3.11 getSymbolType()

```
Type SymbolsTable::getSymbolType (
            string name )
```

Get the symbol type.

Parameters

name	The name of the symbol.

Returns

Type - The type of the symbol.

4.14.3.12 setSymbolDefinitionStatus()

```
void SymbolsTable::setSymbolDefinitionStatus (
            string name,
            bool hasValue )
```

Sets the definition status of a symbol (whether it has been assigned a value).

Parameters

name	The name of the symbol.
hasValue	Whether the symbol has a value assigned.

4.14.3.13 setSymbolUsage()

```
void SymbolsTable::setSymbolUsage (
            string name,
            bool isUsed )
```

Sets the usage status of a symbol.

Parameters

name	The name of the symbol.
isUsed	Whether the symbol has been used.

4.14.3.14 symbolHasAValue()

```
bool SymbolsTable::symbolHasAValue (
            string name )
```

Returns true if the symbol has been assigned a value.

Parameters

name	The name of the symbol.

Returns

bool - Whether the symbol has a value.

4.14.3.15 symbolIsUsed()

```
bool SymbolsTable::symbolIsUsed (
            string name )
```

Returns true if the symbol is used.

Parameters

name	The name of the symbol.

Returns

bool - Whether the symbol has been used.

Index