VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PRINCIPLES OF PROGRAMMING LANGUAGES

INTERMEDIATE REPRESENTATION OPTIMIZATION AND MIPS CODE GENERATION

Instructor: Dr. Nguyen Duc Dung

Student: Chau Dang Minh - 2013748

Ho Chi Minh City, May 2024

Contents

1 Improvements from Last Version

The previous implementation of IR optimizers and the MIPS code generator suffered from poor design, resulting in undebuggable errors during testing. In this new implementation, each worker is explicitly separated as an isolated Visitor.

2 Revision on Compiling Phases

The MT22-MIPS compiler converts a source program in the MT22 language to a program in a MIPS assembly.

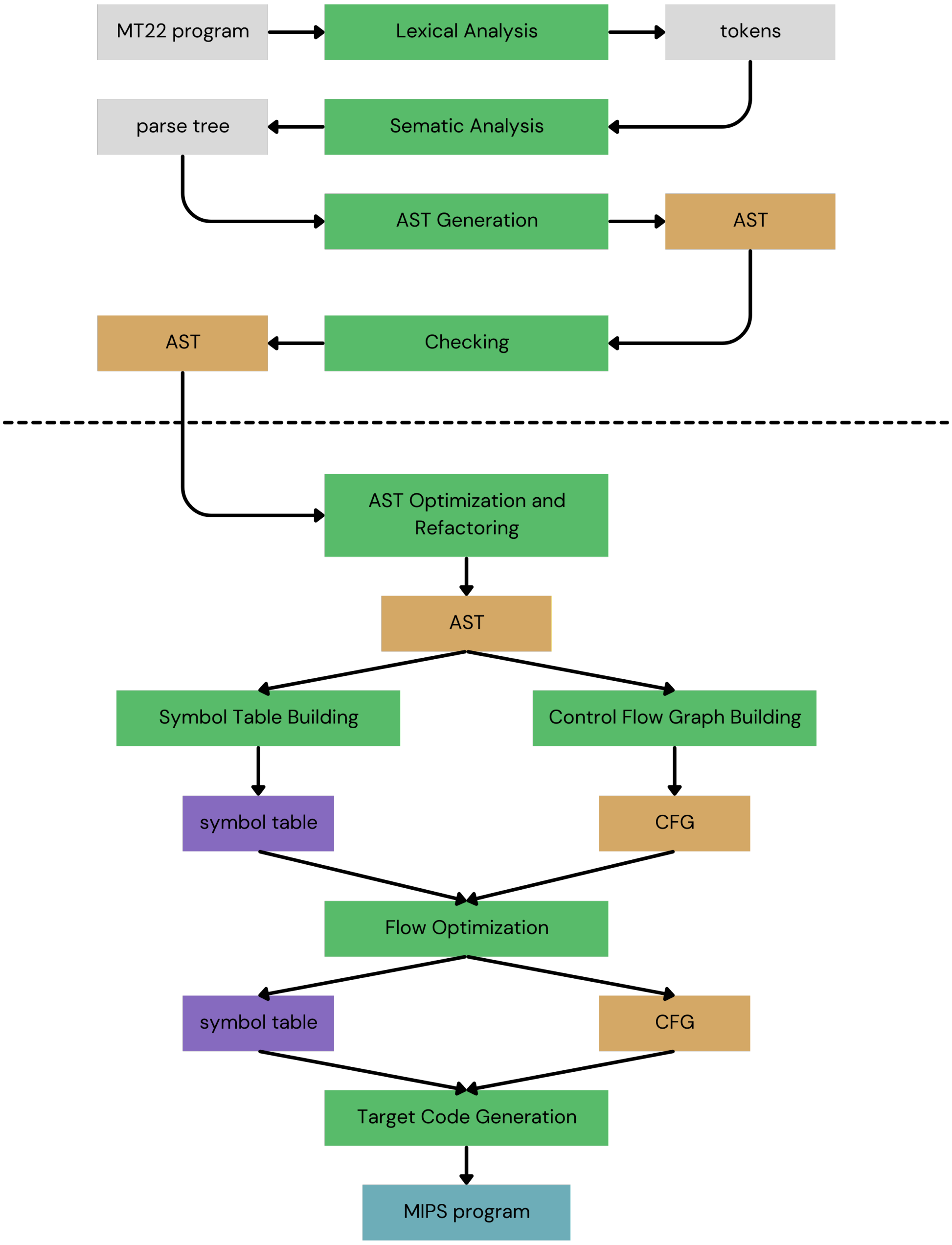

Figure 2.1: Phases of MT22-MIPS compiler

Figure 2.1 illustrates phases of the MT22-MIPS compiler, derived closely to general compilers. The dashed line separates between frontend and backend processes. The frontend has been implemented in class assignments. Here we focus on the backend. Abstract Syntax Tree (AST) is a well-known intermediate representation (IR) for our program, whose major nodes are shown in Figure 2.2. Note that BlockStmt, IfStmt, ForStmt, WhileStmt and DoWhileStmt include at least one Stmt. The AST is produced after the frontend process, then it is refactored and optimized one more time in the backend.

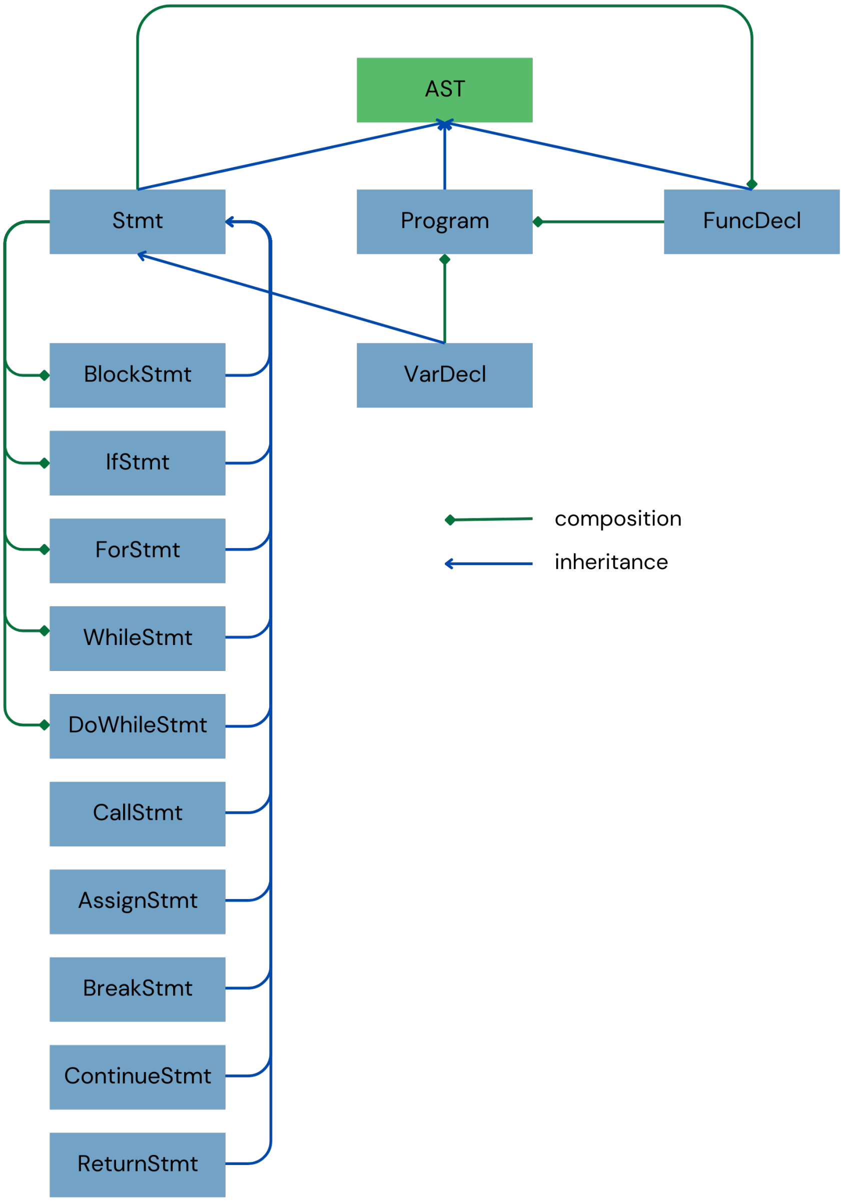

Figure 2.2: Overview of statement relations

Another equivalent IR, the Control Flow Graph (CFG) and an assisting data structure, the Symbol Table. Flow Optimization and Target Code Generation work with the symbol table and the CFG.

3 Backend Design

We use the Visitor design pattern for phases and sub-phases in the backend, as shown in Listing 1. Possible visitees are AST and CFG. Information is passed through visitors through a `VisitData` object. A visitor function takes the data from its caller and returns an *updated data*. This strategy is to ensure purity and to let us visit a visitee just by visiting its children and do later work one-by-one. Python uses pass-by-reference mechanism for a user-defined class, so our implementation does not impact on memory usage. In practice, it is sufficient to encapsulate an object and a context in the data. For each visitor, we have to explicate the visit context. The backend design is illustrated in Figure 3.1

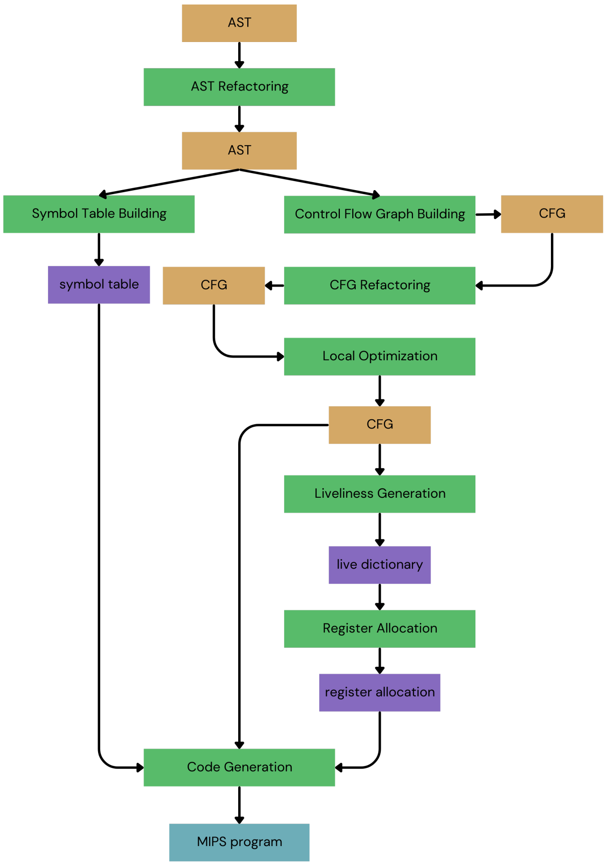

Figure 3.1: Backend design

Listing 1: Visitor pattern

```python
class VisitData:
    def __init__(self,
                 obj, # the object of data structure needed to be visited
                 ctx, # context needed for a visit function to return the complete updated data
                 ):
        # Initialization

class Visitee:
    def accept(self, visitor, visit_data):
        method_name = "visit{}".format(self.__class__.__name__)
        visit_function = getattr(visitor, method_name)
        return visit_function(self, data)

class ExampleVisitee(Visitee): pass

class Visitor:
    def visit(self, visitee, visit_data):
        return visitee.accept(self, data)

    def visitExampleVisitee(self, visitee, visit_data):
        '''
        Instructions...
        '''
        return updated_visit_data
```

3.1 AST Refactor

AST refactor modifies the AST in preparation for later processes. Since variable declaration before using has been checked in frontend, we only need statements that use these variables, while recorded their data types. To sum up, the Refactorer helps us:

- Eliminate dead code after ReturnStmt in FuncDecl.
- Turn Stmt used in IfStmt, WhileStmt, ForStmt and DoWhileStmt to BlockStmt
- Remove VarDecl(name, type, value), while add the name into the symbol table.
- Turn VarDecl(name, type, value) to AssignStmt(Id(name), value, referenced_type), while add the name into the symbol table.
- Turn ForStmt to WhileStmt.
- Unwrap complex BinExpr.

```
class ASTRefactorContext:
class ASTRefactorerContext:
  def __init__(self,
            last_sym # last symbol accessed by the last sub-expression, excluded to BinExpr unwrapper
            used_sum # the symbol that will be used by sub-expressions):
    # Initializations
```

3.2 CFG Building

A control flow graph (CFG) is composed of basic blocks. Each basic block contains a sequence of statements that can be executed sequentially. Listing 2 illustrates the design of CFG and the building context. High level implementations are shown in following figures.

Listing 2: CFG Design and its building context

```
class Block(Visitee):
    def __init__(self, id, name,
               stmts, # list of statements in the block
               cond,  # branching condition

               next,  # next block on no condition

               true,  # branched block on true condition
               false, # branched block on false condition

               jump,  # jumped block on FuncCall or CallStmt
               link,  # linked block after FuncCall or CallStmt
               end,   # the block ending a FuncCall or CallStmt
             ):
        # Initialization

class CFG(Visitee):
    def __init__(self, blocks, avail_id):
        # Initialization

        self.obj.blocks = [Block(id=0)]

        # context
        self.ctx.active_block = self.obj.blocks[-1] # any statements rather than IfStmt, WhileStmt and CallStmt will be added to this block when visited

        self.ctx.loop_block = None # , active_block points to this block if ContinueStmt is visited
```

```
# Last Block
if (expr)
  tstmt
else
  fstmt
# Next Block
```

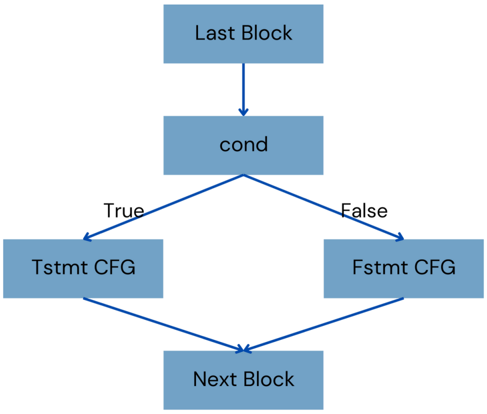

Figure 3.2: CFG of IfStmt

```
# Last Block
while (cond) {
  stmt
}
# Next Block
```

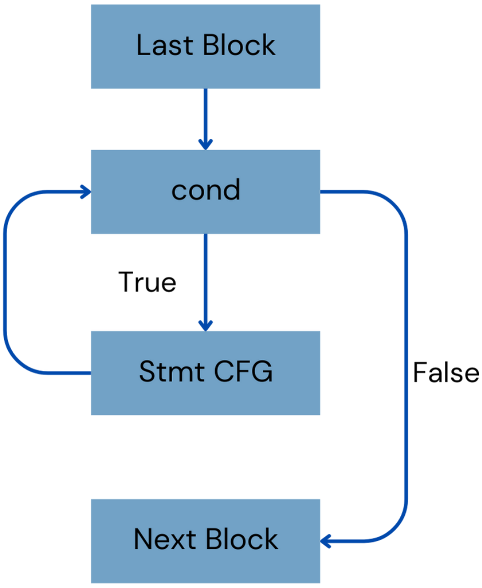

Figure 3.3: CFG of WhileStmt

```

        self.ctx.endloop_block = None # pointed to by a visited BreakStmt.

class CFGBuilderContext:
    def __init__(self,
                active_block, # current active block
                loop_block,  # the first block when entering a loop
                endloop_block # the block after visit statements in loop
                ):
        # Initialization
```

We see the structure of a CFG in Figure 3.7, much simpler than AST in statement types but dataflow expressible.

3.3 CFG Refactor

This component simply remove empty block out of the CFG.

3.4 CFG Local Optimization

Local Optimizer includes

- Algebraic Simplification: whenever there is `Expr` of all literals, compute beforehand if possible.

```
# Last Block
do {
  stmt
} while (cond)
# Next Block
```

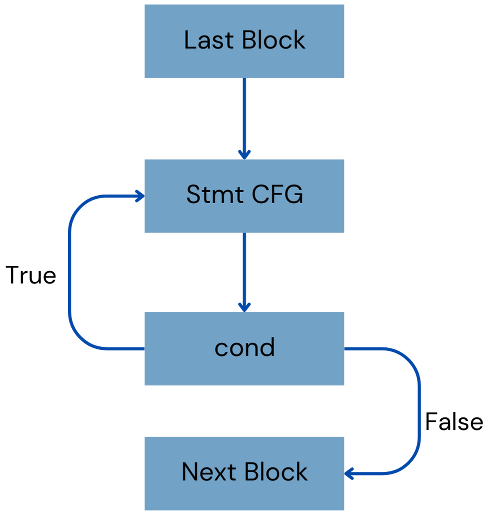

Figure 3.4: CFG of DoWhileStmt

```
  func_name : function void ()
    body
```

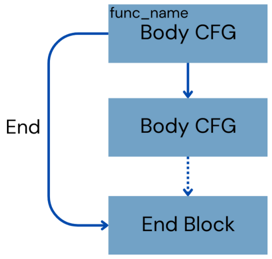

Figure 3.5: CFG of FuncDecl

- Copy Propagation: whenever there is `AssignStmt(Id(x), Id(y))`, replace all future usages of `x` by `y` until the next assignment of `x`.
- Constant Folding: whenever there is `AssignStmt(Id(x), literal)`, replace all future usages of `x` by `y` until the next assignment of `x`.
- Dead Code Elimination: `AssignStmt(Id(x), Id(y))` and `x` does not appear elsewhere in the block, delete this statement.

Each component is implemented as a CFG Visitor, which returns an updated CFG. The components are run iteratively until no changes appear.

3.5 Liveliness Generation

Liveliness is a determination of usage of the symbols. A symbol x is live at statement s if

1. There exists a statement s' that uses s.
2. There is a path from s to s' in the CFG.
3. From s to s', there is no re-assignment of x.

We compute liveliness by passing live information through adjacent statements through a function

$$L(s, x, \text{in} \mid \text{out}).$$

This function determines whether a symbol x is live incoming or outcoming of a statement s. We follow these rules:

1. If a condition C uses x, then x is live incoming this assignment.

$$L(C(x), x, \text{in}) = \text{True}. \tag{3.1}$$

```
    # Last Block
    func_name(params)
    # Next Block
```

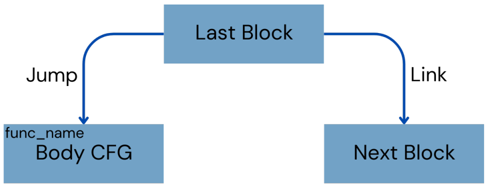

Figure 3.6: CFG of FuncCall and CallStmt

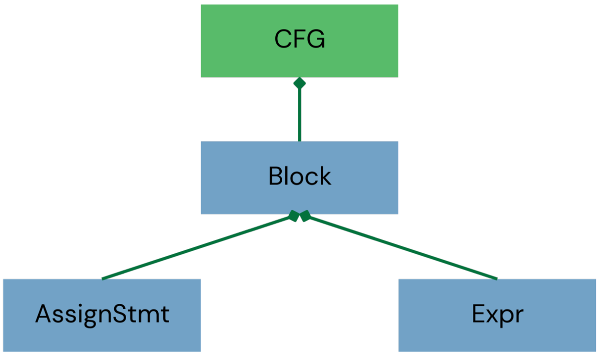

Figure 3.7: The structure of a CFG

2. If an assignment uses x on the right-hand side, then x is live incoming this assignment.

$$L(\ldots := \text{LHS}(x), x, \text{in}) = \text{True}. \tag{3.2}$$

3. If an assignment refers to x on the left-hand side but not the right-hand side, then x is not live incoming this assignment

$$L(x := e, x, \text{in}) = \text{False}. \tag{3.3}$$

4. If a statement does not refer to x, then the liveliness of x does not change when outcoming the statement.

$$L(s, x, \text{in}) = L(s, x, \text{out}) \tag{3.4}$$

5. If x is live incoming a statement s', then x is live outcoming predecessors of s'.

$$L(s, x, \text{out}) = \bigvee \left\{L(s', x, \text{in}) : s' \text{ is a successor of } s\right\}. \tag{3.5}$$

The first three rules are run once. Then the last two rules are run iteratively until no changes appear on the liveliness dictionary. Live symbols at each statement `stmt` is computed as all live symbols outcoming of this statement and incoming of its successor.

Listing 3: Liveliness generator context

```
class LiveGenContext:
  def __init__(self,
               stmt,           # current visited statement
               refered_symbols # current refered symbols
             ):
  # Initialization
```

3.6 Register Allocation

Up to now, we have live symbols at every statement. Two symbols cannot be allocated to the same register if they are live at the same time. Hence, we construct a Register Inference Graph (RIG). Each symbol is represented by a node. There is an edge between two nodes if they are live simultaneously at some point in the program. Since register allocation is NP hard, we follow a heuristic approach. Generally, we try to allocate register to node whose degree is less than or equal to the number of register, then remove this node from RIG, and so on. If such node is not found, we spill the highest-degree node i.e. load and store continuously for respective symbol.

Algorithm 1 Heuristic Register Allocation

```
$r$ : number of registers
$G$ : register inference graph
$S$ : empty stack
while rig.nodes is not empty do
    lst := nodes of degree less than or equal to $r$
    if lst is empty then
        Spill the highest-degree node $N$
        Remove $N$ from $G$
    else
        Add highest-degree node $N$ in lst to $S$
        Remove $N$ from $G$
    end if
end while
while $S$ is not empty do
    Add $N = S$.pop() to $G$
    Choose appropriate register for $N$ based on RIG
end while
```

3.7 Code Generation

Until this point, we got registers allocated to our symbols. The binary expressions are also unwrapped. The Code Generator simply adds all unallocated symbols to memory and uses respective registers for allocated ones.

4 Testing

Command line testing with manual match has been provided in the project repository.

5 Conclusion

Compiler design includes many fields in Computer Science that we have learned in university, from Formal Language Theory, Computer Architecture to Software Engineering. Our implementation reasonably optimizes generated MIPS code through many steps. However, the approach cannot catch up with modern compilers. Refactoring the compiler and further optimization, such as CFG global optimization is our considered future work.

```
data = '''
main : function void() {
a : array[5] of integer = {5,4,3,2,1};
tmp : integer;
for (i = 0; i < 4; i = i + 1){
  for (j = i+1; j < 5; j = j + 1) {
    if (a[i] > a[j]) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
    }
  }
}
}
'''
```

```
'''
Program([
  Func(main, void, [], None,
    Block([
      Assign(a, [5, 4, 3, 2, 1]),
      Assign(i, 0.0),
      While(BinExpr(<, i, 4),
        Block([
          Assign(j, BinExpr(+, i, 1)),
          While(BinExpr(<, j, 5),
            Block([
              If(BinExpr(>, ArrayCell(a, [i]),
                   ArrayCell(a, [j])),
              Assign(tmp, ArrayCell(a, [i]))),
              Assign(ArrayCell(a, [i]), ArrayCell(a,
                   [j])),
              Assign(ArrayCell(a, [j]), tmp),
              Assign(j, BinExpr(+, j, 1))
            ])),
          Assign(i, BinExpr(+, i, 1))]))]))
])
'''
```

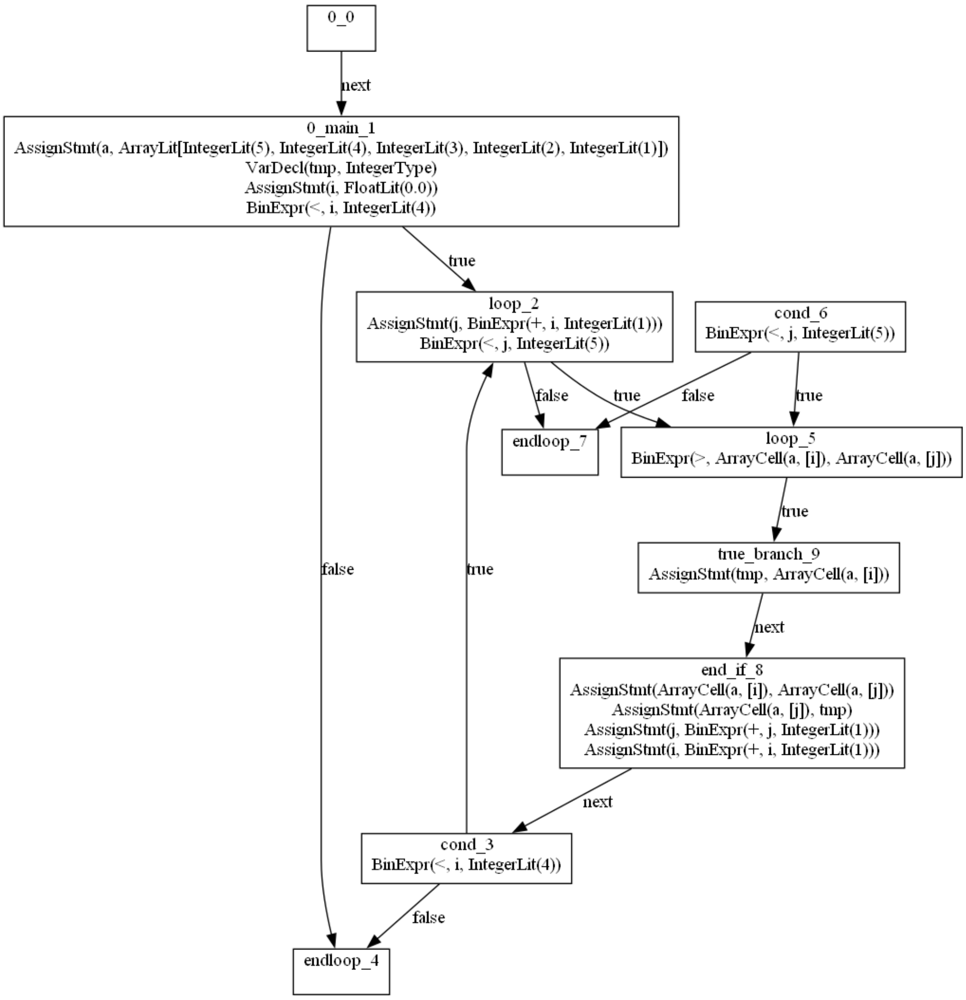
0_0
next
0_main_1
AssignStmt(a, ArrayLit[IntegerLit(5), IntegerLit(4), IntegerLit(3), IntegerLit(2), IntegerLit(1)])
VarDecl(tmp, IntegerType)
AssignStmt(i, FloatLit(0.0))
BinExpr(<, i, IntegerLit(4))
true
loop_2
AssignStmt(j, BinExpr(+, i, IntegerLit(1)))
BinExpr(<, j, IntegerLit(5))
cond_6
BinExpr(<, j, IntegerLit(5))
false
true
false
true
endloop_7
loop_5
BinExpr(>, ArrayCell(a, [i]), ArrayCell(a, [j]))
true
true_branch_9
AssignStmt(tmp, ArrayCell(a, [i]))
false
true
next
end_if_8
AssignStmt(ArrayCell(a, [i]), ArrayCell(a, [j]))
AssignStmt(ArrayCell(a, [j]), tmp)
AssignStmt(j, BinExpr(+, j, IntegerLit(1)))
AssignStmt(i, BinExpr(+, i, IntegerLit(1)))
next
cond_3
BinExpr(<, i, IntegerLit(4))
false
endloop_4

Figure 3.8: A MT22 program, its refactored AST and its corresponding CFG (rendered with Graphiz)