COMP 6421 the Report of Assigment4

Code Generation

Liao Xiaoyun
40102049 sharon.liaoxy@gmail.com

Checklist

1. Memory allocation

1.1 [√] Allocate memory for basic types (integer).
1.2 [√] Allocate memory for arrays of basic types.
1.3 [√] Allocate memory for objects.
1.4 [√] Allocate memory for arrays of objects.

2. Functions

2.1 [√] Branch to a function's code block, execute the code block, branch back to the calling function.
2.2 [√] Pass parameters as local values to the function's code block.
2.3 [√] Upon execution of a return statement, pass the return value back to the calling function.
2.4 [×] Call to member functions that can use their object's data members.

3. Statements

3.1 [√] Assignment statement: assignment of the resulting value of an expression to a variable, independently of what is the expression to the right of the assignment operator.
3.2 [√] Conditional statement: implementation of a branching mechanism.
3.3 [√] Loop statement: implementation of a branching mechanism.
3.4 [√] Input/output statement: execution of a keyboard input statement should result in the user being prompted for a value from the keyboard by the Moon program and assign this value to the parameter passed to the input statement. Execution of a console output statement should print to the Moon console the result of evaluating the expression passed as a parameter to the output statement.

4. Aggregate data elements access

4.1 [√] For arrays of basic types (integer), access to an array's elements.
4.2 [√] For arrays of objects, access to an array's element's data members.
4.3 [√] For objects, access to members of basic types.
4.4 [√] For objects, access to members of array or object types.

5. Expressions

5.1 [√] Computing the value of an entire complex expression.
5.2 [×] Expression involving an array factor whose indexes are themselves expressions.
5.3 [√] Expression involving an object factor referring to object members.

Design

1. Compiler Driver

```
//syntactic parsing
Parser_syntactic parsersyntactic = new Parser_syntactic();
parsersyntactic.loadgrammar();

ASTNode astTree =  parsersyntactic.parse(filename);
parsersyntactic.writeDotFile(astTree);

ConvertAST covertAST = new ConvertAST();
covertAST.filename = filename+"-1";

//generate AST
Node newAST = covertAST.generateNewAST(astTree);
covertAST.writeDotFile(newAST);

ReconstructSourceProgramVisitor RecnScPgVisitor = new ReconstructSourceProgramVisitor();
RecnScPgVisitor.m_outputfilename = reScPath;

//create symbol table
SymTabCreationVisitor STCVisitor   = new SymTabCreationVisitor();
STCVisitor.m_outputfilename = symTabPath;
STCVisitor.m_errorFilename = symTabcheckPath;

//type checking
TypeCheckingVisitor typeCheckVisitor = new TypeCheckingVisitor();
typeCheckVisitor.m_outputfilename = typecheckPath;
typeCheckVisitor.m_symTabfilename = symTabPath;

//computer member size and offset
ComputeMemSizeVisitor CMSVisitor3   = new ComputeMemSizeVisitor();
CMSVisitor3.m_outputfilename = symTabPath;

//code generating
StackBasedCodeGenerationVisitor stackCodeGeneratorVisiter   = new StackBasedCodeGenerationVisitor(moon_outfilePath);

newAST.accept(STCVisitor);
newAST.accept(RecnScPgVisitor);
newAST.accept(typeCheckVisitor);
newAST.accept(CMSVisitor3);
newAST.accept(stackCodeGeneratorVisiter);
```

2. Adding tempVar and moonVarName during type checking

   Creating a new temp var for each operator node and function call node, also giving them a moonVarName which can be used to find the corresponding var faster.

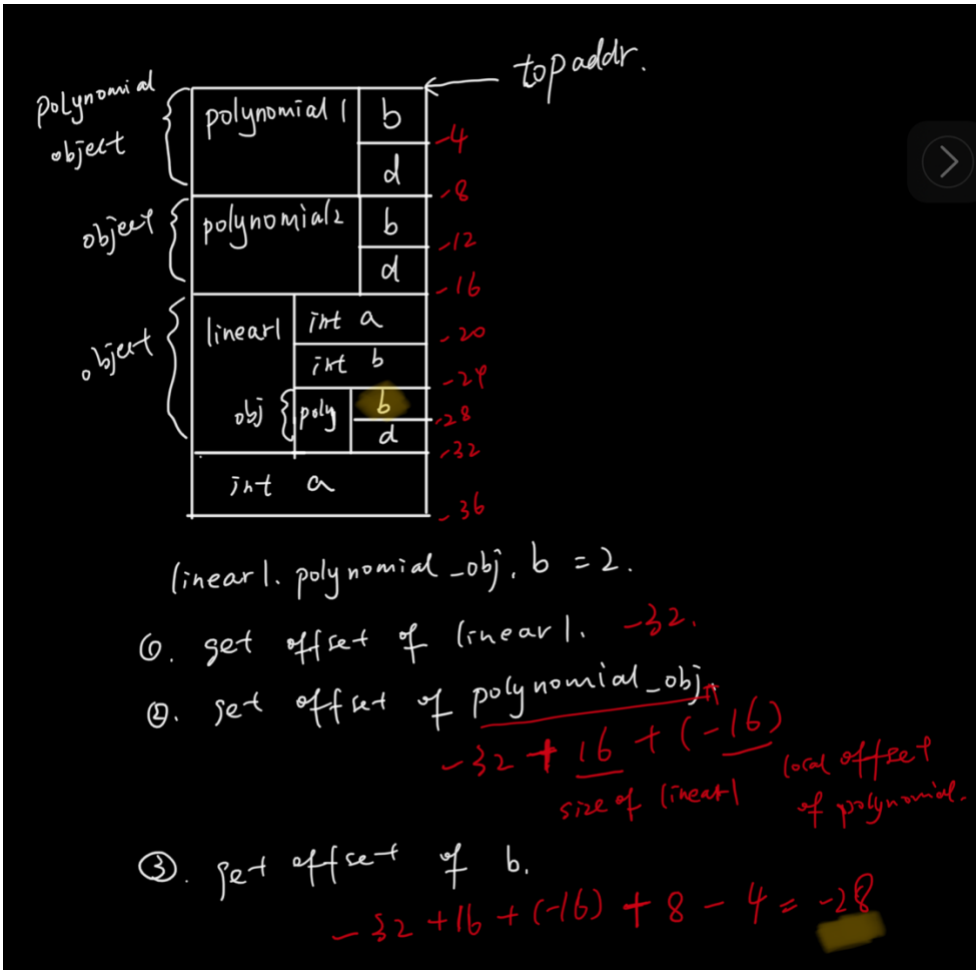
3. Creating a unique function tag for each function

   During type checking, if a function call is valid, then set its function tag as the corresponding function's tag. when in code generation, we can find the correct function quickly, avoid comparing parameters again (overloading). The unique function tags also can be used as unique tags for functions in the moon code.

4. Computing member size and offset

   - Because I used stack frame method to implement the code generation, so we need to computer the offset for each element.

- For computing the offset of the dot operator var, we need to accumulate each operand's offset.



- Computing array element's offset

```
//dimNodeList
int arrayOffset = 0;
ArrayList<Integer> arrayDim = new ArrayList<>();

for(Node dimNode : dimNodeList.getChildren()){
    //IdNode, NumNode, addOp, FuncCallNode, multOp,
    System.out.println("dimNode type:"+ dimNode.getClass().getSimpleName());
    //just accept constant num now
    arrayDim.add(Integer.parseInt(dimNode.m_data));
}

for(int j=0; j<arrayDim.size(); j++){
    int dimSize = 1;
    int num = arrayDim.get(j);
    for(int k = j+1; k<arrayDim.size(); k++){
        dimSize *= dimList.get(k);
    }
    arrayOffset += num*dimSize;
}
arrayOffset = arrayOffset*varEntry.m_size;
offset_var += -arrayOffset;
```

- Getting the size of different type of element

```
public int sizeOfTypeNode(String type) {
    System.out.println("get type size:"+type);
    int size = 0;
    if(type.equals("integer"))
        size = 4;
    else if(type.equals("float"))
        size = 8;
    else if(type.equals("void")){
        size = 0;
    }else if(type.equals("typeerror")){
        size = 0;
    }else {
        size = globalTable.lookupName(type).m_size;
    }
    return size;
}
```

5. Code generation

   In code generation, we need to focus on two parts, which are the main function and function definition. We start from function definitions to generate moon code.

   - Function definition

```
public void visit(FuncDefNode p_node) {
    // propagate accepting the same visitor to all the children
    // this effectively achieves Depth-First AST Traversal

    m_moonExecCode += m_mooncodeindent + "% processing function definition: "  + p_node.m_moonVarName + "\n";
    //create the tag to jump onto
    // each function has a unique tag which generated in type checking stage
    // and copy the jumping-back address value in the called function's stack frame
    m_moonExecCode += String.format("%-10s", ((FuncEntry)p_node.m_symtabentry).tag)  + "sw -4(r14),r15\n" ;
    //generate the code for the function body
    for (Node child : p_node.getChildren())
        child.accept( p_visitor: this);
    // copy back the jumping-back address into r15
    m_moonExecCode += m_mooncodeindent + "lw r15,-4(r14)\n";
    // jump back to the calling function
    m_moonExecCode += m_mooncodeindent + "jr r15\n";
};
```

     In functions, there are variable declaration, function call, dot operation of variable, assignment statement, and control statement, and so on.

   - Dot operation of variable

     Function calls, expressions, assignment statements, and local variables all have entries in the local table, except the variables accessed by the dot operator. So, we need to compute the offset for those variables. In the stage of computing member size and offset we have set the offset for those variables, so we can get the offset from the entries of the var node directly.

```
public void visit(VarNode p_node){
    System.out.println("visit VarNode");
    int offSet = p_node.m_symtabentry.m_offset;
    System.out.println("var offset:"+ p_node.m_symtabentry.m_name + ","+offSet);
}
```

- Function call

1) Getting the correct function entry

   In function call node, there is an important attribute this is funCallEntry:

```
public String tag = "";
public SymTabEntry funcCallEntry = null;
```

   In type checking stage, we have confirmed which functions were called in the function call statements' and set the funCallEntry in function call nodes as this function entry, so that we can access the correct function quickly in the code generation stage.

2) Passing parameter

   Computing the offset for each parameter.

```
for(int i = 0; i<paramInFunc.size(); i++){
    Node varNode = aparamList.getChildren().get(i);
    SymTabEntry param = paramInFunc.get(i);
    //only support integer param now
    if(param.m_type.equals("integer") || param.m_type.equals("float")){
        int offset = 0;
        if (varNode.getClass().getSimpleName().equals("VarNode")){
            offset = varNode.m_symtabentry.m_offset;
        }else {
            offset = p_node.m_symtab.lookupName(varNode.m_moonVarName).m_offset;
        }

        m_moonExecCode += m_mooncodeindent + "lw " + localregister1 + "," + offset + "(r14)\n";
        //p_node.m_symtab.m_size, size of the current scope
        // minus p_node.m_symtab.m_size, to create a new start addr
        int offsetofparam = -p_node.m_symtab.m_size + param.m_offset;
        m_moonExecCode += m_mooncodeindent + "sw " + offsetofparam + "(r14)," + localregister1 + "\n";
    }
}
```

3) Jumping to the function stack frame

```
// make the stack frame pointer point to the called function's stack frame
m_moonExecCode += m_mooncodeindent + "addi r14,r14," + -p_node.m_symtab.m_size + "\n";
// jump to the called function's code
// here the function's name is the unique tag created in the type checking stage
m_moonExecCode += m_mooncodeindent + "jl r15," + p_node.tag + "\n";
// upon jumping back, set the stack frame pointer back to the current function's stack frame
m_moonExecCode += m_mooncodeindent + "subi r14,r14," + -p_node.m_symtab.m_size + "\n";
// copy the return value in memory space to store it on the current stack frame
// to evaluate the expression in which it is
m_moonExecCode += m_mooncodeindent + "lw " + localregister1 + "," + -p_node.m_symtab.m_size + "(r14)\n";
m_moonExecCode += m_mooncodeindent + "sw " + p_node.m_symtab.lookupName(p_node.m_moonVarName).m_offset + "(r14),
this.m_registerPool.push(localregister1);
```

   Using the unique function tag created in the type checking stage as the function label.

- Assignment statement

  The left-hand side and right-hand side of assignment must be Id node, or tempVar node or var node.

  Firstly, getting the offsets of two sides of the assignment.

  Then, generating lw and sw codes for this assignment.

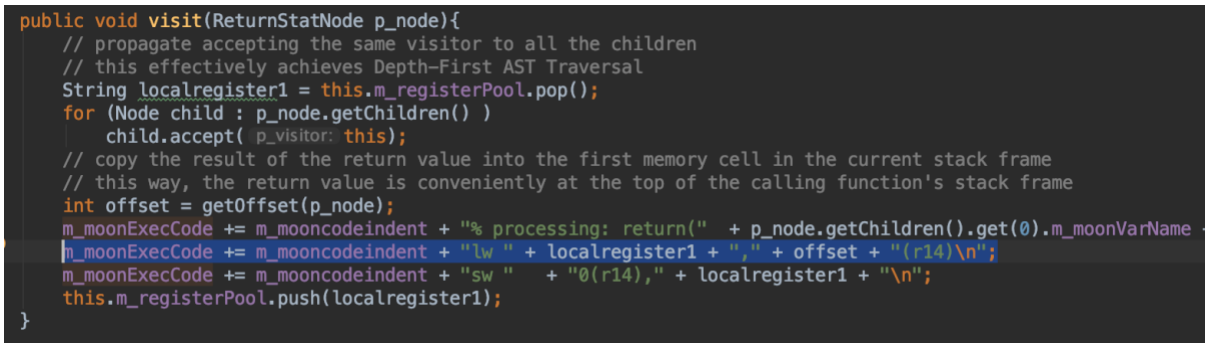
```
int leftOffSet = 0;
int rightOffSet = 0;
System.out.println("p_node.getChildren().get(0).getClass().getSimpleName():"+ p_node.getChildren().get(0).getCla
if(p_node.getChildren().get(0).getClass().getSimpleName().equals("VarNode")){
    leftOffSet = p_node.getChildren().get(0).m_symtabentry.m_offset;
    System.out.println("leftOffSet : "+ leftOffSet);
}else {
    leftOffSet =  p_node.m_symtab.lookupName(p_node.getChildren().get(0).m_moonVarName).m_offset;
}

if(p_node.getChildren().get(1).getClass().getSimpleName().equals("VarNode")){
    rightOffSet = p_node.getChildren().get(1).m_symtabentry.m_offset;
    System.out.println("rightOffSet : "+ rightOffSet);
}else {
    rightOffSet =  p_node.m_symtab.lookupName(p_node.getChildren().get(1).m_moonVarName).m_offset;
}

m_moonExecCode += m_mooncodeindent + "% processing: "  + p_node.getChildren().get(0).m_moonVarName + " := " + p_
// load the assigned value into a register
m_moonExecCode += m_mooncodeindent + "lw "   + localregister2 + "," + rightOffSet + "(r14)\n";
// assign the value to the assigned variable
m_moonExecCode += m_mooncodeindent + "sw "   + leftOffSet + "(r14)," + localregister2 + "\n";
```

- Expression

  The handle of expression is similar to the assignment's, getting the offsets of two sides of the operator, then generate codes.

- Return statement

  Getting the offset of the return variable, then generate moon code.



```
public void visit(ReturnStatNode p_node){
    // propagate accepting the same visitor to all the children
    // this effectively achieves Depth-First AST Traversal
    String localregister1 = this.m_registerPool.pop();
    for (Node child : p_node.getChildren() )
        child.accept( p_visitor: this);
    // copy the result of the return value into the first memory cell in the current stack frame
    // this way, the return value is conveniently at the top of the calling function's stack frame
    int offset = getOffset(p_node);
    m_moonExecCode += m_mooncodeindent + "% processing: return("  + p_node.getChildren().get(0).m_moonVarName
    m_moonExecCode += m_mooncodeindent + "lw " + localregister1 + "," + offset + "(r14)\n";
    m_moonExecCode += m_mooncodeindent + "sw "   + "0(r14)," + localregister1 + "\n";
    this.m_registerPool.push(localregister1);
}
```

- Write and read statement

  Write and read statements are similar to the function calls. Computing the offset of parameters and passing parameters, jumping to the write or read function stack frame to executing function, after the execution was done, return to point of the function call.

```
public void visit(WriteStatNode p_node) {
    // propagate accepting the same visitor to all the children
    // this effectively achieves Depth-First AST Traversal
    System.out.println("visit WriteStatNode");
    // First, propagate accepting the same visitor to all the children
    // This effectively achieves Depth-First AST Traversal
    for (Node child : p_node.getChildren())
        child.accept( p_visitor: this);
    // Then, do the processing of this nodes' visitor
    // create a local variable and allocate a register to this subcomputation
    String localregister1 = this.m_registerPool.pop();
    String localregister2 = this.m_registerPool.pop();

    int offset = 0;
    if (p_node.getChildren().get(0).getClass().getSimpleName().equals("VarNode")){
        offset = p_node.getChildren().get(0).m_symtabentry.m_offset;
    }else {
        offset = p_node.m_symtab.lookupName(p_node.getChildren().get(0).m_moonVarName).m_offset;
    }
    //generate code
    m_moonExecCode += m_mooncodeindent + "% processing: put("  + p_node.getChildren().get(0).m_moonVarName + ")\
    // put the value to be printed into a register
    m_moonExecCode += m_mooncodeindent + "lw " + localregister1 + "," + offset + "(r14)\n";
    m_moonExecCode += m_mooncodeindent + "% put value on stack\n";
    // make the stack frame pointer point to the called function's stack frame
    m_moonExecCode += m_mooncodeindent + "addi r14,r14," + (-p_node.m_symtab.m_size) + "\n";
    // copy the value to be printed in the called function's stack frame
    m_moonExecCode += m_mooncodeindent + "sw -8(r14)," + localregister1 + "\n";
    m_moonExecCode += m_mooncodeindent + "% link buffer to stack\n";
    m_moonExecCode += m_mooncodeindent + "addi " + localregister1 + ",r0, buf\n";
    m_moonExecCode += m_mooncodeindent + "sw -12(r14)," + localregister1 + "\n";
    m_moonExecCode += m_mooncodeindent + "% convert int to string for output\n";
    m_moonExecCode += m_mooncodeindent + "jl r15, intstr\n";
    // receive the return value in r13 and right away put it in the next called function's stack frame
    m_moonExecCode += m_mooncodeindent + "sw -8(r14),r13\n";
    m_moonExecCode += m_mooncodeindent + "% output to console\n";
    m_moonExecCode += m_mooncodeindent + "jl r15, putstr\n";
    // make the stack frame pointer point back to the current function's stack frame
    m_moonExecCode += m_mooncodeindent + "subi r14,r14," + (-p_node.m_symtab.m_size) + "\n";
```

- If statement

  In the if statement, we don't do depth-first AST traversal for this if statement node, generating code for node’s children in order instead.

```
public void visit(IfStatNode p_node) {
    System.out.println("visit IfStatNode");
    if(p_node.m_symtab!=null)
    {
        String localregister1 = this.m_registerPool.pop();
        p_node.getChildren().get(0).accept( p_visitor: this);
        int id = getID();
        int offset = getOffset(p_node);

        m_moonExecCode += m_mooncodeindent + "% processing: if("  + p_node.getChildren().get(0).m_moonVarName + ")\n";
        m_moonExecCode += m_mooncodeindent + "lw " + localregister1 + "," + offset + "(r14)\n";
        m_moonExecCode += m_mooncodeindent + "bz " + localregister1 + ",else"+id+"\n";

        p_node.getChildren().get(1).accept( p_visitor: this);
        m_moonExecCode += m_mooncodeindent + "j endif"+id + "\n";

        m_moonExecCode += "else"+id+ "\n";
        if(p_node.getChildren().size()>2){
            p_node.getChildren().get(2).accept( p_visitor: this);
        }
        m_moonExecCode += "endif"+id + "\n";
        this.m_registerPool.push(localregister1);
    }
};
```

- While statement

  While statement is similar to if statement, generating code for node’s children in order.

```
public void visit(WhileStatNode p_node) {

    System.out.println("visit WhileStatNode");
    if(p_node.m_symtab!=null)
    {
        String localregister1 = this.m_registerPool.pop();
        int id = getID();
        int offset = getOffset(p_node);
        m_moonExecCode += m_mooncodeindent + "% processing: while("  + p_node.getChildren().get(0).m_moonVarName + ")\n";
        m_moonExecCode += m_mooncodeindent + "j gowhile"+id + "\n";

        m_moonExecCode += "gowhile"+id+ "\n";

        p_node.getChildren().get(0).accept( p_visitor: this);

        m_moonExecCode += m_mooncodeindent + "lw " + localregister1 + "," + offset + "(r14)\n";
        m_moonExecCode += m_mooncodeindent + "bz " + localregister1 + ",endwhile"+id+"\n";

        p_node.getChildren().get(1).accept( p_visitor: this);
        m_moonExecCode += m_mooncodeindent + "j gowhile"+id + "\n";

        m_moonExecCode += "endwhile"+id + "\n";
        this.m_registerPool.push(localregister1);
    }
};
```

Tool

1. https://dreampuf.github.io/GraphvizOnline

   This website can help me to generate a readable AST.

2. Astvisitor

   I used this patten to implement moon code.

3. Moon processor simulator