Report of Designing My MiniJava Compiler

——Practice for Compiler Design, Fall 2016

Li YanHao, School of EECS, Peking University

Contents

Chapter 1: MiniJava Compiler Overview

In this Chapter, I am going to introduce the ultimate goal of designing the MiniJava Compiler, while the goal of each stage will be introduced in the particular part of Chapter 2. More, the pre-stage of building my compiler will be thoroughly discussed in the second part of this chapter.

MiniJava is an extremely simple programming language designed for this course. It is actually a subset of Java, and is simple enough so that its grammar can be described with BNF(Find more details in “minijava.jj”). Actually MiniJava does not support Java’s indispensable features like function overloading, so it is actually a toy language that can never be used in the real production scene. Nevertheless, it is still complex enough for beginners of compiler designing. Implementing a compiler for MiniJava is challenging enough, especially when I’m not that familiar to Java.

According to the requirements on syllabus, we are supposed to design and implement a compiler for MiniJava. By writing a compiler we master the knowledge learned on Compiler Design course. At least, I hope so. :D

1. Goals of my MiniJava compiler

First of all, I am supposed to write a compiler instead of an interpreter. Therefore, the ultimate goal of my compiler is: correctly translate the input Minijava code to MIPS instructions as efficient as possible. To be more specific, there are three other kinds of intermediate languages called Piglet, Spiglet and Kanga. Each kind of these languages has different grammar and various restrictions. The code become more and more low-level during the translating process.

The correctness of an compiler can be explained from two aspects. One is that it should be able to do semantic analysis to a certain extent, which is the goal of Type Check stage. For example, when we writing C code and compile it with GCC, it can tell where the semantic errors occur so that user can fix the problem. The other is that running result of the translated code should remain the same as the code before translation, which is the goal of last four stages of my compiler. In fact, if a compiler cannot meet this

requirement, it has no value at all. Remember that the correctness is always the most important thing to a program, not just compiler.

However, the correctness is only the basic requirement. The efficiency of a compiler is also crucial to users. We should do our best to let it become faster and faster. And the code after translation should contain least useless instruction and occupy least register.

Did I miss something so far? Of course, and this is not a problem that can be ignored. Where are lexical analysis and syntax analysis? They should be done before semantic analysis. Luckily we do not need to write our own AST generator and visitor ~~as our school's students did 10 years ago~~, because we have got JavaCC and JTB.

2. Pre-stage: JavaCC&JTB

JavaCC stands for Java Compiler Compiler, and JTB stands for Java Tree Builder. JavaCC helps us doing lexical and syntax analysis and JTB automatically generates the AST of the input code, which brings us great convenience to write a compiler. Not only do they generate an AST, they also throw exceptions when error occurs during lexical analysis or syntax analysis.

```
/**
 * Grammar production:
 * f0 -> "{"
 * f1 -> ( Statement() )*
 * f2 -> "}"
 */
public class Block implements Node {
   public NodeToken f0;
   public NodeListOptional f1;
   public NodeToken f2;

   public Block(NodeToken n0, NodeListOptional n1, NodeToken n2) {
      f0 = n0;
      f1 = n1;
      f2 = n2;
   }
```

Example of a Node

Given a file with prefix ".jj", executing a series of instructions(More details on Slides3-2) will generate two folders named "visitor" and "syntaxtree", and also a series of java class which helps building the AST.

Classes in folder called “syntaxtree” defined the data structure of the AST and generate a specific class for every identifier mentioned in BNF. Classes in folder called “visitor” provide various template of visitor, including GJDepthFirst.java that implement a DFS visitor. The node on the AST has its own attributes, which allows us to visit them while rewriting visitors. The following code snippet gives an example of a node definition, where we can find the attributes mentioned before.

I have explained what JavaCC and JTB have given me. And then I am going to explain how they work in my compiler. Suppose that we are writing a compiler which translate A to B, and “A.jj” is the grammar description file of language A. The implementing procedure is like the following:

1. Build tree with JavaCC&JTB according to “A.jj”.
2. Design data structure and organize file directories.
3. Select appropriate visitor template and rewrite it.

Since I have explained the role those auto-generated code play in my designing procedure, the detailed discussion about my code will focused on how I designed the translating procedure and how I rewrite the visitor.

```
/**
 * f0 -> Identifier()
 * f1 -> "="
 * f2 -> Expression()
 * f3 -> ";"
 */
public MPiglet visit(AssignmentStatement n, MIdentifier argu) {
   MPiglet _ret=new MPiglet();
   MPiglet mp1 = n.f0.accept(this, argu);
   MPiglet mp2 = n.f2.accept(this, argu);
   MVar mv = mp1.getVar();
   if (mv.isTemp())
     _ret.append(new MPiglet("MOVE " + mp1, nCurrentTab));
   else
     _ret.append(new MPiglet("HSTORE TEMP 0 " + mv.getOffset(), nCurrentTab));
   _ret.append(mp2);
   return _ret;
}
```

Example of a Rewritten Visitor

Chapter 2: Implementation Details

In this chapter, we will thoroughly discuss the details of my compiler, including data structures and algorithms. Each stage has its own specific unique goal, and of course has its own difficulties. I will explain my idea when facing the problem, and go through those bugs I met when debugging. However, some of my code is quite annoying and hard to understand, so I will try to explain the goal of that part and hide the implementation detail in case you get confused.

Talk is cheap. Show me the code. Let's start from type check.

1. Stage0 - Type Check

1. Goal and Design

As is mentioned before, the code coming to the first stage should have passed the lexical check and syntax check. The goal of this stage is to do Latent Semantic Analysis. The following mistakes should be found out during this stage:

a) Repeated definition

b) Undefined identifier

c) Mismatched type

d) Illegal array

e) Inherit circulation

f) Mismatched argument

To explain those mistakes more clearly, the sample code is shown as the following picture.

```
class MyClass {
    boolean myBoolean;
    // REFERENCE of undefined class
    MyClass2 myClass;
    // REDEFINITION of variable
    boolean myBoolean;
    public int getMyInt () {
        // REFERENCE of undefined variable
        return myInt2;
    }
    // REDEFINITION of method
    public int getMyInt () {
        // REFERENCE of undefined method
        return this.getMyInt2 ()
    }
}
```

The symbol table is the essential part of this stage. The symbol table describes the logical structure of the program. It should record all identifiers appeared in the code. And the symbol table should support efficient query operations. Therefore, we can quickly get necessary information, like the position of a variable and whether a method's name has been defined before. Now, it is not hard to give out a plan to solve this problem. Traverse the AST for the first time to build a symbol table, and traverse for the second time to do type check according to the symbol table. The basic procedure design is shown in the following code snippet:

```
public class Main{
    public static void main(String args[]){
        try{
            //InputStream in = new FileInputStream("/Users/liyanhao/Desktop/course/comp
            //Node root = new MiniJavaParser(in).Goal();
            Node root = new MiniJavaParser(System.in).Goal();
            MType allClassList = new MClassList();
            //build
            root.accept(new BuildSymbolTableVisitor(), allClassList);
            //typecheck
            root.accept(new TypeCheckVisitor(), allClassList);
            if (ErrorPrinter.getsize() == 0){
                System.out.println("Program type checked successfully");
            }
            else{
                System.out.println("Type error");
            }
            ErrorPrinter.printAll();

        }catch (ParseException e){
            //Parse error occurs.
            e.printStackTrace();
        }catch (TokenMgrError e){
            //Lexical error occurs.
            e.printStackTrace();
        }catch (Exception e){
            //just in case.
            e.printStackTrace();
        }
    }
}
```

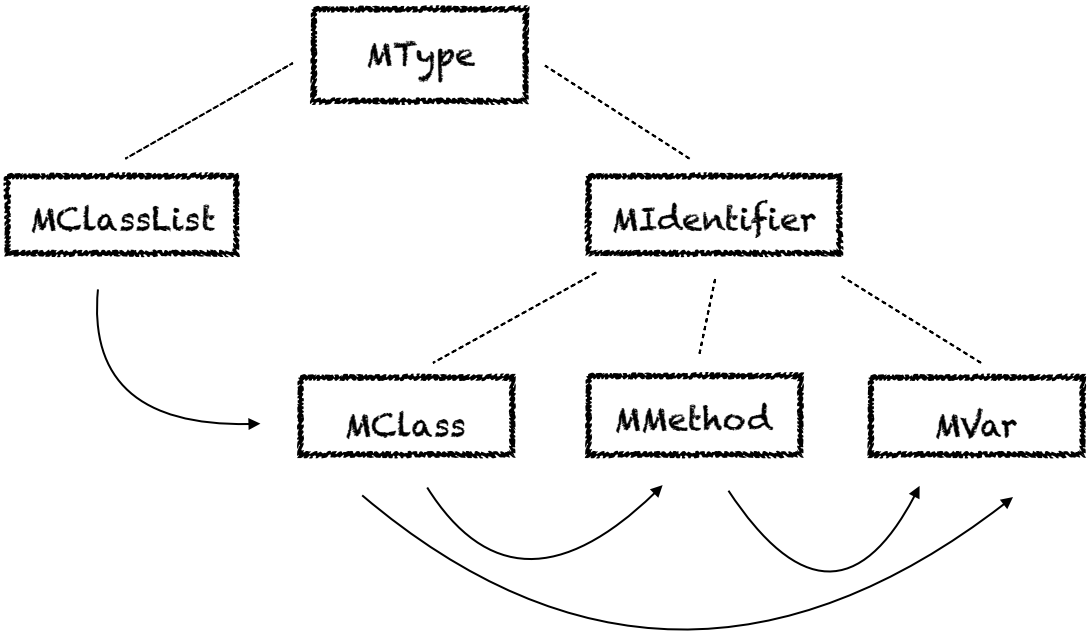
Brief Summary of Type Check

2. Symbol Table

I designed my symbol table structure according to the given slides. However I think the inheritance relations designed on slides is not clear. I defined a class called "MType", which is the father class for all other classes. It should record attributes like position of an identifier if been required. Two classes are derived from "MType". They are "MClassList" and "MIdentifier". "MClassList" is the biggest data structure of my symbol table, which has a list containing all defined classes. "MIdentifier" is an abstract concept of classes,

methods and variables. “MClass”, “MVar” and “MMethod” are derived from it. As is depicted by the name, “MVar” stands for a variable, “MMethod” stands for a method and “MClass” stands for a class. In my design, “MClass” should have two tables while one for its methods and another for its variables. “MMethod” should have a table for its variables. Therefore, I have also defined an interface called “VarContainer” for convenience. It says that an identifier that contains variables should support query and insertion operations. To distinguish different kind of identifiers, an attribute named “type” is defined in “MType”. And you can tell an identifier is not a boolean because this attribute reads “int”.

The following graph describes my symbol table’s structure.



Dashed line means inheritance.
Solid line from A to B means A contains B

All files can be found in package “minijava.symbol”.

3. Implementation Details

The type check is not only done in the second traverse procedure. Errors like repeated definition can be found out when building symbol table. Other errors like mismatched type should be found out when traversing the AST for the second time.

1. First round.

The visitor for first round is called “BuildSymbolTableVisitor.java” in package “minijava.visitor”.

As we mentioned before, the first round is for symbol table building. Since we only care about the symbols, only define sentences' visitor will be rewritten. To be more specific, only the following class's node visitor will be rewritten:

- Goal
- MainClass
- ClassDeclaration
- ClassExtendsDeclaration
- VarDeclaration
- MethodDeclaration
- FormalParameter

It's very easy to rewrite these codes. For example, ClassDeclaration means that a MClass should be put into the AllClassList. The exception goes to main class. A special type called "String[]" will be given to its argument and a class called "main" will be added into the class table, which means that the main class is only treated as a common class.

```
/**
 * f0 -> "boolean"
 */
public MType visit(BooleanType n, MType argu) {
   return new MType("boolean", n.f0.beginLine, n.f0.beginColumn);
}

/**
 * f0 -> "int"
 */
public MType visit(IntegerType n, MType argu) {
   return new MType("int", n.f0.beginLine, n.f0.beginColumn);
}
```

Confirming Variables' Type

```
public MType visit(MethodDeclaration n, MType argu) {
   MType _ret=null;
   n.f0.accept(this, argu);

   MType type = n.f1.accept(this, argu);
   MIdentifier id = (MIdentifier)n.f2.accept(this, argu);
   MMethod newMethod = new MMethod(id.getName(), type.getType(), (MIdentifier)argu, type.getRow(), type.getCol());
   String msg = ((MClass)argu).insertMethod(newMethod);
   if (msg != null) {
     ErrorPrinter.print(msg, newMethod.getRow(), newMethod.getCol());
   }
```

Inserting a Method to the Class

Repeated definition and inheritance circulation are done when inserting an identifier to a table. The circulation inheritance check is slightly different from repeated definition but basically the same. It can be implemented because “MClass” records its base class’s name. The following two code snippets show how I implement this function.

```
HashSet<String> baseNameSet = new HashSet<String>();
while (basename != null) {
  if (basename.equals(newClass.getName())) {
    ErrorPrinter.print("class: \'" + basename + "\'' circular extends itself", id.getRow(), id.getCol());
    break;
  }
  else if(baseNameSet.contains(basename)){
    break;
  }
  baseNameSet.add(basename);
  MClass baseClass = allClassList.getClass(basename);
  if (baseClass != null){
    basename = baseClass.getBaseClassName();
  }
  else{
    break;
  }
}
```

Check if There is Circulation

```
@Override // implements interface
public String insertVar(MVar newVar){
    if (this.varList.containsKey(newVar.getName())){
        // in table; report error
        return "Redundant Variable Declaration: \'" + newVar.getName() + "\'";
    }
    else{
        this.varList.put(newVar.getName(), newVar);
        return null;
    }
}
```

Inserting Defined Name into Method Table

2. Second round.

The visitor for first round is called “TypeCheckVisitor.java” in package “minijava.visitor”.

In this round, all other errors should be found out. Before we come to details, the concept of matching should be strictly defined. When A and B are both variables, A matches B means they are of the same type. When A and B are both classes, then A matches B means they are of the same class OR one is another’s ancestor. The method called “classEqualsOrDerives” in “MClassList” judges whether two classes match. And the following case should be checked:

• Return type mismatch

- Assignment mismatch
- Array index not integer type
- Print only allows integer type
- Expression operate same type
- If & While only allows boolean type
- Argument mismatch when calling procedure

The Basic Idea of this part is to check whether the identifier is the correct type. See the following snippet for a example.

```
/**
 * f0 -> Identifier()
 * f1 -> "["
 * f2 -> Expression()
 * f3 -> "]"
 * f4 -> "="
 * f5 -> Expression()
 * f6 -> ";"
 */
public MType visit(ArrayAssignmentStatement n, MType argu) {
   MIdentifier id = (MIdentifier)n.f0.accept(this, argu);
   MVar newVar = ((VarContainer) argu).getVar(id.getName());
   checkVarEqual(id, newVar, "int[]", "Not an array \'" + id.getName() + "\'");

   MType exp1 = n.f2.accept(this, argu);
   checkExpEqual(exp1, "int", "Index is not an Integer");

   MType exp2 = n.f5.accept(this, argu);
   checkExpEqual(exp2, "int", "Type mismatch, assignment value is not an Integer" );

   n.f6.accept(this, argu);
   return null;
}
```

Check Type Match When Using an Array

Another error is the argument passed to procedure. The check function is defined in class “MMethod”. It record the number of parameters and their types when building the symbol table. So what I need to do here is just calling its query API for correct parameter number and types, and then check whether the calling is legal.

```
public String insertParam(MVar newParam){
    this.paramList.add(newParam);
    return insertVar(newParam);
}
public MVar getParam(int th){
    if (paramList.size() <= th){
        return null;
    }
    else{
        return paramList.get(th);
    }
}
```

API Implementation

The error of undefined variables is found out in this process. The basic idea is to check whether it's in the corresponding table. It's just a hash table's query operation in essence. See the following snippet for a example.

```
//check {exp} is whether the type of {target}
public void checkExpEqual(MType exp, String target, String errmsg){
 if (exp == null){
   System.err.println("null exp in checkExpEqual?");
   return;
 }
 if (target!= null){
   if (!allClassList.classEqualsOrDerives(exp.getType(), target) && !exp.getType().equals(target)){
     ErrorPrinter.print(errmsg, exp.getRow(), exp.getCol());
   }
 }
}
```

Valid Expressions Check Example

I did not meet annoying bugs in the very first part of my compiler. Considering it's the first time to rewrite visitor, I spend over half of my time on learning how to use JTB. It's easy to understand the idea of traversing a AST, but in fact the details of visiting a node is not easy at all. For example, when you want to access a particular attribute of a node, you have to go to the definition to check. Things start to get better when I get familiar to its rules.

2. Stage1 - MiniJava to Piglet

1. Goal and Design

In this stage, we are supposed to translate MiniJava code into Piglet code. The Piglet code is a kind of intermediate code which does not restrict the number of register to be used. The most important part of this stage is to allocate memory for all kinds of data structure.

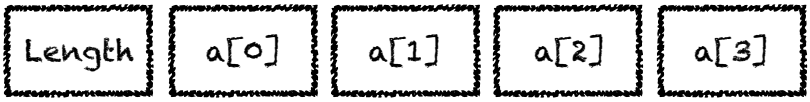
```
public class Minijava2Piglet{
    public static String translate(String raw){
        InputStream in = new ByteArrayInputStream(raw.getBytes());
        try{
            Node root = new MiniJavaParser(in).Goal();
            MClassList allClassList = new MClassList();
            //build symbol table
            root.accept(new BuildSymbolTableVisitor(), allClassList);
            Minijava2PigletVisitor newvis = new Minijava2PigletVisitor();
            allClassList.updateVarAndMethodTable();
            newvis.setAllClass(allClassList);
            newvis.setCurrentTemp(allClassList.alloc(20));

            MPiglet smackdown = root.accept(newvis, new MIdentifier()); //WWE meme... nevermind
            return smackdown.toString();
        }catch (ParseException e){
            e.printStackTrace();
        }
        return "fuck you";
    }
}
```

There are two parts of translating. The first is building the symbol table, while the other is translating. Building the symbol table is almost the same as the type check stage, but memory allocating should be done here. Translating part is not that difficult if the allocation has been done, the main job goes to address calculation. The main procedure has been given as above. We are not done yet! More need to be discussed on how to allocate memory for arrays and classes.

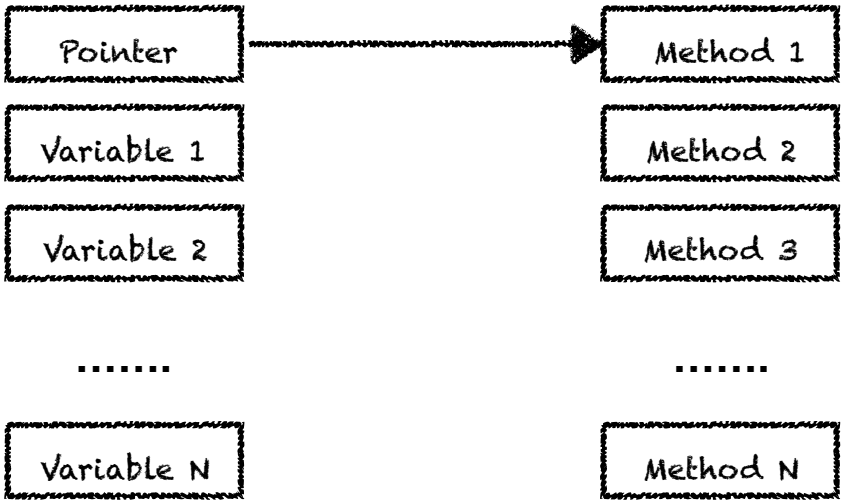
1. Array

The memory allocation strategy of arrays is a very natural idea. The first byte is saved for the length, while the rest bytes store the data. The return value of allocation is the address of the first byte, pointing to the length of it. This idea is showed by the picture below: (int[] a = new int[4])



2. Class

Comparing to arrays, classes are more complex since a class contains both variables and methods. According to the grammar of Piglet, only boolean and integer are allowed to be operated. Therefore, the key part of translating a class is to separate its methods and data. My idea refers to others' implementation and the standard of Java. A class contains two tables. One is the variable table, the other is the method table. The idea is presented as the picture below.



The idea of separating rises from a standard of Java: A class will not have memory allocated for it until it's been instantiated. Therefore the separated method table allows only one method table is created for a class. When an object is instantiated, the address of its method table pointer is returned.

Another important part of translating class is about the inheritance. MiniJava does not allow overload, but ancestor's function can be overwritten by its son. This feature is implemented with the function called "UpdateBaseClass" in package "minijava.symbol". Details can be found in source code.

3. Procedure

The essential part of translating is to pass the correct object. In MiniJava, an object is not separated, so the procedure can correctly operate the object itself. However, only integers and booleans are allowed to be operated. A solution to this is using temp0 as a special temp which pass the address of the object. More, if the number of arguments is bigger than 20, then allocate memory to save those arguments.

2. Implementation Details

1. MPiglet

This class named "MPiglet" represents a single sentence of Piglet. This kind of design will appear frequently in the following stages, because my whole compiler translates code based on single sentences. It is actually a special string builder with tab record. It implemented method similar to string, like append. The special part about it is that it record the class that contains this sentence, and it records the number of tabs. With these features, we can translate the MiniJava correctly and print formatted code. The following picture shows a part of the "MPiglet".

```
public MPiglet(String str, int nCurrentTab) {
    // append tab to the code
    this.isHead = true;
    this.pigCode = new StringBuilder("");
    for (int i = 0; i < nCurrentTab; ++i)
        this.pigCode.append("   ");
    this.pigCode.append(str);
}
```

2. Translating expressions

With all these preparation, we can finally start our translation. The translating part is almost the easiest thing in this stage. All you need to do is to calculate address according to your memory allocation and literally translate the expression. You can find more details in file named “Minijava2PigletVisitor.java” in package “minijava.visitor”. I will just give a few examples to show how to translate while DFSing the AST.

```
/**
 * f0 -> PrimaryExpression()
 * f1 -> "+"
 * f2 -> PrimaryExpression()
 */
public MPiglet visit(PlusExpression n, MIdentifier argu) {
  MPiglet _ret = new MPiglet("PLUS");
  _ret.append(n.f0.accept(this, argu));
  _ret.append(n.f2.accept(this, argu));
  return _ret;
}
```

E.G.1: Translate to Postfix Expression While Traversing

```
/**
 * f0 -> "while"
 * f1 -> "("
 * f2 -> Expression()
 * f3 -> ")"
 * f4 -> Statement()
 */
public MPiglet visit(WhileStatement n, MIdentifier argu) {
  String label1 = this.getNextLabel(), label2 = this.getNextLabel();
  MPiglet _ret = new MPiglet(label1, 0);
  _ret.append(new MPiglet("CJUMP", nCurrentTab));
  _ret.append(" " + n.f2.accept(this, argu).toString() + " " + label2);
  _ret.append(n.f4.accept(this, argu));
  _ret.append(new MPiglet("JUMP " + label1, nCurrentTab));
  _ret.append("\n"+label2);
  _ret.append(new MPiglet("NOOP", nCurrentTab));
  return _ret;
}
```

E.G.2: Translate the While Expression

```
/**
 * f0 -> PrimaryExpression()
 * f1 -> "["
 * f2 -> PrimaryExpression()
 * f3 -> "]"
 */
public MPiglet visit(ArrayLookup n, MIdentifier argu) {
  MPiglet _ret = new MPiglet();
  String temp1 = getNextTemp(), temp2 = getNextTemp(), temp3 = getNextTemp();
  addBegin(_ret);
  _ret.append(new MPiglet("MOVE " + temp1, nCurrentTab));
  _ret.append(n.f0.accept(this, argu));
  _ret.append(new MPiglet("MOVE " + temp2, nCurrentTab));
  _ret.append(n.f2.accept(this, argu));
  _ret.append(new MPiglet("MOVE " + temp2 + " PLUS 4 TIMES 4 " + temp2, nCurrentTab));
  _ret.append(new MPiglet("MOVE " + temp1 + " PLUS " + temp1 + " " + temp2));
  _ret.append(new MPiglet("HLOAD " + temp3 + " " + temp1 + " 0", nCurrentTab));
  _ret.append(new MPiglet("RETURN " + temp3, nCurrentTab));
  addEnd(_ret);
  return _ret;
}
```

E.G.3: Accessing an Element in Array

```
/**
 * f0 -> "new"
 * f1 -> "int"
 * f2 -> "["
 * f3 -> Expression()
 * f4 -> "]"
 */
public MPiglet visit(ArrayAllocationExpression n, MIdentifier argu) {
  MPiglet _ret = new MPiglet();
  String temp1 = getNextTemp(), temp2 = getNextTemp(), temp3 = getNextTemp();
  addBegin(_ret);
  _ret.append(new MPiglet("MOVE " + temp1, nCurrentTab));
  _ret.append(n.f3.accept(this, argu));
  _ret.append(new MPiglet("MOVE " + temp2 + " TIMES 4 " + temp1, nCurrentTab));
  _ret.append(new MPiglet("MOVE " + temp2 + " PLUS 4 " + temp2, nCurrentTab));
  _ret.append(new MPiglet("MOVE " + temp3 + " HALLOCATE " + temp2, nCurrentTab));
  _ret.append(new MPiglet("HSTORE " + temp3 + " 0 " + temp1, nCurrentTab));
  _ret.append(new MPiglet("RETURN " + temp3, nCurrentTab));
  addEnd(_ret);
  return _ret;
}
```

E.G.4: Allocating Memory for an Array

3. Difficulties and what to improve

I think this part is the most difficult one for me. I spent almost three whole days on it, but still have bugs with the submitted version. The bug that I did not solve is the initialization of an array. Though TA told me that initialization is an option for compiler, I still think that the performance of my compiler should obey the Java's standard. More, it is really the most buggy part. I spend a lot of time dealing with the update part, and I really want to say that overriding is really a stupid feature.

As for the test sample, I used the sample on the server. I don't know whether it's a data-oriented coding process, but I just assume that my compiler works correctly. Bad things didn't happen when translating Minijava into MIPS, so I am not gonna worry about this anymore.

3. Stage2 - Piglet to SPiglet

1. Goals and Design

Actually, I just want to say that why do we need such a stage? I mean, why don't we just skip the Piglet and translate MiniJava into SPiglet from the very beginning?

No more complaints and let's take a look at SPiglet. The only thing we need to do is to translate compound sentences into simple sentences. Just traverse the AST of Piglet code and translate those need to be translated. How? Put the result into a temp and return it. That's it.

2. Implementation Details

I am not going to explain what kind of sentences I need to translate. ~~Actually, this checkpoint's sense of presence is so weak that I have almost forget the details.~~ Examine examples ~~and feel my code from your heart.~~

```
/**
 * f0 -> Label()
 * f1 -> "["
 * f2 -> IntegerLiteral()
 * f3 -> "]"
 * f4 -> StmtExp()
 */
public MSpiglet visit(Procedure n) {
  MSpiglet _ret=new MSpiglet(n.f0.f0.tokenImage + "[" + n.f2.f0.tokenImage + "]", 0);
  _ret.append(new MSpiglet("BEGIN", 0));
  MSpiglet sp1 = n.f4.accept(this);
  _ret.append(sp1);
  _ret.append(new MSpiglet("RETURN " + sp1.getTemp(), TAB_NUM));
  _ret.append(new MSpiglet("END", 0));
   return _ret;
}
```

E.G.1: ...

```
/**
 * f0 -> "HSTORE"
 * f1 -> Exp()
 * f2 -> IntegerLiteral()
 * f3 -> Exp()
 */
public MSpiglet visit(HStoreStmt n) {
 MSpiglet _ret=new MSpiglet();
 MSpiglet sp1 = n.f1.accept(this);
 MSpiglet sp2 = n.f3.accept(this);
 _ret.append(sp1);
 _ret.append(sp2);
 _ret.append(new MSpiglet("HSTORE " + sp1.getTemp() + " " + n.f2.f0.tokenImage + " " + sp2.getTemp(), TAB_NUM));
   return _ret;
}
```

E.G.2: ...

4. Stage3 - Spiglet to Kanga

1. Goal and Design

In the stages before, there is no restriction on the number of registers. However, in real world, you'll only have several registers so you have to arrange this precious resource very carefully. The Kanga language starts to use running stack to pass arguments when calling procedure. More, there is no return sentences.

The main task of this stage is to arrange registers. Another thing that need to be done is to pass arguments correctly into the stack. For me, traversing is done for twice. The first time is to do the liveness analysis, and the second time is to translate according to the first time's result.

```
public class Spiglet2Kanga {
    public static String translate(String str) {
        InputStream is = new ByteArrayInputStream(str.getBytes());
        try{
            Node root = new SpigletParser(is).Goal();
            LivenessVisitor v1 = new LivenessVisitor();
            Spiglet2KangaVisitor v2 = new Spiglet2KangaVisitor();
            Env env = new Env();
            root.accept(v1, env);
            env.alloc();
            root.accept(v2, env);

            return env.KangaCode.toString();
        }catch (ParseException e){
            e.printStackTrace();
        }
        return "fuck you";
    }
}
```

Brief Summary of What This Stage Should Do

Here gives the problem description of data-flow analysis and solutions.

It is the process of collecting information about the way the variables are used, defined in the program. Data-flow analysis attempts to obtain particular information at each point in a procedure. Usually, it is enough to obtain this information at the boundaries of basic blocks, since from that it is easy to compute the information at points in the basic block. In forward flow analysis, the exit state of a block is a function of the block's entry state. This function is the composition of the effects of the statements in the block. The

entry state of a block is a function of the exit states of its predecessors. This yields a set of data-flow equations:

For each block b:

$$out_b = trans_b(in_b)$$

$$in_b = join_{p \in pred_b}(out_p)$$

In this, $trans_b$ is the transfer function of the block ***b***. It works on the entry state in_b, yielding the exit state out_b. The join operation ***join*** combines the exit states of the predecessors $pred_b$ of ***b***, yielding the entry state of ***b***.

After solving this set of equations, the entry and/or exit states of the blocks can be used to derive properties of the program at the block boundaries. The transfer function of each statement separately can be applied to get information at a point inside a basic block.

Each particular type of data-flow analysis has its own specific transfer function and join operation. Some data-flow problems require backward flow analysis. This follows the same plan, except that the transfer function is applied to the exit state yielding the entry state, and the join operation works on the entry states of the successors to yield the exit state.

The entry point (in forward flow) plays an important role: Since it has no predecessors, its entry state is well defined at the start of the analysis. For instance, the set of local variables with known values is empty. If the control flow graph does not contain cycles (there were no explicit or implicit loops in the procedure) solving the equations is straightforward. The control flow graph can then be topologically sorted; running in the order of this sort, the entry states can be computed at the start of each block, since all predecessors of that block have already been processed, so their exit states are available. If the control flow graph does contain cycles, a more advanced algorithm is required.

What I’ve done is the backward flow analysis. The basic algorithm is like the following:

```
for i ← 1 to N
    initialize node i
while (sets are still changing)
    for i ← 1 to N
        recompute sets at node i
```

In my code, the ***out*** set was calculated as the following code snippet. “nextStmt1” and “nextStmt2” record the exit point of a block. It actually construct a flow graph. My code calculate the ***out*** set, which can reveal the set of variables that can be used before redefined. And therefore register arrangement can be done later.

```
for (int i = 0; i < stmtList.size(); ++i){
    NStmt stmt = stmtList.get(i);
    if (!stmt.isUnconditionJump && i+1 < stmtList.size())
        stmt.nextStmt1 = stmtList.get(i+1);

    if (stmt.jumpLabel != null)
        stmt.nextStmt2 = labelStmt.get(stmt.jumpLabel);
}

while (true){
    boolean flag = false;
    for (NStmt stmt : stmtList){
        flag |= update(stmt, stmt.nextStmt1);
        flag |= update(stmt, stmt.nextStmt2);
    }
    if (!flag) break;
}
```

Backward Flow Analysis

2. Implementation details

The code can be found in package “spiglet”.

1. First round

I’ve defined data structures to implement my translation code. “Env.java” record the scanning process. It record the current block and the translating sentence. “NStmt.java” describes a single sentence of SPiglet. It record the type of the sentence, and necessary information for the coming data-flow analysis, including entry point and exit point. The definition of basic block can be found in “NMethod.java”. It implement the backward flow analysis and allocate register for variables.

The basic idea of liveness visitor is translate every sentence into an object of “NStmt”. What it need to do is to record the necessary information. After finishing record, call “env.alloc” to do data-flow analysis and allocate

register. The register allocation strategy is simple. Arrange from s0 to s7 then from t0 to t7. If the number of register needed is too big, then spill argument. In the first round, only a number is given to the variable which will be used later in the second round.

The following pictures show what my code do.

```
/**
 * f0 -> "CJUMP"
 * f1 -> Temp()
 * f2 -> Label()
 */
public String visit(CJumpStmt n, Env argu) {
    String _ret=null;
    NStmt stmt = new NStmt("CJumpStmt");
    stmt.jumpLabel = n.f2.f0.tokenImage;
    stmt.addUsedTemp(n.f1.f1.f0.tokenImage);
    argu.currentMethod.addStmt(stmt);
    return _ret;
}
```

Instantiating an NStmt object

```
public void allocReg(){
    buildGraph();
    nSpilledPara = Math.max(0, nParaSize-4);
    regForTemp = new TreeMap<Integer, Integer>();
    nSpilledSize = 0;
    for (int temp1 : tempSet)
        regForTemp.put(temp1, nSpilledSize++);
}

public int getReg(int t){
    if (tempSet.contains(t) == false) return -1;
    return regForTemp.get(t);
}
```

Allocating Register Number

2. Second round

The real register allocating comes in this part. The following code snippets describes my register allocating strategy.

```
public String getReg(int t){return (t < 8) ? "s"+t : "t"+(t-8);}
public void move(String str, Env argu){
  if (argu.moveToReg == -1) return;
  int t = argu.moveToReg;
  argu.moveToReg = -1;

  String _ret = "";
  if (t < 18){
    _ret = getReg(t);
    argu.append("MOVE " + _ret + " " + str);
  }
  else{
    argu.append("MOVE v0 " + str);
    argu.append("ASTORE SPILLEDARG " + (t+argu.currentMethod.nSpilledPara) + " v0");
  }
}
```

Register Allocation Strategy

The other part is not that difficult if the register is correctly arranged. Just remember that “PASSARG #” starts from 1 instead of 0, which means “SPILLEDARG #-1” is the correct instruction.

```
/**
 * f0 -> "HLOAD"
 * f1 -> Temp()
 * f2 -> Temp()
 * f3 -> IntegerLiteral()
 */
public String visit(HLoadStmt n, Env argu) {
   String _ret=null;
   String r2 = n.f2.accept(this, argu);

   int t = Integer.valueOf(n.f1.f1.f0.tokenImage);
   t = argu.currentMethod.getReg(t);
   if (t < 18)
     argu.append("HLOAD " + this.getReg(t) + " " + r2 + " " + n.f3.f0.tokenImage);
   else{
     String rv = "v" + (argu.vReg++);
     argu.append("HLOAD " + rv + " " + r2 + " " + n.f3.f0.tokenImage);
     argu.append("ASTORE SPILLEDARG " + (t+argu.currentMethod.nSpilledPara) + " " + rv);
   }

   return _ret;
}
```

E.G.1: Translating HLOAD

```
/**
 * f0 -> "TEMP"
 * f1 -> IntegerLiteral()
 */
public String visit(Temp n, Env argu) {
   String _ret=null;
   int t = Integer.valueOf(n.f1.f0.tokenImage);
   t = argu.currentMethod.getReg(t);
   if (t < 18){
     if (t < 8)
       _ret = "s" + t;
     else
       _ret =  "t" + (t-8);
   }
   else{
     if (argu.isPassingPara >= 0)
       _ret = "v0";
     else
       _ret = "v" + (argu.vReg++);
     argu.append("ALOAD " + _ret + " SPILLEDARG " + (t+argu.currentMethod.nSpilledPara));
   }
   if (argu.isPassingPara >= 0){
     if (argu.isPassingPara <= 3)
       argu.append("MOVE a" + argu.isPassingPara + " " + _ret);
     else
       argu.append("PASSARG " + (argu.isPassingPara-3) + " " + _ret);
     argu.isPassingPara++;
   }
   return _ret;
}
```

E.G.2: Translating TEMP into Running Stack

3. Things that I need to improve

Professor said that this is the most difficult part of the whole compiler, and the register arrangement is free for everyone to try. Redundant code can be removed during this stage. However, I was in a busy time period when doing my checkpoint 4. So I do some robust thing on this stage. Code like “procedure[*][*][20]” appears frequently in my translated code.

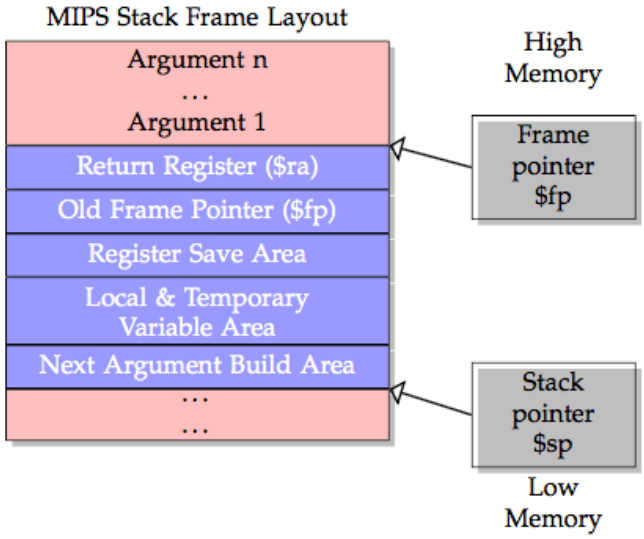
I feel really ashamed for my less-efficient compiler. If I could have another chance, I will try my best to remove the redundant code and arrange my register in a much more careful way. Nevertheless, I still want to say that a benchmark should be provided on this stage. Definitely I will have more reason to improve my compiler if I can see my compiler’s performance in a real number. I guess there is not another chance for me to write a minijava compiler when I am in Peking University. It is probably the last time in my life to write a compiler, so it’s a pity for me. Whatever.

5. Stage4 - Kanga to MIPS

1. Goals and Design

Another easy stage. The Kanga code should be translate to MIPS instructions that can correctly run on the SPIM simulator. I thought it was not a trash job until I realized that this SPIM was a next level shit. Misinterpreted instructions like “addu r1 r2 i” can run on it.

Anyway, the goal of this is to maintain a stack that fit the MIPS standard. The MIPS stack standard is depicted by the following picture. (Credit goes to Zhang Chi.)



2. Implementation

Just don't forget to add a ‘\n’ after a print statement. Do not use $a0 and $v0 incase there is a system call. Here comes the example code snippets:

```
void beginProc(String name, int stacknum, Env argu){
    argu.append(".text\n.globl " + name);
    argu.append(name + ":");
    argu.append("sw $fp,-8($sp)"); // frame pointer
    argu.append("sw $ra,-4($sp)"); // return address
    argu.append("move $fp,$sp"); // exchange fp & sp
    argu.append("addi $sp,$sp, " + ((~((stacknum+3)*4))+1)); // update sp
}

void endProc(int stacknum, Env argu){
    argu.append("lw $ra,-4($fp)");
    argu.append("lw $fp,-8($fp)");
    argu.append("addi $sp,$sp, " + (stacknum+3)*4);
    argu.append("jr $ra\n"); // return
}
```

E.G.0: Maintaining the Stack

```java
/**
 * f0 -> "PRINT"
 * f1 -> SimpleExp()
 */
public ReturnType visit(PrintStmt n, Env argu) {
   ReturnType r1 = n.f1.accept(this, argu);
   if (n.f1.f0.choice instanceof IntegerLiteral)
       argu.append("li $a0, " + r1.value);
   else if (n.f1.f0.choice instanceof Label)
       argu.append("la $a0, " + r1.value);
   else if (n.f1.f0.choice instanceof Reg)
       argu.append("move $a0, " + r1.value);
   else
       assert(false);

   argu.append("li $v0, 1"); // syscall 1
   argu.append("syscall");

   argu.append("la $a0, endline"); // print need to print an endline
   argu.append("li $v0, 4"); // syscall 4
   argu.append("syscall");

   return null;
}
```

E.G.1: System Call to Implement PRINT

```java
/**
 * f0 -> "CALL"
 * f1 -> SimpleExp()
 */
public ReturnType visit(CallStmt n, Env argu) {
   ReturnType _ret=null;
   ReturnType r1 = n.f1.accept(this, argu);
   Node choice = n.f1.f0.choice;

   if (choice instanceof IntegerLiteral){
       argu.append("li $v0," + r1.value);
       argu.append("jalr $v0");
   }
   else if (choice instanceof Label)
       argu.append("jal " + r1.value);
   else if (choice instanceof Reg)
       argu.append("jalr " + r1.value);
   else assert(false);

   return _ret;
}
```

E.G.2: Translating CALL

6. Final Stage - Minijava to MIPS

In my code design, every translation from A to B is implemented with a class called “A2B.java”. It reads A code from standard input and output translated B code to standard output. Then the final procedure is organized in the code snippet below. You can find more details in package “all”.

```
public class Main{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        String str = new String("");
        while(sc.hasNext())
            str += sc.nextLine() + "\n";
        str = Minijava2Piglet.translate(str);
        str = Piglet2Spiglet.translate(str);
        str = Spiglet2Kanga.translate(str);
        str = Kanga2Mips.translate(str);
        System.out.println(str);
    }
}
```

Here is the end of my detail description. Hooray!

Chapter 3: Conclusions

In this chapter, I am going to conclude what I learned from writing this compiler. And of course, I am going to point out some problems about this course as professor required. Some of them come with my specific advice, the others don't. After all, I basically enjoyed this course and will recommend it to my friends.

1. What I have learned from building a compiler

What did I learn after spending more than 3 months writing my own compiler? I have to say that the first thing come to my head is that my programming skill in Java improved a lot, really. I took Java Programming last semester but all I did was just translating C++ into Java. However, this course forced me to understand some unique features of java, including inheritance and generics. It is also the first time for me to write such a big Java project on my own. Designing and Debugging are both challenging for me. I guess I will never forget those nights when I was scratching my head, staring at the screen and trying to fix a bug.

Another thing is that, I have a deeper understanding on the latter part of the theoretical course about compiler through writing my own compiler. I always believe that practicing is the second best way to learn something(teaching is the best :D). AST was just abstract like its name, and intermediate code was like imagination. However, those definitions come to real when building your own compiler. Thanks to this course. (I swear it's not sarcasm.)

~~I almost forget to mention that the TA of this course is one of the best I've ever met. Does his WeChat count as what valuable things I have got in this course? Just kidding. :D~~

2. What can be improved in this course

I have to admit that this course is well-designed as a practice of compiler design. However, it is not designed to be a synchronized part of the theoretical course. Nothing useful was taught during the first month, and the reason is that we had not finished the syntax part. In fact, mastering the use

of Yacc and Lex does not need the knowledge of lexical and syntax part. I think a student majored in computer science should be able to learn how to use them himself. This flaw directly resulted in the lack of time when doing the latter checkpoints. To be honest, I did not have enough time to think about optimization and implement it when allocating registers on checkpoint 4. Please consider skipping the first two lectures, because they are completely useless for us when building a compiler.

Another problem is that the optional part's requirement is not clearly listed in the handout materials. Nobody knows what they can do as a bonus and nobody knows what bonus they can get if they do more. I could not deny that I'm a score-driven student, and I guess most of students in Peking University are the same. For me, I just do not want to spend time doing extra works when I even don't know what bonus I can get. Well, I feel regretted now that I did not spend much time trying to optimize my compiler when translating Spiglet to Kanga. However, I have convinced myself that it's not my fault. Please consider this and clarify the optional part next year.

The last thing I would like to mention is not that specific as the above problems. When I writing my own compiler, I feel like I am doing this as instructed. I know the difference between those languages, but why do I have to divide my compiler into 5 parts? I still don't know now. I hope that the professor of this course could focus more on the idea about how a compiler is designed, instead of how can I implement a specific feature.

Appendix

Tools used:

JavaCC 6.0
JTB 1.3.2
QtSpim 9.0.

Reference Sites:

UCLA CS132 —— http://web.cs.ucla.edu/~palsberg/course/cs132/project.html
Github —— Others' implementation of MiniJava compiler
Google
Stack Overflow

Thanks to:

Gao Zheng, TA.
Wang ZiChen.
Wang ZiChang.

The source code can be found on the server.
This is only the report of my compiler.