Assignment 5: Code Generation

Zhenyan Zhu - z277zhu Zijian Feng - z63feng Richella Li - j2333li

Instructions

- To build the compiler, run **mak**e at the root directory.
- Make sure it has a output dir in the root dir
- To compile a list of java file, we use the following usage:
- ./joosc [java file containing Test methods] [other java test files] [stdlib files]
- All the asm file produced will be stored in output/

Design and Implementation

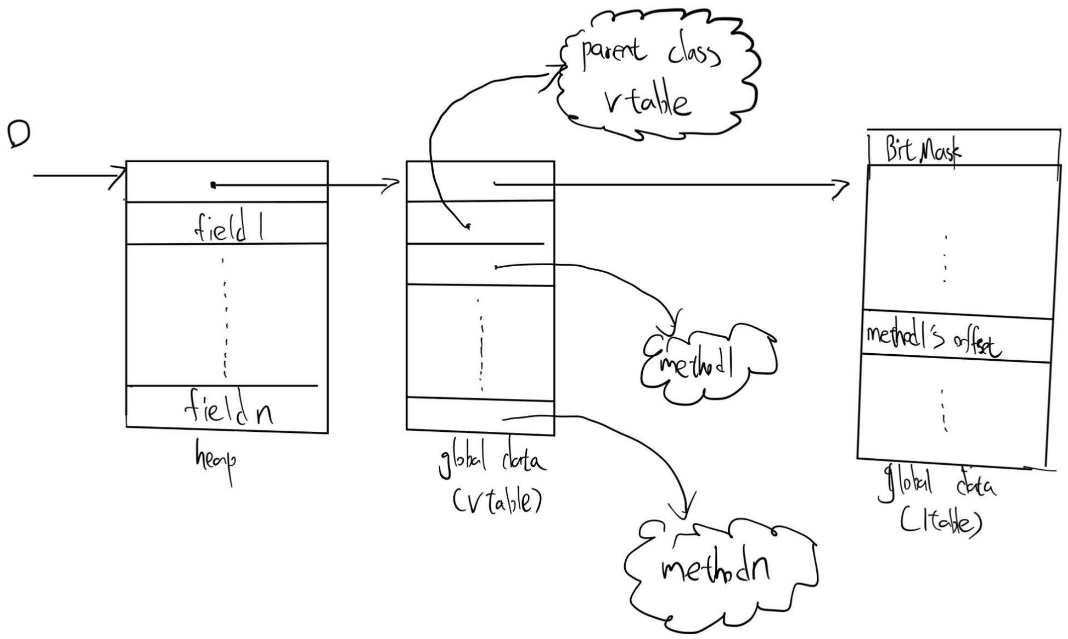
- Project file structure

  Our project is developed with java and a structure with sbt as our build tool.
  The structure of the files are listed below:

```
- build (store our intermediate files while building.)
- lib (store the third-party library and also the package we built)
- output (produced output asm file here)
- src (store the source files)
  |-- ast (the definitions of every Node in the AST Tree).
  |-- backend (asm file, ir translator and register allocator)
  |-- dataflowAnalysis (the definition of control flow graph, aka, CFG)
  |-- exception (the definition of SemanticError thrown during compiling.)
  |-- flex (jflex/cup config files)
  |-- hierarchy (the implementation of Hierarchy Checking)
  |-- lexer (parser/lexer generated by flex)
  |-- type (TypeChecking implementation including Environment builder and Type checker)
  |-- utils (some common tools functions used regarding the AST tree.)
  L-- visitors (TypeCheckVistor, StaticCheckVistor and IRVisitor that used in the Visitor Design
Pattern)
- stdlib (java standard library source code)
- test (store the test cases and some bash scripts for testing)
- boot.sh (boot the docker env)
- Dockerfile (to build the docker env)
- Makefile (build this compiler)
```

- IR Node
  - After type checking, we implement an IR translator. It stores:
    - `List<CompilationUnit>` **`comps`**: store ast node of each compilation unit
    - `List<CompUnit>` **`ir_comps`**: IR version of above comps

        - `IRTranslatorVisitor` **`visitor`**: IR translator has its own visitor which visits each AST node and translates it into its IR version. We travel them in the order of passing compilation units.
    - Special IR node - CompUnit:
        - `Set<String>` **`externStrs:`** store labels from other CompUnits that are used here.
        - `Set<String>` **`definedLabels:`** store labels defined in the CompUnit
        - `List<FieldDecl>` **`staticFields:`** store static fields in the CompUnit
    - Init static field: we create our static field initialization process after all the IR translation is done. We produce an IR version of the static field initialization process and store it as a FuncDecl in the first CompUnit.
- Calling Conversion
    - The first argument is the receiver, and then followed by normal arguments.
    - Before calling the function, caller saves all the arguments into the stack.
    - Callee is responsible for saving the caller's ebp. Before returning, it should also pop temp storage and caller's ebp.
- Canonicalize IR Code
    - We also canonicalize each ir node using the visitor design pattern (`CanonicalizeVisitor`). We visit each ir node in ir_comps and store its own canonical type in canonicalized_node field in the ir node
    - Canonicalize Cjump
        - We canonicalize Cjump by canonicalize Cjump's target expression and separate the false label to a individual Jmp IR
- Object memory layout



    - Memory layout for each object is shown above: each object is pointed to some starting address of a heap. The first slot is the memory address that points to its class Vtable that is stored in the global data. The other slots store field values in the order of class superiority and then field defined order.
    - In each class's Vtable, the first slot is the memory address of the Itable that is stored in the global data. The second slot is the memory address of its super class's Vtable (if the class is java.lang.Object, the value of that slot is 0), which is used for instanceOf test. The other slots store the method addresses of that class's containMap.
    - In each class's Itable, the first slot is the BitMask. When performing an 'and' operation between the value of the BitMask and the hashCode of one interface method, we will get the position of that interface method in the Itable, which stores the value of its implemented method offset in Vtable.
- Tiling Canonicalized IR Code
    - To follow the tiling algorithm, we implement a special tiling visitor (`TilingVisitor`), it will skip some nodes that have been tiled by a parent node and go straight to child nodes given by the parents, and bind them together after visiting all child nodes:
        - A 'Tile' is a list of 'Code' (Code is the AST for x86 assembly), and each 'Code' and 'Tile' has its own toString method, which can directly write assembly files by java.io.PrintWriter
        - `Pair<List<Node>, Tile> res_pair = n.tiling(`**`this`**`):` tile the current node, produce its child nodes that we want the visitor to continue visiting and producing tiling from that point, and its own tiling.
- Register Allocation
    - The register allocation algorithm we implemented is the naive way: putting every abstract register onto the stack. In tiling visitor, whenever we tiled a functionDecl, we will keep track of this function at Register.java. Then, whenever we create a new instance of abstract register, we will add such register into the corresponding functionDecl's chunk. (A chunk is a hashmap that maps each abstract register to its offset of ebp, all abstract registers in that chunk will be put onto the stack when the function is called)
    - When generating real assembly codes, we can divide instruction into four categories:
        - BinOpCodeS: an instruction that reads two operands and stores the result in the first operand (add, sub, imul…). When allocate register for this kind of code, we load the value of abstract registers into ecx, edx, eax (allocated in order if
        - needed); perform the instruction, then store the value back to the left operand
        - BinOpCodeL: an instruction that reads two operands and will not store the value in the first operand (cmp, test …). The allocation step is similar to BinOpCodeS, but we do not perform the store back instruction.
        - UnaryOpCodeS: an instruction that does some process and stores the result in the first operand (pop, not …). We set its operand to ecx, perform the instruction, and store back to the memory address of the original abstract register.
        - UnaryOpCodeL: an instruction that loads the first operand and does some process (call, push, jmp, jcc …). We loads the value of the original abstract register into ecx and do process.
- Create Assembly

    - We generate an assembly file based on each compilation unit, for the first compilation unit we see, we will add some extra procedures: add `global _start`, `_start:`, and call static field initialization process `call staticInit`. Finally, call test process, extract the return code and exit.
    - For the remaining part of the first asm file and all other asm file generation, we follow the same procedure:
        - Section .text
        - Extern labels needed by using in `externStrs` each ir_comp.
        - Tile code and allocate register, then write the asm of the ir_comp following the extension.
        - Section .data
        - Write string literal used in the ir_comp
        - If the ir_comp is a classdecl, we write its itable and vtable here, note all the objects from the same class use the same itable.
        - Write static fields declared in the ir_comp if there are any.

Testing:

- Multiple object from the same class
- Nested static fields
- Complex Inheritance cases
- Method Invocation
- String addition for class case and nested cases.
- Array cases

Known issues:

The only issue is that our scope environment can only save fields, methods and constructors with different names (different methods/constructors can have the same name for overloading). However, after finishing A3, we notice that fields can have the same name with constructors as well as methods, and if we want to solve this case, everything would be restructured and rewritten, so we ignore this case.

Work

- Zhenyan Zhu - z277zhu
    - Designed and implemented IR visitor framework
    - Translated expressions(non-OO), statements(non-OO) into IR
    - Designed and implemented Canonicalize visitor framework
    - Canonicalized Move, BinOp, Mem, Eseq
    - Designed and implemented Tiling visitor framework
    - Designed and implemented assembly AST/Tile structure
    - Tiled Move, BinOp, Mem into abstract assembly
    - Designed and implemented whole register allocation

    - Designed Object memory layout
    - Reconstructed Name disambiguation to extract receiver and following fields for each ambiguous name
    - Designed and implemented static/non-static fieldAccess
    - Designed and implemented non static MethodInvocation
    - Moved Vtable/Itable to global data section
    - Designed and implemented InstanceOfTest
    - Designed and implemented Cast between ClassOrInterfaceType
    - Designed and implemented Cast between Array and Object
    - Designed and implemented implicit conversion between numeric types
    - Designed and implemented String concat
    - Designed and implemented String Literal, stored in global data
- Zijian Feng - z63feng
    - Translated conditional expr, array creation and array access into IR.
    - Tilling memory access to string
    - Tilling label, cjump, jump, move memory, name.
    - Design and Implemented Cjump statement.
    - Design and Implemented Interface and Itable.
    - Design and Implemented addition for String, Object.
    - Testing script for a5.
- Richella Li - j2333li
    - Translated MethodInvocation, MethodDecl, IfStmt, WhileStmt, RelatioExpr, EqualityExpr, ConditionalExpr into IR
    - Design and Implemented Calling Conversion
    - Canonicalize Call, Move
    - Tile MethodInvocation, MethodDecl
    - Designed and implemented ASM file structure
    - OO: implemented classInstanceCreation, constructor call, heap malloc, vatble, and implemented MethodMap and FieldMap to recognize offsets.
    - Designed and implemented Static Field Initialization process

Thoughts

How much time did you spend on the assignment? What did you like or dislike about the assignment? What would you change about it?

Richella Li	**Zhenyan Zhu**	**Zijian Feng**	**Total Hours**
100	**120**	**80**	**300**

Like:

- We really like the idea of IR and Canonicalized IR. Compared with directly translating it into asm, the intermediate steps make the translation steps more clearer and easier.

- Using the simulator and Canon checker, we can easily verify the correctness of IR translation of non-OO codes.
- Translation in stages is surely instructive for the code generation. If not, this part would be far more difficult.
- The vtable and itable design is ingenious.

Dislike:

- The final assembly translation and register allocation is very tricky to debug and not indirect to detect anything strange.
- Sometimes during the implementation, we struggled to find sources for some definitions, like X86 instruction and so on. Some related textbooks included in the sources would be really helpful.
- Instructions to implement the details of Casting/fieldAccess/Register allocation are insufficient

What aspects of your chosen programming language did you find helpful in writing the compiler?

- Java has plenty of third-party libraries, such as Jflex and Jcup, which gives us lots of help when implementing the front-end of the compiler.
- Java's instanceOf and Cast operations make coding easily
- Each class in java can overwrite its toString method, which makes us clearly observe the inner structure of objects

What aspects caused annoyance?

- Java does not support any reference propagation. That is, for many scenarios where a pointer is needed, we cannot replace the whole Object.
- When throwing an exception in a method, we have to annotate the type of exception. If this method is called in other methods, we have to annotate the exceptions in those methods as well, which is quite annoying.

Any other thoughts?

z63feng: After learning joos,we learnt every detail of Java grammar. We clearly understand every stage for a statement to run. Moreover, we saw how extremely powerful the visitor design pattern is. Without a design pattern, combining every single implementation will be a total mess.

z277zhu: Writing Compiler is painful, challenging as well as interesting. CS444 taught me how a real OO language is implemented in detail and how to design, build, and implement a large project cooperating with other smart students. Besides, professor Yizhou taught us a lot of theoretical algorithms that modern compilers will choose to implement. In A5, I made several incorrect decisions that produced potential but dangerous bugs that took us a long period to fix, such as we used Temp("_THIS") for every this literals in the whole programs, but when there are nested methods calls, the value of that Temp will be overwritten many times and we can hardly reload previous values. All in all, completing such a huge project is really fulfilling and gives me plenty of confidence for future study.

j2333li: This term learning experience offers me a brand new view of programming. I could never dig down so deeply into a programming language. The time spent on implementing the compiler did help me understand the true core of the programming language much better. Moreover, the freedom of making our own decisions regarding the compiler implementation gave me not only happiness but also frustration. There were many times we designed our own compiler structure, implemented partially and then found it not working. We struggled with such failure for a lot of time, but the failure also taught us a lot. Every time we failed, we stood up with a clearer idea of what we should do next. This course also gives me the opportunity to be with a bunch of small students and working towards a big project. I feel really thankful for this valuable experience.